# Flexible Data-Aware Scheduling for Workflows over an In-Memory Object Store

Francisco Rodrigo Duro*, Javier Garcia Blas†, Florin Isaila‡, Justin M. Wozniak§, Jesus Carretero¶ and Rob Ross‖
*†‡¶Computer Architecture, Communications and Systems Group
University Carlos III Madrid, Madrid, Spain
§‖Mathematics and Computer Science Division
Argonne National Laboratory, Lemont, IL, USA
{frodrigo*, fjblas†, florin‡, jcarrete¶}@arcos.inf.uc3m.es, {wozniak§, rross‖}@mcs.anl.gov

***Abstract*—This paper explores novel techniques for improving the performance of many-task workflows based on the Swift scripting language. We propose novel programmer options for automated distributed data placement and task scheduling. These options trigger a data placement mechanism used for distributing intermediate workflow data over the servers of Hercules, a distributed key-value store that can be used to cache file system data. Thus, mixed-mode scheduling techniques can be used for exploiting the locality of intermediate data in either compulsory or advisory mode. We demonstrate that these new mechanisms can significantly improve the performance of many-task workflows with up to 73x for writes, up to 130x for reads, and up to 86x for the whole workflow. These results are due mainly to reducing the contention on the shared file system, exploiting the data locality, and trading off locality and load balance.**

***Index Terms*—scientific workflows; file systems; data locality; load balance; high performance**

## I. Introduction

Scientific workflows consist of interdependent tasks that communicate through intermediate storage abstractions, typically files. The increasing availability of data generated by high-fidelity simulations and high-resolution scientific instruments in domains as diverse as climate [1], experimental physics [2], bioinformatics [3], and astronomy [4], has shown the file system to be a substantial performance bottleneck. While typical high-performance computing (HPC) systems use a monolithic parallel file system, data-intensive workflow implementations must borrow techniques from the big data computing (BDC) space, such as exposing data storage locations and scheduling work to reduce data movement.

Swift [5] is a system for the rapid and reliable specification, execution, and management of large-scale science and engineering workflows. Swift software combines a simple scripting language to enable the concise, high-level specifications of complex parallel computations, as well as mappers for accessing diverse data formats in a convenient manner. The Swift/T runtime execution engine [6] can manage the dispatch of many (trillions of) tasks to many (millions of) processors [7], whether on parallel computers, campus grids, or multisite grids.

This work addresses the performance bottleneck posed by the limited scalability of shared file systems to Swift workflow execution. We propose a novel scalable solution for improving the performance of Swift/T through Hercules, a distributed in-memory put/get store. More exactly, the paper makes the following contributions. First, we design and implement a novel scheme for using Hercules as a fully distributed store for both metadata and data involved in the execution of any workflow. Second, we design and implement a novel informed data placement mechanism for Hercules, which allows a higher-level implementation to control the data placement system wide based on specific criteria. Third, we propose novel workflow-aware task and data placement mechanisms that combine Swift load-balancing capabilities and Hercules data and metadata distribution functionality for implementing various locality-aware and load-balancing policies. The data placement strictly enforces data locality and constrains task placement. The locality-aware hard task placement mechanism strictly enforces task locality and can be used for enforcing the execution of a task on the node where the data resides. The locality-aware soft task placement offers a best-effort alternative, which allows users to dynamically trade-off locality and load balance.

The remainder of the paper is organized as follows. Section II shortly describes Swift and experimentally motivates the problem. Section III presents the new Swift/T task scheduling policies. Section IV describes the Hercules architecture and introduces the novel data placement policies. Section V evaluates the performance of the proposed solution. Section VI presents related work. Section VII presents our conclusions.

## II. Swift overview

This section contains a short overview of the Swift system, meant for facilitating the readability of the paper. At the end of this section we experimentally motivate the problem to be addressed, namely, the lack of scalability of the shared file system.

Swift/T differs from other workflow systems in multiple aspects. It is as much a programming language as it is a workflow language, since it translates into a high-performance MPI program for execution on large-scale machines. It emphasizes hierarchical programming, in which experimental ensembles and campaigns are expressed in the high-level Swift language,

```
 1: app (file file_output) write_file(){
 2:  "./write-local.sh" file_output;
 3: }
 4: app (void) read_file(file file_input){
 5:   "java read-local.class" file_input;
 6: }
 7: foreach i in [0:n-1] {
 8:
 9:
10:
11:    file f<file_name[i]> = write_file();

12:
13:
14:    read_file(f);
15: }
```

Fig. 1. Example of a data-parallel Swift workflow.

```
 1: app (file file_output) write_file(){
 2:  "./write-local.sh" file_output;
 3: }
 4: app (void) read_file(file file_input){
 5:   "java read-local.class" file_input;
 6: }
 7: foreach i in [0:n-1] {
 8:   set_rank(filename[i], i % nnodes)
 9:   rank1 = get_rank(filename[i])
10:   location L1 = location(rank1, SOFT, RANK)
11:    file f<file_name[i]> =
           @location = L1 write_file();
12:   rank2 = get_rank(filename[i])
13:   location L2 = location(rank2, SOFT, RANK)
14:   @location = L2 read_file(f);
15: }
```

Fig. 3. Example of a data-parallel Swift workflow, in which files are placed round-robin over the ranks (line 8). A best-effort task placement policy (SOFT) is used in lines 11 and 14 for placing both the write_file and read_file tasks on the same node with the file.

while performance-critical computation is expressed in C/C++ and Fortran. Scripting language interpreters for Python, R, Julia, and Tcl, may be optionally embedded and accessed [8], easing development.

### A. Swift language

The Swift language is a scripting language for describing data-parallel workflows. A workflow node can be a program written in any other language and is treated by Swift as a black box. The code snippet in Figure 1 illustrates a data-parallel file copy operation. Lines 1-3 define a workflow node corresponding to a bash script creating a file and returning a Swift variable representing the file. Lines 4-6 define a workflow node corresponding to a Java program reading the file received as an argument. Lines 7-15 show how *n* parallel instances of these mini-workflows can be launched with a foreach construct. The `write_file` Swift function creates the file by running the shell script, and the `read_file` Swift function launches the Java program passing the file created by the shell script as parameter. On that example, where the `read_file` task depends on the `write_file` task, the data dependencies are automatically managed by the Swift runtime.

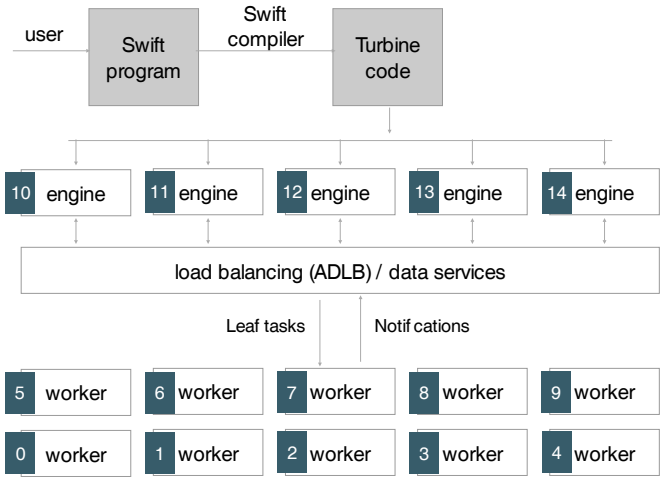



Fig. 2. Overview of Swift/T system: Swift code is compiled into the Turbine code. Turbine code is passed to the Turbine runtime engines, which launch tasks on workers, perform load balancing, and enforce data dependencies.

### B. Swift/T system

The implementation of the Swift system for supercomputers is called Swift/T. Swift/T, illustrated in Figure 2, is composed of three main components: the Swift compiler, the Turbine runtime, and the workers. The Swift compiler compiles Swift programs into an intermediary Turbine code. The Turbine runtime performs three main functions. First, it enforces the data dependencies between tasks, for instance the dependency between `read_file` and `write_file` in Figure 1. For that, it uses a key/value store for sharing Swift program variables among distributed nodes. Additionally, it employs a pub/sub mechanism for notifying the fulfillment of task dependencies. Second, it employs an adaptive load-balancing technique for choosing the workers on which the tasks run. Third, it launches the tasks on the workers.

For supporting full application portability, external files are not saved in the key/value store, which is used exclusively for internal Swift variables. In turn, external files are accessed through a shared file system. In Figure 1, the Swift `file_name[i]` variable represents the name of the file and it is associated with the variable $f$, which has the Swift type `File`. For `File` types Swift uses *futures* for the intercommunication between data-dependent tasks. A consumer task registers the file of interest to the Turbine key/value store and registers for a notification when a producer task finishes. If the data have already been produced, it is served immediately. Otherwise, the task is blocked until the producer finishes and Turbine runtime notifies the consumer.

The Swift/T system runs as an MPI program consisting of *n* processes identified by ranks from 0 to *n*-1. The lower ranks are assigned to workers and the higher ranks to runtime engines. An example of this rank assignment is shown in Figure 2, where the ranks are displayed in the black boxes placed inside the workers and engines.

### C. Experimental motivation

The current shared parallel file systems used in most supercomputer systems are known to have scalability issues when thousands of nodes access concurrently. For illustrating how this problem affects the performance of Swift workflows, we present in this section an evaluation of the simplest possible workflow shown in Figure 1. We use as workflow tasks two simple sequential C programs, one that creates a file on a shared file system (GPFS) and another one that reads the file. The Swift program executes *n* instances of this simple workflow. We performed both weak-scaling and strong-scaling evaluations. For weak scaling, we vary the number of tasks from 64 to 1,024; each task writes or reads 256 MB, and the aggregate data written or read vary from 16 to 256 GB. For strong scaling, the total number of tasks is 1,024, and only the number of workers varies from 64 to 1,024. In all cases the aggregate data written or read is 256 GB. The detailed experimental setup is shown in Section V.

Relying on file I/O for data communication between tasks exposes every possible bottleneck of the underlying shared file system. In addition to the data locality problem, file system contention increases proportionally with the number of nodes working concurrently in parallel tasks.

Figures 4a and 4b show the average read and write throughput per task while increasing the number of workers concurrently accessing the shared file system. The average task write performance drastically degrades, losing more than 90% of the achieved throughput for strong scalability. The average task read performance also drops over 80% for weak scalability. For strong scalability the last increase in performance of the reads can be attributed to caching effects: the amount of file system client-side memory increases with the number of nodes for the same data amount. Even in this case, however, the performance for 1,024 tasks is below 15% of that measured with 64 concurrent tasks. Figure 4c shows how the aggregated throughput is less penalized and is stabilized near the 1100 MB/s mark. This behavior can be explained because of the total available bandwidth of the GPFS that is shared between every concurrent task running, resulting in stable aggregated throughput but drastic degradation in the throughput achieved by each task individually.

The figure shows that the file system does not scale with a moderate increase in the number of concurrent workers. This problem is caused by both data and metadata access contention. The data contention is exacerbated by the locality agnosticism of the Swift scheduler, which places dependent read tasks on different nodes from the write tasks, causing the data to go over the shared file system.

### D. Problem definition

In this paper we target improving the performance of any application consisting of a large number of intercommunicating tasks deployed dynamically on processes of a parallel application (e.g., MPI) scheduled through a batch scheduler on a large cluster (in the same way as any Swift program). These tasks communicate through intermediary files that are dynamically generated and uniquely named. Two outstanding access patterns need to be optimized for these workflow applications: sequentially writing the whole file and sequentially reading the whole file.

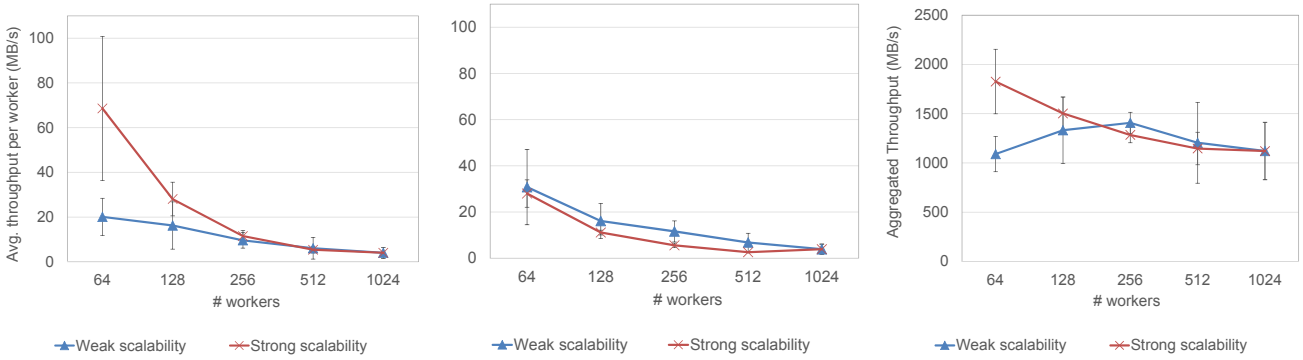
Based on these requirements, we design a solution that avoids most overheads involved in a shared parallel file system such as locking protocols for enforcing consistency and protection.

## III. Task placement for supporting locality-aware dynamic scheduling

The default scheduling scheme in Swift discussed in Section II is data locality agnostic. In this mode, the task is first placed in the work queue of the node where it is created. If no work stealing events are triggered, the task remains there until a worker attached to the server obtains the task. If another server steals the task, it can be executed on any worker in the system. Thus, this scheme is not predictable, but it allows total flexibility to the load balancer. The application of this scheme in a data-intensive use case, however, would result in a great deal of network traffic since locality is not applied.

In order to allow applications to take advantage of data locality, our solution leverages a mechanism from the Swift language for placing tasks on given ranks (specific workers) or nodes (*n* workers can be running on the same node). This is achieved by using a special type of variable called `location` and task annotation, as illustrated in Figure 3, which adds the necessary code to the example from Figure 1 between lines 8 and 14. Line 8 enforces that file $i$ is created on node $i\%nnodes$, where $nnodes$ is the number of nodes on which the program runs and line 9 returns a rank running on the node where the file is stored. Line 10 initializes a `location` variable, which is used in line 11 in a best-effort policy for placing the task locally to the data. Lines 12-14 have the same functionality for `read_file` task as lines 9-11 for `write_file` task. By commenting out line 8, the program uses the default data placement of Hercules. By commenting out lines 8-10 and the location construct from line 11, the program uses the default data placement and the default task placement for `write_file` task.

The `location` type has three attributes. The first one is the MPI rank on which a task will execute, identifying a specific worker. The second argument indicates whether the task placement is strict (`HARD`, used in the example) or advisory (`SOFT`). The third argument indicates whether the placement has to be done on the exact rank given by the first argument (`RANK`, used in the example) or on the node on which the worker process `rank` is running (`NODE`). The value `NODE` gives to the Swift load balancer more flexibility to




(a) Average throughput performance per worker of write operations in MB/s.

(b) Average throughput performance per worker of read operations in MB/s.

(c) Aggregated throughput performance of I/O operations in MB/s.

Fig. 4. Average throughput for read and write operations over GPFS for locality-agnostic execution of $n$ instances of a simple workflow.

place the tasks. The Cartesian product of the last two variables provides four modes that can be used for task placement:

- HARD RANK enforces the task placement on the given rank (worker).
- HARD NODE enforces the task placement on the node where the process with the given rank runs and dynamically selects the best-available worker in the node.
- SOFT RANK offers best-effort task placement on the given rank (worker). The load balancer can decide to move the task on another worker running on the same node or on a different node.
- SOFT NODE offers best-effort task placement on the node of the given rank. The load balancer can decide to move the task on a worker running in a different node.

The HARD RANK scheme offers predictable behavior. Load balancing, however, suffers greatly as tasks are forced to execute on a specific worker (see §V). This feature could be used selectively to perform application operations that strictly require access to a node-local file or some specific component (e.g., a node-local database). It could be useful in a system without Hercules or another data movement technique. However, it falls far short of our complete system with Hercules.

The other three schemes successively relax the task placement, letting the load balancer more freedom to improve the overall system load balance.

The most relaxed scheme is SOFT NODE. In this mode, tasks are given a rank on which to execute. If this rank is busy, another rank on the same node may take the task. If these ranks are all busy, another worker under the same server may take the task. If a server is the target of a steal and all of its tasks are soft targeted, a soft-targeted task may be taken.

These last three schemes offer various degrees of a mix of predictable and dynamic behavior. They prevent a block in dataflow progress by allowing idle workers to continue making progress even if network data movement is required, but they boost the likelihood of local data access if the necessary data locality control mechanisms described in the next section are used.

## IV. Scalable data sharing

The four schemes offer various degrees of task placement control. However, task placement has to be complemented with data locality control in order to open up an optimization space that can be used for trading off load balance and locality. In this section we describe our scalable data-sharing solution.

Our scalable aim is to improve four main aspects that contribute to the file I/O bottleneck illustrated above: metadata scalability, data scalability, locality exploitation, and file system server scalability. We will explain how our solution addresses these four issues. First, however, we give an overview of Hercules [9], the system that we have extended in this work for supporting Swift many-task workflows.

### A. Hercules overview

Hercules [9] is a key-value distributed in-memory store based on Memcached [10]. Figure 5 shows an overview of the Hercules architecture consisting of a client-side library (upper side of the figure) and a server-side back-end (lower side of the figure).

Data are distributed over the servers by consistently hashing a key. Therefore, in the original design, data location cannot be controlled, and data are always cached on the calculated server.

Hercules clients can access data through a key-value store interface, a POSIX-like interface, or MPI-IO interface. For POSIX and MPI-IO, the file name is used as key.

### B. Novel mechanisms for scalable data sharing

This section describes the Hercules-based solution for efficiently supporting many-task workflow applications as discussed in Section II-D.

For achieving *server scalability*, Hercules servers are deployed on-demand on every Swift node (one instance per node, not per worker) before the application is started. The

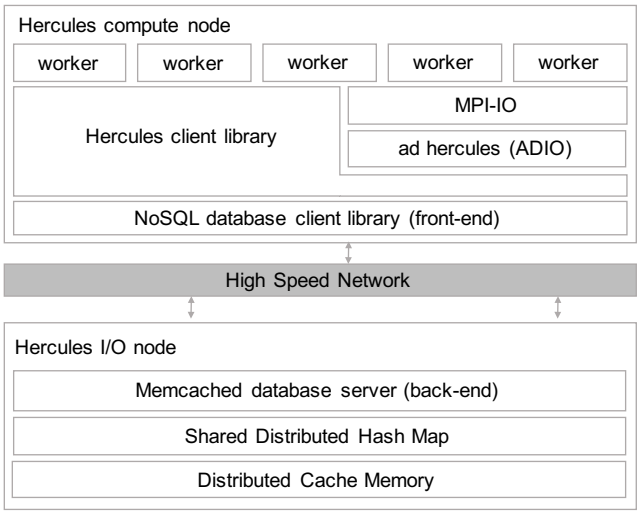



Fig. 5. Hercules architecture, consisting of a client-side library (upper side of the figure) and a server-side back-end (lower side of the figure).

main advantage of this approach is that the total memory scales with the number of worker nodes and it is available to applications as a cooperative distributed cache. In contrast, the file system suffers from two main problems. First, the client caches are non-cooperative. When the locality is not exploited, the exchange of data between tasks happens through file system server caches. Second, there is a limited amount of cache at a fixed number of file servers. Thus the access performance is likely to degrade with a substantial increase in the number of processes/tasks as a result of contention caused by both metadata accesses (large number of files) and data accesses (larger data volume). Nevertheless, this omnipresence of servers on every application nodes allows each client to potentially benefit from having data stored locally on the same node, as discussed below.

Hercules naturally offers *data access scalability* by distributing data of intermediary files over all servers. For *metadata scalability* our new solution employs two approaches. First, data management is decoupled from the metadata management: data, and metadata are stored separately in distinct objects. Optionally, the file data can be associated with the file metadata through a pointer stored in the metadata. Second, the metadata is distributed over all servers. For each file the metadata object is assigned to a server based on algorithmic-hashing using the file name as key.

For enabling *locality exploitation* by workflow engines running on top of Hercules, we leverage the omnipresence of servers on all application nodes and the separation of data and metadata management discussed above. For exposing and exploiting data locality, we offer two new Hercules API calls:

- int `get_rank(filename)` a function that returns the rank corresponding to a server where a given file is cached if the file exists
- void `set_rank(filename, rank)` a function that enforces a future file to be created at a server running on a node with given rank

The `get_rank` function allows a workflow engine to locate a file. This information can be used for scheduling a task close to the data. The `set_rank` function allows placing the data on a given server based on user-specified criteria. This function can be used to control the load or capacity balance of a Hercules server or application node. These functions rely on looking up the file metadata at a server identified by hashing the file name and getting or setting the metadata accordingly.

Since our solution targets intermediary files whose life cycle consists of being entirely written, followed by being read in their entirety, followed by deletion, there is no need for costly locking algorithms to enforce consistency, as in the case of general-purpose parallel file systems such as GPFS [11] and Lustre [12]. Sequential consistency is naturally enforced by internally reading a buffer after write completion.

## V. Evaluation

We evaluated our solution on 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. The employed interconnect is InfiniBand QDR 4 GB/s per link. GPFS is configured 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 results are 3200 MB/s using *dd*. We use TCP/IP over InfiniBand, with a maximum recorded throughput of 4 gigabit/s measured with *iperf*, and use 80% of the available main memory of the node running the service. In every Hercules case in the following evaluation, one I/O node is launched on each worker node.

### A. Filecopy microbenchmark

Our first evaluation is based on a Swift microbenchmark performing a data-parallel file copy, as shown in Figure 1. Unlike the example from Figure 1, workflow external applications `write_task` and `read_task` were written in C. The `write_task` sequentially writes a file, and the `read_task` sequentially reads a file. The microbenchmark can be run for different numbers of workers, numbers of files, file size, and chunk size (buffer size used in the POSIX function calls).

In this first evaluation, we compared the performance achieved by GPFS with our proposed solution in a weak-scalability scenario identical to the one evaluated in Figure 4. The objective of this case is to evaluate the scalability of both GPFS and Hercules under I/O contention.

We scaled the number of nodes from 8 to 128, using 8 workers per node, fully utilizing the eight cores available on each Fusion compute node, up to a maximum of 1,024 workers accessing in parallel the storage system. We used one write task and one read task per worker. Each write task creates a 256 MB file in GPFS or Hercules with a chunk size of 8 MB. The read tasks read the same amount of data.

In this experiment we compared the default GPFS configuration with four configurations based on Hercules summarized in Table I corresponding to the cross-product of two

TABLE I
FOUR HERCULES CONFIGURATIONS EVALUATED FOR THE FILE COPY BENCHMARK.

| Data Placement (DP) | Task Placement (TP) | Notation |
|---|---|---|
| default | locality agnostic | DP-def TP-loc-ag |
| informed | locality agnostic | DP-inf TP-loc-ag |
| default | locality aware | DP-def TP-loc-aw |
| informed | locality aware | DP-inf TP-loc-aw |

configurations for data placement and task placement. The data placement is either the default (algorithmic hashing) or informed (balanced data distribution), while the task placement is either locality aware or locality agnostic. In all cases we used the HARD NODE mode for task placement.

Figures 6a and 6b show the average throughput achieved by each worker for write and read operations, respectively. In this case the timing of each task contains only the effective execution, that is, the time waiting in the scheduling queue as a result of load imbalances and other Swift overheads is not included. Figure 6c shows in logarithmic scale the overall I/O throughput of the whole workflow calculated as the total amount of data accessed in write and read operations divided by the total execution time. This time includes all the Swift overheads and the idleness due to load imbalances. All times are based on 5 repetitions of each experiment.

First, we note that Hercules performs significantly better than GPFS for write tasks, read tasks, and the whole workflow. The Hercules performance is up to 73x higher than GPFS for write operations exploiting locality (20x for locality-agnostic cases), up to 130x for read operations (24x for locality-agnostic cases), and up to 86x for the whole workflow. The performance per worker in our solutions remains stable when the problem scales, while for GPFS it degrades because of contention. Two main factors explain this difference. First, Hercules-based solutions avoid the contention by distributing both metadata and data accesses over all the servers. Second, the locality awareness significantly reduces the network traffic and leaves more bandwidth available for cases when the data are stored remotely. For Hercules, the performance improvement provided by locality awareness over locality agnosticism is more than 2x in all cases but one.

In Figures 6a and 6b we note that the average throughput of individual write and read tasks is better for DP-def TP-loc-aw than for DP-inf TP-loc-aw. However, the overall workflow throughtput is worse for DP-def TP-loc-aw than for DP-inf TP-loc-aw. We explain these results by the fact that individual task times do not include the additional overhead caused by load imbalance. In contrast, the overall workflow time includes these imbalance effects. For DP-inf TP-loc-aw the informed data placement helps the Swift scheduler to better take advantage of the cluster resources and improve the load balance. A more detailed explanation of this phenomenon is described in Section V-C.

### B. MapReduce-like workflow

The second evaluation case is based on a MapReduce-like workflow that performs both computation and I/O. We have implemented in C and Swift an application that receives *n* text files as input and counts how many words of different sizes are in all files. The output is a text file including the number of words with size 1 on the first line, size 2 on second line, and so on. Each job consists of three tasks. A *map task* reads an input file from GPFS, tokenizes the file, and produces a temporary file over GPFS/Hercules. A *count task* reads a file produced by the previous task, counts the number of words with the same length contained, and produces a temporary file over GPFS/Hercules. A *merge task* takes two files produced previously and merges their contents. The final resulting file is stored in GPFS.

The application is fully configurable in 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 are executed. In the Hercules case, the initial input files are read from GPFS, and the final output file is written to GPFS, while any other I/O operations are 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.

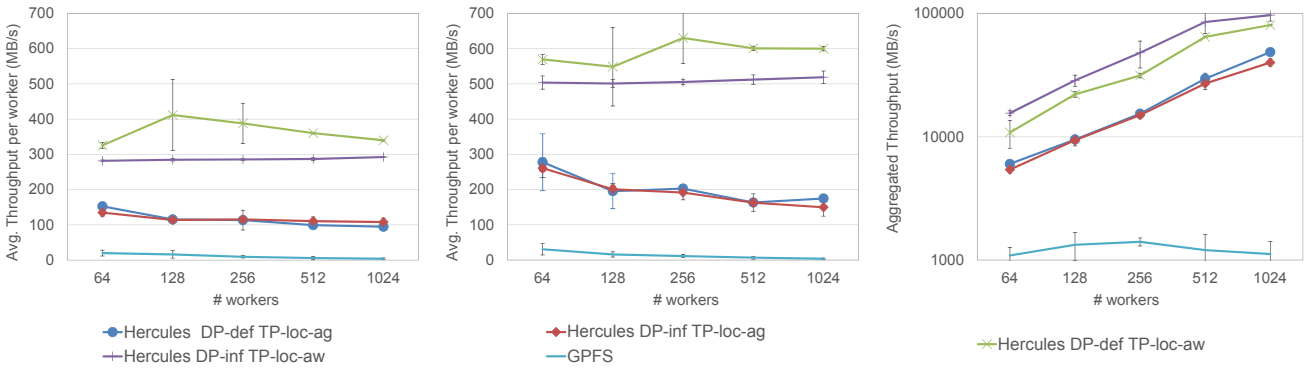
For this workflow we compared the default GPFS-based solution with different configurations of our solution. The total work consists of processing 256 text files containing 256 MB of text data. The selected chunk size is 32 MB, and 8 workers run per node (one per core).

Figure 7a shows the breakdown of the average time per worker on each phase of the workflow for 64 workers (8 compute nodes). The timing of each phase of the workflow does not include the Swift and the load imbalance overheads. The total execution time, plotted as a line, includes these overheads.

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 Figure 8. Even the initial read phase, where files are read from GPFS, is positively affected by 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.

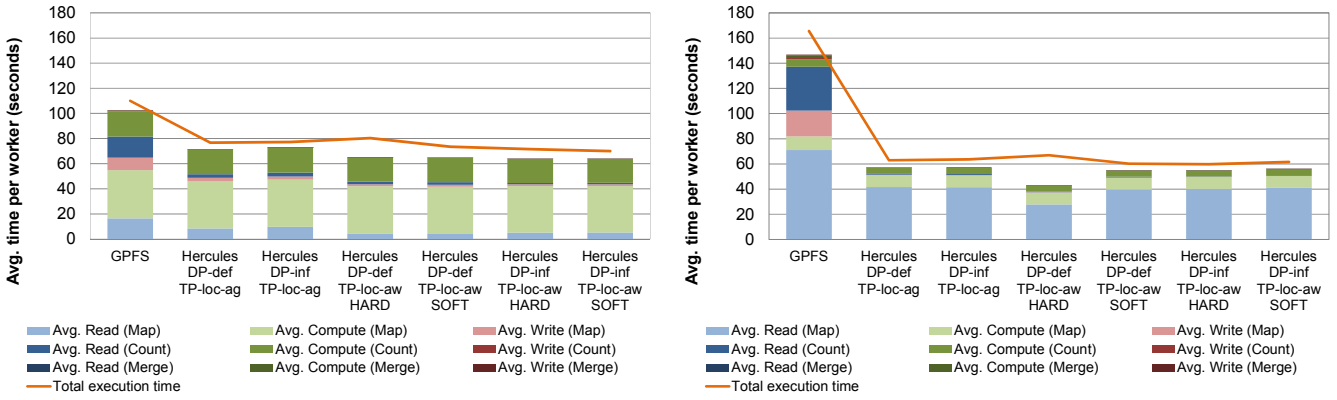
Figure 7a shows that our solution can speed up total execution times by up to 1.57. This amount of time is especially substantial given that I/O operations represent less than 45% of the execution time per worker. If we focus on I/O operations, the times are reduced from 43 seconds to under 8 seconds, more than a 5x improvement. Furthermore, if we focus only on the operations directly accomplished over Hercules, the improvement is over 10x, in line with the results seen in Section V-A.

Figure 7b 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 nodes). The results are similar to the previous case, but the solutions based on Hercules and




(a) Average write throughput per worker in MB/s. (b) Average read throughput per worker in MB/s. (c) Overall I/O throughput in MB/s.

Fig. 6. Filecopy microbenchmark weak-scalability throughput performance results. The problem size scales with the number of workers, executing one write task and one read task per worker. File size is 256 MB with a chunk size of 8 MB.




(a) Average execution time in seconds of each phase of the workflow using 64 workers (8 nodes). (b) Average execution time in seconds of each phase of the workflow using 256 workers (32 nodes).

Fig. 7. Breakdown of the average execution times of each phase in a workflow composed of 256 jobs (767 tasks) for 256 MB text files. Task scheduling policy is denoted HARD for "NODE, HARD" and SOFT for "NODE, SOFT".

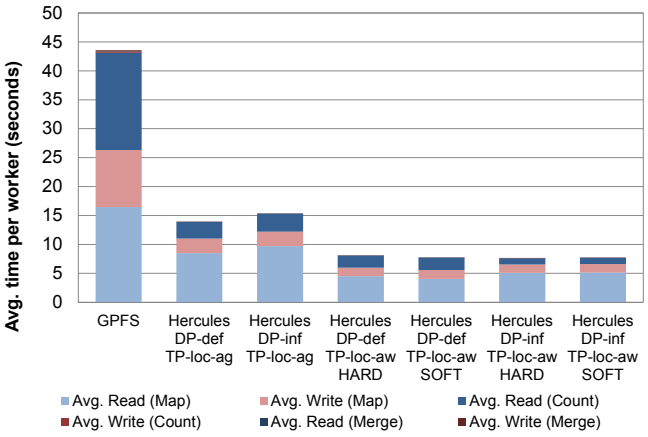



Fig. 8. Breakdown of the MapReduce-like workflow task times including only the I/O phases for 64 workers (8 nodes).

Swift are capable of taking 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 of 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, while the speedup of I/O operations is up to 4.48. If we focus only on operations directly related with Hercules (map write operation, count read operation, count write operation, and merge read operation) the speedup in time is greater than 100x. Figure 9 shows a breakdown of these operations for the target configurations.

For Hercules-related cases the worst performance is obtained for the default configuration. When data locality is exploited, the time needed for I/O operations is reduced, especially in the cases where the data locality is strictly enforced (HARD). In contrast, when best-effort data locality is used (SOFT), the 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,

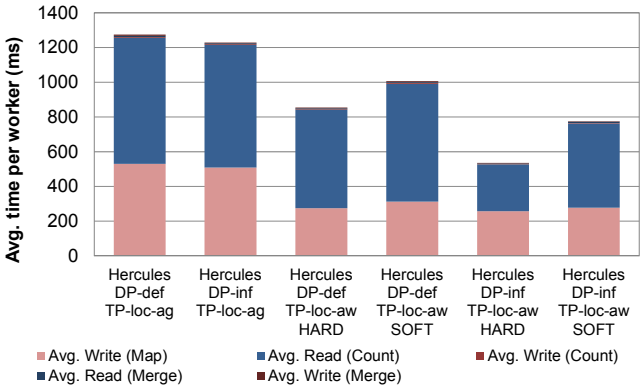



Fig. 9. Breakdown of the MapReduce-like workflow task times for 256 workers (32 nodes). This figure only shows the I/O phases directly performed over Hercules: map-write, count-read, count-write, and merge-read.

because this configuration trades off locality for load balance. A more detailed analysis of this trade-off is discussed in the next section.

### C. Data-locality and load-balance analysis

In the previous two sections we have seen that an improvement in the I/O performance does not necessary translate into an improvement in the total workflow time. This is mainly due to the classical trade-off between data locality and load balance; that is, improving data locality may hurt load balance and vice versa.

For illustrating this analysis, we plotted the behavior of each worker for four Hercules configurations from Figure 7b, as specified below. Figure 10 shows the time spent by each worker in computing (in green) and idle states (in red).

Figure 10a displays the behavior for the default configuration. This 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.

Figure 10b shows the behavior for strictly enforcing the data locality and using the default data placement. As Figure 7 shows, the time needed by each worker to complete its assigned tasks is substantially reduced, and the best performance is obtained. Nevertheless, the total execution time (yellow line) is the worst among all configurations. As Figure 10b shows, forcing data locality without a proper data placement implies that some workers remain underutilized (24.48% of idle time), while some tasks still are left because the data are in another node, causing an increase of the total execution time.

The behavior of best-effort data-locality task placement and default data-placement strategy is depicted in Figure 10c. 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 Figure 10d achieves the best total execution time, 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.

## VI. Related work

This section discusses related work in three areas: distributed key-value stores, the storage I/O bottleneck in scientific workflows, and in situ and in-transit computation.

### A. Distributed key-value stores

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. Memcached [10] is a simple RAM-based key-value store that provides high performance with no durability and minimal consistency guarantees. It provides a single, completely flat hash table with opaque keys and values. Redis [13] provides similar functionality to Memcached, as well as a range of data structures including hash tables and lists and the ability to subscribe to data items. Other more sophisticated key-value stores that are highly scalable, use disk storage, and provide consistency and durability guarantees include Dynamo [14] and Cassandra [15].

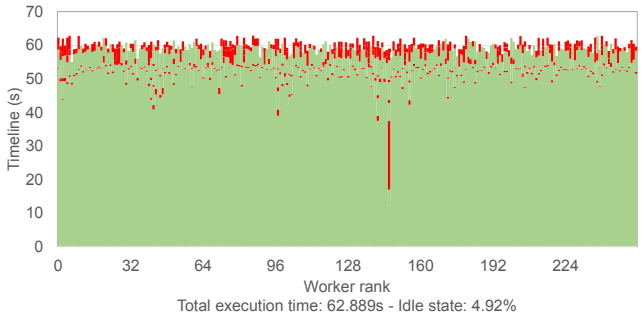
While these key-value systems provide a range of options for data storage, they do not take advantage of high-performance message passing available on many clusters. Moreover, they do not provide all the coordination primitives that were required in Turbine to implement a dataflow language such as Swift.

### B. Storage I/O bottleneck in scientific workflows

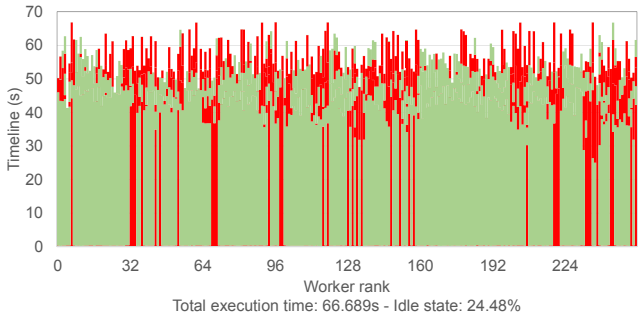
The increasing popularity in many-task computing and workflow engines has exposed I/O bottlenecks in such scenarios. This new problem has led several researchers to investigate new solutions.

One approach involves using extended file attributes in MosaStore [17] to provide communication between the workflow engine and the file system through the use of hints about the data. The workflow engine can provide these hints directly to the file system or the file system can infer the patterns analyzing the data accesses. The MosaStore approach is radically different from Hercules using a centralized metadata server instead of the fully distributed, easy and flexible deployment approach of our proposed solution.

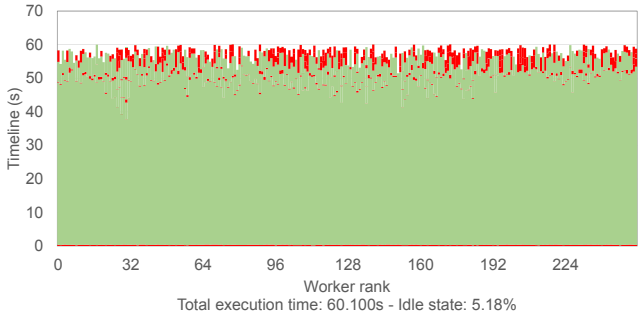
The AMFS shell [18] offers programmers a simple scripting language for running parallel scripting applications in memory on large-scale computers. The objective of this solution is similar to the combination of Swift/T and Hercules, but Swift/T is able to automatically solve more dynamic data dependencies in the full-featured Swift language. AMFS and Hercules share the distributed metadata approach. Another difference consists in the ability of AMFS shell programs to explicitly specify in-memory or persistent storage, whereas Hercules can be



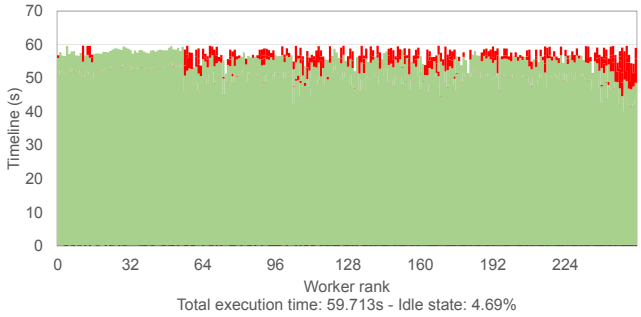

(a) Timeline in seconds for each worker, using default data-placement and locality-agnostic Swift/T scheduler.




(b) Timeline in seconds of for each worker, using default data-placement and enforcing locality-aware scheduling (HARD, NODE).




(c) Timeline in seconds for each worker, using default data-placement and best-effort locality-aware scheduling (SOFT, NODE).




(d) Timeline in seconds for each worker, using informed data placement and enforcing locality-aware scheduling (HARD, NODE).

Fig. 10. Detailed figure of the timeline execution status of each worker on a MapReduce program with 256 jobs (767 tasks) running on 32 nodes (256 workers). Light green represents busy status; dark red represents idle/wait status.

deployed with persistence enabled in a transparent way for the programmer.

HyCache+ [19] is a distributed storage middleware that allows I/O to effectively leverage the high bisection bandwidth of the high-speed interconnect of massively parallel high-end computing systems. HyCache+ acts as the primary place for holding hot data for the applications (e.g., metadata, intermediate results for large-scale data analysis) and only asynchronously swaps cold data on the remote parallel file system. Some similarities between HyCache+ and Hercules are their fully distributed metadata approach, use of compute network instead of the shared storage network, and the high scalability capabilities. HyCache+ relies on POSIX, whereas Hercules offers the possibility of using a POSIX-like interface and get/set operations. HyCache+ focuses on enhancing parallel file systems in a generic way, whereas Hercules has been designed to work specifically with a many-task engine, exposing and exploiting data locality in current applications. HyCache+ and Hercules share similar ideas, but Hercules is ready to be deployed to improve many-task I/O performance focusing on easy and flexible deployment options.

Parrot [20] is a tool for attaching existing programs to remote I/O systems through the POSIX file system interface; and Chirp [21] is a user-level file system for collaboration across distributed systems such as clusters, clouds, and grids. They are usually combined to easily deploy a distributed file system ready to use with current applications through a POSIX API. Many characteristics are shared with the 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 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.

The main design goal of MemEFS [22], an in-memory locality-agnostic file system, is dynamic scalability. The design goal of our solution is locality awareness and exploitation focused on many-task workflows running as parallel applications on large-scale platforms.

Prior work in workflow scheduling for grid-like systems [23] solves the problem of mapping an abstract DAG description onto physical resources and replicas by using a heuristic over some time window. In contrast, Hercules focuses on a single cluster and exploits rich user hints to improve I/O performance by a best-effort strategy.

### C. In-situ and in-transit computation

In-situ and in-transit locality exploitation has become a critical capability for pushing scalability beyond the current level. Aspects currently addressed include data staging [24], in situ and in-transit data analysis and visualization [25],

[26], [27], publish-subscribe paradigms for coupling large-scale analytics [28], and flexible analytics placement tools [29]. We share with this systems the goal of reducing data movement, improving the locality exploitation, and overcoming the current file system bottleneck caused by large data set accesses on large-scale infrastructures. However, most of the works cited are applied to parallel scientific applications, whereas we focus in our work on many-task workflows.

## VII. Conclusions

In this paper we have explored novel methods for improving the performance of many-task workflows based on the Swift scripting language. This work extends the optimization space of Swift runtime with novel dimensions for data placement and task scheduling. A novel data placement mechanism can be used for distributing intermediate workflow data by using custom criteria over the servers of the Hercules key-value store. A novel scheduling technique is used for exploiting the locality of intermediate data in either compulsory or advisory mode. We demonstrate that these new mechanisms can be used for significantly improving the performance of many-task workflows with up to 73x for writes, up to 130x for reads, and up to 86x for the whole workflow. These results are due mainly to reducing the contention on the shared file system, exploiting the data locality, and trading off locality and load balance. Thus, while the experiments were performed on an HPC-oriented scientific cluster, our solution allows the developer to mix in a range of BDC techniques to boost performance, without rewriting the workflow logic.

In this paper we have evaluated a user-driven data placement strategy. In the future we plan to explore novel techniques for adaptive data and task placement by taking into consideration dynamic load conditions and capacities.

## References


[1] M. Woitaszek, J. M. Dennis, and T. R. Sines, "Parallel high-resolution climate data analysis using Swift," in *Proc. MTAGS at SC*, 2011.

[2] J. M. Wozniak, K. Chard, B. Blaiszik, R. Osborn, M. Wilde, and I. Foster, "Big data remote access interfaces for light source science," in *Proc. Big Data Computing*, 2015.

[3] A. O'Driscoll, J. Daugelaite, and R. D. Sleator, "Big data, Hadoop and cloud computing in genomics," *Journal of Biomedical Informatics*, vol. 46, no. 5, pp. 774–781, 2013.

[4] D. L. Jones, K. Wagstaff, D. R. Thompson, L. D. Addario, R. Navarro, C. Mattmann, W. Majid, J. Lazio, R. Preston, and U. Rebbapragada, "Big data challenges for large radio arrays," in *Proc. IEEE Aerospace Conference*, 2012.

[5] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Comput.*, vol. 37, no. 9, pp. 633–652, 2011.

[6] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for high performance many-task applications," *Fundamenta Informaticae*, vol. 28, no. 3, 2013.

[7] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *Proc. SC*, 2014.

[8] J. M. Wozniak, T. G. Armstrong, K. C. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster, "Interlanguage parallel scripting for distributed-memory scientific computing," in *Proc. WORKS at SC*, 2015.

[9] F. R. Duro, J. G. Blas, and J. Carretero, "A hierarchical parallel storage system based on distributed memory for large scale systems," ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 139–140.

[10] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004.

[11] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," ser. FAST '02. Berkeley, CA: USENIX Association, 2002.

[12] "Lustre file system. http://www.lustre.org." 2013.

[13] T. Macedo and F. Oliveira, *Redis Cookbook*. Sebastopol: O'Reilly Media, 2011.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[15] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[16] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu, "The case for a versatile storage system." *Operating Systems Review*, vol. 44, no. 1, pp. 10–14, 2010.

[17] L. Costa, H. Yang, E. Vairavanathan, A. Barros, K. Maheshwari, G. Fedak, D. Katz, M. Wilde, M. Ripeanu, and S. Al-Kiswany, "The case for workflow-aware storage: An opportunity study," *Journal of Grid Computing*, pp. 1–19, 2014.

[18] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, and I. Foster, "Parallelizing the execution of sequential scripts," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 31:1–31:12.

[19] D. Zhao, K. Qiao, and I. Raicu, "HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems," in *IEEE/ACM CCGrid*, 2014.

[20] D. Thain and M. Livny, "Parrot: Transparent user-level middleware for data-intensive computing." *Scalable Computing: Practice and Experience*, vol. 6, no. 3, 2005.

[21] D. Thain, C. Moretti, and J. Hemmes, "Chirp: a practical global filesystem for cluster and grid computing," *Journal of Grid Computing*, vol. 7, no. 1, pp. 51–72, 2009.

[22] A. Uta, A. Sandu, S. Costache, and T. Kielmann, "MemEFS: an elastic in-memory runtime file system for escience applications," in *11th IEEE International Conference on eScience*, 2015.

[23] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gila, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, pp. 219–237, 2005.

[24] T. Jin, F. Zhang, Q. Sun, H. Bui, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C. Chang, and M. Parashar, "Leveraging deep memory hierarchies for data staging in coupled data intensive simulation workflows," in *IEEE Cluster 2014*, 2014.

[25] M. Dreher and B. Raffin, "A flexible framework for asynchronous In situ and in transit analytics for scientific simulations," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, USA, May*, pp. 277–286.

[26] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and I/O acceleration on leadership-class systems." in *LDAV*, D. Rogers and C. T. Silva, Eds. IEEE, 2011, pp. 9–14.

[27] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *CLUSTER*. Beijing, China: IEEE, Sep. 2012.

[28] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics." in *CCGRID*, 2014, pp. 246–255.

[29] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "FlexIO: I/O middleware for location-flexible scientific data analytics," in *IPDPS 2013, Cambridge, USA, May*, pp. 320–331.