Lawrence Berkeley National Laboratory
LBL Publications

Title
Programming Abstractions for Managing Workflows on Tiered Storage Systems

Permalink
https://escholarship.org/uc/item/4g49x4z1

Journal
ACM Transactions on Storage, 17(4)

ISSN
1553-3077

Authors
Ghoshal, Devarshi
Ramakrishnan, Lavanya

Publication Date
2021-11-30

DOI
10.1145/3457119

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-ShareAlike License, availalbe at
https://creativecommons.org/licenses/by-sa/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Programming Abstractions for Managing Workflows on Tiered Storage Systems

DEVARSHI GHOSHAL and LAVANYA RAMAKRISHNAN, Lawrence Berkeley National Laboratory, USA

Scientific workflows in **High Performance Computing** (**HPC**) environments are processing large amounts of data. The storage hierarchy on HPC systems is getting deeper, driven by new technologies (NVRAMs, SSDs, etc.) There is a need for new programming abstractions that allow users to seamlessly manage data at the workflow level on multi-tiered storage systems, and provide optimal workflow performance and use of storage resources. In previous work, we introduced a software architecture **Managing Data on Tiered Storage for Scientific Workflows (MaDaTS)** that used a **Virtual Data Space** (**VDS**) abstraction to hide the complexities of the underlying storage system while allowing users to control data management strategies. In this article, we detail the data-centric programming abstractions that allow users to manage a workflow around its data on the storage layer. The programming abstractions simplify data management for scientific workflows on multi-tiered storage systems, without affecting workflow performance or storage capacity. We measure the overheads and effectiveness introduced by the programming abstractions of MaDaTS. Our results show that these abstractions can optimally use the storage capacity in lesser capacity storage tiers, and simplify data management without adding any performance overheads.

CCS Concepts: • **Software and its engineering** → **Abstraction, modeling and modularity**; *Dataflow architectures*; *Middleware*; • **General and reference** → Design;

Additional Key Words and Phrases: Data management, scientific workflows, multi-tiered storage, burst buffer

ACM Reference format:
Devarshi Ghoshal and Lavanya Ramakrishnan. 2021. Programming Abstractions for Managing Workflows on Tiered Storage Systems. *ACM Trans. Storage* 17, 4, Article 29 (October 2021), 21 pages.
https://doi.org/10.1145/3457119

1 INTRODUCTION

Scientific datasets are growing exponentially and are complemented by advancements in new storage technologies (NVRAMs, SSDs, etc.) and the use of multiple filesystems (burst buffers, parallel file systems, etc.) have resulted in increased storage tiers on **High Performance Computing** (**HPC**) systems. Large amounts of data from scientific workflows need to be managed on tiered

This work and the resources at NERSC are supported by the U.S. Department of Energy, Office of Science and Office of Advanced Scientific Computing Research (ASCR) under Contract No. DE-AC02-05CH11231.
Authors' address: D. Ghoshal and L. Ramakrishnan, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, California 94720; emails: {dghoshal, lramakrishnan}@lbl.gov.

This work is licensed under a Creative Commons Attribution-ShareAlike International 4.0 License.

© 2021 Copyright held by the owner/author(s).
1553-3077/2021/10-ART29 $15.00
https://doi.org/10.1145/3457119

storage systems. Such data management can be extremely tedious and complex due to the underlying architecture of the storage systems, amount of data processed, and complexity of scientific workflows. Existing methods and tools have used queueing models and historical information about the data to manage and improve I/O performance on tiered storage systems [17, 21]. However, current abstractions and tools provide limited control for scientific applications to manage data on HPC systems and are often insufficient to allow fine-grained control based on user preferences.

Scientific workflows on HPC systems execute simulations and data analyses for scientific discovery. Today, scientific workflows largely rely on applications or custom user scripts to manage their data on tiered storage. For example, users often "stage" the input data to a storage layer prior to executing the workflow. This gives rise to performance and productivity challenges when moving to future HPC systems, where the memory storage hierarchy is getting deeper, driven by new technologies and the need to minimize I/O costs. Automation allows us to asynchronously stage the data or perform selective staging of data to improve performance or balance costs. However, our experiences working with various science groups show a need for a semi-automated architecture. Our discussion with different science groups revealed that complete automation hinders HPC users who often need to exert control on the data management. Thus, our software architecture is driven by the needs of different scientific workflows and user groups.

In previous work, we introduced a software architecture **Managing Data on Tiered Storage for Scientific Workflows** called **MaDaTS** [14] that used a **Virtual Data Space (VDS)** abstraction to hide the complexities of the underlying storage systems. VDS maps the datasets of a workflow onto *virtual data objects* that can co-exist across the different tiers of a storage system during the execution of a workflow. VDS provides an opportunity that allows users to program a workflow around its data. In this article, we present the programming abstractions that allow users to use MaDaTS for managing workflows on tiered storage systems. Using the programming abstractions, programmers map workflows and data on a virtual data space, and let VDS optimize the data management using workflow and data properties. However, programmers can also implement their own data management strategies by operating on VDS. The VDS abstraction provides the necessary programming constructs for defining new data management strategies with the evolving memory-storage hierarchy. VDS is designed to operate with different architectures including computional storage architectures, where the compute would move closer to storage. VDS's data-centric programming model can be leveraged resulting in an execution plan that accounts for the compute moving closer to the storage.

The programming abstractions of MaDaTS provide control to the programmers to manage data and workflows. For example, users can control which datasets in a workflow are temporary without necessarily worrying about when and how to remove the datasets. Alternatively, users can also control which temporary datasets need to persist without worrying about the mechanics of copying/moving the datasets to a persistent store. Finally, through VDS users can also control the use of storage tiers, or can use the entire storage hierarchy for the workflow data. This is useful for controlling the flow of data through the storage stack, and managing the lifecycle of data on different storage tiers as per user needs. Our results show that these programming abstractions simplify data management for complex workflows on multi-tiered storage systems without adding any performance overheads. The abstractions also optimally use the storage space based on the characteristics of data and storage systems.

The rest of the article is organized as follows. We present the background in Section 2. We describe the programming abstractions of MaDaTS in Section 3. We present our results in Section 4 and discuss related work in Section 5. Finally, we present the conclusions in Section 6.

2 BACKGROUND

In this section, we provide an overview of the data lifecycle on tiered storage systems, existing storage abstractions and a background on MaDaTS [14].

2.1 Scientific Workflows

Scientific workflows are widely used in HPC environments to manage the software pipelines for scientific discovery [27]. Workflows can be composed using a workflow description language, through ad-hoc scripts, or using workflow tools that chain together a series of tasks with inputs and outputs in a pipeline. The traditional model of capturing workflows using a **directed acyclic graph** (**DAG**) is process-centric. In this model, data management becomes secondary to task execution, and data management steps often come before and after the execution of the workflow, and are managed explicitly by users or applications. Existing workflow tools [9, 30] do not provide mechanisms to implicitly manage data on HPC resources. Current user methods are less than optimal in performance and effectiveness. For example, users often "stage" the input data to a storage layer prior to executing the workflow. However, some of this data can be asynchronously staged as the workflow executes. Also, selective staging of data might be sufficient for improving performance while balancing costs. But scientists often face challenges in automating these due to the interplay of workflow tasks and data, and the underlying complexities of the storage systems. Thus, there is a need for simplified programming abstractions that can allow users to seamlessly control data management strategies for scientific workflows running on HPC systems.

2.2 Tiered Storage Systems

SSDs have added one more tier to the storage hierarchy in HPC systems in the form of Burst Buffers [20, 29]. It resides between compute nodes and the high-capacity **parallel file system** (**PFS**). The existence of different storage systems has given rise to different storage tiers and hence, different filesystem interfaces. These filesystems often differ in the way data is stored in the underlying storage system and are accessed by the users. For example, the storage systems on different supercomputers like Cori [4] and Titan [1] provide separate namespaces for the different storage layers. Alternatively, unified and single namespace filesystems have also been implemented on tiered storage systems [2]. Programming abstractions are required to manage workflow data that hide different storage-specific data management interfaces from the users. The VDS abstractions in MaDaTS hide the different data management interfaces in multi-tiered storage systems. In the case of storage systems that use a unified interface, MaDaTS acts as a workflow engine as the data movement is already handled by the storage system and the storage tiers are unknown to MaDaTS (i.e., using a single namespace). In the case where the storage tiers are unified but with different namespaces, MaDaTS can make intelligent decisions about the placement and distribution of data based on workflow structure, data properties, and storage characteristics.

Users keep the input and output datasets on persistent long-term storage. These storage systems provide high resilience, but lack high performance. For this reason, users often copy/move these datasets onto faster storage tiers, typically a PFS, or a burst buffer that provides high I/O throughput. This process of temporarily moving the data to a faster storage is called staging, and several optimizations have been proposed to improve the effectiveness of data staging on HPC systems [22–24, 33]. However, for scientific workflows, data remains alive even after one or more tasks using it have finished execution. This introduces different data management challenges when managing workflow data on tiered storage systems. For example, datasets that are shared between different workflow tasks need to be staged for longer durations to avoid overheads of frequent stage-in and stage-out operations. Scientific collaborations may also choose to stage or store their

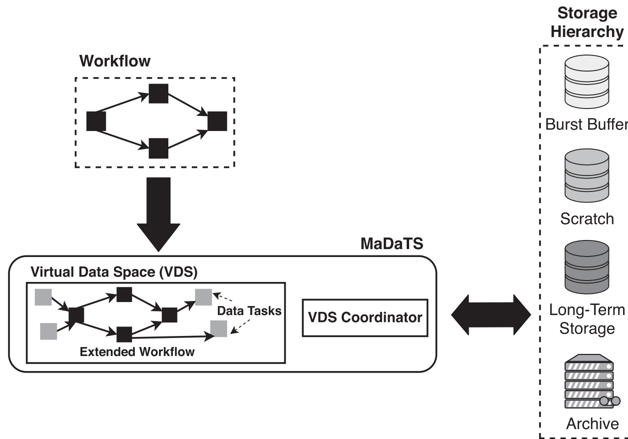


Fig. 1. In MaDaTS, users map workflows and data to a virtual data space (VDS) and a VDS coordinator manages the data during workflow execution on tiered storage systems.

data on specific tiers based on usage frequency. Hence, there is a need for programming data management of workflows.

2.3 MaDaTS

Figure 1 shows the high-level architecture of MaDaTS, and the role of VDS in the system. MaDaTS provides an integrated data management and workflow execution framework on HPC resources. A workflow is often represented as a DAG and VDS provides a data-centric view of the workflow akin to a *data DAG*. The traditional model of capturing workflows using a DAG is **task-centric**. In this model, data management becomes secondary to task execution. Users explicitly stage the data to and from different storage layers (e.g., scratch, archive), and execute the workflow tasks separately. The data movement between the storage layers is often constrained to the beginning and to the end of workflow execution. On the contrary, the **data-centric model** of workflows [6] allows us to make data management decisions by inspecting the data properties and data use during the entire lifecycle of a workflow. Data objects are treated as first-class entities in workflows, resulting in a data-driven workflow execution. By using the data-centric model, MaDaTS manages both data and tasks explicitly depending upon the data properties, providing transparent and efficient data management for workflow data. This removes a significant burden from the programmers because they only need to define the data properties, and MaDaTS takes care of moving the data, executing the tasks, and selecting the appropriate storage. In order to manage workflow tasks on HPC systems, MaDaTS uses Tigres [15], which is a workflow library that provides templates to compose, run, and manage workflow tasks on desktops and HPC clusters. MaDaTS uses existing abstractions in the Tigres and the batch scheduler of an HPC system to manage the compute resources. By default, MaDaTS executes the tasks locally on the available resources. Programmers can also specify resource requirements for the tasks on HPC systems, in which case resources are requested and allocated by the underlying batch scheduler. MaDaTS can also be used with other workflow managers.

MaDaTS implements a VDS coordinator that manages virtual data objects through VDS. It is responsible for managing the data on the memory-storage hierarchy, and preparing a data execution plan. Based on the data management strategy, VDS coordinator creates data movement tasks and

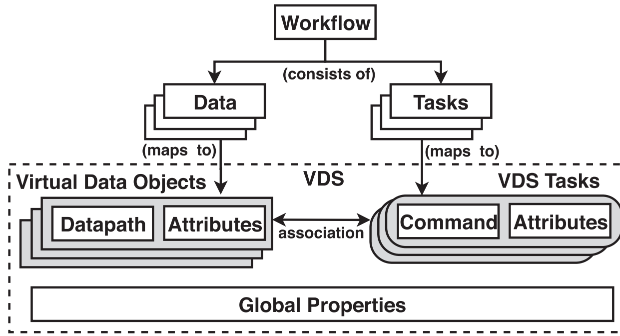


Fig. 2. A VDS in MaDaTS is a mapping of a workflow into the data space, where workflow data are mapped to abstract data items, called virtual data objects. Users add virtual data objects to the VDS and create associations to workflow tasks, resulting in a data dependency graph. Users may also define certain attributes for each virtual data object. A set of global properties define the characteristics of a VDS, which control the data management strategies for a workflow.

associates them with the corresponding virtual data objects. Essentially, VDS coordinator manages an extended workflow to manage data and tasks in an integrated manner. MaDaTS implements different data management strategies using VDS. These data management strategies use data properties, structure of the workflow, and storage system characteristics for managing the lifecycle of data on different storage tiers. Our previous work [14] describes and evaluates the data management strategies. However, programmers can use the programming abstractions to implement their own strategies, and seamlessly control the dataflow through different storage layers.

3 PROGRAMMING ABSTRACTIONS

VDS provides an abstraction that allows users to map their data into a common namespace. Users can use a programming interface to map their workflows and data objects to VDS. Figure 2 conceptually describes the process of mapping a workflow on to VDS. A user can use the programming abstractions of to compose a workflow in a data space and specify data properties that are hints to the VDS for data management. The primary entity in VDS is a virtual data object that represents each data element of a workflow and maps them to the underlying file system. A data element can be a file, a collection of files, or a directory containing multiple files and subdirectories. MaDaTS uses these hints and data management strategies to create data tasks that orchestrate the movement of data across the memory-storage hierarchy. The data tasks in the workflow interact with the storage system to move data between the storage layers.

3.1 VDS

VDS supports the data-centric model of workflows that supports the needs of big data and multi-tiered hierarchical storage [6]. Data management is going to be the principal challenge with increasing volume and rate of data. Performance and efficiency of workflows now depend on efficiently utilizing the storage resources to manage data during execution. The data-centric model of workflows allows us to make data management decisions by examining each data object over the lifecycle, rather than just looking at individual tasks. In the data-centric model of a workflow, data objects are treated as first-class parameters to the workflow tasks. This results in a data-driven workflow execution, where a task can be executed as soon as the data it uses becomes available.

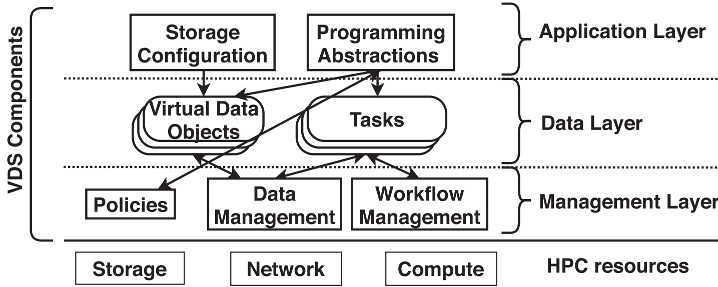


Fig. 3. VDS comprises (a) an application layer that uses a storage configuration to manage multiple tiers of the storage hierachy, and the abstractions to program VDS, (b) a data layer that consists of virtual data objects and VDS tasks, and (c) a management layer that manages data and workflows on HPC resources using the virtual data objects and VDS tasks.

Figure 3 shows the design and implementation of VDS. It comprises (a) an application layer comprised of the storage configuration and programming abstractions, (b) a data layer that consists of the virtual data objects and tasks, and (c) a management layer that manages workflow execution and data on multi-tiered storage systems.

3.2 Application Layer

The application layer provides the user an interface to program VDS for managing workflow data on tiered storage systems. Our current implementation provides a Python interface. The application layer includes a storage configuration and an API to access the underlying programming abstractions. The storage configuration tells VDS about the storage tiers to be used. It is a YAML file that lists the different storage tiers and their properties. Each storage tier is identified by its unique filesystem mount point. The properties of each storage tier include the filesystem (Lustre/HPSS/GPFS), performance characteristics like throughput, IOPS, and latency, and persistency characteristics like volatile vs. non-volatile storage. When new storage tiers are added or removed, this configuration file is updated. The configuration file can be either defined by the system administrator or by the application programmer. MaDaTS is currently implemented to work with POSIX-compliant filesystems, HPSS and burst buffers, and abstracts the storage hierarchy based on the storage configuration. This has two advantages. First, it allows programmers to configure storage tiers that are specific to a workflow. Second, it unifies storage tiers from multiple systems into a single data space.

Users use the programming abstractions to create a VDS and add virtual data objects to the VDS. Users then create and associate workflow tasks to the virtual data objects. The abstractions are also used to specify data properties and data management strategies, based on which new virtual data objects and tasks may be created in VDS.

3.3 Data Layer

The data layer consists of the virtual data objects, and their mapping to the underlying storage tiers and respective datasets of a workflow. A virtual data object is an abstract entity in VDS that is uniquely identified by an object identifier. Each object identifier is a combination of a storage identifier and path to the data object relative to the storage mount point. Figure 4 shows the mapping of a data object in the storage system to a virtual data object in VDS. The example maps the data object `/global/scratch/sample/foo` to a virtual data object `scratch:sample/foo`. Since

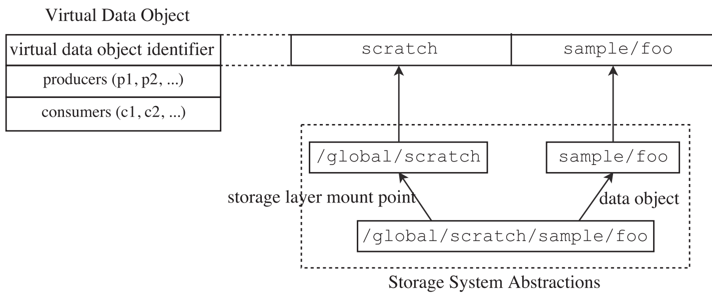


Fig. 4. Example virtual data object in VDS that corresponds to a data object in the storage system.

`/global/scratch/sample/foo` represents the absolute path of the data object `foo` on the `scratch` storage (mounted on `/global/scratch`), the corresponding object identifier for the virtual data object in VDS is `scratch:sample/foo`. The storage identifier uniquely identifies the storage layer in the multi-tiered storage hierarchy. Storage system abstractions (e.g., the underlying filesystem interfaces) resolve the virtual data object identifier to the corresponding physical location of the data object on the filesystem.

In addition to the identifier, a virtual data object also consists of the workflow tasks that operate on the corresponding filesystem objects. Each task is either a producer that writes to a filesystem object, or a consumer that reads from the corresponding filesystem object. Hence, a virtual data object is an encapsulation of a filesystem object, along with its producers and consumers. VDS uses this data-centric view of the workflow that is mapped to a collection of virtual data objects to manage workflow tasks and data on tiered storage systems.

A virtual data object also has attributes that define the I/O and data management characteristics of workflow data. Examples of these attributes include *size*, *persistence*, and *type*. Users can set the attribute values to control the different aspects of data management. For example, if users set the "type" of virtual data objects to be "static," then the underlying datasets are not copied/moved between the storage tiers during workflow execution. Users can also use these properties to indicate which virtual data objects need to persist beyond the lifetime of a workflow. This allows VDS to copy any intermediate data to a persistent store so that users can share it between multiple workflows or use it for future analysis.

3.4 Management Layer

Internally, VDS manages the lifecycle of a virtual data object through several operations. These operations on virtual data objects facilitate managing data for scientific workflows. There are four operations on virtual data objects: (a) `add`, (b) `copy`, (c) `replace`, and (d) `delete`. The `add` operation creates a virtual data object on a VDS. The `copy` operation creates another version of a virtual data object on a different storage layer, but with the same attributes and associations as the original virtual data object. The `replace` operation overwrites one virtual data object with another, also replacing any task parameter that uses the old virtual data object. The `delete` operation removes a virtual data object from VDS. The different operations on virtual data objects are listed in Table 1.

The `copy` operation on a virtual data object updates the preferred location of the filesystem object on a storage tier. In order to move the filesystem object to the preferred storage tier, VDS creates "data tasks," and associates them with the virtual data objects. The data tasks can be of three types: (i) *setup*, that creates necessary directories for staging the datasets, (ii) *move*, that moves datasets between two storage tiers, and (iii) *cleanup*, that removes data by deleting virtual

Table 1. Operations on Virtual Data Objects in VDS

Operation	Description
$add(V)$	adds a virtual data object V on the VDS
$copy(V, s)$	copies virtual data object V to a storage tier s
$replace(V, V')$	replaces a virtual data object V with V'
$delete(V)$	removes virtual data object V from the VDS

ALGORITHM 1: Defining a data management strategy using VDS

Input: A virtual data space VDS
1: s = select_fast_storage()
2: **for** v in $VDS.vdos$ **do**
3: $VDS.copy(v, s)$
4: **end for**

data objects that have no future use in the workflow. VDS makes sure that there are no duplicate tasks managing the same data at the same time. A cleanup task is associated with a virtual data object, if it is non-persistent. For temporary but persistent data objects, additional tasks are created by VDS to copy data to a persistent store prior to cleaning up. The cleanup operation is analogous to garbage collection, freeing up space by removing datasets that will no longer be used by the workflow.

Instead of updating each and every access to a data object in a workflow, `copy` updates the access to the corresponding data object for all the dependent workflow tasks in a single operation. This is because it associates the same set of tasks to the destination virtual data object that is associated with the source. The `copy` operation also replaces all the task associations to the source virtual data object with a *move* data task, that transfers the object from the source to the destination storage. Hence, it simplifies workflow data management on tiered storage systems. As an example, the algorithm for managing a workflow using VDS by moving the datasets to a fast storage is listed below.

Algorithm 1 selects all the virtual data objects in a VDS and makes copies of them on a fast storage tier. Each copy operation in VDS may generate one or more data tasks. For an input dataset, VDS creates a *stage-in* data task that moves the dataset to the fast storage tier, and replaces all references to the input dataset with the staged dataset. Similarly for an output dataset, the copy operation creates a *stage-out* data task that moves the dataset from a staged storage tier to a persistent storage, as specified by the user. The data management strategy defined above is then registered to MaDaTS with a unique name. Programmers use the `strategy` attribute of a VDS to select the data management strategy. MaDaTS provides three pre-defined data management strategies—passive, storage-aware, and workflow-aware [14]—but programmers can also define, register, and use custom data management strategies based on their workflow needs.

The copy operation replaces all intermediate dataset references with the staged dataset references. If any virtual data object that maps an intermediate dataset is set to *persist*, VDS creates additional stage-out tasks to move the dataset to a persistent storage. Finally, if the VDS is programmed to *auto-cleanup*, additional data tasks are created to remove the intermediate datasets. In addition to data tasks, virtual data objects are also associated with "compute tasks." A compute task refers to a workflow task that primarily processes or analyzes data. This separation of concerns between compute and data tasks allows MaDaTS to allocate separate resources for data management

and workflow execution. For example, a compute task may be executed through a batch queue system, whereas a data task may be executed on a separate I/O node with high bandwidth and low latency. Each task in VDS is identified by a *command* and a set of *attributes*. A command refers to a task executable. The attributes of a task specify the runtime information for the task. Examples of task attributes include parameters to the executable command, the expected runtime of the task, and the number of nodes to be requested for executing the task. These options are further used by MaDaTS to schedule and execute tasks on the HPC resources.

A VDS exists during the entire lifetime of a workflow execution and its data lifecycle. It is automatically destroyed when a workflow execution ends. But the underlying filesystem objects retain their state. Hence, any persistent filesystem objects that are created on different storage layers persist beyond the workflow execution. If programmers decide to persist intermediate data, multiple copies of a dataset may exist on different storage tiers. By default, MaDaTS does not remove duplicate datasets for two reasons. First, in the case of volatile storage, the duplicate datasets will not persist beyond the workflow execution. Removing the datasets in such a case is a redundant operation and may introduce additional overheads. Second, if multiple workflows share datasets, then they will not be moved multiple times between the storage tiers, minimizing the overall data movement operations. For automated removal of intermediate datasets that are persisted beyond a workflow's lifetime, programmers set the *auto-cleanup* option in VDS to *true*. This ensures that there is only one copy of the dataset on the storage system.

3.5 Example: Programming the Virtual Data Space

In this section, we provide an example to program VDS. The programming abstractions of MaDaTS allow users to create a VDS, add virtual data objects and tasks to the VDS, and manage VDS. Table 2 lists the abstractions and associated operations in MaDaTS. Users first create a VDS and map the data to virtual data objects.

```
vds = VirtualDataSpace()
vdo = vds.map("/path/to/dataset")
```

Users then create tasks and associate them with the virtual data objects. Each task uses a command that can be executed independent of the other tasks. The parameters to the task are the different arguments that are used for executing the original workflow task. However, any parameters that refer to workflow datasets are replaced by the corresponding virtual data objects.

```
# Create tasks
prod = Task(command="...")
prod.params = ['...', '...']
cons = Task(command="...")
cons.params = ['...', vdo, '...']
# Associate tasks to virtual data objects
vdo.add_producer(prod)
vdo.add_consumer(cons)
```

Users can associate virtual data objects with tasks by adding tasks as producers and/or consumers of the data objects. This task association tells VDS about the data and task dependencies in the workflow. This lets VDS schedule the data management tasks alongside the compute tasks to optimize workflow runtime and use of storage capacity. Hence by default, VDS does not copy any intermediate data to a persistent storage. In order to save intermediate datasets, users make the corresponding virtual data object persistent.

Table 2. The Programming Abstractions of MaDaTS

Abstraction	Description
task = **Task**(*command*)	Creates a task with an executable command. It has to be associated with a virtual data object.
vdo = **VirtualDataObject**(*dataset*) *vdo*.**add_producer**(*task*) *vdo*.**add_consumer**(*task*)	Map a dataset to a virtual data object. This does not add the virtual data object to the VDS. Tasks are associated as producers or consumers of a virtual data object.
vds = **VirtualDataSpace**() *vdo* = *vds*.**map**(*dataset*) *vds*.**add**(*vdo*)	Create a virtual data space. A virtual data object has to be explicitly added to a VDS, or a dataset needs to be mapped to a virtual data object in a VDS.
dag = **get_workflow_dag**(*vds*)	Get the workflow as a directed acyclic graph (DAG) of executable tasks from a VDS.
manage(*vds*)	Manage data and workflows through VDS. Virtual data objects and task associations have to be established prior to calling **manage**().
validate(*vds*)	A utility function to check if any dataset of a workflow is not mapped as a virtual data object in VDS. For datasets that are not mapped to VDS, they will not be managed by MaDaTS during workflow execution.

Users program the VDS using virtual data objects and VDS tasks. manage() abstracts the data management strategies, hiding the complexities of the underlying storage systems.

```
vdo.persist = True
```

Once a VDS is created and all the virtual data objects are added, users set up the global properties (e.g., data management strategy) and initiate VDS management as

```
vds.strategy = ...
manage(vds)
```

The `manage` operation generates scripts for managing tasks and data for the workflow. MaDaTS generates separate scripts for data and compute tasks. For each data task that is associated with a virtual data object, a script is generated to manage the associated operations on the dataset. For example, if a setup task is associated with the virtual data object, then a script is generated that prepares necessary directories prior to moving the data. Similarly, for a cleanup data task associated with a virtual data object, a script is generated that removes the dataset when all tasks associated with it finish execution. For each compute task, the arguments are parsed and a script is generated with the execution command. If there are additional properties for the compute tasks that specify the HPC scheduler, the number of CPUs to be requested, and the expected walltime, then a batch script is generated with all the specified information and the execution command. The dependencies between the tasks are managed by the order in which the virtual data objects are accessed. A DAG is generated that consists of the execution order of the scripts. The Tigres workflow library uses the DAG to manage data and execute tasks. Based on the task and virtual data object attributes, it submits these scripts through a batch scheduler or runs it locally on a

```
task_definitions = ({
    task = 'task1';
    inputs = ['vdo_in1', 'vdo_in2'];
    outputs = ['vdo_out1'];
    cpus = 1024;
    walltime = '00:30:00'; ...}, ...);
data_properties = ({
    name = 'vdo_out1';
    size = '1G';
    persist = true; ...}, ...);
data_management_strategy = 'STRATEGY';
```

Fig. 5. A workflow description contains task definitions, hints to the data elements, and a data management strategy for MaDaTS.

workstation. Once all the scripts have executed, and all compute and data tasks associated with virtual data objects have finished, the workflow execution terminates and the VDS is destroyed.

3.6 MaDaTS Command-Line Interface

MaDaTS also provides a **command-line interface** (**CLI**) that uses the programming abstractions to translate workflow descriptions into VDS, and manage them on HPC resources. The workflow descriptions for MaDaTS include resource requirements (including number of CPUs, walltime, etc.) in addition to the tasks, their dependencies, and their input/output datasets. The workflow description may contain "hints" on the data that help MaDaTS to select the storage layer where the data can be moved during workflow execution. These hints can be provided by the users as additional annotations to a workflow description, describing various properties about the data. The hints provide information about the storage and quality of service requirements for workflows. For example, a data hint `persist = true` suggests that the data needs to be persisted for long-term storage and MaDaTS needs to stage the data out to a persistent storage. Similarly, a user might specify a data size that helps MaDaTS decide the appropriate storage layer for the workflow data. In the absence of data hints, if the data management strategy requires the data to be moved, MaDaTS always moves the data to the storage layer with the highest throughput. Thus, users can skip specifying the data hints and MaDaTS still tries to aggressively optimize the I/O performance of the workflow.

Figure 5 provides a partial example of a workflow description, annotated with data hints. In this workflow description, a task `task1` is defined with certain inputs and outputs. Each input and output of the workflow task maps to a unique virtual data object which has an identifier. Each task definition contains information about resource requirements for executing the task. In this example, the size and persistence hints are specified for `vdo_out1`. Our current implementation accepts hints about the data size, persistence, and replication. These hints help MaDaTS to select the appropriate storage layer for accessing the data in the workflow. The existing set of hints is identified by our use cases, where workflows have both small (in KBs) and large (in GBs) datasets and where only part of the output data needs to be preserved for long-term storage. However, this is extensible to include other properties/hints since the data hints are specified as name-value pairs.

4 EVALUATION

In this section, we evaluate the capabilities and tradeoffs of the programming abstractions, and analyze the different data management strategies of MaDaTS.

4.1 Evaluation Setup

We evaluate the programming abstractions of MaDaTS on NERSC's Cori supercomputer. It is a Cray XC40 supercomputer with 2,388 Intel Xeon "Haswell" processor nodes and 9,688 Intel Xeon Phi **"Knight's Landing"** (**KNL**) nodes. For our experiments, we use only Haswell nodes. Each node has 32 cores and has 128 GB DDR4 2133 MHz memory and four 16 GB DIMMs per socket. Each core has its own L1 and L2 caches, with 64 KB and 256 KB, respectively. We use 32 nodes to execute each parallel phase of the workflow and one node to execute each sequential phase.

Cori provides multiple storage options with five different file systems that provide different throughput, storage space, and data retention policies. Cori's "burst buffers" provide fast persistent storage options that are built using Intel's P3608 SSDs, delivering 1.8 PB of usable capacity operating at 1.7 TB/s. The burst buffer on Cori is managed by the Cray Datawarp API. Users submit job scripts specifying datawarp directives to initiate automatic data transfers to and from burst buffers. The other storage options on Cori include "scratch," which is a Lustre file system with peak performance of approximately 700 GB/s, and "project," which has GPFS with a peak performance of 40 GB/s. "Home" on Cori is another GPFS filesystem, which is used for permanently storing small datasets and the default filesystem for login nodes. Additionally, Cori also provides a **High Performance Storage System** (**HPSS**), which is the long-term tape archival storage.

The job scripts on Cori are executed through the Slurm batch scheduler. Cori's queues are used to schedule and manage jobs by providing users with different **quality of service** (**QoS**) options. These options determine the amount of time spent by a job waiting to acquire resources. For our evaluation, we use realtime service to minimize unpredictable queue wait times. We run each experiment at least five times and for our plots, use the results that have minimum queue wait times in order to remove any bias due to unpredictable queue wait times.

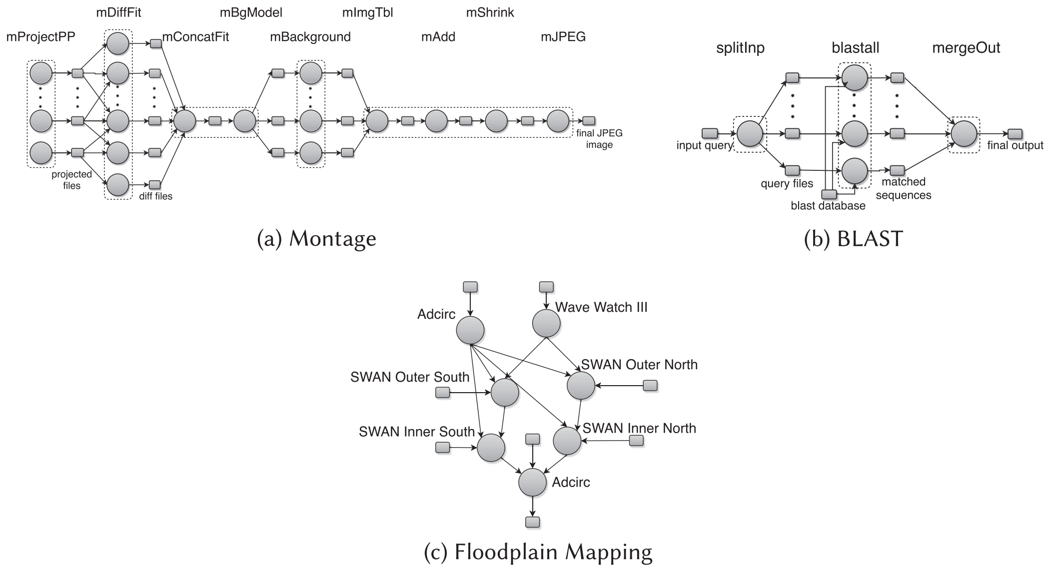
For managing the workflow tasks on HPC, we use the Tigres workflow library [15]. We use two versions of each workflow: one that uses the programming abstractions of MaDaTS to map the workflows to VDS, and the other is the original version of the workflows managed using batch job scripts. The workflow written in MaDaTS manages data and tasks using the VDS abstractions, and executes them using Tigres. The original workflow copies/moves data explicitly before and after each stage of the workflow. Dependencies between the workflow stages are managed by the Slurm batch scheduler.

4.2 Workloads

Figure 6 shows the three scientific workflows we use for our experiments. These workflows are selected because of their complex structure and the amount of data processed. For each workflow, we configure the storage hierarchy differently to evaluate the runtime performance, user-level control, and dynamic data management using the VDS abstractions. We select the storage tiers for each workflow based on their input and output data characteristics. For example, workflows that need intermediate datasets to be persisted post workflow execution, used the scratch filesystem as the staging tier. On the other hand, if the final outputs of a workflow are large enough and need long-term storage, they are moved to an archival storage. Table 3 lists the set of storage tiers configured for evaluating each workflow.

Montage is a data-intensive workflow that assembles a jpeg image from sky survey data (fits files). Montage is a combination of sequential and parallel tasks and requires all the input data in a single directory prior to executing the workflow. Each fits file is 1 MB in size, and a total of 55 GB of data is processed for degree 8.0 of the Montage workflow.

BLAST is a compute-intensive workflow that matches protein sequences against a large database (>6 GB). BLAST splits an input file (7,500 protein sequences for our tests) into multiple small



(a) Montage

(b) BLAST

(c) Floodplain Mapping

Fig. 6. Scientific workflows: Each workflow exhibits different data access and execution patterns. Montage assembles an image for survey M17 on band j from 2mass Atlas images. BLAST performs protein sequence matching using a shared large protein database. Floodplain Mapping focuses on accurately simulating the storm surges in the coastal areas of North Carolina.

Table 3. Storage Hierarchy Used for Different Workflows to Evaluate MaDaTS

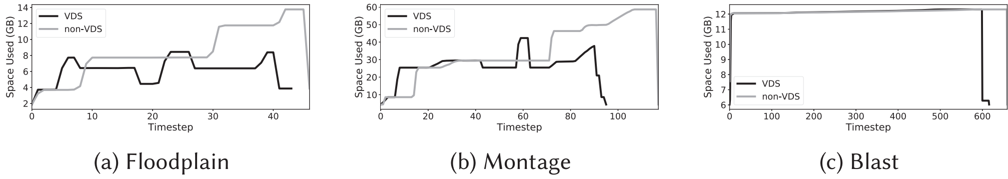
Workflow	Input-tier	Staging-tier	Output-tier
Montage	Project	Scratch	Home
Blast	Scratch	Burst Buffer	Home
Floodplain	Project	Scratch	Archive

The storage hierarchy is selected based on the different characteristics of input and output data for each workflow.

files (a few KBs) and then uses parallel tasks to compare the data in those files to that of a large shared database. It finally merges all the outputs from the parallel tasks into a single file.

In addition to the two scientific workflows, we also emulate the **Floodplain Mapping** workflow [3] with synthetic datasets. It focuses on accurately simulating the storm surges in the coastal areas of North Carolina. It uses a total of 14 GB of input and output data. Each stage of the emulated workflow simply reads and writes data from/to the filesystem. The files are read and written using the dd utility, keeping the file sizes the same as that defined in the original workflow definition.

We evaluate the effectiveness and efficiency introduced by the abstractions in MaDaTS using the different ways in which VDS can be programmed. Specifically, we evaluate the different configurations of VDS in MaDaTS. The *default* configuration copies the input and output datasets between storage tiers, but does not copy any intermediate output to persistent storage. This configuration also does not clean up any intermediate data from the storage tiers. The *cleanup* configuration removes unused datasets from the storage tiers, and only saves the final output datasets to a persistent storage. The *persist* option saves all intermediate data to a long-term persistent storage. Finally, the *cleanup + persist* option saves the intermediate data to a long-term persistent storage, and also removes them from the staging tier. The original workflow programs (referred to as



(a) Floodplain (b) Montage (c) Blast

Fig. 7. Storage space used over time using the VDS abstractions in MaDaTS versus the space used by the original (non-VDS) workflow program. VDS optimally uses the storage capacity because it removes datasets that will no longer be used during workflow execution. The original workflow program only removes the data when workflow execution completes.

non-VDS workflows) are hand-optimized versions of Tigres workflows that stage-in the input datasets, execute the tasks, stage-out the output datasets, and finally, clean up all intermediate datasets.

4.3 VDS Overheads and Tradeoffs

In this section, we evaluate the overheads, complexity, and other tradeoffs of using the VDS abstraction for managing workflows on tiered storage systems.

Storage Capacity. Figure 7 compares the use of storage capacity over time between the original and MaDaTS' version of the workflows. The measurements are taken in equal intervals during the entire duration of the workflow execution. Each time interval is depicted as a timestep on the X-axis of the graph. The Y-axis shows the total amount of space used across all the storage tiers on Cori. For these results, we use VDS with the cleanup option enabled, as the original non-VDS version of the workflows remove unused intermediate and staged datasets at the end of the workflow execution. Workflow runs with MaDaTS have fewer timesteps because they run faster than the corresponding original versions.

As can be seen from the graph, both Floodplain and Montage workflows with VDS use 30%–40% less overall space than the original non-VDS execution of the workflow. VDS automatically and asynchronously removes intermediate and staged datasets as the workflow stages execute. Instead of removing all the datasets at the end, VDS uses the workflow execution information to manage storage space, and removes the datasets that will no longer be used by the future steps of the workflow. For the Blast workflow, space usage is not that significantly lower as compared to the other workflows. This is because the input dataset is substantially smaller (few KBs in size) as compared to the large protein database (≈7 GB) that is used for the majority of the workflow execution. Both, MaDaTS and the original version of the workflow, stage the large protein database to the fast tier (burst buffer on Cori, in this case) that remains on the burst buffer for the entire lifetime of the workflow.

Workflow Runtime. Figure 8 shows the total runtime of the workflows with different VDS configurations in MaDaTS and compares the results to the original run of the workflow. The default VDS configuration results in the fastest execution of the workflows. This is because it only optimizes the runtime and not the use of storage space, resulting in fewer data movement steps in the workflow. It only stages in/out the datasets and uses the fast tier for intermediate datasets. There are no data cleanup and/or data stage-out tasks for intermediate datasets. The persist and cleanup options result in saving and cleaning out the datasets, respectively, from the staging area, resulting in additional tasks, some of which are at the end of the workflow execution (for output datasets). Although this creates additional data tasks and datasets, it has very small overheads as compared to the rest of the workflow run because the data tasks are managed concurrently with the compute tasks.

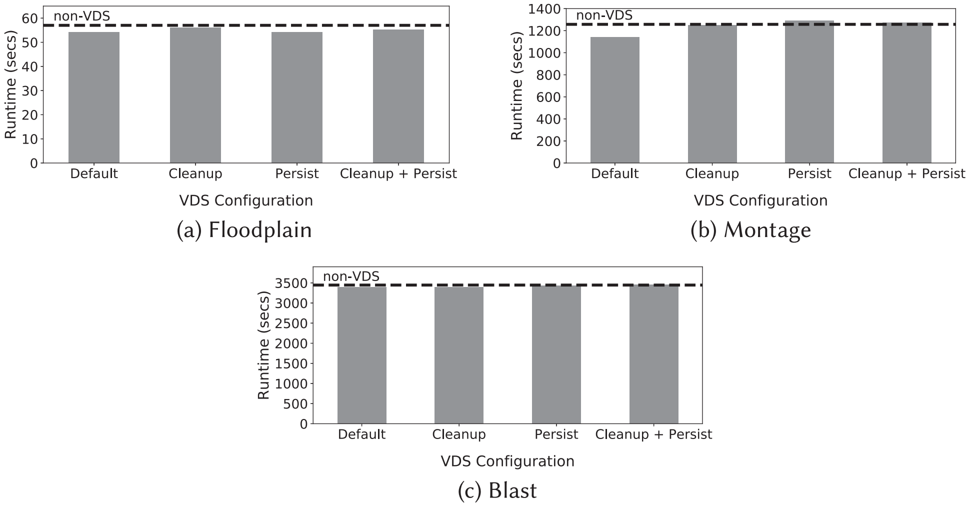
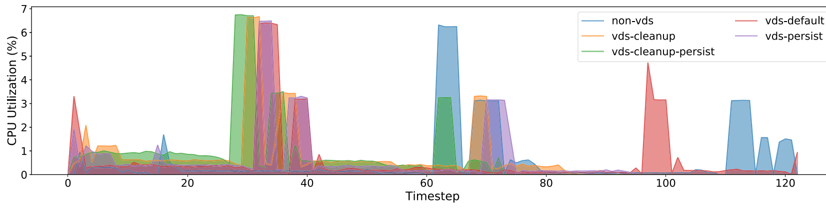


Fig. 8. Workflow runtime with different VDS configurations in MaDaTS. The original version (non-VDS) of the workflows performs only as fast as the worst-case VDS configuration (i.e., with cleanup and data persistence). Overlapping data and compute tasks minimize the overheads of data management in MaDaTS.

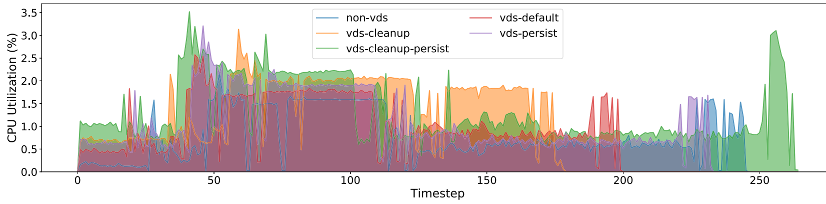
The original version of the workflow performs only as fast as the VDS version with cleanup and data persistence (which is the worst-case scenario with VDS because it creates tasks to copy all datasets of the workflow and also creates cleanup tasks to remove all intermediate datasets). The non-VDS workflow performance gets affected because it copies data only in phases before and after the workflow that adds data movement overheads to the workflow. For Blast, the data tasks are very small compared to the computation. Hence, the runtimes are unaffected by the changes in configuration options.

CPU Utilization. Figure 9 shows the amount of CPU used with different configurations in VDS for managing the workflows and their data. The different options in VDS creates additional data tasks to be executed. However, the data tasks themselves are all I/O bound and have minimal impact on the overall CPU utilization. For the experiments, we execute the workflows on a single node and monitor their CPU usage over time. We collect the CPU usage at every 0.5 s interval. Hence, the X-axis shows the logical timestep when the CPU usage is measured. The Y-axis shows the CPU used in percentages. The figure shows that the maximum CPU used by the VDS-enabled workflows is comparable to the non-VDS versions of the workflows. For all the workflows, VDS option with *cleanup + persist* enabled, show a small increase in the average CPU utilization (<1.5%) over the non-VDS version. This small overhead is due to the creation of additional data tasks for saving all intemediate datasets to a persistent storage, and removing them from the temporary storage prior to finishing the workflow execution. For all other configurations in VDS, the average CPU utilization results in <0.5% increase over the non-VDS version of the workflows.

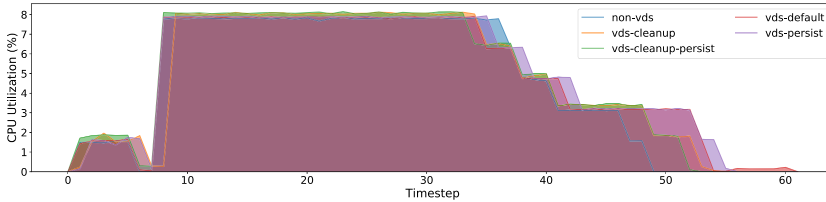
Data Tasks. Table 4 evaluates the amount of data movement involved in each workflow. The default configuration that does not do any cleanup and intermediate data saving have the fewest data tasks and data movement operations, which only correspond to the number of data stage-in and stage-out tasks for input and output datasets, respectively. This shows that instead of copying in and out the datasets for each stage of the workflow, VDS copies the datasets in/out as per the data use in the workflow. The best-case non-VDS version of the workflows also have fewest data movement operations because only the input and output datasets are moved. But programmers



(a) Floodplain



(b) Montage



(c) Blast

Fig. 9. CPU utilization with different options in VDS. The workflows show a small increase in the average CPU utilization (<1.5%), specifically when the *cleanup + persist* option in VDS is enabled due to the creation of additional tasks.

Table 4. Number of Data Tasks Created by VDS Based on the Different Options when Data is Always Used from the Fastest Storage Tier

Workflow	**Data-tasks/Data-movement-operations**					
	VDS				**non-VDS**	
	Default	**Cleanup (C)**	**Persist (P)**	**C+P**	**Best-case**	**Worst-case**
Floodplain	6/5	17/5	12/11	23/11	-/5	-/16
Montage	3/2	13/2	10/9	20/9	-/2	-/10
Blast	4/3	10/3	6/5	12/5	-/3	-/7

The total number of data tasks is the sum of data setup, movement and cleanup tasks. The numbers on the right side show the number of data movements. The best case for the non-VDS versions of the workflows move only the input and output datasets. The worst-case non-VDS also moves intermediate datasets for each stage of the workflows. VDS optimizes data movements based on dataset properties and VDS configuration, whereas for non-VDS versions of the workflows, programmers have to manually move the data for each workflow stage.

have to explicitly implement the data movement operations, keeping track of the placement and distribution of specific datasets on different storage tiers. If all the intermediate datasets are to be saved, then the number of data tasks in VDS equals the number of total (including the input, output, and intermediate) datasets in the workflow and a setup task for preparing necessary directories on the target storage tier. In the worst case, VDS has persist enabled for all intermediate datasets. VDS (with "persist" option enabled) moves fewer datasets as compared to the worst-case non-VDS

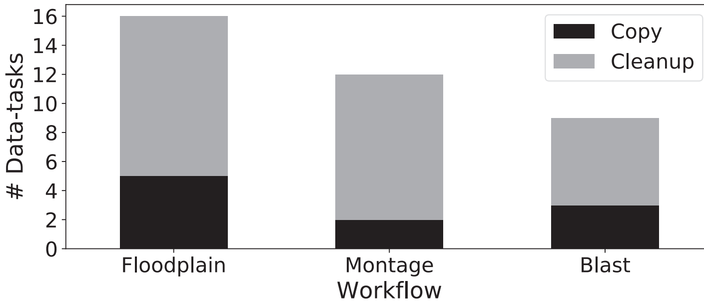


Fig. 10. Number of different types of data tasks created within VDS for the different workflows when users set auto-cleanup to true. There are as many cleanup tasks as there are intermediate datasets in the workflow. However, there are only as few data movement operations as there are persistent datasets for the workflow. Hence, many datasets are temporarily created on fast storage and removed resulting in larger number of cleanup tasks, but fewer data movements.

Table 5. Number of Scripts and **Lines of Code** (**LoC**) for Each Workflow With and Without Using MaDaTS

Workflow	**Workflow scripts**		**Total LoC**	
	MaDaTS	Original	MaDaTS	Original
Floodplain	1	8	86	97
Montage	1	10	78	102
Blast	1	4	48	37

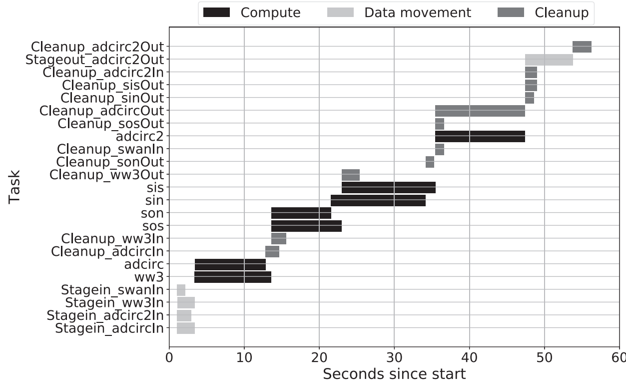
The original scripts (batch job scripts) have been hand optimized to have minimum lines of code as required to execute the workflow and associated tasks. The lines of code exclude the core Tigres workflow programs that are common to both MaDaTS and batch-scripts, and only include the programs written to submit the specific tasks and manage data on HPC systems.

version of the workflow because of the data movement optimizations based on dataset properties, workflow structure, and the VDS configuration. On the contrary, the non-VDS workflows move data in and out of the storage tiers during each stage of the workflows.

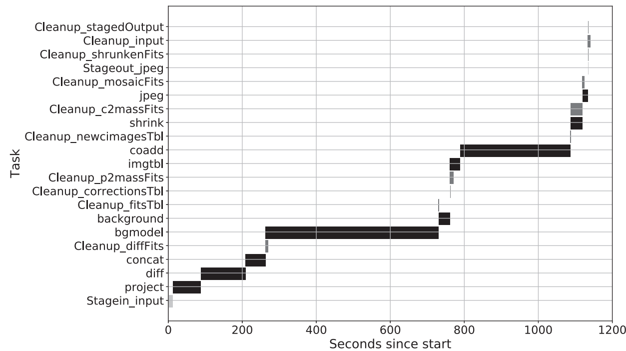
For VDS configurations with "cleanup" enabled, there are more data tasks than there are datasets in the workflow. However, these additional data tasks are cleanup tasks that asynchronously remove unused datasets from the storage tier as the workflow executes. Figure 10 shows the types of data tasks when the "cleanup" option is enabled in VDS. As can be seen from the graph, the majority of these data tasks are responsible for removing unused intermediate datasets from the storage system.

Programming Complexity. In order to evaluate the programming complexity of MaDaTS, we compare the complexity between writing a script using the programming abstractions of MaDaTS and the hand-optimized original workflow scripts that use batch scheduler dependencies and job scripts to manage workflow and data on HPC systems. We measure the complexity using two metrics: (a) lines of code and (b) the number of scripts written to manage a workflow and its data. We used cloc [5] to count the lines of code.

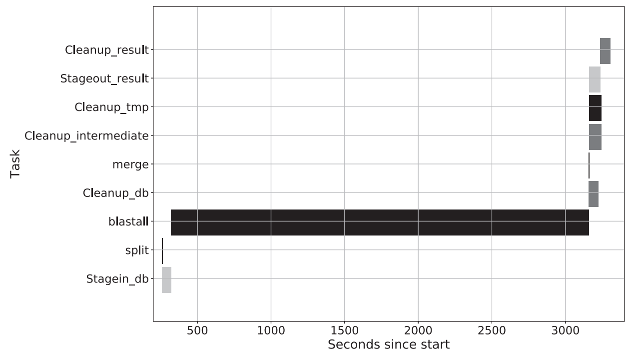
Table 5 lists the lines of code for each workflow. For Floodplain and Montage, the MaDaTS programs use fewer lines of code for data management and workflow execution on multi-tiered



(a) Floodplain



(b) Montage



(c) Blast

Fig. 11. Timeline of task execution and data management while using the MaDaTS programming model for workflows using "cleanup" option enabled. The data movement tasks are prefixed by "Stagein" and "Stageout" based on whether the data is moved in or out, respectively, and the cleanup tasks are prefixed by "Cleanup." MaDaTS optimizes workflow performance by overlapping data operations with task execution.

storage hierarchy. Blast has fewer lines of code for the original workflow scripts because only one stage of the workflow (blastall) is executed through the batch scheduler. This results in fewer lines of code as there are no dependency handlers and batch scheduler directives for the scripts, except one.

Table 5 also lists the number of scripts written to manage each workflow. A single MaDaTS program manages workflow execution and data management in an integrated and efficient way, whereas for the original workflow programs, there are separate job scripts for each stage of the workflow, and there is an additional controller script that needs to be written and submitted to manage the dependencies between the jobs. This creates an additional complexity, when not using MaDaTS' data-centric programming model.

Compute and Data Management. Figure 11 shows Gantt charts of MaDaTS programmed workflows. The figure shows the execution timeline of compute and data tasks (for staging in and out the datasets, as well as for removing unused datasets) in each workflow. For this experiment, we enable only the "cleanup" option in VDS, so no intermediate dataset is persisted or staged-out. As can be seen from the figure, each compute task in the workflow starts as soon as the input data for the specific task is staged-in, rather than staging-in all the input datasets. Similarly, the stage-out tasks are also overlapped with other cleanup tasks. In case the intermediate datasets are also staged-out, the stage-out data tasks will be executed with any overlapping compute tasks, in addition to the cleanup tasks. This is more efficient than separating out the data movement and cleanup tasks at the beginning and end of a (non-VDS) workflow. Hence, the graph shows how VDS optimizes the storage space and the runtime simultaneously by taking into account the scope of a dataset for the entire workflow rather than individual tasks of the workflow.

5 RELATED WORK

In this section, we highlight related work on data management in scientific workflows, abstractions for managing data, and approaches to optimize data movements between multi-tiered storage systems.

Tiered Storage Systems. Previous research has shown that workflow performance is limited by the data movement costs between the storage tiers [14], and I/O characteristics of the workflows [8]. Previous approaches have also proposed efficient data migration between SSDs and HDDs using task deadlines and I/O characteristics of the workloads [16, 32]. Just-in-time staging approaches have also been proposed to minimize data movement costs between multiple tiers of the storage system [22, 33]. Our previous work has also focused on placing data and improving performance on tiered storage systems based on user hints and data requirements [14, 25]. In this article, we introduce the programming abstractions necessary to simplify data management for workflows and provide different levels of control to the users, while balancing storage capacity and workflow performance on tiered storage systems.

Data Management. Previous work on managing data for scientific workflows proposed using workflow-aware storage systems [26, 28]. Such storage systems use data-aware scheduling that creates replicas of data or places the data near computation for minimizing performance bottlenecks. However, their work does not focus on multi-tiered storage systems. Custom file system interfaces [7] to minimize data movement between storage tiers based on workflow patterns have also been proposed. Workflow systems managing data over a wide-area network [18] make data management decisions based on network and storage parameters. They specifically focus on strategies for efficiently transferring the data over WAN. FlexIO [34] provides abstractions for placing analytics along the I/O path of a simulation and analysis workflow to optimize data movements and application performance. There is a lack of suitable abstractions that allow users to efficiently manage data based on the workflow characteristics.

Abstractions for Managing Data. Previous works have proposed different abstractions for managing data. Franklin et al. [13] proposes data management abstractions to hide the differences of multiple data sources through a common set of services. Memory abstractions like RDDs simplify in-memory computation for complex data analytics [31]. Several data abstractions have also

been proposed for accessing data in Grid environments and data catalogs [10, 12, 19]. BitDew [11] provides a programming abstraction for transparent data management on desktop grids. These abstractions are used for managing data efficiently in distributed setups, but do not consider any storage hierarchy. In this article, we propose a programming abstraction that provides users with different levels of control to manage workflow data on tiered storage systems.

6 CONCLUSIONS AND FUTURE WORK

In this article, we present a programming model that uses a VDS abstraction for simplifying workflow and data management on tiered storage systems. The programming abstractions of MaDaTS provide different levels of control to the users while balancing workflow performance and use of storage capacity. Our results show that programmers can obtain optimal results with less complexity and fewer lines of code.

The current implementation of VDS in MaDaTS manages data between POSIX compliant filesystems. Our future implementations will focus on supporting non-POSIX filesystems and cloud storage. We plan to integrate high-performance data transfer protocols like Globus into MaDaTS for efficient data transfer between remote storage systems, hiding the complexities of data management over WAN.

ACKNOWLEDGMENTS

The authors would like to thank Sudharshan S. Vazhkudai for his valuable suggestions and feedback.

REFERENCES

[1] 2013. Titan. Retrieved on October 7, 2021 from https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.
[2] 2015. Alluxio. Retrieved on October 7, 2021 from http://www.alluxio.org/.
[3] 2015. North Carolina Floodplain Mapping Program. Retrieved on October 7, 2021 from http://www.ncfloodmaps.com/.
[4] 2016. Cori. Retrieved on October 7, 2021 from http://www.nersc.gov/users/computational-systems/cori/.
[5] 2018. Count Lines of Code. Retrieved on October 7, 2021 from https://github.com/AlDanial/cloc.
[6] Asif Akram, J. Kewley, and Rob Allan. 2006. A data centric approach for Workflows. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)*.
[7] Chao Chen, Michael Lang, Latchesar Ionkov, and Yong Chen. 2016. Active burst-buffer: In-transit processing integrated into hierarchical storage. In *2016 IEEE International Conference on Networking, Architecture and Storage (NAS'16)*.
[8] Christopher Daley, Devarshi Ghoshal, Glenn Lockwood, Sudip Dosanjh, Lavanya Ramakrishnan, and Nicholas Wright. 2016. Performance characterization of scientific workflows for the optimal use of burst buffers. In *11th Workshop on Workflows in Support of Large-Scale Science (WORKS'16)*.
[9] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13, 3 (2005), 219–237.
[10] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. DataSpaces: An interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 15, 2 (2012).
[11] Gilles Fedak, Haiwu He, and Franck Cappello. 2009. BitDew: A data management and distribution service with multiprotocol file transfer and metadata abstraction. *Journal of Network and Computer Applications* 32, 5 (2009), 961–975.
[12] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. 2002. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management (SSDBM'02)*. IEEE Computer Society.
[13] Michael Franklin, Alon Halevy, and David Maier. 2005. From databases to dataspaces: A new abstraction for information management. *ACM Sigmod Record* 34, 4 (2005).
[14] Devarshi Ghoshal and Lavanya Ramakrishnan. 2017. MaDaTS: Managing data on tiered storage for scientific workflows. In *ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'17)*. ACM Press, 12 pages.
[15] Valerie Hendrix, James Fox, Devarshi Ghoshal, and Lavanya Ramakrishnan. 2016. Tigres workflow library: Supporting scientific pipelines on HPC systems. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'16)*.

[16] Stephen Herbein et al. 2016. Scalable I/O-Aware job scheduling for burst buffer enabled HPC clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*.

[17] I. Iliadis, J. Jelitto, Y. Kim, S. Sarafijanovic, and V. Venkatesan. 2015. ExaPlan: Queueing-based data placement and provisioning for large tiered storage systems. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 218–227. https://doi.org/10.1109/MASCOTS.2015.41

[18] Chen Jin, Scott Klasky, Stephen Hodson, Weikuan Yu, Jay Lofstead, Hasan Abbasi, Karsten Schwan, Matthew Wolf, W Liao, Alok Choudhary, et al. 2008. Adaptive IO system (adios). *Cray User's Group*.

[19] David T. Liu and Michael J. Franklin. 2004. GridDB: A data-centric overlay for scientific grids. In *the 30th International Conference on Very Large Data Bases*.

[20] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*.

[21] X. Meng, C. Wu, J. Li, X. Liang, Y. Bin, M. Guo, and L. Zheng. 2014. HFA: A hint frequency-based approach to enhance the I/O performance of multi-level cache storage systems. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS'14)*. 376–383. https://doi.org/10.1109/PADSW.2014.7097831

[22] Henry M. Monti, Ali R. Butt, and Sudharshan S. Vazhkudai. 2013. On timely staging of HPC job input data. *IEEE Transactions on Parallel and Distributed Systems* 24, 9 (2013).

[23] Ramya Prabhakar, Sudharshan S. Vazhkudai, Youngjae Kim, Ali R. Butt, Min Li, and Mahmut Kandemir. 2011. Provisioning a multi-tiered data staging area for extreme-scale machines. In *2011 31st International Conference on Distributed Computing Systems (ICDCS'11)*.

[24] Melissa Romanus, Fan Zhang, Tong Jin, Qian Sun, Hoang Bui, Manish Parashar, Jong Choi, Saloman Janhunen, Robert Hager, Scott Klasky, Choong-Seock Chang, and Ivan Rodero. 2016. Persistent data staging services for data intensive in-situ scientific workflows. In *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing (DIDC'16)*. ACM, New York, NY, 8 pages.

[25] W. Shin, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan. 2019. Data jockey: Automatic data management for HPC multi-tiered storage systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. 511–522. https://doi.org/10.1109/IPDPS.2019.00061

[26] Hyogi Sim, Youngjae Kim, Sudharshan S. Vazhkudai, Devesh Tiwari, Ali Anwar, Ali R. Butt, and Lavanya Ramakrishnan. 2015. AnalyzeThis: An analysis workflow-aware storage system. In *2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[27] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields. 2014. *Workflows for e-Science: Scientific Workflows for Grids*. Springer Publishing Company.

[28] Emalayan Vairavanathan, Samer Al-Kiswany, Lauro Beltrão Costa, Zhao Zhang, Daniel S. Katz, Michael Wilde, and Matei Ripeanu. 2012. A workflow-aware storage system: An opportunity study. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*. IEEE, 326–334.

[29] Teng Wang, Sarp Oral, Michael Pritchard, Kevin Vasko, and Weikuan Yu. 2015. Development of a burst buffer system for data-intensive applications. *CoRR*.

[30] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Computing* 37, 9 (2011).

[31] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, 15–28.

[32] G. Zhang, L. Chiu, C. Dickey, L. Liu, P. Muench, and S. Seshadri. 2010. Automated lookahead data migration in SSD-enabled multi-tiered storage systems. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*.

[33] Zhe Zhang, Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma, Gregory G. Pike, John W. Cobb, and Frank Mueller. 2007. Optimizing center performance through coordinated data staging, scheduling and recovery. In *The 2007 ACM/IEEE Conference on Supercomputing (SC'07)*. ACM, New York, NY.

[34] Fang Zheng, Hongbo Zou, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Jai Dayal, Tuan-Anh Nguyen, Jianting Cao, Hasan Abbasi, Scott Klasky, et al. 2013. FlexIO: I/O middleware for location-flexible scientific data analytics. In *2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13)*. IEEE, 320–331.

Received August 2020; revised November 2020; accepted March 2021