Pegasus and DAGMan From Concept to Execution: Mapping Scientific Workflows onto Today's Cyberinfrastructure

Ewa Deelman[1], Miron Livny[2], Gaurang Mehta[1], Andrew Pavlo[2], Gurmeet Singh[1], Mei-Hui Su[1], Karan Vahi[1], R. Kent Wenger[2]
[1]*USC Information Sciences Institute, Marina Del Rey, CA 90292*
[2] *University of Wisconsin Madison, Madison, WI 53706*

Abstract: In this chapter we describe an end-to-end workflow management system that enables scientists to describe their large-scale analysis in abstract terms, then maps and executes the workflows in an efficient and reliable manner on distributed resources. We describe Pegasus and DAGMan and various workflow restructuring and optimizations they perform and demonstrate the scalability and reliability of the approach using applications from astronomy, gravitational-wave physics, and earthquake science.

Keywords: workflow management, workflow optimization, abstract workflow, workflow provenance, cyberinfrastucture

Introduction

Scientific workflows are becoming an enabler of complex scientific analyses. They provide a representation of complex analyses composed of heterogeneous models designed by groups of scientists. At the same time, workflows have also become a useful representation that is used to manage the execution of large-scale computations. This representation not only facilitates overall creation and management of the computation but also builds a foundation upon which results can be validated and shared. Since workflows formally describe the sequence of computational and data management tasks, it is easy to trace back how particular data were derived. Workflows have also become a tool capable of bringing sophisticated analysis to a broad range of users, enhancing scientific collaboration and education. There are many workflow systems currently being developed [7]. In this chapter we concentrate on two complimentary systems Pegasus [28, 31] and DAGMan [22, 34] that together enable efficient and robust execution of large-scale scientific workflows in distributed environments.

In order to facilitate workflow creation, scientists need to be allowed to formulate the workflows in a way that is meaningful to them using high-level abstractions that specify the overall structure of the analysis and the data to be operated on (via a visual or textual interface) in a resource-independent way. This abstract workflow (or workflow instance) is important because it uniquely identifies the analysis to be conducted at the application level without including operational details of the execution

environment. The instance can thus be published along with the results to describe how a particular data product was obtained.

In order to support the abstract workflow specifications which let scientists concentrate on the science rather than on the operational aspects of the cyberinfrastructure (such as the Open Science Grid [2] or the TeraGrid [1]), mapping technologies are needed to automatically interpret and map the user-defined workflows onto the available resources. This is an approach analogous to traditional computer programming methods, where high-level languages are used to describe the computation without needing to specify the use of specific registers or memory locations. In this analogy, the "workflow mapping engine" is a compiler that translates between the high-level specifications and the underlying execution system and optimizing the executables based on the target architecture. The mapping includes finding the appropriate software and computational resources where the execution can take place as well as finding copies of the data indicated in the workflow instance. The mapping process can also involve workflow restructuring geared towards optimizing the overall workflow performance as well as workflow transformation geared towards data management and provenance information generation. The mapping process is usually automated and in our work it is done by Pegasus [28, 31].

The result of the mapping process is an executable workflow, which can be executed by a workflow engine that follows the dependencies defined in the workflow and executes the activities defined in the workflow nodes. DAGMan [35], our workflow engine relies on the resources (compute, storage, and network) defined in the workflow to perform the necessary actions. As part of the execution, data are generated along with their associated metadata and any provenance information that is collected.

The separation of concerns between workflow generation, workflow mapping, and workflow execution allows us to design software in a modular way and to optimize the components based on their functionality. Additionally it allows us to interface to a variety of workflow generation systems.

In this chapter we describe Pegasus and DAGMan and illustrate the workflow optimizations and restructuring techniques and their use in large-scale scientific applications.

1. Pegasus and DAGMan

Currently, Pegasus takes a workflow description in a form of a Directed Acyclic Graph in XML format (DAX). We recognize that not every scientist would be willing to write workflows in XML (or write scripts to generate XML), and that users may want to use simple languages or point and click GUIs. Thus we have integrated Pegasus into a variety of workflow instance generation systems such as VDL (Virtual Data Language) [33], Wings [36], Triana [52], and most recently we have developed a prototype integration with Kepler [44].

Of particular interest is the integration with Wings [37]. Wings uses rich semantic descriptions of components and workflow templates expressed in terms of domain ontologies and constraints. Wings has a workflow template editor to compose components and their data flow. Wings assists the user by enforcing the constraints specified for the workflow components. Wings also helps with data selection to ensure

the datasets selected conform to the requirements of the workflow template. With this information, Wings generates a workflow instance that specifies the computations (but not where they will take place) and the new data products. For all the new data products, it generates metadata attributes by propagating metadata from the input data through the descriptions and constraints specified for each of the components.

In some cases, scientific applications want to provide users with an interface, which is only in the form of a metadata query. For example, in astronomy, users often do not want to know the details of the underlying system, instead they want to retrieve images of an area of the sky of interest to them. In such cases Pegasus is usually integrated into a portal environment where the user is presented with a web form to fill in the desired metadata attributes. Inside the portal, the workflow instance is generated automatically based on the user's input and is given to Pegasus for mapping and then to DAGMan for execution [50]. Examples of this approach can be seen in the Montage project (an astronomy application) [18, 40], the Telescience portal (a neuroscience application)[42], and the Earthworks portal (an earthquake science application) [46]. In all these applications, Pegasus and DAGMan are being used to run the application workflows on national infrastructure such as the TeraGrid.

Pegasus can map workflows onto a variety of target resources such as those managed by PBS [39], LSF [55], Condor [32], and individual machines. Figure 1 shows an overview of the workflow generation and execution process. The executable workflow produced by Pegasus has directives to DAGMan for the execution of the workflow components. These directives include remote job execution, data movement, and data registration. Authentication to remote resources is done via GSI [54]. During the workflow execution, we capture provenance information about the execution tasks. Provenance includes a variety of information such the hosts where the tasks have executed, the runtime, environment variables, etc.

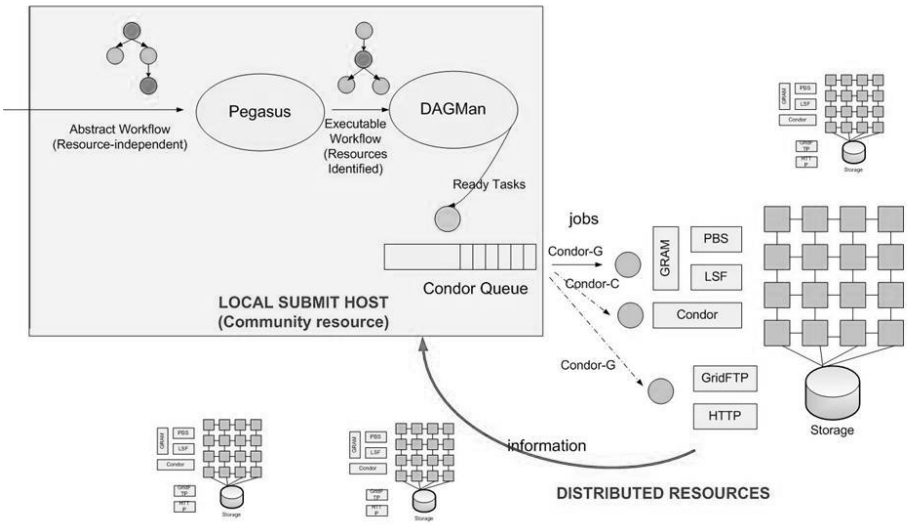

Figure 1: A General Overview of Pegasus and DAGMan Managing Workflows on the National Cyberinfrastructure.

The workflow mapping and workflow optimizations we perform as well as the scalability and robustness of our workflow execution enable us to efficiently and reliably run large-scale scientific workflows. Below we describe some of these key features.

1.1.Automatically locating physical locations for both workflow components and data.

Mapping the workflow instance to an executable form involves finding the resources that are available and can perform the computations, the data that is used in the workflow, and the necessary software. We assume that data may be replicated in the environment and that users publish their data products into some data registry. This registry can be a private or community resource. Some communities, such as Laser Interferometer Gravitational-Wave Observatory (LIGO) [13] maintain project-wide registries of the data coming off the detectors. Pegasus uses the logical filenames referenced in the workflow to query a data registry service such as the Globus Replica Location Service (RLS) [21] to locate the replicas of the required data. Given the set of logical filenames, RLS returns a corresponding set of physical file locations. Optionally, Pegasus also adds nodes to the workflow to register the final and intermediate workflow data products into the registry. In this way, new data products can be easily discovered by the user, the community, or another workflow. In order to be able to find the location of the logical application component names (transformations) defined in the workflow instance, Pegasus queries the Transformation Catalog (TC) [25] and obtains the physical locations of the transformations (on possibly several systems) and the environment variables and libraries necessary for the proper execution of the software. Pegasus also supports staging of statically linked executables on demand. In that case, the executables are treated as input data for the corresponding workflow tasks. The executables are transferred to the remote grid sites along with other input data required by the jobs.

1.2.Finding appropriate resources to execute the components

Pegasus queries cyberinfrastructure monitoring services (the Globus Monitoring and Discovery Service (MDS) [24], the OSG VORS system [6], or any information service) to find the available resources and their characteristics (machine load, scheduler queue length, available disk space, and others). This information is combined with information from the Transformation Catalog to make scheduling decisions. When making a resource assignment, Pegasus prefers to schedule the computation where the data already exist; otherwise it makes a random choice or uses simple scheduling heuristics such as min-min [19], or HEFT [53]. Oftentimes it is difficult to apply sophisticated scheduling algorithms because of the incomplete or out-of-date information about resources, in-exact or missing models of the application performance, and the size of the workflows that may result in significant runtimes for the scheduling algorithms.

Pegasus also uses information services to find information about the location of the data movement services (GridFTP [10], RFT [11], or SRB [14] servers) that can perform wide-area data transfers, job managers [23] that can schedule jobs on the remote sites, storage locations, where data can be pre-staged, shared execution directories, site-wide environment variables, etc. This information is necessary to

produce the executable workflow that describes the necessary data movement, computation and catalog updates. Registries of code and data as well as information services allow Pegasus to provide a level of abstraction to the user and give it the freedom to automatically optimize the workflow execution.

1.3.Performance optimization through workflow restructuring

During the mapping process, Pegasus is able to restructure the workflow to improve the overall workflow performance. One of the optimizations involves clustering of workflow tasks so that they are treated as one for the purpose of submission to a remote site and where they are then expanded automatically into a sequence of tasks or a parallel set of tasks (via MPI [38]) for the purpose of execution.

Pegasus currently implements level- and label- based clustering. In level-based clustering, tasks at the same level can be clustered together. The user can specify either the number of clusters to be created per level or the number of tasks to be grouped in a cluster. We apply clustering techniques in applications such as Montage [5, 16, 17], an application, where there are often many (~10,000) short duration (order of seconds) tasks. This type of application usually incurs large overheads if each task is submitted individually. Figure 2 shows a small Montage workflow in its original form (left), clustered with two clusters per level (middle) and two tasks per cluster (right).

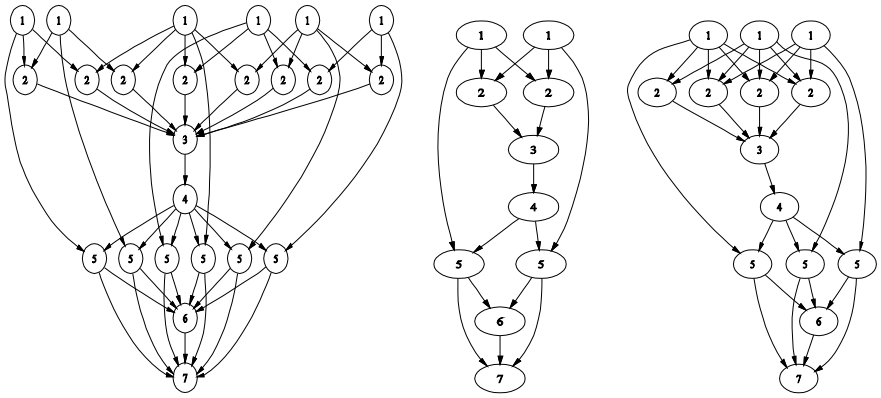

Figure 2. Montage workflow: original (left) clustered with two clusters per level (middle) and two tasks per cluster (right).

In label-based clustering, the user can label the tasks in the workflow to be clustered together. The tasks in the workflow with the same label are grouped into a single cluster. Figure 3(1) shows a workflow where tasks are labeled as *cluster_1* and *cluster_2* and the resulting clustered workflow is shown in Figure 3(2). Note that if the tasks in the clusters are executed sequentially implicit dependencies between tasks are added. Any clustering scheme can be implemented using an appropriate labeling scheme.

Each cluster whether generated using level- or label- based clustering must satisfy the convexity requirement that dictates that all paths between any two tasks in a cluster must be completely contained within it. The cluster shown in Figure 4 is non-convex

since the path from t1 to t3 through t4 is not contained within the cluster. The difficulty here is that t4 must start execution after t1 has completed and before t3 starts execution. Thus it creates co-scheduling requirements between clusters. However, due to the best effort nature of the execution environment, it is not possible to achieve co-scheduling without explicit resource control.

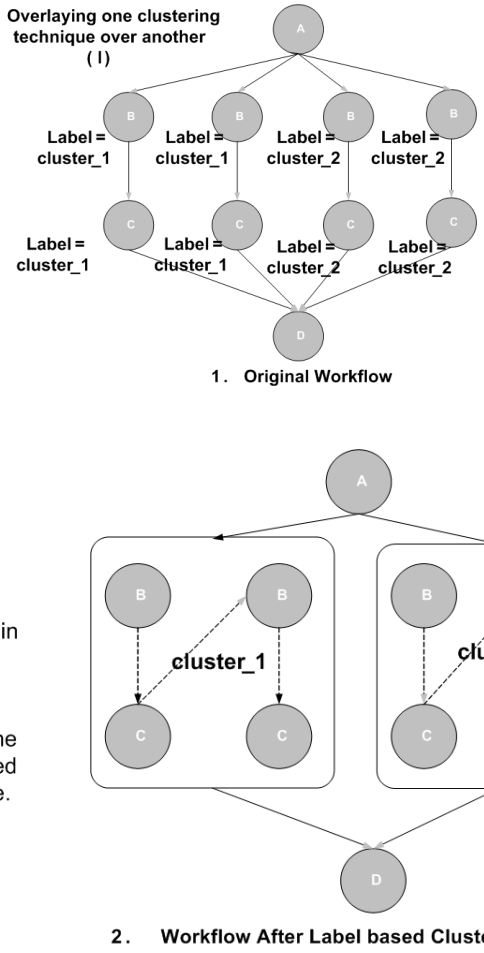

Figure 3. Example of label based clustering.

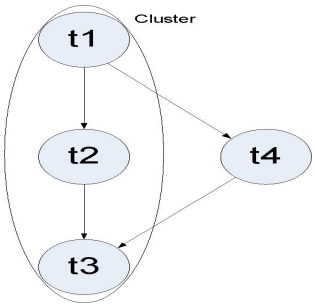

Figure 4. A non-convex cluster.

Pegasus does error checking to ensure that each cluster created by grouping the tasks with the same label satisfies the convexity requirement. Note that the clusters generated using level-based clustering trivially satisfy the convexity requirement since all the tasks at a level are independent of each other and no path exists between them. Another restriction of clustering is that the tasks within a cluster be scheduled to the same resource.

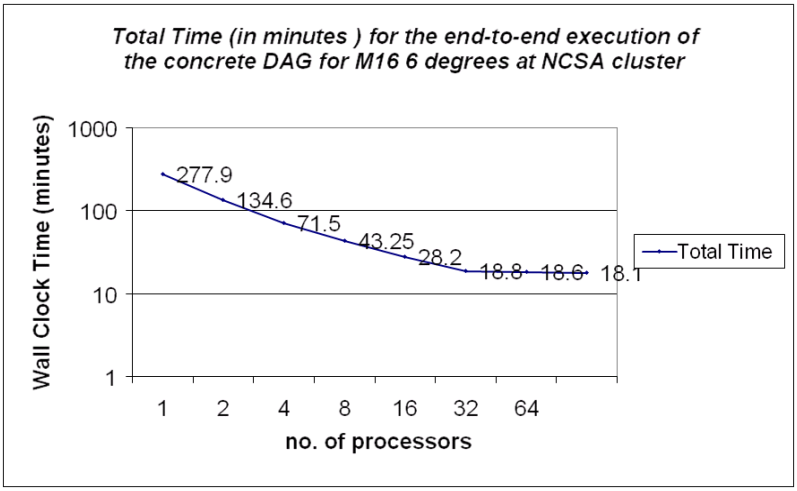

Figure 5: Improving Workflow Scalability With the Use of Clustering Techniques [40].

Figure 5 shows the results of the application of this optimization to Montage. The figure shows the results when Pegasus groups the nodes of the workflow into as many tasks per cluster as there are processors. Pegasus then schedules the clusters through DAGMan. Each cluster is executed as an MPI job on the TeraGrid. The figure shows a good speedup of approximately 15 on 32 processors. As the number of processors increases, the speedup decreases due to sequential aspects of the Montage workflow structure.

1.4.Workflow footprint optimization

In many execution environments, data storage available for executing applications can be limited. In our recent work [48, 51] we are studying mechanisms for minimizing the amount of disk space a particular workflow requires by adding nodes to the workflow to explicitly remove the data when they are no longer needed. The purpose of the cleanup job is to delete the data file from a specified computational resource to make room for the subsequent computations. Since a data file can be potentially replicated on multiple resources (in case the compute tasks are mapped to multiple resources) the decisions to add cleanup jobs are made on a per-resource basis. The algorithm is applied after the executable workflow has been created but before the workflow is executed. Details of the algorithm are described in [51], here we just provide a simple illustration.

Figure 6 shows an executable workflow containing 7 compute jobs {0,1,..,6} mapped to two resources {0,1}. The algorithm first creates a subgraph of the executable workflow for each execution resource used in the workflow. The subgraph of the workflow on resource 0 contains jobs {0,1,3,4} and the subgraph on resource 1 contains jobs {2,5,6}. The cleanup nodes added to this workflow using our algorithm are shown in Figure 7. The cleanup job for removing file f on resource r is denoted as C_{fr}.

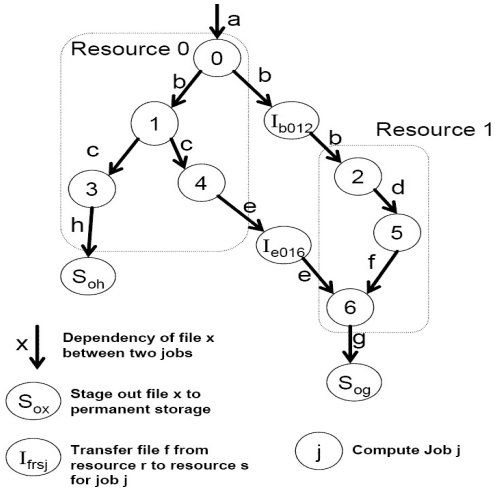

Figure 6: The Original Executable Workflow Mapped by Pegasus to Two resources.

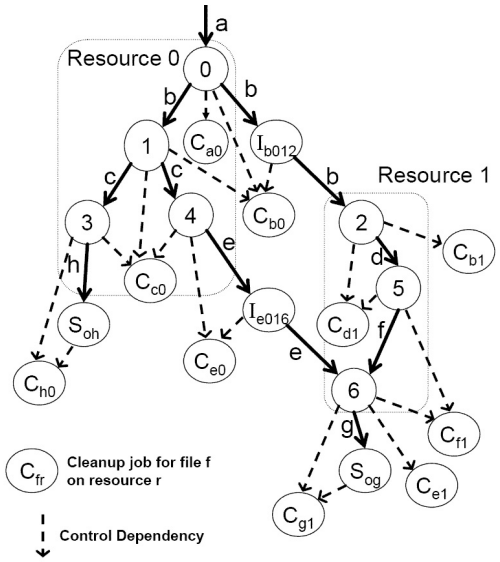

Figure 7**:** The Workflow with Dynamic Cleanup Nodes Added.

For each task in the subgraph, a list of files either required or produced by the task is constructed. For example, the list of files for task 1 mapped to resource 0 contains files b and c. For each file in the list, a cleanup job for that file on that resource is created (if it does not already exist) and the task is made the parent of the cleanup job. Thus, a cleanup job, C_{c0}, which will remove file c on resource 0 is created and task 1 is made the parent of this cleanup job. The cleanup jobs for some files might already have been created as a result of parsing previous tasks. For example, the cleanup job C_{b0} for removing file b on resource 0 already exists (as a result of parsing task 0). In this case the task being parsed is added as a parent of the cleanup job. Thus, task 1 is added as a parent of cleanup job C_{b0}. When the entire subgraph has been traversed, there exists one cleanup job for every file required or produced by tasks mapped to the resource. If a file required by a task is being staged-in from another resource, then the algorithm makes the cleanup job for the file on the source resource a child of the stage-in job, ensuring that the file is not cleaned up on the source resource before it is transferred to the target resource. For example, file b required by task 2 mapped to resource 1 is being staged-in from resource 0 using stage-in job I_{b012}, and so the cleanup job for file b on resource 0 (C_{b0}) is made a child of I_{b012}. Finally, if a file produced by a task is being staged-out to a storage location, the cleanup job is made a child of the stage-out job. For instance, the cleanup job C_{h0} for removing file h on resource 0 is made a child of the stage-out job S_{oh} that stages out file h to permanent storage. By adding the appropriate dependencies, the algorithm makes sure that the file is cleaned up only when it is no longer required by any task in the workflow. In this version of the algorithm there are as many cleanup jobs as there are files. For large scientific workflows, this solution is not scalable, so for the final version of the algorithm, we reduced the number of clean up jobs to be at most as many as there are computational tasks in the workflow.

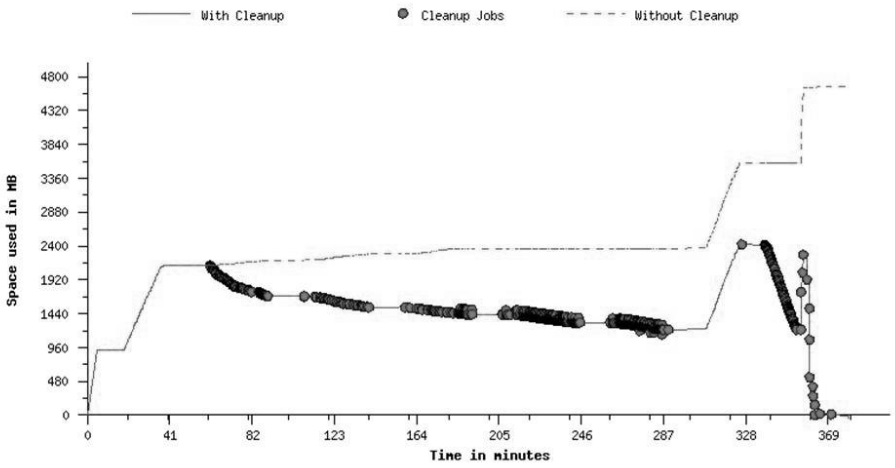

Figure 8: Actual Data Space Optimization for a Montage Workflow. The figure shows the amount of space used by a workflow over time on resources of Open Science Grid with and without space optimization[51].

In real experiments using the resources of the Open Science Grid, we were able to reduce the workflow data footprint of a 2 square degree Montage workflow by approximately 48% [51]. Figure 8 shows the storage space used by the workflow without and with cleanup during the execution of the workflow. Cleanup opportunities are widespread in the workflow as can be seen by the execution of cleanup jobs in the figure leading to a significant reduction in maximum storage used. For some workflows such as LIGO these opportunities might be concentrated towards the end of the workflow execution resulting in relatively small savings in the storage space. For these workflows we use restructuring techniques in order to minimize the amount of storage space used. We are currently evaluating those techniques using real applications and cyberinfrastructure deployments.

1.5.Adapting to the changing execution environment

Very large workflows (on the order of thousands of tasks) should not be mapped all at once as the state and the availability resources can change significantly over the execution time of the workflow. Rather, portions of the workflow need to mapped and executed before the remaining portions are mapped and executed. In our work we have developed a workflow partitioning technique where the workflow is divided into smaller sub-workflows [26]. The partitioning algorithm is a pluggable component of Pegasus. The only restriction is that the partition preserves the dependencies in the original workflow and that there cannot be any cyclic dependencies between the partitions, or in other words, the partitions need to be convex as defined in the clustering section. Currently, we have implemented three basic partitioning algorithms: level-based, which creates partitions based on the depth of the workflow—all tasks at the same depth are in the same partition; single-node where each node is its own partition, and label-based partitioning. Single node partitioning is equivalent to just-in-time mapping, where each task is mapped to a resource only when it is ready to run. In label-based partitioning, the partitions are built based on the labels assigned to them allowing for a flexible partition structure. Figure 9 shows how a label-based

partitioning works. Figure 9 (a) shows the original workflow labeled for partitioning. Figure 9 (b) shows this workflow partitioning into three partitions. In this workflow Partition 1 will execute first, then the other two partitions can execute in parallel. We also notice that although this partitioning is correct as it does not violate any dependencies, it does introduce additional dependencies, in particular between tasks C and D and task B. This may result in performance degradation.

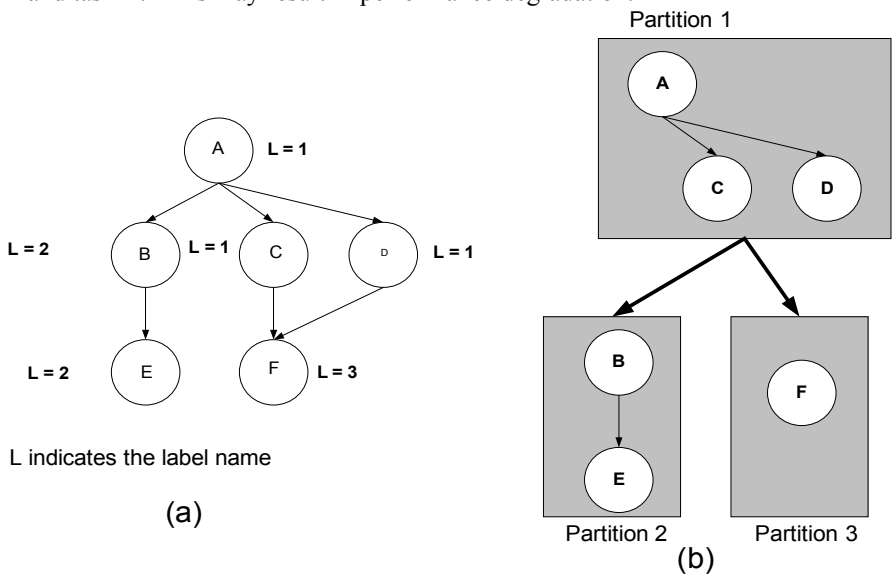

Figure 9: (a) A Workflow Labeled for Partitioning. (b) Resulting workflow.

Each partition is mapped by Pegasus and executed by DAGMan before the dependent partitions are treated in the same fashion. The entire process of managing the mapping and execution of workflows and following the dependencies between them is performed by DAGMan. The workflow partitioning approach combined with node clustering is often used by LIGO workflows running on the OSG and in SCEC workflows running on the TeraGrid to adjust to the dynamic grid conditions and to improve the overall workflow performance.

1.6.Support for reliable execution across a variety of platforms

Pegasus' partitioning and data reuse play an important role in the reliability of the workflow execution, especially in very dynamic execution environments. Breaking up the workflow into smaller pieces and mapping and executing at that level of granularity enables us to re-plan and re-execute the individual partitions when failures during execution occur [28]. If a partition fails during the mapping or the execution, we can trigger a retry (or several retries) of that partition. Additionally when the re-planning occurs within the partition, not everything needs to be re-executed because of the data-reuse capabilities of Pegasus. If for example the original sub-workflow consisted of three sequential tasks *t1, t2, t3*, each producing file *f1, f2, f3* respectively and task *t3* failed, then upon re-planning, Pegasus would discover that files *f1* and *f2* already exist and thus tasks *t1* and *t2* do not need to be re-executed and the only task that will be re-mapped and re-executed is task *t3*.

Additional reliability is provided by the DAGMan execution engine. DAGMan sits as a layer "above" the batch system in the software stack. DAGMan utilizes the batch system's standard API and logs in order to submit, query, and manipulate jobs, and does not directly interact with the jobs independently. While DAGMan currently only works with Condor [43] as a batch system, it can uses Condor's grid abilities (known as Condor-G) to submit to many other batch and grid systems. DAGMan reads the logs of the underlying batch system to follow the status of submitted jobs rather than invoking interactive tools or service APIs. Reliance on simpler, file-based I/O allows DAGMan's own implementation to be simpler, more scalable and reliable across many platforms, and therefore more robust. For example, if DAGMan has crashed while the underlying batch system continues to run jobs, DAGMan can recover its state upon restart (by reading logs provided by the batch system) and there is no concern about missing callbacks or gathering information if the batch system is temporarily unavailable ─ it is all in the log file.

Workflow management includes not only job submission and monitoring but job preparation, cleanup, throttling, retry, and other actions necessary to ensure the successful workflow execution. DAGMan attempts to overcome or work around as many execution errors as possible, and in the face of errors it cannot overcome, it allows the user to resolve the problem manually and then resume the workflow from the point where it last left off. This can be thought of as a "checkpointing" of the workflow, just as some batch systems provide checkpointing of jobs.

1.7.Scalability of Pegasus and DAGMan

The scalability of our workflow management system can be seen in the context of scientific applications that use our technologies.

The Southern California Earthquake Center (SCEC) [8] uses our tools to produce more accurate seismic hazard maps. These maps, generated as part of the CyberShake project [45], indicate the maximum amount of shaking expected at a particular geographic location over a certain period of time. The hazard maps are used by civil engineers to determine building design tolerances. Figure 10 shows the results of one of two major CyberShake executions. The run was performed on the TeraGrid in the fall of 2005 and used Pegasus and DAGMan. The two workflows ran over a period of 23 days and processed 20TB of data using 1.8 CPU Years. The total number of tasks in the two workflows was 261,823. CyberShake workflows have tasks with very varied runtimes as can be seen in Figure 11. The task times range from l minute to almost 70 hours.

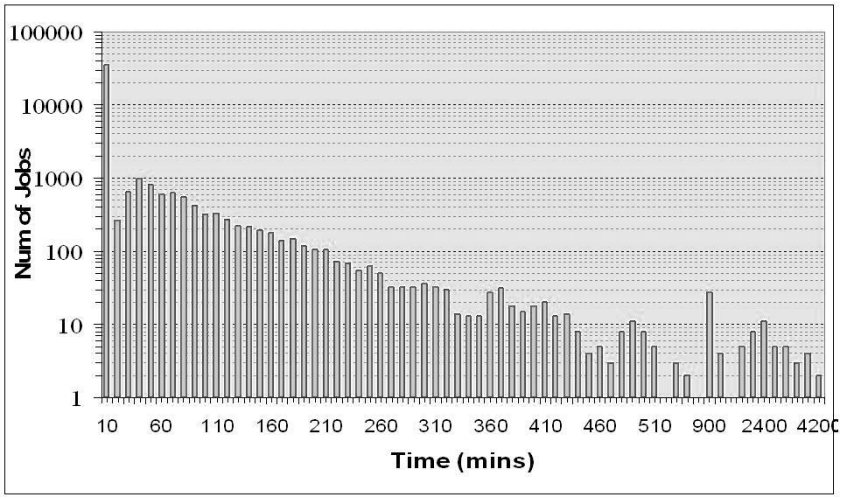

Figure 10: Execution of SCEC Workflows on the TeraGrid in 2005 [29, 30].

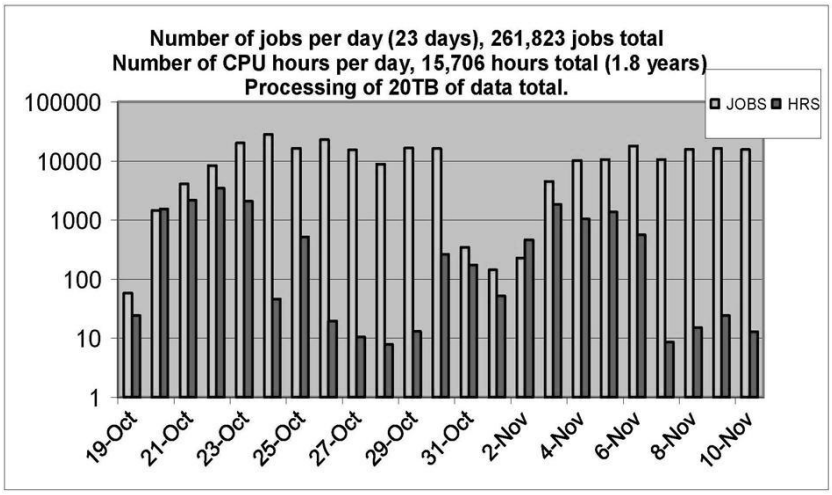

Figure 11: Distribution of Seismogram Tasks in the SCEC CyberShake Workflow.

After several months of validation, result interpretation, code modification and improvement, SCEC conducted 10 more runs of CyberShake in the Spring of 2006. The total number of tasks was over 212,000 and the runtime was approximately 8 CPU months. Since then the scientists have been analyzing the results and trying to understand why the results differ from the traditional calculations. Now, a new wave of simulations is underway. As the result of many simulations, SCEC scientists now believe that CyberShake may be more accurate than traditional methods because it models rupture directivity and sedimentary basin effects which contribute to the shaking experienced at different geographic locations. As a result more accurate hazard maps can be created. SCEC is also using Pegasus and DAGMan in the Earthworks Portal [46], a TeraGrid Science Gateway, hosted at Washington University that allows users to configure and execute earthquake wave propagation simulations structured as workflows through a simple portal interface.

Pegasus and DAGMan are used in the LIGO project to map binary inspiral analysis workflows onto the OSG [20]. A month of LIGO data requires many thousands of jobs, running for days on hundreds of CPUs. Figure 12 illustrates the use of OSG for the LIGO workflows over the period of November 2006 to early January 2007. The figure was created using the Monalisa monitoring software used on OSG [9]. The workflows were run across several OSG sites and used a total of 2.5 CPU years of computing over a period of two months.

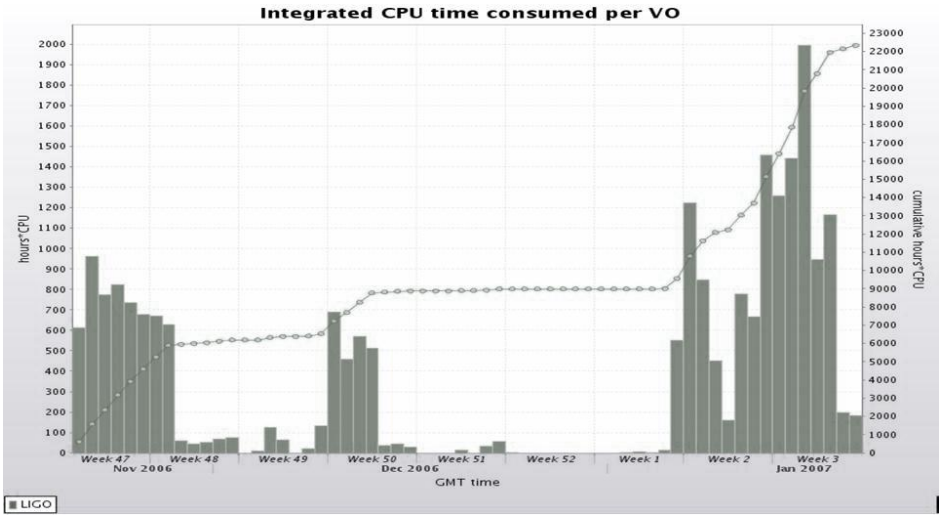

Figure 12: LIGO's CPU Hours Usage of OSG Resources.

Currently DAGMan is also used by a broad range of application in both scientific and commercial domains. Some examples of DAGMan use in science are in the areas of bioinformatics, machine translation, computational fluid dynamics, high-energy physics, and others. In particular DAGMan is used in production by BioMagResBank (BMRB) [3] at UW-Madison to execute BLAST workflows, which consist of approximately 500 nodes and run for approximately 24 hours on 100 processors. During a workflow run, the BMRB's BLAST setup compares 4,000 sequences against multiple databases containing a total of ~4.5 million entries. Another use of DAGMan is the NMI build and test infrastructure [4]. This infrastructure automatically builds a set of software packages every day on a variety of platforms and performs automated tests on the resulting builds. The builds are represented as DAGMan workflows, where the dependencies reflect the dependencies between the builds of the software packages. Figures 13 and 14 show some of the details of the execution of the workflows over 28 months starting in October of 2004. Over that time period there were ~2,800 workflows submitted each month, with a total of over 105,000 workflow tasks executed each month.

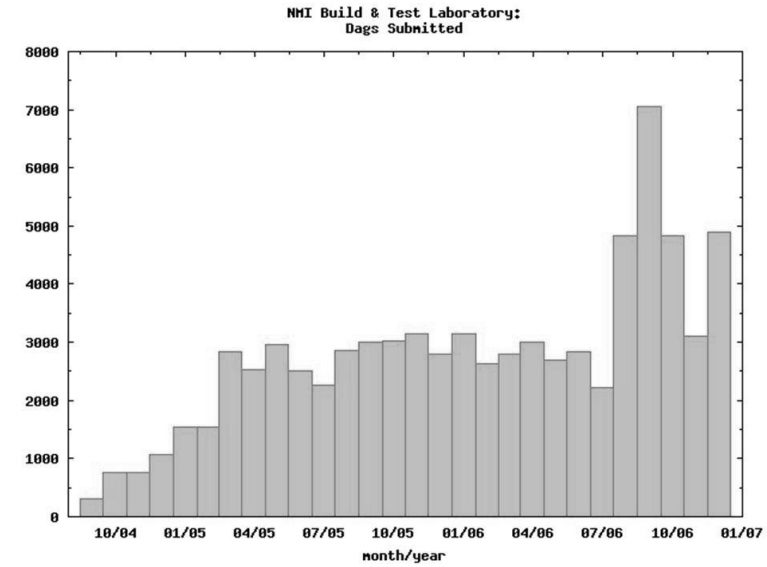

Figure 13: The Number of Workflow Submitted Each Month in Support of the NMI Build and Test.

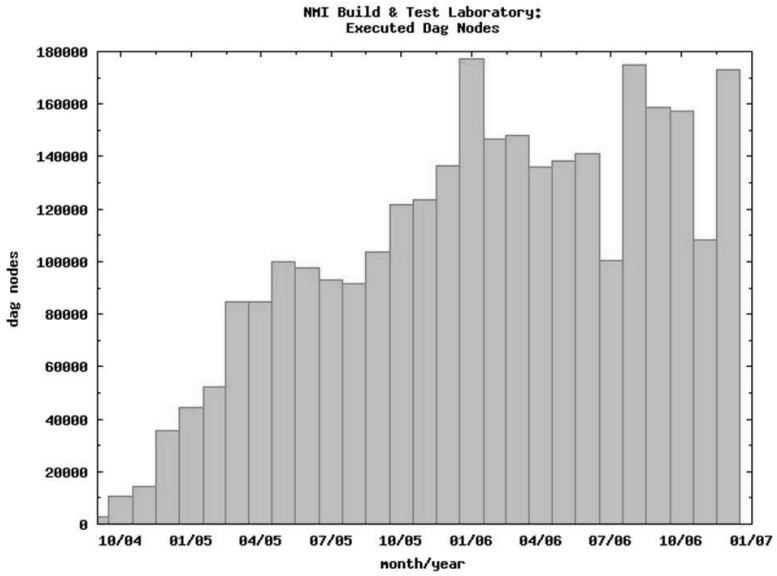

Figure 14: The Number of Workflow Tasks Submitted Each Month in Support of the NMI Build and Test.

1.8.Provenance Tracking

One promising area in large-scale applications in general and in workflow technologies in particular is the generation, management and querying of provenance which provides information about how data was produced. In our current system, Pegasus stores provenance information related to the execution of the workflow components. This information describes the execution environment of the tasks, where

they were executed, how long the execution took place, which files were used and generated and other task-level information. We are also exploring the use of provenance tracking capabilities to record the workflow transformations performed by Pegasus during the workflow mapping process [41].

2. Other Applications Using Pegasus and DAGMan

Pegasus and DAGMan are used in astronomy, and in particular in the Montage application which delivers science-grade mosaics of the sky. Our workflow technologies were used to transform a single-processor Montage code into a complex workflow and parallelized computations to process larger-scale images. Montage workflows mapped by Pegasus to the existing cyberinfrastructure are characterized by tens of thousands of executable tasks and the processing of thousands of images. Recently, Montage was used to make a scientific discovery: the verification of a bar in the spiral galaxy M31 [15]. Although there have been hints of a bar in M31 from optical data, none of the analyses were convincing because the effects of interstellar extinction at optical wavelengths were severe. However, the universe is much more transparent in the infrared, and this enabled astronomers to overcome the effects of interstellar extinction. There was one more problem: the variable background in the infrared images hid the structure of the galaxy. By using Montage, which was able to rectify the backgrounds to a common level, the astronomers were finally able to see the structure.

Pegasus and DAGMan are also used in the Telescience project [42] and portal to support 3D reconstruction of electron tomography images. The UCSD scientists plan to continue to rely on our workflow technologies to expand the set of Grid applications they support within their portal environment and to develop new techniques that can provide real-time feedback from the 3D reconstruction to the scientists manipulating the instrument. Data mining and natural language processing applications are new user communities that are exploring the use of our workflow technologies to manage the large-scale computations on today's cyberinfrastructure.

Within GriPhyN [12], Pegasus was made available as part of the Virtual Data System (VDS) and interfaced with VDL. Today several applications use Pegasus and DAGMan it in that fashion. Among them are: a climate modeling application, where simulations which used to take 2.5 months to run manually, took only 2.5 days to run using our tools [47]. Another example is the GADU project (Genome Analysis and Database Update) [49], where the researchers at Argonne National Laboratory use Pegasus and DAGMan to conduct genome analysis on OSG.

3. Conclusions

We have shown the ability of the end-to-end Pegasus/DAGMan to efficiently and robustly execute computational workflows in a variety of application domains. Although we were able to address many issues faced by today's large-scale analyses running on the distributed cyberinfrastructure resources, many challenges still remain.

Through our work with data-intensive computations, we have recognized that data management is at least just as important as computation scheduling. When data sets

being processed or generated by workflows are significant—sometimes on the order of Terabytes, it is critical to employ workflow scheduling techniques that are aware of the storage necessary to support the successful workflow execution.

Since many scientific collaborations are managing their experimental data across the distributed environment, it is also necessary for the workflows to be aware of these systems and collaborate with them in order to support the overall analysis. Examining these data-versus computation-placement issues that are being implemented by independent systems remains a challenge and we plan to continue our work in this area [27].

Acknowledgments

This work was supported by the National Science Foundation under grant CNS 0615412 and EAR-0122464. 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 and provided by the NSF TeraGrid.

The authors would like to thank the many collaborators from the science projects, among them: Tom Jordan, Phil Maechling, Robert Graves, David Meyers, and Scott Callaghan (SCEC), Kent Blackburn, Duncan Brown, Scott Koranda, and Britta Daudert (LIGO), Bruce Berriman, John Good, and Dan Katz (Montage), Steven Peltier and Abel Lin (UCSD-NCMIR), Yolanda Gil, Jihie Kim and Varun Ratnakar (Wings), Luc Moreau, Simon Miles, and Paul Groth (PASOA provenance), Rizos Sakellariou (workflow scheduling), and Jens Voeckler (remote execution).

References