Pegasus, a Workflow Management System for Science Automation

Ewa Deelman[a,*], Karan Vahi[a], Gideon Juve[a], Mats Rynge[a], Scott Callaghan[b], Philip J. Maechling[b], Rajiv Mayani[a], Weiwei Chen[a], Rafael Ferreira da Silva[a], Miron Livny[c], Kent Wenger[c]

[a] *University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*
[b] *University of Southern California, Los Angeles, CA, USA*
[c] *University of Wisconsin at Madison, Madison, WI, USA*

Abstract

Modern science often requires the execution of large-scale, multi-stage simulation and data analysis pipelines to enable the study of complex systems. The amount of computation and data involved in these pipelines requires scalable workflow management systems that are able to reliably and efficiently coordinate and automate data movement and task execution on distributed computational resources: campus clusters, national cyberinfrastructures, and commercial and academic clouds. This paper describes the design, development and evolution of the Pegasus Workflow Management System, which maps abstract workflow descriptions onto distributed computing infrastructures. Pegasus has been used for more than twelve years by scientists in a wide variety of domains, including astronomy, seismology, bioinformatics, physics and others. This paper provides an integrated view of the Pegasus system, showing its capabilities that have been developed over time in response to application needs and to the evolution of the scientific computing platforms. The paper describes how Pegasus achieves reliable, scalable workflow execution across a wide variety of computing infrastructures.

Keywords: Scientific workflows, Workflow management system, Pegasus

1. Introduction

Modern science often requires the processing and analysis of vast amounts of data in search of postulated phenomena, and the validation of core principles through the simulation of complex system behaviors and interactions. The challenges of such applications are wide-ranging: data may be distributed across a number of repositories; available compute resources may be heterogeneous and include campus clusters, national cyberinfrastructures, and clouds; and results may need to be exchanged with remote colleagues in pursuit of new discoveries. This is the case in fields such as astronomy, bioinformatics, physics, climate, ocean modeling, and many others. To support the computational and data needs of today's science applications, the growing capabilities of the national and international cyberinfrastructure, and more recently commercial and academic clouds need to be delivered to the scientist's desktop in an accessible, reliable, and scalable way.

Over the past dozen years, our solution has been to develop workflow technologies that can bridge the scientific domain and the available cyberinfrastructure. Our approach has always been to work closely with domain scientists—both in large collaborations such as the LIGO Scientific Collaboration [1], the Southern California Earthquake Center (SCEC) [2], and the National Virtual Observatory [3], among others, as well as individual researchers—to understand their computational needs and challenges, and to build software systems that further their research.

The Pegasus Workflow Management System, first developed in 2001, was grounded in that principle and was born out by the Virtual Data idea explored within the GriPhyN project [4, 5, 6]. In this context, a user could ask for a data product, and the system could provide it by accessing the data directly, if it were already computed and easily available, or it could decide to compute it on the fly. In order to produce the data on demand, the system would have to have a recipe, or workflow, describing the necessary computational steps and their data needs. Pegasus was designed to manage this workflow executing on potentially distributed data and compute resources. In some cases the workflow would consist of simple data access, and in others it could encompass a number of interrelated steps. This paper provides a comprehensive description of the current Pegasus capabilities and explains how we manage the execution of large-scale workflows running in distributed environments. Although there are a number of publications that focused on a particular aspect of the workflow management problem and showed quantative performance or scalability improvements, this paper provides a unique, intergrated view of Pegasus system today and describes the system features derived from research and work with application partners.

*Corresponding address: USC Information Sciences Institute, 4676 Admiralty Way Suite 1001, Marina del Rey, CA, USA, 90292

Email addresses: deelman@isi.edu (Ewa Deelman), vahi@isi.edu (Karan Vahi), gideon@isi.edu (Gideon Juve), rynge@isi.edu (Mats Rynge), scottcal@usc.edu (Scott Callaghan), maechlin@usc.edu (Philip J. Maechling), mayani@isi.edu (Rajiv Mayani), weiweich@acm.org (Weiwei Chen), rafsilva@isi.edu (Rafael Ferreira da Silva), miron@cs.wisc.edu (Miron Livny), wenger@cs.wisc.edu (Kent Wenger)

The cornerstone of our approach is the separation of the workflow description from the description of the execution environment. Keeping the workflow description resource-independent i.e. abstract, provides a number of benefits: 1) workflows can be portable across execution environments, and 2) the workflow management system can perform optimizations at "compile time" and/or at "runtime". These optimizations are geared towards improving the reliability and performance of the workflow's execution. Since some Pegasus users have complex and large-scale workflows (with O(million) tasks), scalability has also been an integral part of the system design. One potential drawback of the abstract workflow representation approach with compile time and runtime workflow modifications is that the workflow being executed looks different to the user than the workflow the user submitted. As a result, we have devoted significant effort toward developing a monitoring and debugging system that can connect the two different workflow representations in a way that makes sense to the user [7, 8].

Pegasus workflows are based on the Directed Acyclic Graph (DAG) representation of a scientific computation in which tasks to be executed are represented as nodes, and the data- and control-flow dependencies between them are represented as edges. This well-known, DAG based declarative approach has been used extensively to represent computations from single processor systems, to multi-CPU and multi-core systems, and databases [9, 10, 11, 12, 13, 14]. Through the abstraction of DAGs, we can apply the wealth of research in graph algorithms [15, 16] to optimize performance [17, 18] and improve reliability and scalability [6, 19]. By allowing a node to represent a sub-DAG, we apply recursion to support workflows with a scale of O(million) individual tasks.

This paper describes the Pegasus system components, the restructuring that the system performs, and presents a case study of an important earthquake science application that uses Pegasus for computations running on national cyberinfrastructure. The paper is organized as follows. Section 2 provides a general overview of Pegasus and its subsystems. Section 3 explores issues of execution environment heterogeneity and explains how Pegasus achieves portability across these environments. Section 4 discusses the graph transformations and optmizations that Pegasus performs when mapping a workflow onto the distributed environment ahead of the execution. At runtime, Pegasus can also perform a number of actions geared towards improving scalability and reliability. These capabilites are described in Section 6. As with any user-facing system, usability is important. Our efforts in this area are described in Section 7. We then present a real user application, the CyberShake earthquake science workflow, which utilizes a number of Pegasus capabilities (Section 8). Finally, Sections 9 and 10 give an overview of related work and conclude the paper.

2. System Design

We assume that: 1) the user has access to a machine, where the workflow management system resides, 2) the input data can be distributed across diverse storage systems connected by wide area or local area networks, and 3) the workflow computations can also be conducted across distributed heterogeneous platforms.

In Pegasus, workflows are described by users as DAGs, where the nodes represent individual computational tasks and the edges represent data and control dependencies between the tasks. Tasks can exchange data between them in the form of files. In our model, the workflow is *abstract* in that it does not contain resource information, or the physical locations of data and executables referred to in the workflow. The workflow is submitted to a workflow management system that resides on a user-facing machine called the *submit host*. This machine can be a user's laptop or a community resource. The target execution environment can be a local machine, like the submit host, a remote physical cluster or grid [20], or a virtual system such as the cloud [21]. The Pegasus WMS approach is to bridge the scientific domain and the execution environment by mapping a scientist-provided high-level workflow description, an *abstract workflow* description, to an *executable workflow* description of the computation. The latter needs enough information to be executed in a potentially distributed execution environment. In our model it is the workflow management system's responsibility to not only translate tasks to jobs and execute them, but also to manage data, monitor the execution, and handle failures. Data management includes tracking, staging, and acting on workflow inputs, intermediate products (files exchanged between tasks in the workflow), and the output products requested by the scientist. These actions are performed by the five major Pegasus subsystems:

Mapper. Generates an executable workflow based on an 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 can also restructure the workflow to optimize performance and adds transformations for data management and provenance information generation.

Local Execution Engine. Submits the jobs defined by the workflow in order of their dependencies. It manages the jobs by tracking their state and determining when jobs are ready to run. It then submits jobs to the local scheduling queue.

Job Scheduler. Manages individual jobs; supervises their execution on local and remote resources.

Remote Execution Engine. Manages the execution of one or more tasks, possibly structured as a sub-workflow on one or more remote compute nodes.

Monitoring Component. A runtime monitoring daemon launched when the workflow start executing. It monitors the running workflow, parses the workflow job, and tasks logs and populates them into a workflow database. The database stores both performance and provenance information. It also sends notifications back to the user notifying him or her of events such as failures, success, and completion of tasks, jobs, and workflows.

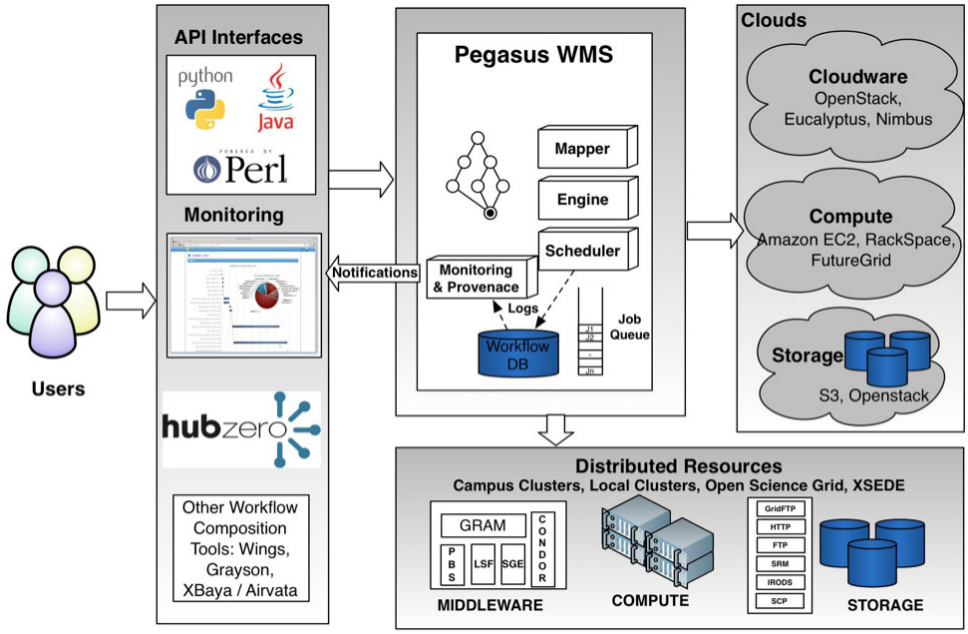

Figure 1: Pegasus WMS Architecture.

Scientists can interact with Pegasus via the command line and API interfaces, through portals and infrastructure hubs such as HubZero [22], through higher-level workflow composition tools such as Wings [23] and Airvata [24], or through application-specific composition tools such as Grayson [25]. Independent of how they access it, Pegasus enables scientists to describe their computations in a high-level fashion independent of the structure of the underlying execution environment, or the particulars of the low-level specifications required by the execution environment middleware (e.g. HTCondor [26], Globus [27], or Amazon Web Services [28]). Figure 1 provides a high level schematic illustrating how scientists interface with Pegasus and subsystems to execute workflows in distributed execution environments. In the remaining of this Section, we describe the Pegasus workflow structure, and give more details about the main system components.

2.1. Abstract Workflow Description for Pegasus

The abstract workflow description (DAX, Directed Acyclic graph in XML) provides a resource-independent workflow description. It captures all the tasks that perform computation, the execution order of these tasks represented as edges in a DAG, and for each task the required inputs, expected outputs, and the arguments with which the task should be invoked. All input/output datasets and executables are referred to by logical identifiers. Pegasus provides simple, easy to use programmatic API's in Python, Java, and Perl for the DAX generation [29]. Figure 2 shows a Python DAX generator code of a simple "hello world" program that executes two jobs in a serial order: `hello` and `world`. The corresponding DAX generated by running this program is shown in Figure 3. In this example, the first job with `ID0000001` refers to an executable called hello identified by the tuple (namespace, name, version). It takes an input file identified by a logical filename (LFN) `f.a`, and generates a single output file identified by the LFN `f.b`. Similarly, the second job takes as input file `f.b` and generates an output file `f.c`.

The Pegasus workflow can also be defined in a hierachical way, where a node of a DAX represents another DAX. The ability to define workflows of workflows helps the user construct more complex structures. In addition, it can also provide better scalability. As users started running larger workflows through Pegasus, the size of the DAX file increased correspondingly. For instance, the XML representation for a Cybershake workflow [30] developed in 2009 approached 1GB and containing approximately 830,000 tasks. Because many of the optimizations that Pegasus applies requires the whole graph in memory, Pegasus needs to parse the entire DAX file before it can start mapping the workflow. Even though we employ state of the art XML parsing techniques [31], for large workflows the Mapper can be inefficient and consumed large amounts of memory. Structuring the workflow in a hierarchical fashion solves this problem. The higher-level DAX has a smaller workflow to map. The lower-level DAX is managed by another instance of Pegasus at runtime as explained further in Section 5.1.

2.2. Workflow Mapping

In order to automatically map (or plan) the abstract workflow onto the execution environment, Pegasus needs information about the environment, such as the available storage systems, available compute resources and their schedulers, the location of the data and executables, etc. With this information, Pegasus transforms the abstract workflow into an executable workflow, which includes computation invocation on the target resources, the necessary data transfers, and data registration. The executable workflow generation is achieved through a series of graph refinement steps performed on the underlying

```python
#!/usr/bin/env python

from Pegasus.DAX3 import *
import sys
import os

# Create a abstract dag
dax = ADAG("hello_world")

# Add the hello job
hello = Job(namespace="hello_world",
            name="hello", version="1.0")
b = File("f.b")
hello.uses(a, link=Link.INPUT)
hello.uses(b, link=Link.OUTPUT)
dax.addJob(hello)

# Add the world job (depends on the hello job)
world = Job(namespace="hello_world",
            name="world", version="1.0")
c = File("f.c")
world.uses(b, link=Link.INPUT)
world.uses(c, link=Link.OUTPUT)
dax.addJob(world)

# Add control-flow dependencies
dax.addDependency(Dependency(parent=hello,
                             child=world))

# Write the DAX to stdout
dax.writeXML(sys.stdout)
```

Figure 2: Python DAX generator code for Hello World DAX.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<!-- generator: python -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX"
            version="3.4" name="hello_world">

    <!-- describe the jobs making
        up the hello world pipeline -->
    <job id="ID0000001" namespace="hello_world"
                    name="hello" version="1.0">

        <uses name="f.b" link="output"/>
        <uses name="f.a" link="input"/>
    </job>

    <job id="ID0000002" namespace="hello_world"
                    name="world" version="1.0">

        <uses name="f.b" link="input"/>
        <uses name="f.c" link="output"/>
    </job>

    <!-- describe the edges in the DAG -->
    <child ref="ID0000002">
        <parent ref="ID0000001"/>
    </child>
</adag>
```

Figure 3: Hello World DAX (Directed Acyclic graph in XML).

DAG, successively transforming the user's abstract workflow to the final executable workflow. As part of this process, Pegasus can also optimize the workflow. We describe this process of compile time workflow restructuring in detail in Section 4.

To find the necessary information, Pegasus queries a variety of catalogs. It queries a *Replica Catalog* to look up the locations for the logical files referred to in the workflow, a *Transformation Catalog* to look up where the various user executables are installed or available as stageable binaries, and a *Site Catalog* that describes the candidate computational and storage resources that a user has access to. For example, in the hello world example in Figure 4, the Mapper will look up the Replica Catalog for files `f.a`, `f.b`, `f.c`, and the Transformation Catalog for executables identified by tuples (`hello_world`, `hello`, 1.0), (`hello_world`, `world`, 1.0). For the sake of brevity, we will not focus on the details of the different catalogs in this publication. Pegasus supports various catalog implementations, including file-based and database based implementation [29].

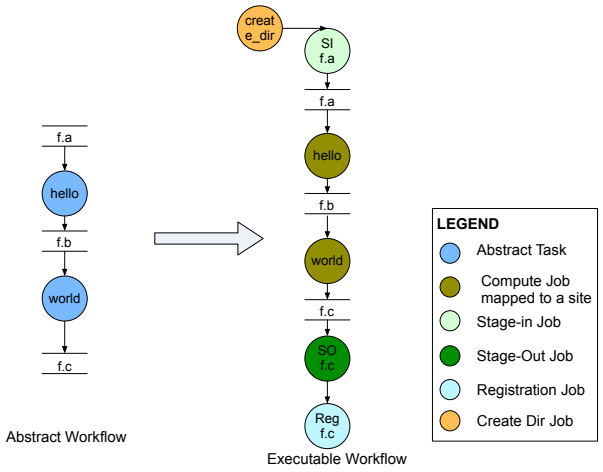

Figure 4: Translation of a simple Hello World DAX to an Executable Workflow

Once the Mapper decides where to execute the jobs and where to retrieve the data from, it can generate an executable workflow. The abstract tasks nodes are transformed into nodes containing executable jobs. Data management tasks are added, they represent data staging of the required input data, staging out the output products back to user specified storage systems, and cataloging them for future data discovery. Additionally, the workflow is also transformed and optimized for performance and reliability, as described in the following sections. Figure 4 shows an executable workflow with the nodes added by the Mapper for a simple hello world DAX.

2.3. *Workflow Execution*

After the mapping is complete, the executable workflow is generated in a way that is specific to the target workflow engine. It is then submitted to the engine and its job scheduler on the submit host. By default, we rely on *HTCondor DAGMan* [26] as the workflow engine and *HTCondor Schedd* as the scheduler. In this case the executable workflow is a HTCondor DAG describing the underlying DAG structure. Each node in the DAG is a job in the executable workflow and is associated with a job submit file that describes how each job is to be executed. It identifies the executable that needs to be invoked, the arguments with which it has to be launched, the environment that needs to be set before the execution is started, and the mechanism of how the job is to be submitted to local or remote resources for execution. When jobs are ready to run (their parent jobs have completed) DAGMan releases jobs

to a local Condor queue that is managed by the *Schedd* daemon. The Schedd daemon is responsible for managing the jobs through their execution cycle. HTCondor supports a number of different universes, which is a HTCondor term for execution layouts. By default, jobs run in the local HTCondor pool, but with the help of an extendable built-in job translation interface, the Schedd can interface with various remote job management protocols such as Globus GRAM [32], CREAM CE [33], and even SSH to submit jobs directly to remote resource managers such as SLURM [34], PBS [35], LSF [36], and SGE [37].

3. Portability

The abstract format used by Pegasus enables workflow sharing and collaboration between scientists. It also allows users to leverage the ever changing computing infrastructures for their workflows, without having to change the workflow description. It is Pegasus' responsibility to move workflows between different execution environments, and set them up to match the mode that the execution environment supports. This matching also takes into account how the data is managed on the target sites, such as the tools and file servers used to stage in the data to the site. In this section we focus on the different execution environments that Pegasus supports, the various remote execution engines used to run in these environments, and the supported data management strategies.

3.1. Execution Environment

A challenge for workflow management systems is to flexibly support a wide range of execution environments, yet provide optimized execution depending on the environment targeted. This is important as different scientists have access to different execution environments. In addition, we have seen over the years that Pegasus WMS has been developed that the type of execution environments and how they access them changes over time. Our users have used local clusters, a variety of distributed grids, and more recently computational clouds.

In the simplest case, the workflow can be executed locally, using a specialized, lightweight execution engine that simply executes the jobs in serial order on the *submit host*. This lightweight, shell-based engine does not rely on HTCondor to be installed or configured on the submit host. If users want to leverage the parallelism in the workflow while running locally, they can execute their workflow using HTCondor DAGMan and Schedd configured for local execution. These two modes enable users to easily port their pipelines to Pegasus WMS and develop them without worrying about remote execution. It is important to note that once a user's workflow can be executed locally using Pegasus WMS, users don't have to change their codes or DAX generators for remote execution. When moving to remote execution, Pegasus relies on HTCondor DAGMan as the workflow execution engine, but the Schedd is configured for remote submissions. The Mapper leverages, extends, and fine-tunes existing HTCondor functionality to make the workflow execute efficiently on remote resources. Currently, the most common execution environments for Pegasus WMS workloads are campus HTCondor pools, High Performance Computing (HPC) clusters, distributed computing infrastructures, and clouds.

3.1.1. HTCondor Pool

The most basic execution environment for a Pegasus workflow is a HTCondor pool. A set of local machines, which may or may not be in the same administrative domain, and one or more submit machines, are brought together under a common central HTCondor manager. The advantage of this execution environment is that it is very easy to use, with no extra credentials or data management components required. Authentication and authorization are handled by the operating system, and data management is either handled directly on top of a shared filesystem or internally with HTCondor doing the necessary file transfers. HTCondor is a very flexible and extensible piece of software, and can be used as a scheduler/manager for setups described in the following sections on distributed and cloud execution environments.

3.1.2. HPC Cluster Environment

The HPC cluster execution environment is made up of a remote compute cluster with interfaces for job control and data management. Common interfaces include Globus GRAM/GridFTP, CreamCE, and Bosco [38] if only SSH access is provided to the cluster. The cluster is usually configured for parallel job execution. The cluster usually also has a shared filesystem which Pegasus WMS can use to store intermediate files during workflow execution. Examples of HPC clusters include campus clusters, and HPC infrastructures such as XSEDE [39] in the U.S. and EGI [40] in Europe.

3.1.3. Distributed Execution Environment

Whereas the HPC cluster provides a fairly homogenous and centralized infrastructure, distributed computing environments like the Open Science Grid (OSG) [41] provide access to a somewhat heterogeneous set of resources, geographically distributed, and owned and operated by different projects and institutions. Executing a workflow in distributed environments presents a different set of challenges than the HPC cluster environment, as the distributed environment does not typically provide a shared filesystem, which can be used to share files between tasks. Instead, the workflow management system has to plan data placement and movements according to the tasks' needs. The preferred job management in distributed systems is through the HTCondor glideins, pilot jobs used to execute workflow tasks. Systems such as GlideinWMS [42] overlay a virtual HTCondor pool over the distributed resources to allow a seamless distribution of jobs.

3.1.4. Cloud Execution Environment

Cloud infrastructures are similar to the distributed infrastructures, especially when it comes to data management. When executing workflows in the cloud, whether a commercial cloud such as Amazon Web Services (AWS) [43], or science clouds like FutureGrid [44], or private cloud infrastructures such as those managed for example by OpenStack [45], the workflow

will have to handle dynamic changes to the infrastructure at runtime. Examples of these changes could be adding or removing virtual machine instances. Pegasus WMS can be configured to use clouds in many different ways, but a common way is to run an HTCondor pool across the virtual machine instances, and use a cloud provided object storage service, such as Amazon S3, for intermediate file storage. With a strong focus on reliability, HTCondor is particularly good at handling resources being added and removed, and the object storage services provide a centralized, yet scalable, data management point for the workflows.

The Pegasus WMS Mapper can also plan a workflow to execute on a mix of different execution environments. This is particularly useful when the workflow contains jobs with mixed requirements, for example some single core high-throughput jobs and some parallel high-performance jobs, with the former requiring a high-throughput resource like the Open Science Grid, and the latter requiring a high performance resource like XSEDE. Another common job requirement is software availability, where certain software is only available on a set of resources and another set of software is available on a disjoint set. Software availability is expressed in the Transformation Catalog, and the Mapper will map jobs to the correct execution environment, and add data management jobs to make sure data is moved to where it is needed.

3.2. Remote Workflow Execution Engines

The mapping between the abstract workflow tasks and the executable workflow jobs is many to many. As a result a job that is destined for execution (that is managed by the local workflow engine and the local job scheduler) can be composed of a number of tasks, and their dependencies. These tasks need to be managed on the remote resource: executed in the right order, monitored, and potentially restarted in case of failures. Depending on the execution environment's layout and the complexity of the jobs that need to be handled, we have developed different remote execution engines for managing such jobs.

3.2.1. Single Core Execution Engine

The simplest execution engine is *Pegasus Cluster*, which is capable of managing a job with a set of tasks runing on a remote resource. It can be used in all the supported execution environments. It is particulary useful when managing a set of fine-grained tasks as described in Section 4.3. Pegasus Cluster is a single-threaded execution engine, which means it runs only one task at the time, and that the input to the engine is just an ordered list of tasks. The execution order is determined by the Mapper by doing a topological sort on the associated task graph for the clustered job. The advantage of using the Pegasus Cluster execution engine is twofold: 1) more efficient execution of short running tasks, and 2) it can reduce the number of data transfers if the tasks it manages have common input or intermediate data.

3.2.2. Non-Shared File System Execution Engine

The *PegasusLite* [46] workflow engine was developed for the case in which jobs are executed in a non-shared filesystem environment. In such deployments, the worker nodes on a cluster do not share a file system between themselves or between them and the data staging server. Hence, before a user task can be executed on a worker node, the input data needs to be staged in and similarly the task's outputs need to be staged back to the staging server when a task finishes. In this case, the Mapper adds data transfer nodes to the workflow to move input data to a data staging server associated with the execution site where the jobs execute. PegasusLite is then deployed on demand when the remote cluster scheduler schedules the job onto a worker node. PegasusLite discovers the best directory to run the job in, retrieves the inputs from the data staging server, executes the user's tasks, stages out the outputs back to the data staging server and removes the directory that it created when it started. All the tasks are executed sequentially by PegasusLite. For more complex executions that utilize multiple cores, we provide an MPI-based engine, PMC described below.

3.2.3. MPI/Shared File System Execution Engine

We primarily developed *Pegasus MPI Cluster (PMC)* [47] to execute large workflows on Petascale systems such as Kraken [48] and Blue Waters [49] where our traditional approach of using Condor Glideins [50] did not work. These Petascale systems are highly optimized for running large, monolithic, parallel jobs, such as MPI codes and use specialized environments, such as the Cray XT System Environment. The nodes in the Cray environment have a very minimal kernel with almost no system tools available, no shared libraries, and only limited IPv4/6 networking support. The networking on these nodes is usually limited to system-local connections over a custom, vendor-specific interconnect. These differences significantly limited our ability to rely on HTCondor Glideins, as they depend on a basic set of system tools and the ability to make outbound IP network connections, and this PMC is good solution for these machines.

The PMC workflow execution engine targets HPC architecture, leverages MPI for communication and the Master-Worker paradigm for for execution. PMC takes in a DAG-based format very similar to HTCondor DAGMan's format and starts a single master process and several worker processes, a standard Master-Worker architecture. The Master distributes tasks to the worker processes for execution when their DAG dependencies are satisfied. The Worker processes execute the assigned tasks and return the results to the Master. Communication between the Master and the Workers is accomplished using a simple text-based protocol implemented using MPI_Send and MPI_Recv. PMC considers cores and memory to be consumable resources, and can thus handle mixes of single core, multi-threaded, and large memory tasks, with the only limitation being that a task has to have lower resource requirements than a single compute node can provide. Similar to DAGMan, PMC generates a rescue log, which can be used to recover the state of a failed workflow when it is resumed.

3.3. Data Management During Execution

The separation between the workflow description and the execution environment also extends to data. Users indicate the

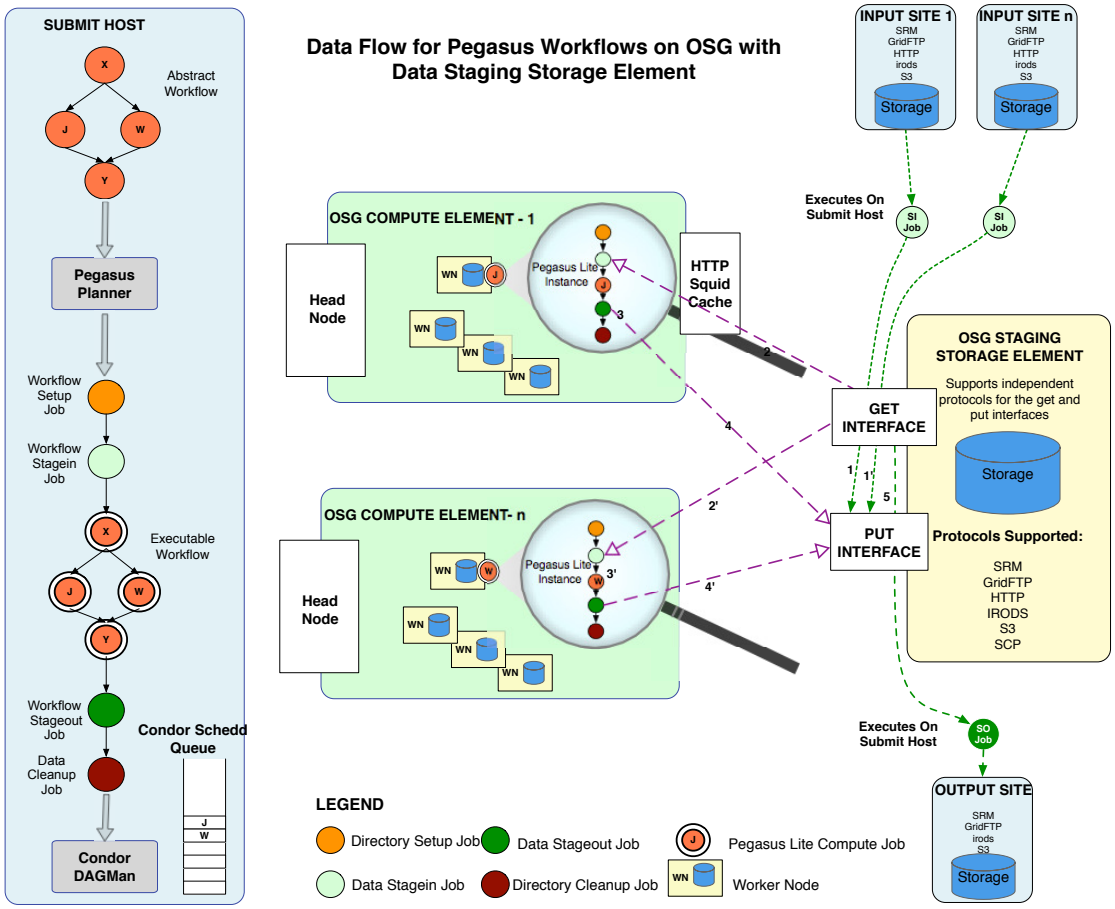

Figure 5: Data flow for a Pegasus Workflows planned with non-shared filesystem mode, running on the Open Science Grid.

needed inputs and the outputs generated by the workflow using logical file names. The Mapper can then decide where to access the input data from (many scientific collaborations replicate their data sets across the wide area for performance and reliability reasons) and how to manage the results of the computations (where to store the intermediate data products). In the end the results are staged out to the location indicated by the user (either the submit host or another data storage site).

Having the ability to move workflows between execution resources requires a certain degree of flexibility when it comes to data placement and data movement. Most workflows require a data storage resource close to the execution site in order to execute in an efficient manner. In an HPC cluster, the storage resource is usually a parallel filesystem mounted on the compute nodes. In a distributed grid environment, the storage resource can be a GridFTP or SRM server, and in a cloud environment, the storage resource can be an object store, such as Amazon S3 or OpenStack data services.

The early versions of Pegasus WMS only supported execution environments with a shared filesystem for storing intermediate data products that were produced during workflow execution. With distributed and cloud systems emerging as production quality execution environments, the Pegasus WMS data management internals were reconsidered and refactored to provide a more open-ended data management solution that allows scientists to efficiently use the infrastructure available to them. The criteria for the new solution were:

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

As a result, Pegasus 4.0 introduced the ability to not only plan workflows against a shared filesystem resource, but also remote object stores resources. Consider a job to be executed on Amazon EC2 with a data dependency on an object existing in Amazon S3. The job will have data transfers added to it during the planning process so that data is staged in/out to/from the working directory of the job. The three user selectable data placement approaches are:

Shared Filesystem. Pegasus uses a shared filesystem on a compute resource for intermediate files. This is an efficient approach but limits the execution of the workflow to nodes, which

have the shared filesystem mounted. It can also have scalability limitations for large-scale workflows.

Non-shared Filesystem. Each job is its own small workflow managed by the PegasusLite engine. It manages data staging of inputs and outputs, to and from where the job is executed. This approach provides a flexible solution when executing workflows in a distributed environment, and enables late binding, or decision making at runtime (see Section 3.2.2). Because jobs are not limited to run where the data already is located, these jobs can be started across different execution environments and across different infrastructures, and can still retrieve the necessary input data, execute the tasks, and stage out the output data. Supported remote files systems include iRODS [51], Amazon S3 [52] and compatible services, HTTP [53], GridFTP [54], SRM [55], and SSH [56].

HTCondor I/O. HTCondor jobs are provided a list of inputs and outputs of the job, and HTCondor manages data staging. The advantages of this approach is that it is simple to set up as no additional services and/or credentials are required, and data transfer status/failures are tied directly to job execution. The downside is a more limited number of protocols than the non-shared filesystem case can provide, and less flexibility in the data placement decisions.

Figure 5 illustrates a Pegasus WMS workflow running on the Open Science Grid in the non-shared filesystem mode. To the left in the Figure is the submit host, where Pegasus maps the abstract workflow to an executable workflow. During execution, data is staged into a central object store, such as a SRM, GridFTP, or iRODS server. When a job starts on a remote compute resource, PegasusLite detects an appropriate working directory for the task, pulls in the input data for the task from the storage element, runs the task, stages out products of the task and cleans up the working directory. The workflow also contains tasks to clean up the storage element as the workflow progresses and to stage the final output data to a long-term storage site.

The main concept in the new data management approach is to be able to place the data staging element anywhere in the execution environment. In the previous Open Science Grid example, the data storage element was an SRM/GridFTP/iRODS server placed near the compute resources. Let's consider the same workflow, but this time running on an HPC cluster with a shared filesystem (Figure 6).

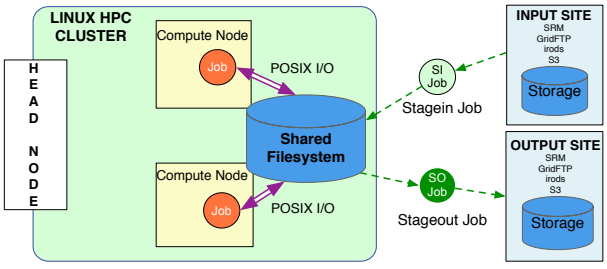

Figure 6: Data management on a shared filesystem resource (only the remote execution environment is shown).

In this example, the data storage element has moved inside the compute resource that is provided by the shared filesystem. By changing the data staging approach to *Shared Filesystem*, Pegasus will optimize the data access patterns by eliminating staging to/from the staging element and set up the workflow tasks to do POSIX based I/O directly against the shared filesystem.

Pegasus WMS has a clean separation between the Mapper deciding what files need to be transferred when, and how those transfers are executed. The execution is handled by a data management component called `pegasus-transfer`. `pegasus-transfer` can be executed anywhere in the system, such as on the submit host in case of transfers to/from remote resources, or on the remote resource to pull/push data from there. The Mapper provides `pegasus-transfer` with a list of transfers to complete, and a set of credentials required to authenticate against remote storage services. `pegasus-transfer` then detects the availability of command line tools to carry out the transfers. Examples of such standard tools are `globus-url-copy` for GridFTP transfers, `wget` for HTTP transfers, or Pegasus internal tools such as pegasus-s3 to interact with Amazon S3. Transfers are then executed in a managed manner. In the case of the source and destination protocols being so different that not one tool can transfer from the source directly to the destination, `pegasus-transfer` will break the transfer up into two steps: using one tool for the transfer from the source to a temporary file, and another tool for the transfer from the temporary file to the destination. This flexibility allows the Mapper to concentrate on planning for the overall efficiency of the workflow, without worrying about transfer details, which can come up at runtime.

3.4. Job Runtime Provenance

Workflow and job monitoring are critical to robust execution and essential for users to be able to follow the progress of their workflows and to be able to decipher failures. Because the execution systems have different job monitoring capabilities and different ways of reporting the jobs status, we developed our own uniform, lightweight job monitoring capability: `pegasus-kickstart` [57] (kickstart). The Pegasus Mapper wraps all jobs with kickstart, which gathers useful runtime provenance and performance information about the job. The information is sent back to the submit host upon job completion (whether it is successful or not) and is integrated into a larger view of the workflow status and progress, as described in Section 7.2.

4. Compile Time Workflow Restructuring

As mentioned in the previous section, the flexibility and portability in terms of the supported execution environments and data layout techniques is achieved by having a separation between the user's description of the workflow and the workflow that is actually executed. The Mapper achieves this by performing a series of transformations of the underlying DAG

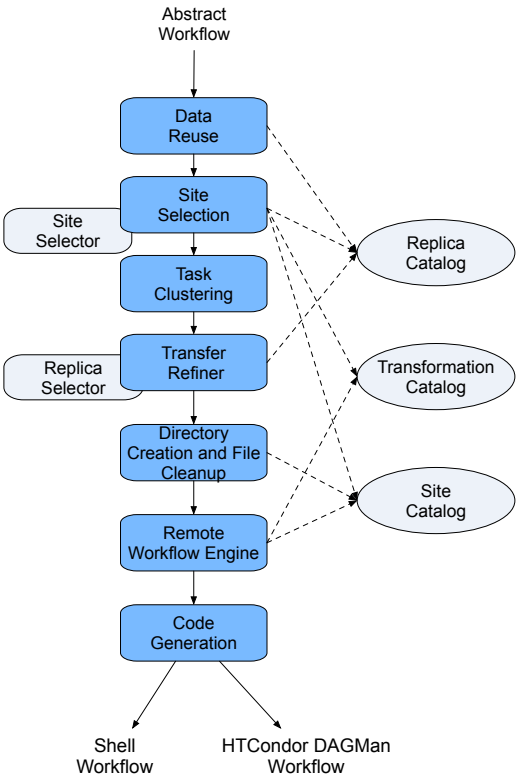

Figure 7: Pegasus Compile Time Workflow Restructuring.

shown in Figure 7. During this process the Mapper also performs a number of transformations to improve the overall workflow performance. The Mapper has a pluggable architecture that allows Pegasus WMS to support a variety of backends and strategies for each step in the refinement pipeline.

In order to better explain the refinement steps, Figure 8 shows a diamond-shaped six node input workflow and the corresponding final executable workflow generated by the Mapper. The following subsections detail this process.

4.1. Data Reuse/Workflow Reduction

Pegasus originated from the concept of virtual data. As a result, when Pegasus receives a workflow description, it checks whether the desired data are already computed and available. The Mapper queries a Replica Catalog to discover the locations for these files. If any of these files exist, then the Mapper can remove the jobs that would generate that data from the workflow. The Mapper can also reason, whether removal of a node in the workflow results in a cascaded removal of the other parent nodes, similar to the tool "make" [58] used to compile executables from source code.

In the case where the workflow outputs are already present somewhere, the Mapper will remove all the compute nodes in the workflow, and just add a data stageout and registration node to the workflow. These jobs will result in transfers of the output to the user-specified location. In the example shown in Figure 8, the Mapper discovers that the file `f.d` exists somewhere, and is able to reason that `Task D` and by extension `Task B` do not need to be executed. Instead, there is a node added to the workflow to stage `f.d` (during the replica selection stage). `Task A` still needs to execute because file `f.a` is needed by `Task C`. Data reuse is optional. In some cases it is actually more efficient to recompute the data than access it.

4.2. Site Selection

In this phase, the Mapper looks at the reduced workflow and passes it to a Site Selector that maps the jobs in the reduced workflow to the candidate execution sites specified by the user. The Mapper supports a variety of site selection strategies such as Random, Round Robin, and HEFT [59]. It also allows to plug in external site selectors using a standard file-based interface [29], allowing users to add their own algorithms geared towards their specific application domain and execution environment.

During this phase, the Mapper is also able to select sites that don't have user executables pre-installed as long as the user has stageable binaries cataloged in the Transformation Catalog and the executable is compatible with the execution site's architecture. This is a fairly common use case, as Pegasus users execute workflows on public computing infrastructure [41, 60], where users executables may not be preinstalled, or they want to use the latest version of their codes.

4.3. Task Clustering

High communication overhead and queuing times at the compute resource schedulers are common problems in distributed computing platforms, such as cloud computing infrastructures and grids [61]. The execution of scientific workflows on these platforms significantly increases such latencies, and consequently increases the slowdown of the applications. In Pegasus, the task clustering technique is used to cope with the low performance of short running tasks, i.e. tasks that run for a few minutes or seconds [62, 63]. These tasks are grouped into coarse-grained tasks to reduce the cost of data transfers when grouped tasks share input data, and save queuing time when resources are limited. This technique can improve as much as 97% of the workflow completion time [64].

Pegasus currently implements level-, job runtime-, and label-based clustering. In the following, we give an overview of these techniques.

Level-based horizontal clustering. This strategy merges multiple tasks within the same horizontal level of the workflow, in which the horizontal level of a task is defined as the furthest distance from the root task to this task. For each level of the workflow, tasks are grouped by the site on which they have been scheduled by the Site Selector. The clustering granularity, i.e. the number of tasks per cluster in a clustered job, can be controlled using any of the following parameters: `clusters.size`, which specifies the maximum number of tasks in a single clustered job; and `clusters.num`, which specifies the allowed number of clustered jobs per horizontal level and per site of a workflow. Figure 9 illustrates how the `clusters.size` parameter affects the horizontal clustering granularity.

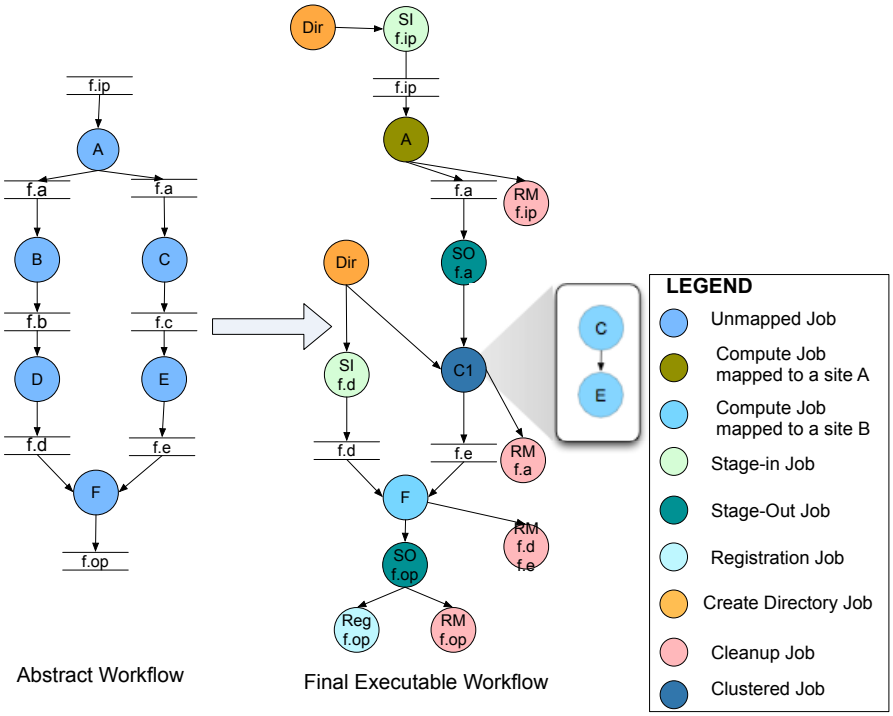

Figure 8: Translation from Abstract to Executable Workflow.

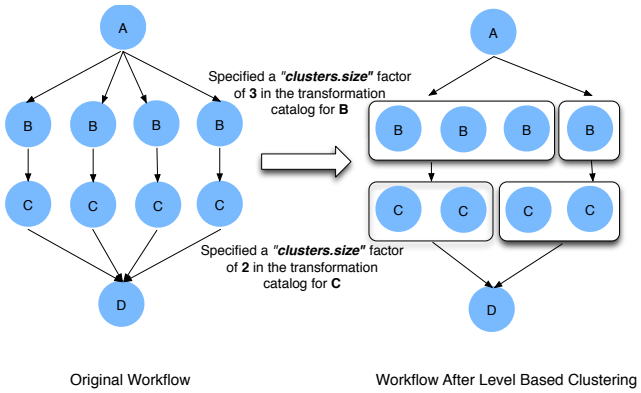

Figure 9: Example of level-based clustering.

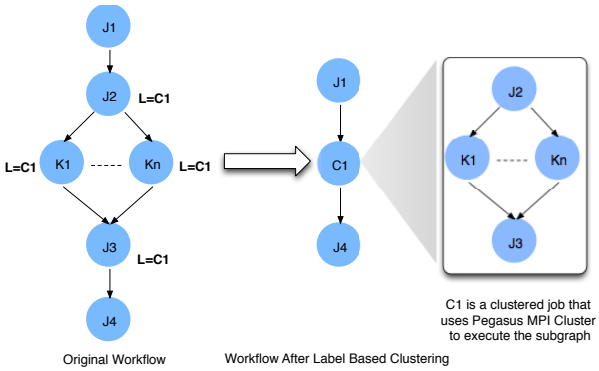

Figure 10: Example of label-based clustering.

Job Runtime-based horizontal clustering. Level-based clustering is suitable for grouping tasks with similar runtimes. However, when the runtime variance is high, this strategy may perform poorly, e.g. all smaller tasks get clustered together, while larger tasks are grouped into another clustered job. In such cases, runtime-based task clustering can be used to merge tasks into clustered jobs that have a smaller runtime variance. Two strategies have been recently added for balanced runtime clustering. The first one groups tasks into a job up to the upper bounded by a `maxruntime` parameter and employs a bin packing strategy to create clustered jobs that have similar expected runtimes [29]. The second strategy allows users to group tasks into a fixed number of clustered jobs and distributes tasks evenly (based on job runtime) across jobs, such that each clustered job takes approximately the same amount of time. The effectiveness of this balanced task clustering method has been evaluated in [65, 66]. Task runtime estimates are provided by the user or a runtime prediction function [63].

Label-based clustering. In label-based clustering, users label tasks to be clustered together. Tasks with the same label are merged by the Mapper into a single clustered job. This allows users to create clustered jobs or use a clustering technique that is specific to their workflows. If there is no label associated with a task, the task is not clustered and is executed as is. Thus, any clustering scheme can be implemented using an appropriate labeling program. Figure 10 illustrates how workflow is executed using Pegasus MPI Cluster as a remote workflow execution Engine. The Mapper clusters the sub graph with same label into a single clustered job in the executable workflow. This job appears as a single MPI job requesting n nodes and m cores on the

remote cluster.

4.4. Replica Selection and Addition of Data Transfer and Registration Nodes

One of the main features of Pegasus is its ability to manage the data for the scientist's workflows. The Mapper in this phase looks up the Replica Catalog to discover the location of input datasets and adds data stage-in jobs. The replica catalog can be a simple file that has the logical-to-physical filename mapping, it can be a database, or a fully distributed data registration system, such as the one used in LIGO [67]. The stage-in jobs transfer input data required by the workflow from the locations specified in the Replica Catalog to a directory on the data staging site associated with the job. Usually the data staging site is the shared filesystem on the execution site or in some cases can be a file server close to the execution site such as SRM server when executing on OSG, or S3 when executing workflows on the Amazon cloud. The data stage-in nodes can also *symlink* against existing datasets on the execution sites.

In the case where multiple locations are specified for the same input dataset in the Replica Catalog, the location from where to stage the data is selected using a *Replica Selector*. Pegasus supports a variety of replica selection strategies such as preferring datasets already present on the target execution site, preferring sites that have good bandwidth to the execution site, randomly selecting a replica, or using a user provided ranking for the URLs, which match a certain regular expression. Again, the scientist has the option of using Pegasus-provided algorithms or supplying their own.

Additionally, in this phase Pegasus also adds data stageout nodes to stage intermediate and output datasets to user specified storage location. The intermediate and final datasets may be registered back in the Replica Catalog to enable subsequent data discovery and reuse. Data registration is also represented explicitly by the addition of a registration node to the workflow.

It is worthwhile to mention that the added data stage-in or stage-out nodes to the workflow don't have a one-to-one mapping with the compute jobs. The Mapper by default creates a cluster of data stage-in or stage-out nodes per level of the workflow. This is done to ensure that for large workflows, the number of transfer jobs executing in parallel don't overwhelm the remote file servers by opening too many concurrent connections. In the Mapper, the clustering of the data transfer nodes is configurable and handled by a *Transfer Refiner*.

4.5. Addition of Directory Creation and File Removal Nodes

Once the Mapper has added the data transfer nodes to the workflow, it proceeds to add directory creation and file removal nodes to the underlying DAG. All the data transfer nodes added by the Mapper are associated with a unique directory on a data staging site. In the case where the workflow is executing on the shared filesystem of an execution site, this directory is created on the filesystem.

After the directory creation nodes are added, the Mapper can optionally invoke the cleanup refiner, which adds data cleanup nodes to the workflow. These nodes remove data from the workflow-specific directory when it is no longer required by the workflow. This is useful in reducing the peak storage requirements of the workflow and is often used in data-intensive workflows as explained below.

Scientific workflows [30, 68, 69, 70] often access large amounts of input data that needs to be made available to the individual jobs when they execute. As jobs execute new output datasets are created. Some of them may be intermediate, i.e. required only to be present during the workflow execution and are no longer required after the workflow finishes, while others are the output files that the scientist is interested in and need to be staged back from the remote clusters to a user defined location.

In case of large data-intensive workflows it is possible that the workflows will fail because of lack of sufficient disk space to store all the inputs, the intermediate files, and the outputs of the workflow. In our experience, this is often the case when scientists are trying to execute large workflows on local campus resources or shared computational grids such as OSG. One strategy to address this is to remove all the data staged and the outputs generated after the workflow finishes executing. However, cleaning up data after the workflow has completed may not be effective for data-intensive workflows as the peak storage used during the workflow may exceed the space available to the user at the data staging site.

The other strategy that we developed and implemented in Pegasus [71, 18] is to cleanup data from the data staging site as the workflow executes and data are no longer required. To achieve this the Mapper analyzes the underlying graph and file dependencies and adds data cleanup nodes to the workflow to "cleanup" data that is no needed downstream. For example, the input data that is initially staged in for the workflow can be safely deleted once the jobs that require this data have successfully completed. Similarly, we can delete the output data once the output has been successfully staged back to the user specified location. When we initially implemented the algorithm, it was aggressive and tried to cleanup data as soon as possible, i.e. in the worst case each compute job can have a cleanup job associated with it. However, this had the unintended side effect of increasing the workflow runtime, as the number of jobs that needed to be executed doubled. In order to address this, we implemented clustering of the cleanup jobs using level-based clustering. This allows us to control the number of cleanup jobs generated per level of the workflow. By default, the Mapper adds 2 cleanup jobs per level of the workflow. The cleanup capabilities of Pegasus as the workflow executes has greatly helped a genomics researcher at UCLA execute large scale and data intensive genome sequencing workflows [72] on the local campus resource, where the per-user scratch space available on the cluster was 15 TB. The peak storage requirement for a single hierarchical workflow in this domain was around 50 TB and the final outputs of the workflow were about 5-10 TB. As with other Pegasus components, the user can provide their own cleanup algorithm.

4.6. Remote Workflow Execution Engine Selection

At this stage, the Mapper has already identified the requisite data management jobs (stage-in, stage-out and cleanup) for the

workflow. The Mapper now selects a remote workflow engine to associate with the jobs. For tasks clustered into a single job, it associates a remote workflow engine capable of executing a collection of tasks on the remote node. For example, it may select Pegasus MPI Cluster or Pegasus Cluster to manage these tasks. Additionally, jobs mapped to a site without a shared filesystem get associated with the PegasusLite execution engine. It is important to note that the Mapper can associate multiple remote workflows engines with a single job. PegasusLite can be used to manage the data movement for the jobs to the worker node, while the actual execution of the clustered tasks can be managed by Pegasus Cluster or Pegasus MPI Cluster.

4.7. Code Generation

This is the last step of refinement process and at this point the Mapper writes out the executable workflow in a form understandable by the local workflow execution engines. Pegasus supports the following code generators:

HTCondor. This is the default code generator for Pegasus. This generator writes the executable workflow as a HTCondor DAG file and associated job submit files. The job submit files will include code targetting the remote execution engines, supporting workload delegation to the remote resources.

Shell. This code generator writes the executable workflow as a shell script that can be executed on the submit host.

The Mapper can easily support other DAG based workflow executors. For example, we have previously integrated Pegasus with the GridLab Resource Management System (GRMS) [73].

5. Runtime Optimizations

The majority of the optimizations of Pegasus workflows takes place in the mapping step, i.e. before the workflow has been submitted to HTCondor and started to run. The advantage of this is that the Mapper knows the full structure of the workflow and can make well-informed decisions based on that knowledge. On the other hand, there are times when knowledge is acquired at runtime, and runtime decisions need to be made. The general Pegasus solution for runtime optimizations is to use hierarchical workflows.

5.1. Interleaving Mapping and Execution

Figure 11 shows how Pegasus mapping/planning can be interleaved with the workflow execution. The top level workflow is composed of four tasks, where task `A3` is a sub-workflow, which is in turn composed of two other sub-workflows `B3` and `B4`. Each sub-workflow has an explicit workflow planning phase and a workflow execution phase. When the top level workflow (DAX `A`) is mapped by the Mapper, the sub-workflow jobs in the workflow are wrapped by Pegasus Mapper instances. The Pegasus Mapper is invoked for sub workflows only when the sub workflow nodes are executed by the DAGMan instance managing the workflow containing these nodes. For example, the mapping for the `A3` sub workflow node (that refers to DAX

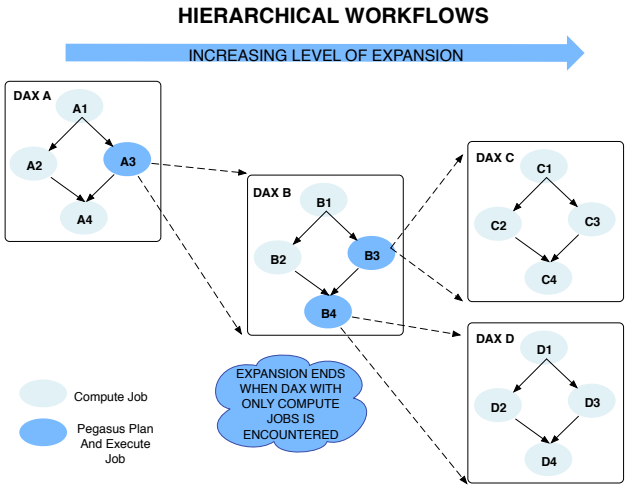

Figure 11: Hierarchical Workflows. Expansion ends when a DAX with only compute jobs is encountered.

`B`) only occurs when DAGMan is executing the workflow corresponding to DAX `A` and reaches the node `A3`. As evident in the Figure, our implementation covers arbitrary levels of expansion. The majority of workflows we see, use a single level of expansion, where only the top level of the workflow has sub workflow nodes (see the earthquake application workflow in Figure 16).

5.2. Just In Time Planning

In distributed environments, resource and data availability can change over time, resources can fail, data can be deleted, the network can go down, or resource policy has been changed. Even if a particular environment is changing slowly, the duration of the execution of the workflow components can be quite large and by the time a component finishes execution, the data locations may have changed as well as the availability of the resources. Choices made ahead of time even if still feasible may be poor.

When using hierarchical workflows, Pegasus only maps portions of the workflow at a time. In our example, jobs `A1`, `A2`, `A4` will be mapped for execution, but the sub-workflow contained in `A3` will be mapped only when `A1` finishes execution. Thus, a new instance of Pegasus will make just-in-time decisions for `A3`. This allows Pegasus to adapt the sub-workflows to other execution environments, if during execution of a long running workflow, additional resources come online, or previously available resources dissapear.

5.3. Dynamic Workflow Structures

In some cases, the workflow structure cannot be fully determined ahead of time, and hierarchical workflows can provide a solution, where part of the workflow needs to generated dynamically. A common occurance of this is in workflows, which contain a data discovery step, and what data is found will determine the structure of the reminding parts of the workflow. For example, in order to construct the workflow description for the

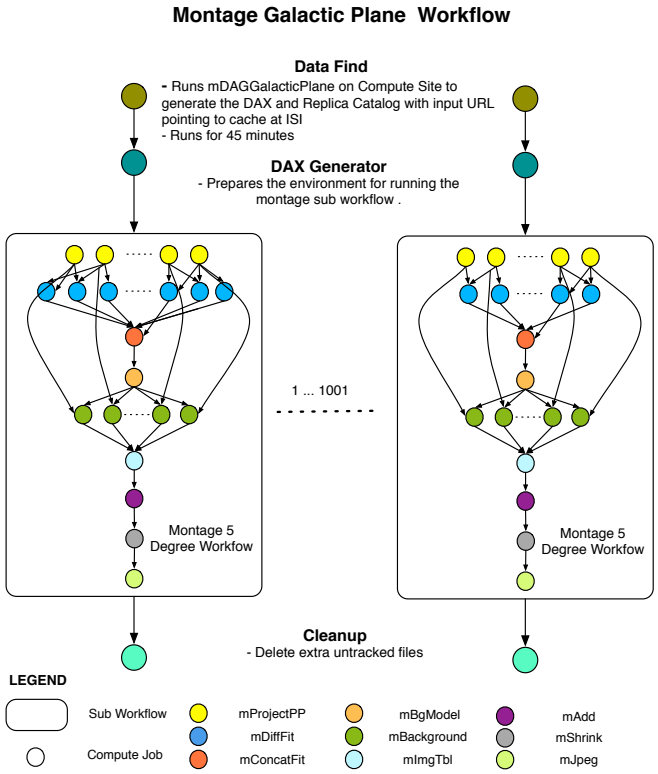

Figure 12: The Galactic Plane workflow, showing the top-level workflow with 1,001 data find tasks and sub-workflows.

Galactic Plane astronomy mosaicking workflow [74], a super-workflow of a set of 1,001 Montage [69] workflows is created. The top-level workflow queries a *datafind service* to get a list of input FITS images for the region of the sky the mosiaic is requested for. Once the top-level workflow has the list of input images, a Montage sub-workflow is created to process those images. Figure 12 shows the structure of the workflow.

6. Reliability

Even with just-in-time planning, when running on distributed infrastructures, failures are to be expected. In addition to infrastructure failures such as network or node failures, the applications themselves can be faulty. Pegasus dynamically handles failures at multiple levels of the workflow management system building upon reliability features of DAGMan and HTCondor. The various failure handling strategies employed are described below.

Job Retry. If a node in the workflow fails, then the corresponding job is automatically retried/resubmitted by HTCondor DAGMan. This is achieved by associating a job retry count with each job in the DAGMan file for the workflow. This automatic resubmit in case of failure allows us to automatically handle transient errors such as a job being scheduled on a faulty node in the remote cluster, or errors occurring because of job disconnects due to network errors.

Data Movement Reliability. As described earlier, low-level transfers are handled by the `pegasus-transfer` subsystem/command line tool. `pegasus-transfer` implements file transfer retries and it can vary the grouping of the given transfers, and the command line arguments given to protocol-specific command line tools. For example, when using *globus-url-copy* for a non-third party transfer (a transfer from a GridFTP to a local file for example), in the initial attempt `pegasus-transfer` will pass `-parallel 6` to `globus-url-copy` to try to use parallel transfers for better performance. The downside to parallel connections is that they can easily fail if there are firewalls in the way. If the initial transfer fails, `pegasus-transfer` will remove the `-parallel 6` argument for subsequent transfers and try the safer, but slower single connection transfer.

Failed Workflow Recovery/Rescue DAGs. If the number of failures for a job exceeds the set number of retries, then the job is marked as a fatal failure that leads the workflow to eventually fail. When a DAG fails, DAGMan writes out a rescue DAG that is similar to the original DAG but the nodes that succeeded are marked done. This allows the user to resubmit the workflow once the source of the original error has been resolved. The workflow will restart from the point of failure. This helps handle errors such as incorrectly compiled codes, faulty application executables, remote clusters being inaccessible either because of network links being down, or the compute cluster being unavailable because of maintenance or downtime.

Workflow Replanning. In case of workflow failures, users also have an option to replan the workflow and move the remaining computation to another resource. Workflow replanning leverages the feature of data reuse explained earlier, where the outputs are registered back in the data catalog as they are generated. The data reuse algorithm prunes the workflow (similar to what "*make*" does) based on existing output and intermediate files present in the data catalog. The intermediate files are shipped to the new cluster/location by the data stage-in nodes added by the planner during the replanning phase. Workflow replanning also helps in the case where a user made an error while generating the input workflow description such as wrong arguments for application codes, incorrect underlying DAG structure resulting in jobs being launched when not all its inputs are present, etc., that led to the workflow failing.

In hierarchical workflows, when a sub-workflow fails and it is automatically retried, the mapping step is automatically executed and a new mapping is performed.

7. Usability

Over the years, as Pegasus has matured and increased its user base, usability has become a priority for the project. Usability touches on various aspects of the software. In our case we want to be able to provide the user with an easy way of composing, monitoring and debugging workflows. In this section we focus on Pegasus' capabilities in these areas and the software engineering practises applied to reach our usability goals.

7.1. Workflow Composition

We have chosen to rely on the expressiveness and power of higher level programming languages to aid the user in describing their workflows. To achieve this, we have focused on providing simple, easy to use DAX APIs in major programming languages used by scientists such as Java, Python and Perl. We have taken great care to keep the format simple and resisted the urge to introduce complicated constructs.

While it can be argued that providing a graphical workflow composition interface would make it easier to create workflows in Pegasus, we believe that a GUI would make complex workflows more difficult to compose and maintain. Graphical workflow tools provide constructs such as foreach loops, which can be used to generate a large number of tasks, but if those constructs are not what the scientist needs, new constructs have to be defined, which usually includes writing custom code. Because Pegasus is just an API on top of well-established programming languages, the scientist can leverage the constructs in those programming languages across the whole workflow composition, not only in the complex areas as in the case of the graphical workflow composition interface.

The level of difficulty of workflow composition depends on how well the codes and executables of the tasks are defined. Scientific users often develop code that is structured to run in their local environments. They make assumptions about their environment, such as hard-coded paths for input and output datasets and where dependent executables are installed. This results in their code failing when run remotely as part of a scientific workflow. Based on our interactions with users over the years, we developed a series of guidelines for writing portable scientific code [29] that can be easily deployed and used with Pegasus. Though most of the guidelines are simple such as propagating *exitcode* from wrapper scripts, providing arguments for input and output file locations, etc., making them available to scientists allows them to write code that can be easily integrated into portable workflows.

7.2. Monitoring and Debugging

Pegasus provides users with the ability to track and monitor their workflows, without logging on to remote nodes, where the jobs ran or manually going through job and workflow logs. It collects and automatically processes information from three different distinct types of logs that provide the basis for the runtime provenance of the workflow and the of jobs:

Pegasus Mapper log. Because of the workflow optimizations the Mapper performs, there is often not a one to one mapping between the tasks and the jobs [75]. Thus, there needs to be information collected to help relate the executable workflow to the original workflow submitted by the user. The Mapper captures the information about the input abstract workflow, the jobs in the executable workflow and how the input tasks map to the jobs in the executable workflow. This information allows us to correlate user-provided tasks in the input workflow to the jobs that are launched to execute them. The Mapper logs this information to a local file in the workflow submit directory. With the detailed logs, Pegasus can generate the provenance of the results, referring to the original workflow. Pegasus can also provide the provenance of the mapping process itself, in case that needs to be inspected [8, 76].

Local Workflow Execution Engine Logs *(DAGMan log).* During the execution of a workflow, HTCondor DAGMan writes its log file (`dagman.out`) in near real-time. This file contains the status of each job within the workflow, as well as the pre-execute and post-execute scripts associated with it.

Job Logs *(Kickstart records).* Regardless of the remote execution engine, all the jobs that are executed through Pegasus are launched by the lightweight C executable `pegasus-kickstart` [57] that captures valuable runtime provenance information about the job. This information is logged by `pegasus-kickstart` as an XML record on its *stdout* and is brought back to the workflow submit directory by HTCondor as each job in the executable workflow completes. These job log files accumulate in the submit directory on the submit host and capture fine-grained execution statistics for each task that was executed. Some of this information includes:

- the arguments a task was launched with and the exitcode with which it exited;
- the start time and duration of the task;
- the hostname of the host on which the task ran and the directory in which it executed;
- the stdout and stderr of the task;
- the environment that was set up for the job on the remote node;
- the machine information about the node that the job ran on, such as architecture, operating system, number of cores on the node, available memory, etc.

Together, all this information contains the provenance for the derived data products. The data from the DAGMan workflow log and the raw kickstart logs from the running jobs are normalized and populated to a relational database by a monitoring daemon called *pegasus-monitord* [75]. It is launched automatically whenever a workflow starts execution and by default populates to SQLite database in the workflow submit directory. We support other SQL backends, such as MySQL, but we chose SQLite as our default as we did not want the setup of a fully featured database such as MySQL to be a prerequisite for running workflows through Pegasus. In the future, we plan to provide support for exporting the runtime provenance stored in our relational datastore, into a format conformant to the provenance community standard [77], provenance data model.

Automatic population of system logs into a database has enabled us to build a debugging tool called *pegasus-analyzer* that allows our users to diagnose failures. Its output contains a brief summary section, showing how many jobs have succeeded and how many have failed. For each failed job, it prints information showing its last known state, information from the kickstart record along with the location of its job description, output, and error files. It also displays any application `stdout` and `stderr` that was captured for the job.

Pegasus also includes a lightweight web dashboard that enables easy monitoring and online exploration of workflows based on an embedded web server written in Python. The dashboard can display all the workflows (running and completed) for a particular user. It allows users to browse their workflows. For each workflow, users can see a summary of the workflow run including runtime statistics (snapshot displayed in Figure 13) such as the number of jobs completed, failed, and workflow walltime, and can browse data collected for each job by the system. The dashboard can also generate different types of charts such as workflow gantt charts (snapshot displayed in Figure 14) illustrating the the progress of the workflow through time and pie charts depicting the distribution of jobs on the basis of count and runtime.

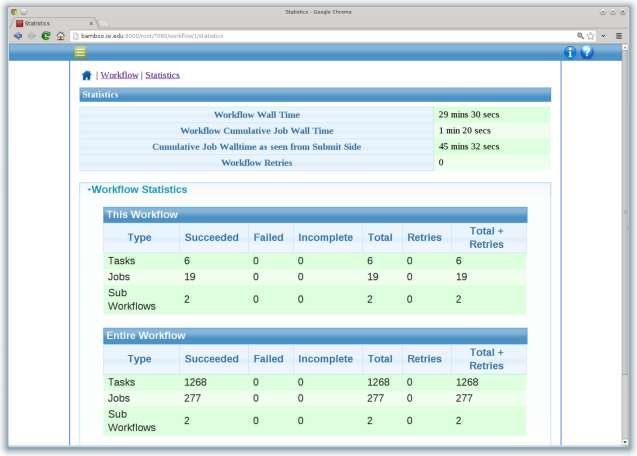

Figure 13: Workflow Statistics displayed in `pegasus-dashboard`.

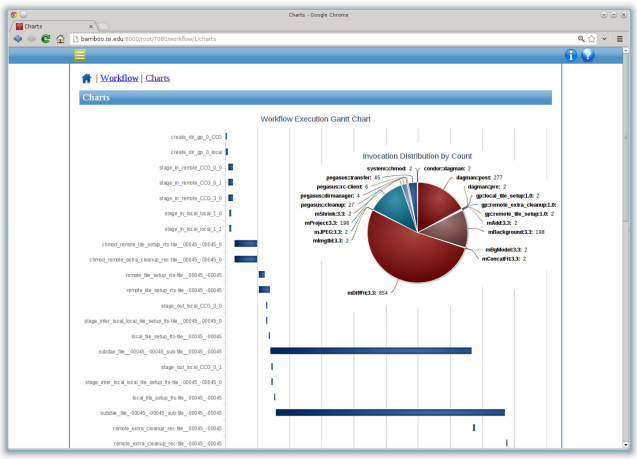

Figure 14: Workflow Gantt Chart and Job Distribution Pie Charts displayed in `pegasus-dashboard`.

7.3. Software Engineering Practices

Initially, Pegasus releases were built as binary distributions for various platforms on the NMI [78] build and test infrastructure and were made available for download on our website. While this worked for some of our users, we realized that even installing a binary distribution was a barrier to adoption for a lot of scientific communities. We needed to provide Pegasus releases as native RPM and DEB packages that could be included in various platform specific repositories to increase the adoption rate. This allows system administrators to easily deploy new versions of Pegasus on their systems. In order to make Pegasus available as a native package we had to move the software to the standard FHS layout. Pegasus releases are now available in the Debian repository and are shipped with Debian Squeeze.

Pegasus has the ability to execute workflows on a variety of infrastructures. We found that changes made for one particular infrastructure often broke compatibility with others. At other times newer versions of the underlying grid middleware, such as HTCondor and Globus, broke compatability at the Pegasus level. It is important for us to find such errors before our users do. In the past few years, we have setup an automated testing infrastructure based on Bamboo [79] that runs nightly unit tests, and end-to-end workflow tests that allow us to regularly test the whole system.

Another area that we have focussed on is improving the software documentation. We have worked on a detailed Pegasus user guide [29] that is regularly updated with each major and minor release of the software. The user guide, also includes a tutorial that helps get started with Pegasus. The tutorial is packaged as a virtual machine image with Pegasus and the dependent software pre-installed and configured. This enables new users of Pegasus to explore the capabilities Pegasus provides without worrying about the software setup.

7.4. Incorporation of User Feedback

User feedback has been the driving force for many usability improvements. As a result, we have streamlined the configuration process, we introduced a single `pegasus-transfer` client that automatically sets up the correct transfer client to stage the data at runtime. We have also focused on providing easier setups of the various catalogs that Pegasus requries. We also provided support for directory-based replica catalogs, where users can point to the input directory on the submit host containing the input datasets, rather than configuring a replica catalog with the input file mappings. The Pegasus 4.x series has improved support for automatically deploying Pegasus auxillary executables as part of the executable workflows. This enables our users to easily run workflows in non-shared filesystem deployments such as Open Science Grid, without worrying about installing Pegasus auxillary executables on the worker nodes.

8. Application Study: CyberShake

Much effort has been put into making workflows executed through Pegasus scale well. Individual workflows can have hundreds of thousands of individual tasks [19]. In this section we describe one of the large-scale scientific applications that uses Pegasus.

As part of its research program of earthquake system science, the Southern California Earthquake Center (SCEC) has developed CyberShake, a high-performance computing software

platform that uses 3D waveform modeling to calculate physics-based probabilistic seismic hazard analysis (PSHA) estimates for populated areas of California. PSHA provides a technique for estimating the probability that the earthquake ground motions at a location of interest will exceed some intensity measure, such as peak ground velocity or spectral acceleration, over a given time period. PSHA estimates are useful for civic planners, building engineers, and insurance agencies. The fundamental calculation that the CyberShake software platform can perform is a PSHA hazard curve calculation. A CyberShake calculation produces a PSHA hazard curve for a single location of interest. CyberShake hazard curves from multiple locations can be combined to produce a CyberShake hazard map (see Figure 15), quantifying the seismic hazard over a geographic region.

A CyberShake hazard curve computation can be divided into two phases. In the first phase, a 3D mesh of approximately 1.2 billion elements is constructed and populated with seismic velocity data. This mesh is then used in a pair of wave propagation simulations that calculates and outputs strain Green tensors (SGTs). The SGT simulations use parallel wave propagation codes and typically run on 4,000 processors. In the second phase, individual contributions from over 400,000 different earthquakes are calculated using the SGTs, then these hazard contributions are aggregated to determine the overall seismic hazard. These second phase calculations are loosely coupled, short-running serial tasks. To produce a hazard map for Southern California, over 100 million of these tasks must be executed. The extensive heterogeneous computational requirements and large numbers of high-throughput tasks necessitate a high degree of flexibility and automation; as a result, SCEC utilizes Pegasus-WMS workflows for execution.

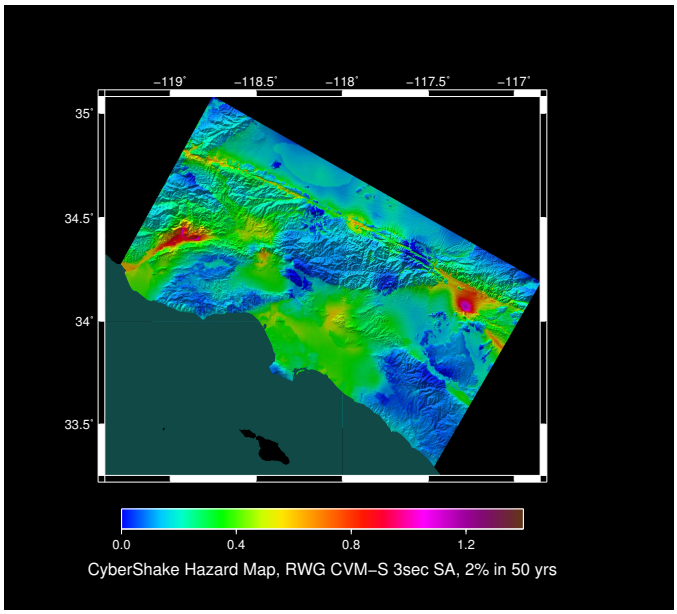

Figure 15: A Cybershake Hazard Map for the Southern California region, representing the ground motion in terms of g (acceleration due to gravity) which is exceeded with 2% probability in 50 years.

CyberShake is represented in Pegasus as a set of hierarchical workflows (Figure 16). Initially, a job is run which invokes RunManager, an executable that updates a database at SCEC which keeps track of the status of hazard calculations for individual locations. The first phase calculations, consisting of large, parallel tasks, are represented as a single workflow, and the second-phase tasks are split up into multiple workflows, to reduce the memory requirements of planning. Both CyberShake computational phases are typically executed on large, remote, high-performance computing resources. Finally, Pegasus transfers the output data back to SCEC storage, where a database is populated with intensity measure information. This intensity measure database is used to construct the target seismic hazard data products, such as hazard curves or a hazard map. Lastly, RunManager is invoked again to record that the calculation is complete.

Originally, the first and second phases were described as discrete workflows for maximum flexibility, so the two phases could be executed on different computational resources. With the development of computational systems, which can support both types of calculations—large parallel tasks as well as quick, serial ones—CyberShake now includes all phases of the calculation in an outer-level workflow, with the phase one, phase two, and database workflows as sub-workflows. Therefore, all the jobs and workflows required to compute PSHA results for a single location are included in the top-level workflow.

Table 1 shows the comparison of CyberShake studies done over the past six years. The table shows how many wall clock hours of computing a particular set of workflows took, the number of geographic sites in the study, the computing systems used, the number of core hours that computations took, the maximum number of cores used at one time. The table also indirectly shows the clustering used: the number of tasks in the original workflow and the number of jobs generated by Pegasus. Finally, the table specifies the number of files and their corresponding total size that was produced during the study. In some cases, the data was not fully captured and is indicated at N/A.

Since the CyberShake 2.2 study, the CyberShake software platform uses PMC to manage the execution of high-throughput tasks on petascale resources, and PMC's I/O forwarding feature to reduce the number of output files by a factor of 60. Using this approach, in Spring 2013 CyberShake Study 13.4 was performed. It calculated PSHA results for 1,144 locations in Southern California, corresponding to 1,144 top-level workflows. Approximately 470 million tasks were executed on TACC's Stampede system over 739 hours, averaging a throughput of 636,000 tasks per hour. PMC's clustering resulted in only about 22,000 jobs being submitted to the Stampede batch scheduler. Pegasus managed 45 TB of data, with 12 TB being staged back for archiving on SCEC's servers. The last, CyberShake study (CySh#14.2) used both CPUs and GPUs for the calculations.

CyberShake workflows are important science calculations that must be re-run frequently with improved input parameters and data. A CyberShake study encompasses hundreds or thousands of independent workflows. Overall these workflows

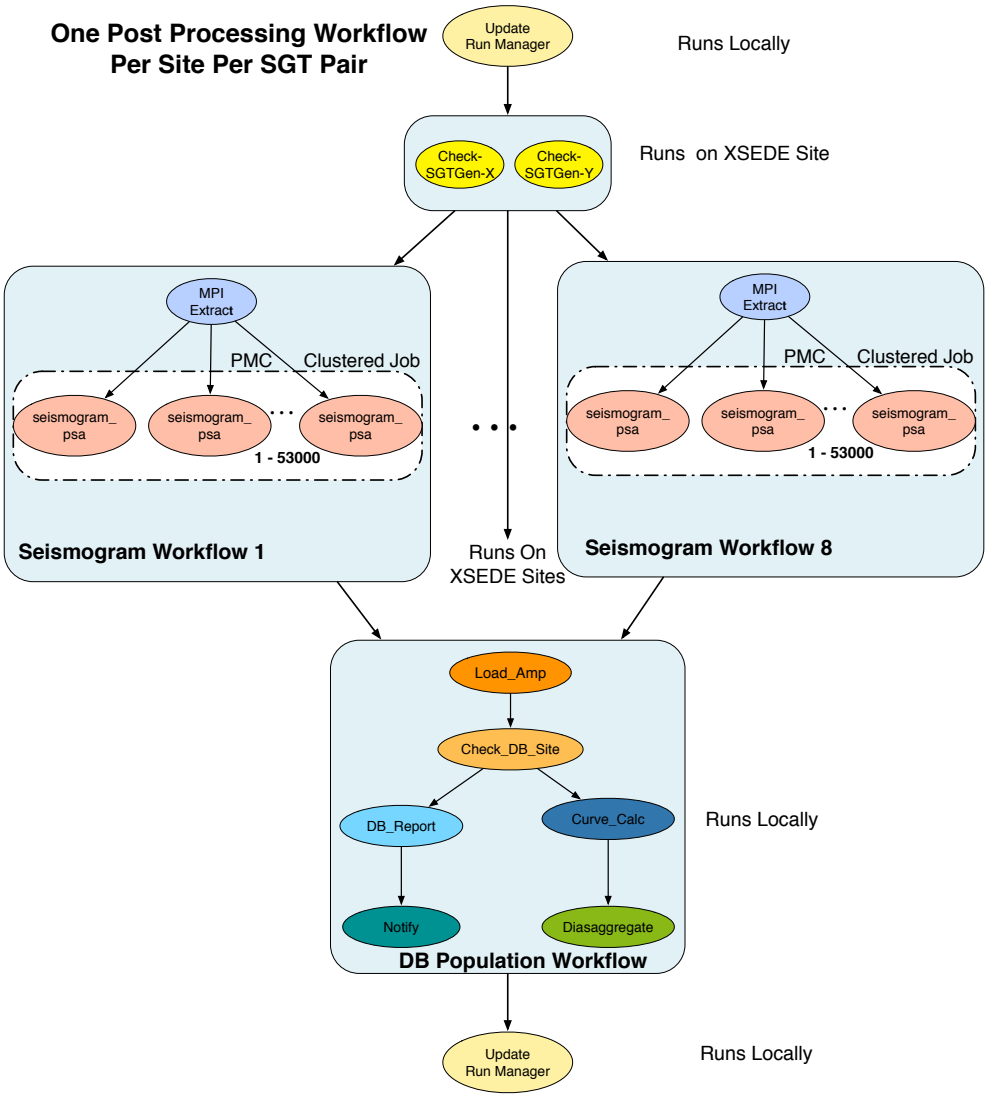

Figure 16: Cybershake Hierarchical Workflow.

Study Name	Year	Geo-graphic Sites	Tasks (million)	Machine	Make-span (hours)	Max Cores	Core Hours (million)	Jobs	Files Produced (million)	Data
CySh #0.5	2008	9	7.5	NCSA Mercury	528	800	0.44	22 (glideins)	7.5	90GB
CySh #1.0	2009	223	192	TACC Ranger	1,314	14,544	6.9	3,952 (Ranger) 3.9 million (Condor)	190	10.6TB
CySh #1.4	2011-2012	90	112	USC HPCC TACC Ranger	4,824	N/A	N/A	N/A	112	4TB
CySh #2.2	2012-2013	217	N/A	USC HPCC NICS Kraken	2,976	N/A	N/A	N/A	N/A	N/A
CySh #13.4	2013	1,132	470 N/A	TACC Stampede NCSA Blue Waters	716	17,600 128,000	1.4 10.8	21,912 N/A	16 2,264	12.3TB 43TB
CySh #14.2	2014	1,144	99.9	NCSA Blue Waters	342	295,040	15.8	29,796 (Blue Waters) 100,707 (Condor)	16	57TB

Table 1: Comparison of CyberShake Studies

could contain up to half a billion tasks, making it impractical to consolidate the workload into a single workflow. Our workflow experiences so far suggest that it would simplify the management of CyberShake studies if workflow tools supported the concept of an ensemble, or a group of workflows, which make up an application. If workflows could be executed and monitored as a group, the benefits of automated management of workflows could be extended to the entire ensemble of workflows without requiring external tools (e.g. RunManager).

9. Related Work

Many different workflow systems have been developed for executing scientific workflows on distributed infrastructures [60, 80]. [68, 80] provide an extensive overview of many of the most influential workflow management systems and their characteristics. Here, we mention some of them.

Taverna [81] is a suite of tools used to design and execute scientific workflows. A Taverna workflow specification is compiled into a multi-threaded object model, where processors are represented by objects, and data transfers from the output port of one processor to the input port of a downstream processor are realized using local method invocations between processor objects. One or more activities are associated to each processor. These activities may consist of entire sub-workflows, in addition to executable software components. Galaxy [82] is an open, web-based platform for data-intensive bioinformatics research. It provides a number of build-in tools, including common visualization tools and supports access to a number of bioinformatics data sets. The system automatically tracks and manages data provenance and provides support for capturing the context and intent of computational methods. Tavaxy [83] is a pattern-based workflow system for the bioinformatics domain. The system integrates existing Taverna and Galaxy workflows in a single environment, and supports the use of cloud computing capabilities. These systems target a specific user community and do not focus on issues of portability, performance, and scalability as Pegasus does.

DIET [84] is a workflow system that can manage workflow-based DAGs in two modes: one in which it defines a complete scheduling of the workflow (ordering and mapping), and one in which it defines only an ordering for the workflow execution. Mapping is then done in the next step by the client, using a master agent to find the server where the workflow services should be run. Askalon [85] provides a suite of middleware services that support the execution of scientific workflows on distributed infrastructures. Askalon workflows are written in an XML-based workflow language that supports many different looping and branching structures. These workflows are interpreted to coordinate data flow and task execution on distributed infrastructures including grids, clusters and clouds. Moteur [86] is a workflow engine that uses a built-in language to describe workflows that targets the coherent integration of: 1) a data-driven approach to achieve transparent parallelism; 2) arrays manipulation to enable data parallel application in an expressive and compact framework; 3) conditional and loop control structures to improve expressiveness; and 4) asynchronous execution to optimize execution on a distributed infrastructure. These systems provide conditional and loop functionalities that are not provided by Pegasus, however the complexity of their workflow language can limit scalability and adoption.

Kepler [87] is a graphical system for designing, executing, reusing, evolving, archiving, and sharing scientific workflows. In Kepler, workflows are created by connecting a series of workflow components called Actors, through which data are processed. Each Actor has several Ports through which input and output Tokens containing data and data references are sent and received. Each workflow has a Director that determines the model of computation used by the workflow, and Kepler supports several Directors with different execution semantics, including Synchronous Data Flow and Process Network directors. Triana [88] is a graphical problem solving environment that enables users to develop and execute data flow workflows. It interfaces with a variety of different middleware systems for data transfer and job execution, including grids and web services. Pegasus does not provide a graphical interface, instead it provides flexible APIs that support complex workflow descriptions. It also provides resource-independent workflow definition.

The Nimrod toolkit [89, 90] is a specialized parametric modeling system that provides a declarative parametric modeling language to express parametric experiments, and automates the formulation, execution, monitoring, and verification of the results from multiple individual experiments using many-task computing (MTC). Nimrod/K [91] is a workflow engine for the Nimrod toolkit built on the Kepler workflow engine. The engine has special Kepler actors that enables enumeration, fractional factorial design, and optimization methods through an API. Although the toolkit is flexible and workflows can be defined and controlled in a programmatic way, it does not provide task clustering, nor all the sophisticated data layouts and the portability that Pegasus supports.

Makeflow [92] represents workflows using a file format very similar to the one used by the Make tool [58]. In Makeflow the rules specify what output data is generated when a workflow component is executed with a given set of input data and parameters. Like Make and Pegasus, Makeflow is able to reduce the workflow if output data already exists. It also maintains a transaction log to recover from failures. Makeflow supports many different execution environments, including local execution, HTCondor, SGE, Hadoop and others. Makeflow, unlike Pegasus, provides only simple shared file system data management capabilities.

Wings [23] is a workflow system that uses semantic reasoning to generate Pegasus workflows based on ontologies that describe workflow templates, data types, and components.

Among these workflow systems, Pegasus distinguishes itself with combination of features such as portability across a wide range of infrastructures, scalability (hundreds of millions of tasks), sophisticated data management capabilities, comprehensive monitoring, complex workflow restructuring, etc.

In addition to workflow systems that target HPC and HTC resources, some systems such as Tez [93] and Ozzie [94] are designed for MapReduce types of models and environments.

10. Conclusions and Future Directions

To-date Pegasus has been used in a number of scientific domains. We enable SCEC to scale up to almost a million tasks per workflow and provide the bridge to run those workflows on national high-end computing resources. We provide LIGO with technologies to analyze large amounts of gravitational wave data. As a result LIGO passed a major scientific milestone, the blind injection test [95, 96]. We supported astronomers in processing Kepler mission data sets [97, 98]. And, we helped individual researchers advance their science [99, 100, 101]. In the coming years, we hope to help to automate more of the scientific processes, potentially exploring the automation from data collection to the final analysis and result sharing.

Pegasus is an ever evolving system. As scientists are developing new, more complex workflows, and as the computational infrastructure is changing, Pegasus is challenged to handle larger workflows, manage data more efficiently, recover from failures better, support more platforms, report more statistics, and automate more work. We are currently working on a service that will help users manage ensembles of workflows more efficiently, and make it easier to integrate Pegasus into other user-facing infrastructures, such as science gateways. We are also investigating many new data management techniques, including techniques to help Pegasus track and discover data at runtime, support disk usage guarantees in storage-constrained environments, and improve data locality on clusters. Finally, we are improving support for new platforms such as clouds and exascale machines by developing task resource estimation and provisioning capabilities [63], and improving the efficiency and scalability of remote workflow execution.

Acknowledgements

Pegasus is funded by The National Science Foundation under the ACI SDCI program grant #0722019 and ACI SI2-SSI program grant #1148515. Pegasus has been in development since 2001 and has benefited greatly from the expertise and efforts of people who worked on it over the years. We would like to especially thank Gaurang Mehta, Mei-Hui Su, Jens-S. Vöckler, Fabio Silva, Gurmeet Singh, Prasanth Thomas and Arun Ramakrishnan for their efforts and contributions to Pegasus. We would also like to extend our gratitude to all the members of our user community who have used Pegasus over the years and provided valuable feedback, especially Duncan Brown, Scott Koranda, Kent Blackburn, Yu Huang, Nirav Merchant, Jonathan Livny, and Bruce Berriman.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

This research was done using resources provided by the Open Science Grid, which is supported by the National Science Foundation and the U.S. Department of Energy's Office of Science.

The Cybershake workflows research was supported by the Southern California Earthquake Center. SCEC is funded by NSF Cooperative Agreement EAR-1033462 and USGS Cooperative Agreement G12AC20038. The SCEC contribution number for this paper is 1911.

[1] LIGO Scientific Collaboration.
URL http://ligo.org

[2] Southern California Earthquake Center.
URL http://scec.org

[3] National Virtual Observatory.
URL http://us-vo.org

[4] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, Workflow management in griphyn, in: J. Nabrzyski, J. M. Schopf, J. Weglarz (Eds.), Grid Resource Management, Kluwer Academic Publishers, Norwell, MA, USA, 2004, pp. 99–116.

[5] V. Nefedova, R. Jacob, I. Foster, Z. Liu, Y. Liu, E. Deelman, G. Mehta, M.-H. Su, K. Vahi, Automating climate science: Large ensemble simulations on the teragrid with the griphyn virtual data system, in: 2nd IEEE International Conference on e-Science and Grid Computing, E-SCIENCE 06, 2006. doi:10.1109/E-SCIENCE.2006.30.

[6] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, S. Koranda, Griphyn and ligo, building a virtual data grid for gravitational wave scientists, in: 11th IEEE International Symposium on High Performance Distributed Computing, HPDC 02, 2002.

[7] D. Gunter, E. Deelman, T. Samak, C. Brooks, M. Goode, G. Juve, G. Mehta, P. Moraes, F. Silva, M. Swany, K. Vahi, Online workflow management and performance analysis with stampede, in: Network and Service Management (CNSM), 2011 7th International Conference on, 2011, pp. 1–10.

[8] S. Miles, P. Groth, E. Deelman, K. Vahi, G. Mehta, L. Moreau, Provenance: The bridge between experiments and data, Computing in Science Engineering 10 (3) (2008) 38–46. doi:10.1109/MCSE.2008.82.

[9] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, M. Wolfe, Dependence graphs and compiler optimizations, in: 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 81, 1981. doi:10.1145/567532.567555.

[10] A. Gerasoulis, T. Yang, On the granularity and clustering of directed acyclic task graphs, IEEE Transactions on Parallel and Distributed Systems 4 (6) (1993) 686–701. doi:10.1109/71.242154.

[11] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Computing Surveys 31 (4) (1999) 406–471. doi:10.1145/344588.344618.

[12] J. Kurzak, J. Dongarra, Fully dynamic scheduler for numerical computing on multicore processors, LAPACK Working Note 220 - 2009.

[13] J. Gray, A. Reuter, Transaction processing: concepts and techniques, Morgan Kaufmann Pub, 1993.

[14] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, J. B. Rothnie, Jr., Query processing in a system for distributed databases (sdd-1), ACM Transactions on Database Systems 6 (4) (1981) 602–625. doi:10.1145/319628.319650.

[15] D. West, Introduction to graph theory, Prentice Hall Upper Saddle River, NJ, 2001.

[16] R. Tarjan, Data structures and network algorithms, Society for industrial and Applied Mathematics, 1998.

[17] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, M. Samidi, Scheduling data-intensive workflows onto storage-constrained distributed resources, in: IEEE International Symposium on Cluster Computing and the Grid (CCGrid), 2007. doi:10.1109/CCGRID.2007.101.

[18] G. Singh, K. Vahi, A. Ramakrishnan, G. Mehta, E. Deelman, H. Zhao, R. Sakellariou, K. Blackburn, D. Brown, S. Fairhurst, D. Meyers, G. B. Berriman, J. Good, D. S. Katz, Optimizing workflow data footprint, Scientific Programming 15 (4) (2007) 249–268.

[19] S. Callaghan, E. Deelman, D. Gunter, G. Juve, P. Maechling, C. Brooks, K. Vahi, K. Milner, R. Graves, E. Field, D. Okaya, T. Jordan, Scaling up workflow-based applications, Journal of Computer and System Sciences 76 (6) (2010) 428–446. doi:10.1016/j.jcss.2009.11.005.

[20] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[21] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Kon-

winski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58. doi:10.1145/1721654.1721672.

[22] M. McLennan, S. Clark, F. McKenna, E. Deelman, M. Rynge, K. Vahi, D. Kearney, C. Song, Bringing scientific workflow to the masses via pegasus and hubzero, in: International Workshop on Science Gateways, 2013.

[23] Y. Gil, V. Ratnakar, J. Kim, P. A. González-Calero, P. T. Groth, J. Moody, E. Deelman, Wings: Intelligent workflow-based design of computational experiments, IEEE Intelligent Systems 26 (1) (2011) 62–72.

[24] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, A. Slominski, A. Douma, S. Perera, S. Weerawarana, Apache airavata: A framework for distributed applications and computational workflows, in: Workshop on Gateway Computing Environments, GCE '11, 2011. doi:10.1145/2110486.2110490.

[25] Steven Cox, GRAYSON Git/README, https://github.com/stevencox/grayson.

[26] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: The condor experience, Concurrency and Computation: Practice and Experience 17 (2-4) (2005) 323–356. doi:10.1002/cpe.v17:2/4.

[27] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: Enabling scalable virtual organizations, International Journal of High Performance Computing Applications 15 (3) (2001) 200–222. doi:10.1177/109434200101500302.

[28] Amazon.com, Inc., Elastic Compute Cloud (EC2), http://aws.amazon.com/ec2.

[29] Pegasus 4.3 user guide, http://pegasus.isi.edu/wms/docs/4.3/.

[30] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, K. Vahi, Cybershake: A physics-based seismic hazard model for southern california, Pure and Applied Geophysics 168 (3-4) (2011) 367–381. doi:10.1007/s00024-010-0161-6.

[31] Sax xerces java parser.
URL http://xerces.apache.org/xerces2-j/

[32] K. Czajkowski, I. T. Foster, N. T. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, A resource management architecture for metacomputing systems, in: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, 1998.

[33] P. Andreetto, S. A. Borgia, A. Dorigo, A. Gianelle, M. Marzolla, M. Mordacchini, M. Sgaravatto, S. Andreozzi, M. Cecchi, V. Ciaschini, T. Ferrari, F. Giacomini, R. Lops, E. Ronchieri, G. Fiorentino, V. Martelli, M. Mezzadri, E. Molinari, F. Prelz, CREAM: A simple, grid-accessible, job management system for local computational resources, in: Conference on Computing in High Energy Physics (CHEP), 2006.

[34] Simple Linux Utility for Resource Management.
URL http://slurm.schedmd.com/

[35] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, D. Tweten, Portable batch system: External reference specification, in: Technical report, MRJ Technology Solutions, Vol. 5, 1999.

[36] IBM Platform Computing Template: LSF.
URL http://www.platform.com/Products/platform-lsf

[37] Oracle Grid Engine.
URL http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html

[38] D. Weitzel, I. Sfiligoi, B. Bockelman, F. Wuerthwein, D. Fraser, D. Swanson, Accessing opportunistic resources with bosco, in: Computing in High Energy and Nuclear Physics, 2013.

[39] Extreme science and engineering discovery environment (xsede).
URL http://www.xsede.org

[40] European grid infrastructure (egi).
URL http://www.egi.eu

[41] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, et al., The open science grid, in: Journal of Physics: Conference Series, Vol. 78, 2007, p. 012057.

[42] I. Sfiligoi, glideinwms—a generic pilot-based workload management system, Journal of Physics: Conference Series 119 (6) (2008) 062044.
URL http://stacks.iop.org/1742-6596/119/i=6/a=062044

[43] Amazon.com, Inc., Amazon Web Services (AWS), http://aws.amazon.com.

[44] Futuregrid.
URL https://www.futuregrid.org/

[45] Openstack.
URL https://www.openstack.org/

[46] K. Vahi, M. Rynge, G. Juve, R. Mayani, E. Deelman, Rethinking data management for big data scientific workflows, in: IEEE International Conference on Big Data, 2013. doi:10.1109/BigData.2013.6691724.

[47] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, P. J. Maechling, Enabling large-scale scientific workflows on petascale resources using mpi master/worker, in: 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond, XSEDE '12, 2012, pp. 49:1–49:8. doi:10.1145/2335755.2335846.

[48] Kraken.
URL http://www.nics.tennessee.edu/computing-resources/kraken

[49] Blue Waters: Sustained Petascale Computing.
URL https://bluewaters.ncsa.illinois.edu/

[50] M. Rynge, G. Juve, G. Mehta, E. Deelman, G. Berriman, K. Larson, B. Holzman, S. Callaghan, I. Sfiligoi, F. Wurthwein, Experiences using glideinwms and the corral frontend across cyberinfrastructures, in: E-Science (e-Science), 2011 IEEE 7th International Conference on, 2011, pp. 311–318. doi:10.1109/eScience.2011.50.

[51] A. Rajasekar, R. Moore, C.-Y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, B. Zhu, iRODS primer: Integrated rule-oriented data system, Synthesis Lectures on Information Concepts, Retrieval, and Services 2 (1) (2010) 1–143. doi:10.2200/S00233ED1V01Y200912ICR012.
URL http://dblp.uni-trier.de/rec/bibtex/series/synthesis/2010Rajasekar.xml

[52] Amazon simple storage service.
URL http://aws.amazon.com/s3/

[53] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext transfer protocol – http/1.1 (1999).

[54] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster, The globus striped GridFTP framework and server, SC '05, IEEE Computer Society, Washington, DC, USA, 2005, p. 54–. doi:10.1109/SC.2005.72.
URL http://dx.doi.org/10.1109/SC.2005.72

[55] A. Lana, B. 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).

[56] T. Ylonen, C. Lonvick, Rfc 4254 - the secure shell (ssh) connection protocol (2006).

[57] J.-S. Vöckler, G. Mehta, Y. Zhao, E. Deelman, M. Wilde, Kickstarting remote applications, in: 2nd International Workshop on Grid Computing Environments, 2006.

[58] Gnu make, http://www.gnu.org/software/make.

[59] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, Parallel and Distributed Systems, IEEE Transactions on 13 (3) (2002) 260–274. doi:10.1109/71.993206.

[60] I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer London, 2007. doi:10.1007/978-1-84628-757-2_4.

[61] W. Chen, E. Deelman, Workflow overhead analysis and optimizations, in: 6th Workshop on Workflows in Support of Large-scale Science, WORKS '11, 2011. doi:10.1145/2110497.2110500.

[62] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Generation Computer Systems 29 (3) (2013) 682–692. doi:10.1016/j.future.2012.08.015.

[63] R. Ferreira da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, M. Livny, Toward fine-grained online task characteristics estimation in scientific workflows, in: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science, WORKS '13, 2013, pp. 58–67. doi:10.1145/2534248.2534254.

[64] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, G. Mehta, Workflow task clustering for best effort systems with

pegasus, in: 15th ACM Mardi Gras Conference, 2008. doi:10.1145/1341811.1341822.

[65] W. Chen, R. Ferreira da Silva, E. Deelman, R. Sakellariou, Balanced task clustering in scientific workflows, in: 2013 IEEE 9th International Conference on eScience, eScience'13, 2013, pp. 188–195. doi:10.1109/eScience.2013.40.

[66] W. Chen, R. Ferreira da Silva, E. Deelman, R. Sakellariou, Using imbalance metrics to optimize task clustering in scientific workflow executions, Future Generation Computer Systems (2014) in press.

[67] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, B. Moe, Wide area data replication for scientific collaborations, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, GRID '05, 2005, pp. 1–8. doi:10.1109/GRID.2005.1542717.

[68] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, Journal of Grid Computing 3 (3-4) (2005) 171–200. doi:10.1007/s10723-005-9010-8.

[69] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M. Su, T. A. Prince, R. Williams, Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking, International Journal of Computational Science and Engineering 4 (2) (2009) 73–87. doi:10.1504/IJCSE.2009.026999.

[70] S. Babak, R. Biswas, P. Brady, D. Brown, K. Cannon, et al., Searching for gravitational waves from binary coalescence, Physical Review D D87 (2013) 024033. doi:10.1103/PhysRevD.87.024033.

[71] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, M. Samidi, Scheduling data-intensive workflows onto storage-constrained distributed resources, in: 7th IEEE International Symposium on Cluster Computing and the Grid, 2007. doi:10.1109/CCGRID.2007.101.

[72] Y. Huang, S. Service, V. Ramensky, C. Schmitt, N. Tran, A. Jasinska, J. Wasserscheid, N. Juretic, B. Pasaniuc, R. Wilson, W. Warren, G. Weinstock, K. Dewar, N. Freimer, A varying-depth sequencing strategy and variant-calling framework for complex pedigrees, in preparation.

[73] GridLab Resource Management System, http://www.gridlab.org/WorkPackages/wp-9.
URL http://www.gridlab.org/WorkPackages/wp-9

[74] M. Rynge, G. Juve, J. Kinney, J. Good, B. Berriman, A. Merrihew, E. Deelman, Producing an infrared multiwavelength galactic plane atlas using montage, pegasus and amazon web services, in: 23rd Annual Astronomical Data Analysis Software and Systems (ADASS) Conference, 2013.

[75] K. Vahi, I. Harvey, T. Samak, D. Gunter, K. Evans, D. Rogers, I. Taylor, M. Goode, F. Silva, E. Al-Shakarchi, G. Mehta, E. Deelman, A. Jones, A case study into using common real-time workflow monitoring infrastructure for scientific workflows, Journal of Grid Computing 11 (3) (2013) 381–406. doi:10.1007/s10723-013-9265-4.

[76] S. Miles, E. Deelman, P. Groth, K. Vahi, G. Mehta, L. Moreau, Connecting scientific data to scientific experiments with provenance, in: e-Science and Grid Computing, IEEE International Conference on, 2007, pp. 179–186. doi:10.1109/E-SCIENCE.2007.22.

[77] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. T. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. G. Stephan, J. V. den Bussche, The open provenance model core specification (v1.1)., 2011, pp. 743–756.

[78] A. Pavlo, P. Couvares, R. Gietzel, A. Karp, I. D. Alderman, M. Livny, C. Bacon, The NMI build & test laboratory: Continuous integration framework for distributed computing software, in: 20th Conference on Large Installation System Administration (LISA), 2006.

[79] Atlassian, Continuous integration & build server - bamboo, https://www.atlassian.com/software/bamboo.

[80] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: An overview of workflow system features and capabilities, Future Generation Computer Systems 25 (5) (2009) 528 – 540. doi:http://dx.doi.org/10.1016/j.future.2008.06.012.

[81] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, J. Bhagat, K. Belhajjame, F. Bacall, A. Hardisty, A. Nieva de la Hidalga, M. P. Balcazar Vargas, S. Sufi, C. Goble, The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud, Nucleic Acids Research 41 (W1) (2013) W557–W561. doi:10.1093/nar/gkt328.

[82] J. Goecks, A. Nekrutenko, J. Taylor, et al., Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences, Genome Biol 11 (8 - 2010).

[83] M. Abouelhoda, S. Issa, M. Ghanem, Tavaxy: Integrating taverna and galaxy workflows with cloud computing support, BMC Bioinformatics 13 (1) (2012) 77. doi:10.1186/1471-2105-13-77.

[84] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, L. Philippe, M. Quinson, F. Suter, A scalable approach to network enabled servers (research note), in: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002.

[85] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek, Askalon: A grid application development and computing environment, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, 2005. doi:10.1109/GRID.2005.1542733.

[86] T. Glatard, J. Montagnat, D. Lingrand, X. Pennec, Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR, International Journal of High Performance Computing Applications (IJHPCA) 22 (3) (2008) 347–360.

[87] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, S. Mock, Kepler: an extensible system for design and execution of scientific workflows, in: Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on, 2004. doi:10.1109/SSDM.2004.1311241.

[88] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: Architecture and applications, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer London, 2007, pp. 320–339. doi:10.1007/978-1-84628-757-2_20.

[89] D. Abramson, J. Giddy, L. Kotler, High performance parametric modeling with nimrod/g: Killer application for the global grid?, in: Parallel and Distributed Processing Symposium, International, IEEE Computer Society, 2000, pp. 520–520.

[90] D. Abramson, B. Bethwaite, C. Enticott, S. Garic, T. Peachey, Parameter exploration in science and engineering using many-task computing, Special issue of IEEE Transactions on Parallel and Distributed Systems on Many-Task Computing 22 (2011) 960 – 973.

[91] D. Abramson, C. Enticott, I. Altinas, Nimrod/k: towards massively parallel dynamic grid workflows, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, 2008, p. 24.

[92] M. Albrecht, P. Donnelly, P. Bui, D. Thain, Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids, in: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET), 2012.

[93] Apache Tez, http://tez.apache.org.
URL http://tez.apache.org

[94] Apache Ozzie, http://oozie.apache.org.
URL http://oozie.apache.org

[95] D. Brown, E. Deelman, Looking for gravitational waves: A computing perspective, in: International Science Grid This Week, 2011.

[96] L. S. Collaboration, "blind injection" stress-tests ligo and virgo's search for gravitational waves! (2011).
URL http://www.ligo.org/news.php

[97] The kepler mission (2011).
URL http://kepler.nasa.gov

[98] G. B. Berriman, E. Deelman, G. Juve, M. Regelson, P. Plavchan, The application of cloud computing to astronomy: A study of cost and performance, in: Proceedings of the e-Science in Astronomy Conference, Brisbane, Australia, 2010.

[99] Yu-Vervet_Monkeys, J. Livny, H. Teonadi, M. Livny, M. K. Waldor, High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas, PLoS ONE 3 (2008) e3197.

[100] V. Marx, Ucla team sequences cell line, puts open source software framework into production newsletter: Bioinform (2010).
URL http://www.genomeweb.com/print/933003

[101] Seqware (2010).
URL http://sourceforge.net/apps/mediawiki/seqware