Rethinking Data Management for Big Data Scientific Workflows

Karan Vahi, Mats Rynge, Gideon Juve, Rajiv Mayani, Ewa Deelman
Information Sciences Institute - University of Southern California
Marina Del Rey, USA
{vahi, rynge, gideon, mayani, deelman}@isi.edu

***Abstract*—Scientific workflows consist of tasks that operate on input data to generate new data products that are used by subsequent tasks. Workflow management systems typically stage data to computational sites before invoking the necessary computations. In some cases data may be accessed using remote I/O. There are limitations with these approaches, however. First, the storage at a computational site may be limited and not able to accommodate the necessary input and intermediate data. Second, even if there is enough storage, it is sometimes managed by a filesystem with limited scalability. In recent years, object stores have been shown to provide a scalable way to store and access large datasets, however, they provide a limited set of operations (retrieve, store and delete) that do not always match the requirements of the workflow tasks. In this paper, we show how scientific workflows can take advantage of the capabilities of object stores without requiring users to modify their workflow-based applications or scientific codes. We present two general approaches, one that exclusively uses object stores to store all the files accessed and generated by a workflow, while the other relies on the shared filesystem for caching intermediate data sets. We have implemented both of these approaches in the Pegasus Workflow Management System and have used them to execute workflows in variety of execution environments ranging from traditional supercomputing environments that have a shared filesystem to dynamic environments like Amazon AWS and the Open Science Grid that only offer remote object stores. As a result, Pegasus users can easily migrate their applications from a shared filesystem deployment to one using object stores without changing their application codes.**

***Index Terms*— Pegasus, workflows, object stores, Pegasus Lite, data staging site, data management, cloud**

I. INTRODUCTION

Flexible and efficient data management is of critical importance in scientific workflow management systems, especially when they operate on large, distributed data sets. Scientific workflows enable the formalization of a set of computational tasks with inter-dependencies. In the era of Big Data, it is necessary to separate the definition of the workflow from the data location and computational sites. Therefore, the workflow management system needs to take a high-level description of a workflow and handle low-level operations such as task execution and data management.

In our experience, many workflow applications require POSIX file systems. That means that each task in the workflow opens one or more input files, read or write a portion of it, and then close it. As a result, workflow management systems need to provide a runtime environment where the tasks in the workflow have access to a POSIX file system for their data operations. Such an environment is readily available on traditional supercomputers [1], [2] usually in the form of a shared parallel file system. Scientists today have multiple alternative computational platforms available to them, including commercial clouds such as Amazon AWS [3], academic science clouds such as FutureGrid [4], [5] and distributed environments such as Open Science Grid [6]. A common feature in cloud and grid infrastructures is some kind of object storage close to the compute nodes. These object stores provide an attractive storage infrastructure, both in terms of scalability to serve a large number of files, and in the amount of storage provided. However, they only provide a limited set of operations to store, retrieve and delete data objects/files. In this paper we present and evaluate a solution for supporting Big Data workflows that rely on object stores for availability and scalability on clouds or grid infrastructures. We introduce the notion of data staging sites where data is placed for the applications to access. Decoupling data storage from the execution site also enables the use of a large number of distributed computational resources, and the migration of work from one execution site to another without the need to move all the data along with it. Our solution also allows for the data staging site to be co-located with the compute site, thereby leveraging the site's shared POSIX file system. This approach works particularly well in traditional supercomputing environments with fast parallel filesystems.

The paper is organized as follows. In Sections 2 and 3, we provide an overview of the different types of files that scientific workflows refer to, where they reside and how they are accessed and made available by workflow systems. Section 4 describes how these files can be made available through object stores for data intensive workflows. In Section 5, we present a data management solution based on the existing Pegasus Workflow Management System [7]. In Section 6 we discuss experiments that highlight the performance penalty a workflow might experience by using object stores exclusively. Finally in Sections 7 and 8 we refer to related work, conclude and explain future work.

II. TYPES OF FILES

In scientific workflows dependencies between jobs usually indicate both control and data dependencies. An edge between two tasks (T1→T2) implies that Task T1 generates an output file that is required as an input file for task T2. Each task in the workflow may read one or more input files, and write out

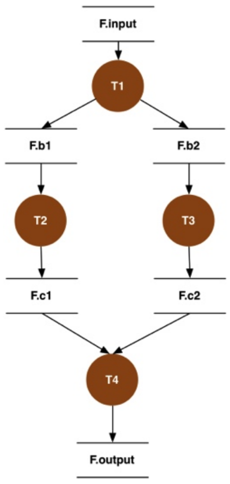

Figure 1: Files in a Workflow.

one or more output files. At the workflow level, we can classify the various files that a workflow refers to into three types: input, intermediate and output files. Each of these file types has a different lifecycle and storage binding.

- **Input Files** are not generated by any task in the workflow and usually reside on the user's own storage server. They can be transferred to another storage server at runtime and retrieved from there, or retrieved directly from the storage server by the tasks. In Figure 1, *F.input* is an input file.
- **Intermediate Files** are files that are generated by the workflow, but do not need to be saved after the workflow completes. They can be deleted once all the tasks that use them have been executed. In Figure 1, *F.b1*, *F.b2*, *F.c1* and *F.c2* are the intermediate files.
- **Output Files** are files that are generated by the workflow that are of interest to the user and need to be saved when the workflow completes. These files are usually transferred to a separate permanent storage during workflow execution. In Figure 1 *F.output* is an output file for the workflow. It is important to note that the output files don't necessarily have to be the outputs of the leaf jobs.

III. Execution Model For Workflows

Scientific workflows are submitted and managed by a workflow management system. The host where the workflow management system resides and coordinates the workflow execution from is referred to as the **submit host**. The workflow tasks execute on **worker nodes**, which can be inside a physical cluster or a virtual machine. The tasks in the workflow can be executed on one or more **compute sites** and require input data that can be present at one or more **input sites**. Intermediate files reside at a **data staging site** for the duration of the workflow. When the workflow has finished the output files are staged to one or more **output sites**, where they are catalogued and stored permanently. Depending on the workflow management system and the target execution environment, one or more of these sites can be co-located. For example, in the case where the input data already resides on the compute site, the compute site and the input data site are the same. Figure 2 shows a simple two-task workflow and illustrates how the files required by the workflow are moved in between the various sites, which are logically distinct from each other.

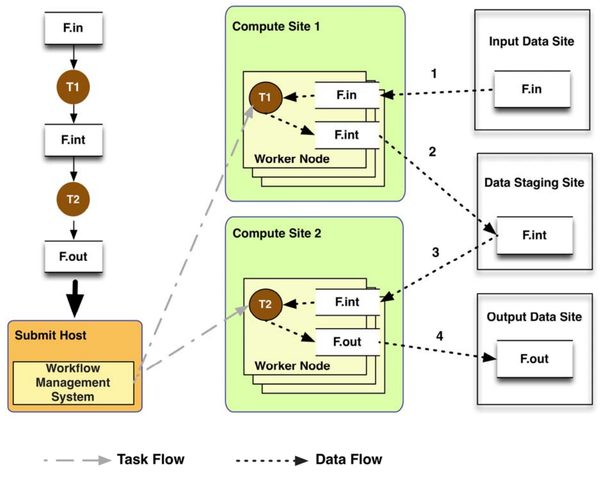

Figure 2: Workflow Execution Model.

IV. Object Storage For Scientific Workflows

An *object store* is a high-level storage service that provides a limited set of operations to store, retrieve and delete *data objects*, which are analogous to files in traditional file systems. Some object stores also support defining and querying metadata for objects. Unlike POSIX file systems, however, object stores do not provide efficient byte-level random access to objects. Unlike traditional files, it is not possible to open an object stored in an object store, read or write a portion of it, and then close it. Instead, a client needs to retrieve an object from an object store, update the data, and then store the data as a new object in the object store.

The internals of the object store can be quite complex in order to provide a reliable, scalable service. For example, Amazon's Simple Storage Service (S3) [8] provides retrieve and delete operations via REST and SOAP APIs. It also distributes multiple replicas of an object across a large set of storage servers to provide redundancy in case of failures.

Many grid storage services and protocols that are designed for data-intensive applications can be thought of as object stores. Services such as GridFTP [9], SRM [10], and iRODS [11], [12] support store/retrieve operations, but do not support random reads and writes. This generalization is important for the purposes of laying out the data management models for different execution environments against different data storage servers.

There are two options when relying on object stores for Big Data workflows: 1) use the store for all three types of data or 2) when available, use a shared file system on the compute node as a data staging site to store intermediate data products.

A. Exclusive Use of Object Stores

In the case where all data is stored in an object store, the workflow management system needs to seamlessly store/retrieve files from the object store, and make the files available to the workflow tasks on a local disk on the worker node. In such a setup, the workflow management system retrieves the inputs and intermediate files from the object store. The workflow task can then do the required POSIX I/O against the local file system, and when the task is complete, the workflow management system can store the intermediate results and outputs to the object store again. This enables the application to do normal POSIX I/O even though the workflow is being executed across distributed resources.

The additional steps the workflow management system has to take to interact with the object store can add some overhead to the workflow execution, but the overhead can be considered a small drawback in comparison to the benefits of being able to use scalable stores and distributing the computations across resources. Since the object store is used to hold both input and intermediate files, the tasks can execute anywhere as long as the workflow system can seamlessly retrieve/store data from/in the object store. In the example illustrated in Figure 3, a task T1 in a workflow can run on a campus compute cluster, and task T2 of the same workflow can run in a virtual machine on Amazon EC2. Both tasks, T1 and T2, use Amazon S3 object store as the data staging site for intermediate files. In general, by separating the data storage from the execution environment, it is easy for the workflow to be run across multiple execution environments. A common use case for this is to "spill over" from local/campus resources to national cyberinfrastructure or cloud resources, when compute demand exceeds what the local/campus resources can provide.

Figure 3 also illustrates the data flow for this workflow. In this case, file F.in is transferred to local file system of the worker node in the cloud on the HPC cluster. Task T1 starts on the node, reads in the input file F.in, and writes out an intermediate file, F.int, to the local file system. F.int is then transferred back to the object store that acts as the data staging site. F.int is then retrieved from the object store to the local file system of the EC2 node. Task T2 starts up and reads the F.int file (created by T1) and writes out the output file F.out to the local disk. The F.out file is then transferred to the object store. All transfers to/from the object store are done by the workflow management system.

The drawbacks of exclusively using object stores for scientific workflows include: duplicate transfers and latencies incurred in transfer of a large number of files. Duplicate transfers are obvious in the case where multiple tasks in the workflow use the same files [13], [14]. The object store will see similar requests for the same object over and over again for different tasks. Object stores usually scale well enough for this to not affect the performance of the workflow. Workflow systems may choose to cache files locally on the node, or a shared file system on the cluster, to alleviate this problem. This is possible as workflows are usually write once, read many—i.e. a file is generated by a single task and is then immutable during the course of the workflow run. A larger concern for overall workflow performance is latency. The design of cloud object stores provides great bandwidth, but the latency for a single retrieve or store operation can be several

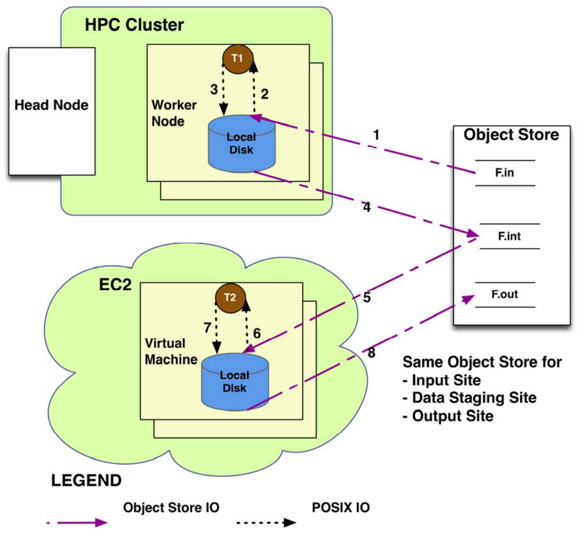

Figure 3: Exclusive Use of Object Stores for Application Data.

seconds long. For data-intensive workflows with a large number of small files this latency can add significantly to the overall workflow execution time. Another issue for large workflows can be the number of intermediate files that need to be shipped to the object store and then retrieved later on by other descendant tasks in the workflow. The duplicate transfers can also have cost implications as commercial object storage providers charge per GB of data transferred and stored, as well as the number of store and retrieve requests [15], [16].

B. Use of Shared File System as Data Staging Site

One of the ways to address the problem of latencies caused by duplicate transfers for input and intermediate files is for the workflow management system to stage the input data on-demand to a shared POSIX compliant file system that is shared across the compute nodes, and then run all the computations on that one resource. The shared file system acts as the data staging site for the workflow and stores all the files that the workflow refers to, for the duration of the workflow run. In this case only the input and output files are placed on the object store. This also helps to lower the cost of executing a workflow using a commercial object store, as the intermediate files are not transferred into or out of the object store. Figure 4 illustrates the data flow for a simple two-task workflow on a single HPC cluster with a shared file system. In this case, file F.in is transferred by the workflow management system to the shared file system of the cluster. Task T1 starts on node C1, reads the input file F.in from the shared file system, and writes the intermediate file F.int to the shared file system. Task T2 then starts up on node C2, reads F.in (created by T1) from the shared file system, and writes its output file, F.out, to the shared file system. F.out is then transferred to the object store by the workflow management system. This approach works particularly well in the traditional supercomputing environment with fast parallel file systems. For example, XSEDE [2] sites have been demonstrated to have great scalability for many scientific workflows [14], [17], [18]. Note that this approach of placing the tasks where the data exists results in a loss of the flexibility of where to place

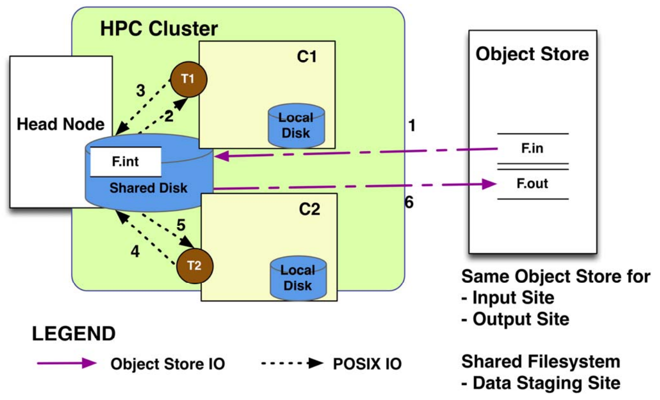

Figure 4: Use of Shared File System and Object Stores for Application Data.

the computation, for example spilling over the computation to another site if the first one is busy.

Both of the above approaches have their own merits, and which method to use depends on the type of workflow and the targeted workflow execution environment. The challenge for workflow management systems is to develop a flexible data management solution that allows the scientist to efficiently use the infrastructure available to them. The solution should have following features:

1. Allow for late binding of tasks and data. Tasks are mapped to compute resources at runtime based on resource availability. The task can discover input data at runtime, and possibly choose to stage the data from one of many locations.
2. Allow for static binding of tasks and data if the scientist has only one compute resource selected.
3. Support access to object stores using a variety of protocols and security mechanisms.

In the following section, we describe how we implemented these features in the Pegasus Workflow Management System.

V. IMPLEMENTATION

The Pegasus Workflow Management System [7] is used by scientists to execute large-scale computational workflows on a variety of cyberinfrastructure, ranging from local desktops to campus clusters, grids, and commercial and academic clouds. Pegasus WMS enables scientists to compose abstract workflows without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor [19], Globus [20], or Amazon EC2). First, a scientist creates catalogs that contain information about the data files and transformations used by the workflow (replica catalog and transformation catalog) as well as information about the available computing resources (site catalog). In many cases the data catalog is maintained by a collaboration, especially when data sets are large, as in the case of LIGO—a gravitational-wave physics project [21]. For large cyberinfrastructures, such as the OSG, site catalogs are maintained by the infrastructure.

Then the scientist provides an abstract representation of their workflow as a directed acyclic graph (DAG). Pegasus WMS takes in this abstract workflow (composed of tasks) and generates an executable workflow (composed of jobs) based on information about available resources. The system has three main components:

- Mapper (Pegasus Mapper): Generates an executable workflow based on the abstract workflow provided by the user or workflow composition system. It finds the appropriate software, data, and computational resources required for workflow execution. The Mapper also restructures the workflow to optimize performance and adds transformations for data management and provenance information generation.
- Execution Engine (Condor DAGMan): Executes the tasks defined by the workflow in order of their dependencies. DAGMan [22] relies on the resources (compute, storage and network) defined in the executable workflow to perform the necessary actions.
- Task Scheduler (Condor Schedd): DAGMan releases workflow tasks to the task executor for submission to local and remote resources.

All the workflow management system components are deployed on the submit host. The jobs are usually executed on remote resources. During the mapping process, Pegasus figures out where the jobs and data are placed. It queries a replica catalog to discover the locations of the input files performing replica selection in case of multiple file locations. It then adds data stage-in and stage-out jobs to the executable workflow, which are responsible for staging in the input data for the workflow from the input storage site and staging out the output data to the output storage site. The advantage of adding separate data stage-in and stage-out jobs to the executable workflow is two fold:

1. The Mapper can perform optimizations at the workflow level to stage-in and stage-out data. For example for large workflows, the mapper can choose to limit the number of independent stage-in jobs created, and thus control the number of connections the storage system sees.
2. No pre-staging of input data required. The data stage-in jobs ensure that data is available to compute jobs when they start executing.

The Mapper can also add cleanup jobs [23], [24] to the workflow that delete intermediate files when they are no longer required, and it can remove jobs from the workflow for which outputs already exist (data reuse). Figure 5 shows an input abstract workflow and the final executable workflow generated by the Mapper. The executable workflow has data management jobs added that stage-in the input data and stage-out the output data, and cleanup jobs that remove files after they are no longer required.

In earlier versions of Pegasus, the stage-in jobs staged in input data to a directory on the shared file system using the data servers on the head nodes of the compute site (i.e. the data staging site and the compute site were always co-located). This static binding of jobs and data at mapping time made it hard to support late binding of jobs, where the execution location is

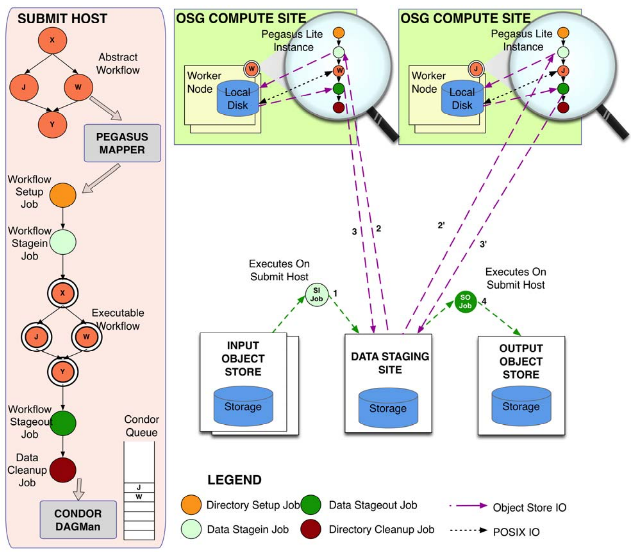

Figure 5: Workflows on OSG using SRM as the data staging site.

determined at runtime according to the resource's policy and configuration.

In our new approach, we have opted for a hybrid model, where the mapper still adds data management jobs for the executable workflow but these jobs are not tied to an execution site. Instead the data is transferred to a **data staging site** associated with the compute site. The **data staging site** is defined as the intermediate data storage site used to manage data for a workflow. It is used to store the input and all intermediate data generated during a workflow execution. The data staging site may or may not be the same as the cluster where the jobs execute. This decoupling enables Pegasus to support the shared file system setup while maintaining enough flexibility to support the late binding of jobs and the use of remote object stores.

We also introduce **Pegasus Lite**, a semi-autonomous lightweight execution engine for Pegasus tasks (with or without dependencies), running as part of the compute job, on remote worker nodes and in clouds VMs. Pegasus Lite receives from Pegasus WMS a set of tasks to manage on the remote resource and is responsible for managing their execution and data needs.

A. Data Staging Site

Separating the data storage from the execution environment enables the late binding of tasks. In Pegasus, the data staging site refers to the intermediate object storage to be used for the workflow. This intermediate object storage is used to store the input data staged in for the workflow from multiple remote servers and the output data that needs to be staged out to an output site. The data staging site also stores any intermediate data products that are generated by jobs as part of the workflow execution.

The data staging site is not required to have a job submission interface. Instead the Pegasus Mapper schedules the data stage-in and stage-out jobs to execute locally on the submit host. For consistency, the data staging site is described as any other site in the configuration, and selected at planning time using the mapper's *--staging-site* option. If not specified, the mapper co-locates the data staging site with the compute site.

While users can choose to let their jobs transfer input data directly from input data servers, they may not find it feasible because of performance or policy reasons. Funneling the input data through a data staging site enables Pegasus to control access to external data infrastructure when accessing the data. For example, a user's input data may be hosted on their desktop or a low performance data server that does not support a lot of concurrent connections. At the same time, the workflow that is executed against the input data set may have thousands of jobs that can execute in parallel. Pegasus' data management jobs, the granularity of which can be controlled independent of the workflow size, allows for the transfer of input data from input servers to a high performance data staging server in a controlled fashion. The jobs can then retrieve input data from this high performance server when they run on distributed resources like OSG.

B. Pegasus Lite

A critical component of this data management approach is the Pegasus Lite workflow engine. The Pegasus Mapper does workflow level reasoning and optimizations, but delegates a set of runtime decisions about a subset of the tasks in the workflow to Pegasus Lite. Pegasus Lite runs on the remote compute node to manage the execution of its tasks. The mapper provides information required for Pegasus Lite to transfer the required data, such as the location of data on the data staging sites, and credentials to access the data. The decision about where to execute a Pegasus Lite job is done at runtime, based on resource availability.

Pegasus Lite uses local information to manage its set of tasks. It discovers the directory in which to execute the tasks, pulls in data, runs the tasks, and stages out data back to the staging site. Pegasus Lite detects the environment and available transfer tools on the system, executes the transfers efficiently with error detection and retries. It can use a variety of data transfer protocols including: S3, GridFTP, SRM, iRODS, HTTP, and SCP. After execution it provides provenance information to the workflow management system on the submit host.

Pegasus Lite performs the following actions:

1. Discover the best working directory on the local disk. The decision is based on space requirements and hints from the environment, such as advertised tmp/scratch spaces.
2. Prepare the node for executing the unit of work. This involves determining whether any supporting software needs to be fetched, such as the Pegasus WMS worker package, which includes the pegasus-transfer tool that is used to get and put data from/to data staging site.
3. Use pegasus-transfer to stage in the input data to the working directory (created in step 1) on the remote node.
4. Execute the task(s).
5. Use pegasus-transfer to stage out the output data to the data staging site.
6. Send provenance information to Pegasus WMS.
7. Remove the working directory created in Step 1.

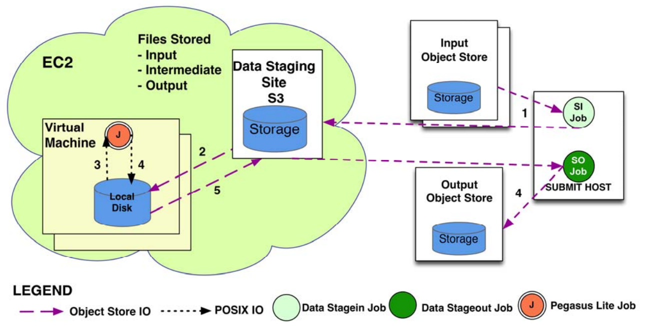

Figure 6. Workflows in EC2 with S3 as the data staging site.

Having a notion of a data staging site as intermediate storage for the workflow that is decoupled from the execution site, and the introduction of Pegasus Lite, gives Pegasus a lot of flexibility in the computing configurations that can be used. We now give a brief overview of the most common configurations supported.

1) Use case: OSG / SRM

The Open Science Grid (OSG) [6] is a consortium of over 100 universities and national labs, and consists of small and large campus clusters that provide software and services. On OSG it is common to use a provisioner, such as GlideinWMS [25], [26] to create compute resource overlays across many OSG sites, and use late binding for the jobs. The infrastructure provided by the Open Science Grid does not provide a shared file system. Instead, OSG provides dedicated high performance data-staging infrastructures, like SRM [10] backed by GridFTP, for the jobs to utilize at runtime. These infrastructures are not always directly co-located with the computational resources, but have high bandwidth and can be used to serve a large amount of files to the compute nodes.

The data flow for the workflows executing on OSG is illustrated in Figure 5. In this setup, the data stage-in jobs are added by the Pegasus Mapper to the executable workflow, to retrieve the input data from multiple input storage sites and store them on a SRM server. The SRM server serves as the data staging site for the workflows. The compute jobs in the workflow are dynamically matched with available worker nodes at runtime by the OSG provisioning tools [25], [26]. The Pegasus Lite instances that start up on the worker nodes are responsible for retrieving input data from the SRM server, executing compute tasks and pushing the output files back to the SRM server. The stage-out jobs added by the Pegasus Mapper transfer the output files for the workflow to the output object store specified by the user.

2) Use case: Amazon EC2 / S3

Most cloud providers offer scalable and highly available object stores such as S3[8] that provide a well-supported scalable storage infrastructure. However, this provides a data mapping challenge for the workflow management system. A common use case is to bring in data from the scientist's home institution, and use the cloud object storage for storing all the workflow datasets (input, intermediate and output files) Figure 6 illustrates such a deployment.

In the cloud, users can choose to let Pegasus WMS stage-in the input from different object stores (can be present outside of the cloud) to the cloud storage as part of the separate data stage-in jobs in the workflow. When Pegasus Lite starts the jobs on the cloud node, the lightweight engine retrieves their input files from the object store in the cloud. The intermediate files are stored back in the cloud object store [8] by Pegasus Lite. If desired, Pegasus can stage the output files to storage outside the cloud, or leave the data where it is.

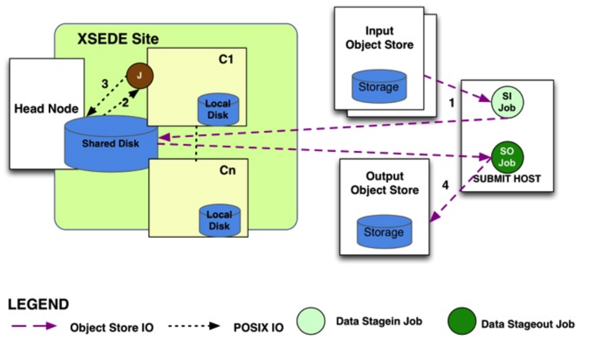

Figure 7: XSEDE with shared file system as the data staging site

3) Use Case: XSEDE

XSEDE is an infrastructure consisting mainly of traditional supercomputers, with a thin layer of grid services providing access to job schedulers and storage systems. Some XSEDE systems such as TACC Ranger and NICS Kraken [27] do not provide large amounts of local disk storage. Instead they provide a Lustre parallel file system [1], [2] as shared scratch storage for the whole cluster. On such systems a common use case is to stage-in the input data to the shared file system and use it to provide POSIX access for the scientific codes in the workflow. Figure 7 illustrates such a deployment. In this case, the data staging site is co-located with the compute site and we rely on the grid storage interface XSEDE sites provide, such as GridFTP, to place and retrieve data from the shared file system.

The placement of the data on the shared file system is managed by the data stage-in jobs added by Pegasus to the executable workflow. Alternatively, users can choose to pre-stage large datasets to the shared file system out of band and the data stage-in jobs will create symbolic links in the workflow's working directory.

VI. EXPERIMENTS

The goal of this work has been to provide an easy to use solution to run data intensive workflows in a variety of environments. The goal was not to improve workflow performance but rather to enable the application to take advantage of a number of different large-scale infrastructures for their data and compute management needs. In fact the increased flexibility on the placement of jobs by using a remote data staging site can be at the expense of the workflow runtimes, by increasing the amount of time the workflow spends doing data transfers. How much extra time will be required, and if it is even noticeable by the user, depends on the structure of the workflow, the amount of data which needs to be staged, the data access pattern, and the performance of the data staging site.

Two workflow experiments were performed to determine what impact, if any, using a data staging site has on workflow runtimes. The two workflows were chosen to have different characteristics: one I/O-intensive workflow and one CPU-intensive workflow. The workflows were run on Amazon EC2 [28] using 10 instances. One m1.xlarge instance was used as a dedicated NFS server, one c1.xlarge instance was used as a dedicated submit node, and 8 c1.xlarge instances were used as compute nodes to run jobs. All instances had software based RAID1 across 4 ephemeral disks in order to maximize local disk I/O performance. On the NFS server, the RAID1 file system was exported as an NFS share, and on the rest of the machines the RAID1 file system was used for job working directories.

The workflows were first executed in a shared file system setup, for which the workflows were configured to read/write directly from/to the NFS file system as described earlier in Figure 7. The workflows were then run on the same set of instances, but with data staging configured to use S3 as shown in Figure 6. Each workflow was run 3 times for each configuration and the runtimes from the 3 runs were averaged together. All input data was pre-staged to S3, and S3 was also indicated as the output site. While setting up the experiments, our predictions were that the I/O intensive workflow would run less efficiently on S3, and that we would not see much of a difference between NFS and S3 for the CPU intensive workflow.

The I/O intensive workflow we used was Montage [13]; an astronomy image application for computing mosaics of images from various sky survey instruments, such as the Spitzer telescope. For this experiment, we choose to run an 8 square degree patch of the 2MASS dataset, which is a fairly large Montage workflow consisting of 12,757 tasks and 7.8 GB of input images. Most of the tasks in this workflow take one or more input images, perform a operation to re-project or combine images, and write an output image. This perpetual reading and writing of the images is what makes Montage I/O intensive. During the workflow execution using S3, we observed a total of 66 GB written, and a total of 101 GB read from S3. The runtimes from our Montage experiment for can be found in Table 1.

The *Wall time* is the time the workflow took from start to finish as seen from the user's point of view. We can see that as expected the S3/local disk executions was longer than the S3/NFS run (129 min vs 70 mins).

All tasks launched in Pegasus WMS are invoked using Kickstart [29] on the remote compute nodes. The kickstart record contains runtime provenance information about the task such as the duration on the remote node, exit code and other useful debugging information. The *Cumulative Kickstart (CK) time* is the sum of all the task runtimes from the kickstart records. The *Cumulative Kickstart time* for the S3, 220 minutes, includes the computation and POSIX I/O time against a fast local filesystem, and can therefore be considered to be the optimal number for the computations. The relatively high CK time for the NFS case shows the poor performance of NFS for this application, which has an overhead of 701 minutes over the S3 case.

	NFS – sharedfs (minutes)	**S3 - nonsharedfs (minutes)**
Wall time	70	129
Cumulative Kickstart time	921	220
Cumulative job time	1030	1196

Table 1. Average run times for the Montage workflow

	NFS -sharedfs (minutes)	**S3 –nonsharedfs (minutes)**
Wall time	57	95
Cumulative Kickstart time	2935	2966
Cumulative Job time	2936	4557

Table 2. Average run times for Rosetta workflow.

The *Cumulative job time* is the sum of all the job runtimes as seen by HTCondor (the workflow job scheduler) on the submit side. Note the jobs can be composed of one or more tasks. In the NFS case the jobs read in input data from the shared filesystem and write out to the shared filesystem. However, jobs are more complex in the S3 case. They are made up of not only the Kickstart wrapped tasks, but also Pegasus Lite steps such as data transfers to/from S3. The runtime of the job seen by HTCondor on the submit host is thus the sum of kickstart time and the time for Pegasus Lite data transfers for the job. Therefore, the overhead of the S3 case is the difference between the *Cumulative job time* and the *Cumulative Kickstart time*, which is 976 minutes. Even though these overheads only differ by 39%, the overall wall times, 70 and 129 minutes, show an increase of 84%. The reason for this difference is the structure and the distribution of the runtimes of the Montage workflow. Data access patterns, and the effect of data staging, are different for the different transformations in the workflow. The second half of the workflow is where the re-projected, background corrected, and scaled images are mosaicked together, and many of these jobs operate on a large number of inputs. Any increase in data access times in this part of the workflow can significantly increase the overall workflow wall time. For example, due to the large number of inputs, the mAdd job, which is on the critical path of the workflow, took 3 times as long in the S3 case as in the NFS case.

For the CPU-intensive workflow experiment we choose a Rosetta [30] workflow. Rosetta is a macromolecular structure modeling code. Our experiments used a Rosetta workflow that contains 200 protein-folding simulations. The tasks in this workflow are CPU-intensive, have no data dependencies between tasks, and all require the same reference database made up of 477 files. Because Rosetta is CPU intensive and just a small amount of I/O is taking place, we expected the runtimes to be about the same regardless of data staging approach. The run times for the Rosetta workflow can be found in Table 2, which shows that the S3 wall times are significantly longer than the NFS times. The explanation for the slowdown in the S3 case is the reference database and the data access pattern. Size wise, the database is a relatively small 391 MB, but it is split into 477 separate files. If the database were contained in one file, it could be quickly transferred to/from S3. However, because it is split, the workflow needs to issue a separate get operation to fetch each piece of the database from S3, which adds a significant amount of overhead to the workflow (both in the walltime and the cumulative job time – which includes the S3 data transfers to the local filesystem). The many small data I/O requests result in a 50% slowdown for the workflow in the S3 case without task clustering when compared to NFS.

In contrast to the Montage experiment, where the large files and continuous I/O strained the NFS server, the light data access pattern of Rosetta, and the fact that the same files are read over and over again, which increases OS level file system caching, makes the Rosetta workflow work very well with NFS as can be seen from the similar Cumulative Kickstart times for Rosetta.

VII. Related Work

There are many approaches, some similar and some outside the scope of this paper, for a workflow management system to handle data. Swift [31] is using a model similar to what is described above in using a shared or local file system at a site as cache, and with the submit host acting as the data staging site. The system first selects a compute site to run a job, and then pushes the data from the submit host to a file system at the site. The job is run, and outputs are staged back the submit host. Intermediate files can be left on the shared file system for subsequent jobs. Our approach is more flexible as the data sites (input, output, and intermediate) can be distinct from the submit host, and use a variety of different protocols. Other workflow systems like Kepler [32], Triana [33] and Taverna [34] focus on streaming workflows or workflows that consist of components that call out to other web services. They have graphical user interfaces that allow users to compose workflows composed of different components and are normally not used to execute workflows that access large datasets. However, both rely on the user to specify data management steps in the workflow, providing limited data management automation.

Kepler [35] introduced a MapReduce actor that allows it to execute composite workflows where some tasks use the Map Reduce [36] paradigm. Hadoop [37] is often used for running data intensive applications, and it does provides a somewhat POSIX-like file system, allowing random reads, but only non-random writes. However, in this case the Hadoop infrastructure is responsible for the partitioning and distributing the input data to the individual nodes. Our approach deals with workflows that need to run on a variety of different execution environments, with codes that are not MapReduce.

On XSEDE, distributed file system like GPFS-WAN [38] has been available for the jobs to access data. Distributed file systems do provide POSIX access to remote input and output files. However, our experience with large datasets on XSEDE indicates that it is better to stage the large datasets to the local parallel file system of the XSEDE cluster and let the jobs access the datasets from there. Chirp [39] is another global distributed file system that has been proposed for grid environments. To access files a Chirp server needs to be launched on the input storage servers and the jobs then can access the input data when launched via Parrot [40] that traps IO system calls and redirects to the Chirp Server. A chirp deployment can be incorporated in our model, where Pegasus Lite can launch the job-using Parrot.

This paper has been mainly focused on flexibility of where data is placed and computations are placed. Much research has been done in focused strategies and optimized data placement in specific environments. Raicu et al [41] have explored use of a data diffusion approach where the data resides on a GPFS persistent storage. They employ various caching strategies at the executor nodes provisioned by Falkon [42] to provide better performance than the shared file system approach based on data locality of the workloads. Chervenak et al. [43] have investigated data placement for scientific workflows running under Pegasus by measuring the influence of data pre-staging. Alspaugh [44] used an open source rule engine called Drools to integrate with GridFTP and Globus RLS [45] for processing input data for workflows.

VIII. Conclusions And Future Work

In this paper we have described how workflow management systems can use object stores to execute large, data-intensive workflows used in Big Data problems. We have described two general approaches: one that exclusively uses object stores to store all the files accessed and generated during a workflow, and another that relies on the shared file system for caching the intermediate data sets. The data management challenge for the workflow system is to support both of these configurations and provide scientists the flexibility of using either approach. We then described the implementation of both approaches in the Pegasus Workflow Management System. Our solution involves decoupling the intermediate data storage site (the data staging site) from the compute site, and introducing an autonomous component called Pegasus Lite that is able to retrieve datasets when a job starts on a remote worker node. We have implemented this solution in Pegasus version 4 and it is now used in production to execute workflows in a variety of execution environments, ranging from traditional supercomputing environments with a shared file system to dynamic environments like the Open Science Grid that only offer remote object stores to store large datasets.

Pegasus Lite currently retrieves the input data for the job from a remote object store. However, the input data can be locally available on the compute nodes. This scenario is common when workflows are executed on dedicated application-specific compute infrastructure such as the LIGO Data Grid [46], where the data is actively replicated out of band on the various grid sites. We plan to introduce pluggable local discovery capabilities to Pegasus Lite that will allow it to discover files locally on the node before retrieving them from a remote object store. Having the flexibility of data and computation placement opens up the ability to study a variety of workflow scheduling and data placement algorithms. We plan to conduct more research in that direction.

IX. Acknowledgment

The National Science Foundation under the OCI SDCI program grant #0722019 and OCI SI2-SSI program grant #1148515 funds Pegasus WMS. We would like to thank Gaurang Mehta, Weiwei Chen and Jens-S. Vöckler for providing input and feedback during the implementation of the approach described in the paper. The use of Amazon EC2 resources was supported by an AWS in Education research grant.

X. References

[1] "XSEDE - Extreme Science and Engineering Environment," 2012, Available: http://www.xsede.org. .

[2] Catlett Charles, "TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications.," *IOS Press*, 2007.

[3] "Amazon Web Services," Available: http://aws.amazon.com/. .

[4] J. Diaz et al., "FutureGrid Image Repository: A Generic Catalog and Storage System for Heterogeneous Virtual Machine Images," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, pp. 560 –564.

[5] "FutureGrid: a distributed testbed, exploring possibilities with Clouds, Grids and High Performance Computing," 2012, Available: https://portal.futuregrid.org/. .

[6] M. Altunay et al., "A Science Driven Production Cyberinfrastructure–the Open Science Grid," *J. Grid Comput.*, vol. 9, no. 2, pp. 201–218, Jun. 2011.

[7] E. Deelman et al., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005.

[8] "Amazon Simple Storage Service," Available: http://aws.amazon.com/s3/. .

[9] W. Allcock et al., "The Globus Striped GridFTP Framework and Server," Washington, DC, USA, 2005, p. 54–.

[10] Abadie Lana and Badin Paolo, "Storage Resource Manager version 2.2: design, implementation, and testing experience.," *Proceedings of International Conference on Computing in High Energy and Nuclear Physics (CHEP 07)*, 2007.

[11] D. Hünich and R. Müller-Pfefferkorn, "Managing large datasets with iRODS - A performance analysis," in *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, 2010, pp. 647 –654.

[12] A. Rajasekar et al., "iRODS Primer: Integrated Rule-Oriented Data System," *Synthesis Lectures on Information Concepts, Retrieval, and Services*, vol. 2, no. 1, pp. 1–143, Jan. 2010.

[13] J.C. Jacob et al., "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *CoRR*, vol. abs/1005.4454, 2010.

[14] R. Graves et al., "CyberShake: A Physics-Based Seismic Hazard Model for Southern California," *Pure and Applied Geophysics*, vol. 168, no. 3, pp. 367–381, 2011.

[15] G. Juve et al., "An Evaluation of the Cost and Performance of Scientific Workflows on Amazon EC2," *Journal of Grid Computing*, vol. 10, no. 1, pp. 5–21, 2012.

[16] "Amazon S3 Pricing," Available: http://aws.amazon.com/s3/pricing/. .

[17] E. Deelman et al., "Managing Large-Scale Workflow Execution from Resource Provisioning to Provenance Tracking: The CyberShake Example," presented at the e-Science, 2006, p. 14.

[18] S. Callaghan et al., "Metrics for heterogeneous scientific workflows: A case study of an earthquake science application," *IJHPCA*, vol. 25, no. 3, pp. 274–285, 2011.

[19] D. Thain et al., "Distributed computing in practice: the Condor experience," *Concurrency - Practice and Experience*, vol. 17, no. 2–4, pp. 323–356, 2005.

[20] I.T. Foster et al., "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *IJHPCA*, vol. 15, no. 3, pp. 200–222, 2001.

[21] A.L. Chervenak et al., "Wide area data replication for scientific collaborations," *IJHPCN*, vol. 5, no. 3, pp. 124–134, 2008.

[22] "Condor DAGMan (Directed Acyclic Graph Manager)," Available: http://research.cs.wisc.edu/condor/dagman/. .

[23] G. Singh et al., "Optimizing workflow data footprint," *Scientific Programming*, vol. 15, no. 4, pp. 249–268, 2007.

[24] A. Ramakrishnan et al., "Scheduling Data-IntensiveWorkflows onto Storage-Constrained Distributed Resources," presented at the CCGRID, 2007, pp. 401–409.

[25] G. Juve et al., "Experiences with resource provisioning for scientific workflows using Corral," *Sci. Program.*, vol. 18, no. 2, pp. 77–92, Apr. 2010.

[26] I. Sfiligoi, "glideinWMS—a generic pilot-based workload management system," *Journal of Physics: Conference Series*, vol. 119, no. 6, p. 062044, Jul. 2008.

[27] "Kraken System Specifications," Available: http://www.nics.tennessee.edu/computing-resources/kraken/. .

[28] "Amazon EC2 Instance Types," Available: http://aws.amazon.com/ec2/instance-types/. .

[29] J.S. Voeckler et al., "Kickstarting remote applications," in *2nd International Workshop on Grid Computing Environments*, 2006.

[30] K. Kaufmann et al., "Practically Useful: What the Rosetta Protein Modeling Suite Can Do for You," *Biochemistry*, vol. 49, no. 14, pp. 2987–2998, Mar. 2010.

[31] Y. Zhao et al., "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," in *Services, 2007 IEEE Congress on*, 2007, pp. 199 –206.

[32] I. Altintas et al., "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, 2004, pp. 423 – 424.

[33] A. Harrison et al., "WS-RF Workflow in Triana," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 3, pp. 268–283, Aug. 2008.

[34] P. Missier et al., "Taverna, Reloaded.," in *SSDBM*, 2010, vol. 6187, pp. 471–481.

[35] J. Wang et al., "Kepler + Hadoop," 2009, pp. 1–8.

[36] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[37] "Apache Hadoop," Available: http://hadoop.apache.org/. .

[38] P. Andrews et al., "Massive High-Performance Global File Systems for Grid computing," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005, p. 53–.

[39] D. Thain et al., "Chirp: A practical global file system for cluster and grid computing," *Journal of Grid Computing*.

[40] D. Thain and M. Livny, "Parrot: An Application Environment for Data-Intensive Computing," *Journal of Parallel and Distributed Computing Practices*, pp. 9–18, 2005.

[41] I. Raicu et al., "Accelerating large-scale data exploration through data diffusion," in *Proceedings of the 2008 international workshop on Data-aware distributed computing*, New York, NY, USA, 2008, pp. 9–18.

[42] I. Raicu et al., "Falkon: a Fast and Light-weight tasK executiON framework," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2007, pp. 43:1–43:12.

[43] A.L. Chervenak et al., "Data placement for scientific applications in distributed environments," in *GRID2007*, 2007, pp. 267–274.

[44] Sara Alspaugh et al., "Policy-Driven Data Management for Distributed Scientific Collaborations Using a Rule Engine," Austin, 2008.

[45] A.L. Chervenak et al., "The Globus Replica Location Service: Design and Experience," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 9, pp. 1260–1272, 2009.

[46] E. Deelman et al., "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," presented at the HPDC, 2002, p. 225–.