# Enabling Large-scale Scientific Workflows on Petascale Resources Using MPI Master/Worker

Mats Rynge[1]
rynge@isi.edu

Gideon Juve[1]
gideon@isi.edu

Karan Vahi[1]
vahi@isi.edu

Scott Callaghan[2]
scottcal@usc.edu

Gaurang Mehta[1]
gmehta@isi.edu

Philip J. Maechling[2]
maechlin@usc.edu

Ewa Deelman[1]
deelman@isi.edu

[1] Information Sciences Institute, University of Southern California
[2] Southern California Earthquake Center, University of Southern California

## ABSTRACT
Computational scientists often need to execute large, loosely-coupled parallel applications such as workflows and bags of tasks in order to do their research. These applications are typically composed of many, short-running, serial tasks, which frequently demand large amounts of computation and storage. In order to produce results in a reasonable amount of time, scientists would like to execute these applications using petascale resources. In the past this has been a challenge because petascale systems are not designed to execute such workloads efficiently. In this paper we describe a new approach to executing large, fine-grained workflows on distributed petascale systems. Our solution involves partitioning the workflow into independent subgraphs, and then submitting each subgraph as a self-contained MPI job to the available resources (often remote). We describe how the partitioning and job management has been implemented in the Pegasus Workflow Management System. We also explain how this approach provides an end-to-end solution for challenges related to system architecture, queue policies and priorities, and application reuse and development. Finally, we describe how the system is being used to enable the execution of a very large seismic hazard analysis application on XSEDE resources.



XSEDE'12, July 16-20, 2012, Chicago, IL, USA



## Categories and Subject Descriptors
F.1.2 [**Computation by Abstract Device**]: Modes of Computation – Parallelism and concurrency

## General Terms
Algorithms, Management, Reliability

## Keywords
Workflow management, task clustering

## 1. INTRODUCTION
Many computational scientists need to execute large-scale, loosely-coupled applications in order to do their research. These applications are often structured as scientific workflows that consist of a few large parallel tasks surrounded by a large number of small, serial tasks used for pre- and post-processing. Although the majority of the individual tasks within these applications are relatively small (less than a few minutes in runtime), in aggregate they represent a significant amount of computation and data. Such fine-grained workflows frequently contain tens of thousands of tasks, and workflows with up to a few million tasks are not uncommon [5]. Together, these tasks may require many tens of thousands of CPU hours of computation and process many terabytes of data.

In order to complete these large-scale computations in a reasonable amount of time, large-scale compute and storage resources are required. Scientists would like to take advantage of state-of-the-art petascale systems such as NICS Kraken and NCSA Blue Waters to reduce their time to solution, but these systems and systems like them are typically not optimized to execute loosely-coupled workloads efficiently. There are several challenges related to software, scheduling, and networking that must be overcome in order to make this possible.

The first challenge scientists face when running loosely coupled jobs on petascale systems is working with software from a variety of diverse sources. Computational scientists often need to integrate multiple scientific codes into pipelines and workflows in order to achieve their goals. These codes are often developed by different teams, over a long period of time, and have been carefully validated to ensure they are scientifically correct, so modifying them to be parallel codes is usually not feasible. Although some optimizations and smaller code changes can be performed, writing new software from scratch to better fit the programming models typically used on petascale systems is often not an option. Scientists need to make existing codes work within the execution environment provided by the resources.

Another major challenge users face when running loosely-coupled applications on petascale systems is that most petascale systems are highly optimized for running large, monolithic, parallel jobs such as MPI codes. Over the past decade, there have been many changes in the system architecture of compute clusters. For example, during TeraGrid and now into XSEDE, the available systems have progressed from traditional Linux clusters to more specialized environments, such as the Cray XT System Environment. The traditional clusters were composed of relatively generic server-class machines with IA64, x86 or x86_64 processors running standard Linux distributions. On such clusters, mixed MPI/serial workloads could be supported using tools such as Condor Glideins [12][15]. In contrast, on newer systems the hardware and software architectures are highly optimized for running parallel applications. For example, when a job is submitted on the Cray XT5 (NICS Kraken), rather than starting on the compute nodes, the job script runs on a service node. In that script, the user can execute the *aprun* command to access a second level scheduler called *ALPS (Application Level Placement Scheduler)*, which handles the placement and execution of parallel processes. In addition, there are significant differences between the traditional clusters and systems like the Cray XT5 at the node level. The former has, in many cases, a complete Linux environment with a full suite of tools running on the compute nodes, while the Cray has a very minimal kernel with almost no system tools available, and no shared libraries. Similarly, a traditional cluster usually has compute nodes with full IPv4/6 networking support in addition to high-speed interconnects such as Infiniband. Sometimes nodes on traditional clusters even have public IPs. In contrast, networking on Cray compute nodes is usually limited to system-local connections over a custom, vendor-specific interconnect. These differences significantly limit the scientists' choice of tools to use for executing loosely-coupled applications. Many existing high throughput computing tools, including Condor Glideins, depend on a basic set of system tools and the ability to make outbound IP network connections, which makes the tools a poor fit for these petascale systems.

Another challenge facing scientists that want to run loosely coupled jobs on petascale systems is scheduling policies. Similar to their architectures, the queue policies and job priorities of petascale systems reflect their preference for large parallel jobs. Many systems place a limit of the number of simultaneous jobs a user can have in the queue and/or have running. On NICS Kraken, for example, a user is only allowed to have 5 running jobs and 5 ready to run jobs in the queue. If more than 5 jobs are submitted, they will be placed in a blocked state and not considered by the scheduler. Also, Kraken gives jobs with over 32,000 cores the highest priority and considers jobs with smaller core counts as backfill. Note that we do not disagree with these policies, we are simply pointing out that they pose a challenge when trying to run workloads that contain of a large number of serial tasks.

Previous approaches to solving these problems have limitations that reduce their effectiveness. Task clustering [23] reduces scheduling overheads and queuing delays by grouping several tasks into a single job, which increases throughput, but often reduces parallelism by forcing independent tasks to be executed in serial. Advance reservations [25] eliminate queuing delays by giving users exclusive access to a set of resources for a limited amount of time, but are not supported on many systems and have a negative impact on resource utilization and QoS. Pilot jobs [15][20][6][16][7] enable more efficient application-level scheduling, but require direct network access to compute nodes, which is often not feasible due to the use of firewalls, private networks, and compute nodes with highly customized networks.

In this paper we describe a new approach to scheduling large, fine-grained workflows on petascale systems. Our approach involves partitioning large, loosely-coupled workflows into independent subgraphs, and then executing each subgraph as a self-contained MPI job. This approach has been implemented in the Pegasus Workflow Management System [10] using a new MPI-based master/worker task scheduling tool called *pegasus-mpi-cluster* (PMC). In the remainder of this paper, we describe how PMC enables us to run large-scale, fine-grained grid workflows on petascale systems. We also describe an example application, the Southern California Earthquake Center (SCEC) CyberShake workflow, which is being deployed on the NICS Kraken system using Pegasus and PMC.

## 2. RELATED WORK

Workflow systems such as Askalon [11], Taverna [18], Triana [24], Kepler [1], and Condor DAGMan [8] are designed to run scientific workflows on the grid. These systems typically assume workflow tasks can be submitted to clusters remotely. Although the majority of resources available through XSEDE and other grids can be accessed using grid middleware such as Globus, these interfaces do not provide the level of throughput required for very fine-grained workflow tasks. In addition, grid middleware submits tasks directly to the local scheduler, which is often not capable of scheduling a large number of small tasks efficiently.

Grid-based master-worker systems such as Condor MW [13], Work Queue [4], Falkon [20], Condor Glideins [12][15][22] and pilot job frameworks [17][16][6] have been used to enable high-throughput scheduling of fine-grained tasks in grids. These systems start worker processes on remote clusters using standard batch jobs, and distribute tasks to those workers from a central master process running outside the target cluster. This approach allows these systems to achieve better throughput by bypassing the normal scheduling path, which is typically not optimized for small tasks. The problem with this approach is that it requires direct TCP connections from the master to the target cluster's compute nodes. This is not feasible on many clusters because of security concerns, firewalls, private networks, and a lack of TCP stacks on compute nodes.

DAGs and other task graph representations are used as the programming model for many parallel applications because they are effective in expressing and optimizing irregular computations [3][2][9][19]. These applications are typically decomposed into function-level tasks that perform very fine-grained computations and access small amounts of data stored in memory. In comparison, the workflow applications we are targeting in this work are composed of relatively coarse-grained, process-level

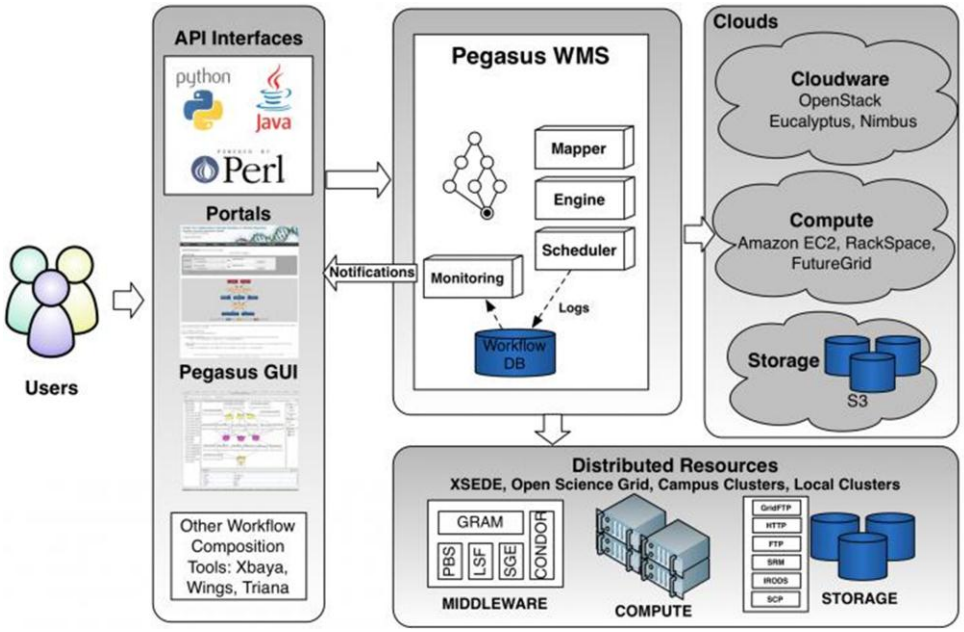



**Figure 1. Pegasus Workflow Management System provides support for running scientific workflows using many types of computing resources currently available.**

tasks that access much larger data sets stored in a shared file system.

# 3. APPROACH

## 3.1 Pegasus Workflow Management System

The Pegasus Workflow Management System [10] is used by scientists to execute large-scale computational workflows on a variety of cyberinfrastructure ranging from local desktops to campus clusters, grids, and commercial and academic clouds, as shown in Figure 1. Pegasus enables scientists to compose abstract workflows without worrying about the details of the underlying execution environment or the particulars of the low-level specifications required by the middleware (Condor, Globus, or Amazon EC2). First, scientists create data stores that contain information about the data files and transformations used in the calculation (RLS [7] and transformation catalog) and information about the available computing resources (called site catalog). Then the scientist provides an abstract representation, a directed acyclic graph (DAG), which may contain sets of different tasks. For example, some tasks may be embarrassingly parallel, while others may have complex dependencies. Pegasus takes in this abstract workflow and generates an executable workflow based on information about available resources. The system is composed of three components:

- **Mapper (Pegasus 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 also restructures the workflow to optimize performance and adds transformations for data management and provenance information generation.
- **Execution Engine (Condor DAGMan):** Executes the tasks defined by the workflow in order of their dependencies. DAGMan relies on the resources (compute, storage and network) defined in the executable workflow to perform the necessary actions.
- **Scheduler/Task manager (Condor Schedd):** manages individual workflow tasks: supervises their execution on local and remote resources.

## 3.2 Executing Large Workflows on Distributed Resources

When executing large workflows on distributed cyberinfrastructure, the Pegasus Mapper usually partitions workflows into manageable clusters that can be executed as single units. The restructuring of the workflow done by the Mapper helps reduce the scheduling overhead of short running jobs. Additionally, on resources with queue policies limiting the number of jobs a user is allowed to have in the queue, clustering can increase throughput. The Mapper also adds data management processes (new workflow nodes) to the executable workflow that stage in the input data required for these clusters of jobs and ship out the output data from the execution sites. The Pegasus Mapper can be configured to use different types of graph clustering techniques to determine how the jobs are clustered and also what executables to use for the clustered job. Some of the supported clustering techniques are listed below:

1. **Horizontal** - The Mapper clusters jobs on the same level of the workflow. The level for a job is determined by doing a Breadth First Search of the underlying DAG. The number of jobs that are clustered into a clustered job is determined by

user provided configuration parameters for different job types.

2. **Runtime-based Horizontal Clustering** - In this technique the Mapper looks at the expected runtimes of the jobs and does the grouping of the jobs on the same level of the workflow based on the *maxruntime* specified by the user for each job type.
3. **Label-based Clustering** - In this technique, the user identifies subgraphs in the abstract workflow description that they want to execute as one cluster. This is achieved by user associating labels with the jobs. Jobs associated with the same label are put in by Pegasus into the same cluster.

Workflows using previous versions of Pegasus WMS have traditionally used *pegasus-cluster*, a C executable that runs clustered jobs sequentially on a node. *pegasus-cluster* allows us to reduce the scheduling overhead of running lots of short running jobs as part of the workflow. However, since *pegasus-cluster* can only run jobs sequentially on a single node, it does not allow us to efficiently schedule tasks on the thousands of cores available on large systems, especially with per user job queue limits in place. In the past, we have used Condor Glideins [12][15][22][21] to overlay a Condor pool on top of resources. Condor Glideins start worker processes on remote clusters using standard batch jobs, and distribute tasks to those workers from a central master process running outside the target cluster. The problem with this approach is that it requires direct TCP connections from the cluster compute nodes to the central master. This is not feasible on many of the large clusters, so we wanted a self-contained solution, which would not require networking to the outside of the cluster.

## 3.3 Pegasus MPI Cluster

In order to efficiently use petascale systems for large workflow applications, and to address the system architecture and network issues outlined above, we have developed an MPI-based task management tool called *pegasus-mpi-cluster* (PMC). Using MPI enables us to leverage the underlying network communications libraries through a common interface that is portable across cutting-edge cyberinfrastructure.

A PMC job consists of a single master process (this process is rank 0 in MPI parlance) and several worker processes. These processes follow the standard master-worker architecture. The master process manages the workflow and assigns workflow tasks to workers for execution. The workers execute the 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.

In order for this approach to work, the target system must have a shared file system to enable the workers to access the executable specified in the workflow, read the workflow input data, and store workflow output data, and the system's compute nodes must allow the worker processes to fork additional processes to execute the tasks. We believe that these requirements are satisfied by most or all of the systems currently available through XSEDE and the DOE, and by upcoming systems such as Blue Waters.

## 3.4 Workflow Descriptions

PMC jobs are expressed as a Directed Acyclic Graph (DAG). Each node in the DAG represents a workflow task, and the edges represent dependencies between the tasks that constrain the order in which the tasks are executed. Each task is a program and a set of parameters that need to be executed. The dependencies typically represent data flow dependencies in the application, where the output files produced by one task are needed as inputs for another.

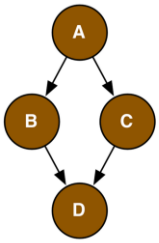
DAGs are expressed using a simple text-based format similar to the one used by Condor DAGMan [8]. There are two record types in a DAG file: TASK and EDGE. The format of a TASK record is:

```
TASK id executable [arguments]
```

Where id is the ID of the task, executable is the path to the executable or script to run, and arguments is an optional argument string to pass to the task. The format of an EDGE record is:

```
EDGE parent child
```

Where parent is the ID of the parent task, and child is the ID of the child task. Figure 2 shows an example DAG file that describes a simple diamond-shaped workflow.




**diamond.dag**

```
TASK A /bin/echo "I am task A"
TASK B /bin/echo "I am task B"
TASK C /bin/echo "I am task C"
TASK D /bin/echo "I am task D"
EDGE A B
EDGE A C
EDGE B D
EDGE C D
```

**Figure 2. Example diamond-shaped workflow (left) and corresponding pegasus-mpi-cluster DAG file (right).**

## 3.5 Workflow Execution

When the PMC job starts, the master marks all workers as idle, parses the DAG file, and marks all tasks in the DAG that have no parents as ready. For each idle worker, the master chooses an arbitrary task, sends a message to the worker to assign the task, and marks the worker as busy. The master continues until there are no idle workers, or no more ready tasks. When a worker receives a task message from the master, it forks a process to execute the task, and sends the result back to the master when the process exits. When the master receives a result message from one of the workers, it marks the worker as idle, and processes the task. If the task succeeded, then it is marked as complete and its children are marked as ready. If the task failed, then it is either re-tried, or marked as a failure.

Currently, the master uses a simple round-robin scheduling approach to match tasks with workers on-demand. In the future we plan to investigate more sophisticated scheduling and load balancing techniques.

## 3.6 Failure Management

Many different types of errors can occur when executing a DAG. One or more of the tasks in the DAG may fail, the PMC job may

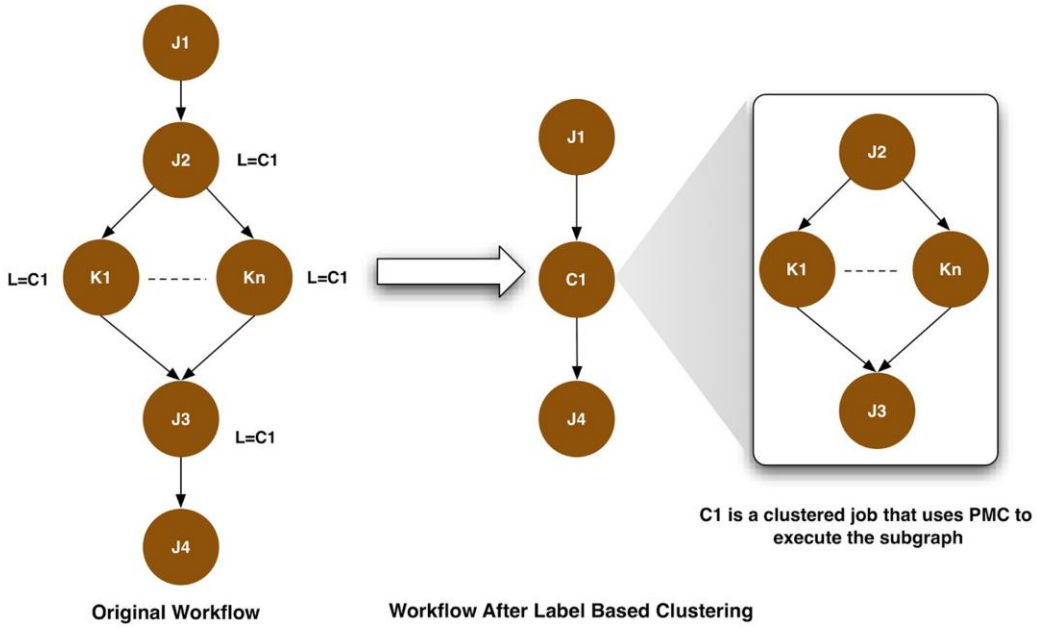



**Figure 3. Example workflow with jobs in a subgraph labeled with same label C1 and corresponding executable workflow after label based clustering (right).**

run out of wall time, the system may crash, etc. In order to ensure that the workflow does not need to be restarted from the beginning after an error, PMC generates a rescue file, which can be used to recover the state of a failed workflow when it is restarted.

The rescue file is a simple text file that lists all of the tasks in the workflow that have finished successfully. This file is updated and flushed to disk each time a task finishes so that it will always be up-to-date if PMC terminates unexpectedly. As such, the rescue file serves as a transaction log for the workflow.

A task is considered to have failed if it returns a non-zero exit code, which indicates that the program called exit with a non-zero value, or the program was killed because of an unhandled signal. When a task fails, PMC can automatically retry it several times. When all the retries of a task fail, then the task is marked as a permanent failure and none of its child tasks can be executed.

When one or more tasks fail all retry attempts, PMC will attempt to complete as much of the remaining workflow as possible before itself exiting with a non-zero exit code to indicate that the workflow has failed. When the workflow is retried, PMC is restarted using the rescue file to skip any tasks that have already been successfully completed.

## 3.7 Integration of pegasus-mpi-cluster into the Pegasus Mapper

The clustered jobs in the Mapper have been traditionally represented as a list of dependent jobs. For label-based clustering, the sub graphs are topologically sorted to ensure correct ordering of jobs when executed using pegasus-cluster. PMC allows the system to exploit the parallelism present in the structure of the subgraphs, when executing them as clustered jobs. To leverage this existing capability we have changed the Pegasus Mapper to represented clustered jobs as DAG's themselves. Additionally, we have introduced a pluggable interface in the Mapper that allows us to use different types of executables to run the clustered job. To integrate *pegasus-mpi-cluster* in Pegasus WMS we have implemented an interface in the Mapper that allows us to wrap a clustered job using *pegasus-mpi-cluster*. As a result, the user now has an option to control both how the workflow should be clustered (the technique to identify the clusters in the workflow) and how to execute the clustered job (sequentially using pegasus-cluster or in parallel using PMC).

Figure 3 shows a workflow where a subgraph (J2, J3, K1--Kn) is marked by labeling the jobs with the same label. The Pegasus Mapper when converting it into an executable workflow, uses label based clustering to cluster the jobs with the same label into a single job C1 (preserving the dependencies in the subgraph). The C1 job is wrapped with PMC for execution. Hence, when the C1 job is submitted during workflow execution to a compute resource, it appears as a single MPI job requesting *n* nodes and *m* cores. PMC then executes the tasks in the job using the master worker paradigm explained earlier. Figure 4 shows the end-to-end process of taking a user provided workflow, mapping it to run on XSEDE resources using PMC.

# 4. EXAMPLE APPLICATION: CYBERSHAKE ON NICS KRAKEN

As part of its research program of earthquake system science, the Southern California Earthquake Center (SCEC) has developed CyberShake [14], a software platform which uses 3D waveform modeling to perform probabilistic seismic hazard analysis (PSHA). PSHA provides a technique for estimating the

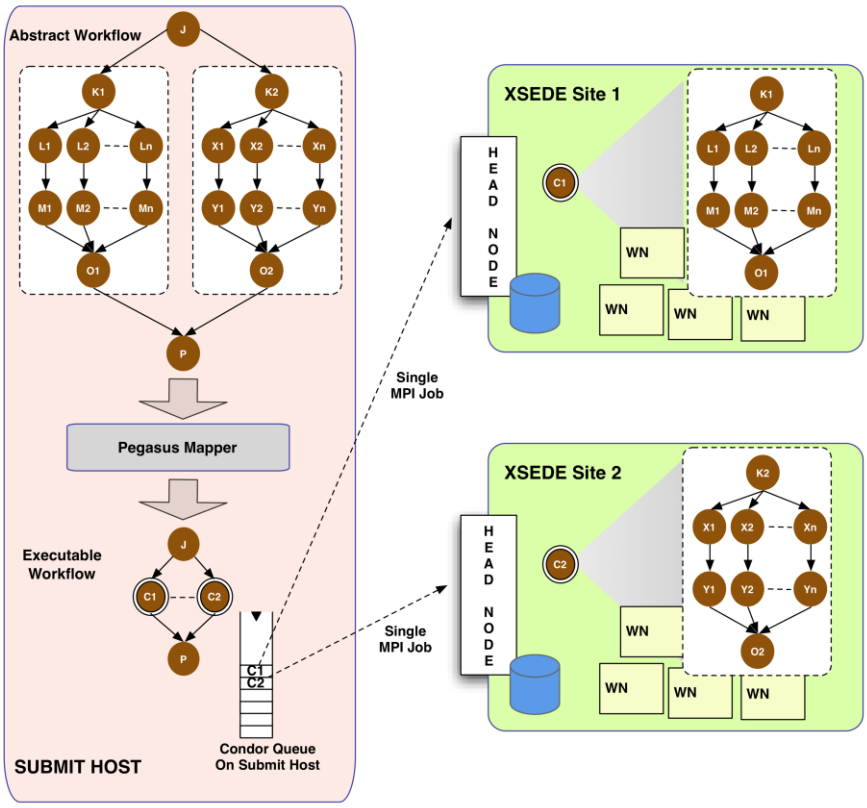



**Figure 4. Distributing a large, fine-grained workflow across cluster resources at different sites. The large workflow is partitioned into independent subgraphs, which are submitted as self-contained MPI jobs to the remote sites. The MPI job uses the master-worker paradigm to farm out individual tasks to worker nodes.**

probability that 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 basic CyberShake calculation produces PSHA information for a single location of interest. The results from multiple runs can be combined to produce a hazard map (see Figure 5), quantifying the seismic hazard over a region.

The current CyberShake processing stream can be divided into two phases. In the first phase, a mesh of approximately 2 billion elements is constructed, populated with seismic velocity information, and used in a pair of wave propagation simulations. In the second phase, individual contributions from over 400,000 different earthquakes are calculated and aggregated to determine the overall hazard. Each of these calculations is executed using short-running serial codes, but about 840,000 task executions are required for computing the hazard for a single location. The extensive computational requirements and large numbers of independent serial jobs necessitate a high degree of automation; as a result, CyberShake utilizes scientific workflows for execution.

Under Pegasus, CyberShake is set up as a set of hierarchical workflows. The wave propagation simulation phase consists of a few smaller jobs to prepare the inputs, and then two large parallel MPI jobs to the actual wave propagation simulation. Next are the seismogram workflows. The number of these can vary slightly but in this case there are 78 workflows, each one with over 10,000 tasks. Targeting PCM execution, each complete seismogram workflow was labeled with the same label to be executed with a

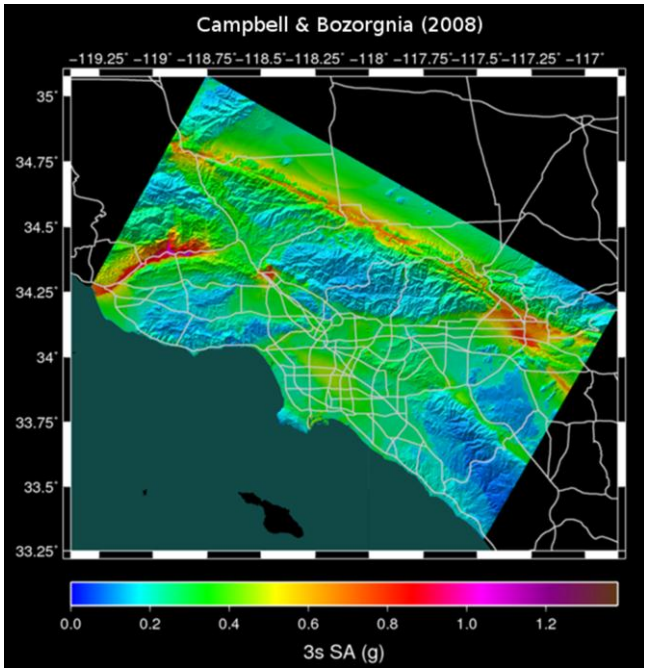



**Figure 5. UCERF2.0-based seismic hazard map of the Los Angeles area showing the geographical variation in peak spectral accelerations expected in the next 30 years when using Campbell and Bozorgnia (2008) attenuation**


**Figure 6. The CyberShake hierarchical workflow containing two wave propagation simulations (SGTGen-X, SGTGen-Y) and 78 seismogram workflows. The figure shows the workflows being mapped to 80 pegasus-mpi-cluster jobs for execution on NICS Kraken**

single pegasus-mpi-cluster instance. As a result, the execution sites would see 80 MPI jobs (two wave propagation simulations, plus 78 PMC jobs). Figure 6 provides an overview of the mapping.

CyberShake has been run successfully on a variety of systems and architectures using Condor Glideins to enable the efficient execution of the short-running serial jobs required by the application [5]. Since SCEC science goals for 2012 include 700 CyberShake runs, the CyberShake team selected NICS Kraken as a target resource due to its large core count and storage. Condor Glideins are unable to run on the XT5 architecture, but PMC enabled CyberShake processing to be performed on Kraken. The CyberShake workflow was partitioned by labeling each of the 78 post processing workflows as individual partitions. Each of these were executed using PMC using 40 core MPI jobs across 5 nodes. Each node has 12 cores, but we only used 8 to get a specific memory per core ratio required by the applications.

## 5. CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated that it is possible to efficiently execute large, fine-grained scientific workflow applications on petascale systems. Our approach involves partitioning workflows into independent subgraphs, and then executing each subgraph as a self-contained MPI job, leveraging the existing parallel execution capabilities of petascale systems. Using this approach, we have been able to run large CyberShake post processing workflows that contain hundreds of thousands of single core serial jobs on NICS Kraken.

We also found that our approach helped when interacting with compute resource staff. Our previous approach, Condor Glideins, usually required interaction with the staff to explain what the jobs were doing, and sometimes special setups like opening up firewalls were required. PMC is much easier to set up and requires no interaction with system administrators because it is based on standard MPI jobs and requires no special network configuration. As a result, domain scientists can more easily adopt this approach across a wide variety of existing systems.

The approach described in this paper is still in its initial stages and there are several areas we plan to investigate in the future. Two such areas are workflow graph partitioning and resource estimation. In order to execute workflows using Pegasus and PMC, the workflow DAG must be partitioned into subgraphs that

can be clustered together and the resource requirements of these subgraphs must be estimated in order to provide appropriate core count and wall time parameters for the MPI jobs. These parameters are critical because they have a significant impact on application performance and resource utilization. Currently the parameters are determined manually by the application developer based on experience and trial and error. In the future we plan to investigate graph partitioning techniques that can automatically identify subgraphs of the workflow that can be executed efficiently using PMC. We also plan to investigate resource estimation techniques that can automatically determine the number of cores and the amount of wall time to request for each subgraph based on the structure of the subgraph and estimates of the runtime of individual tasks.

## 6. ACKNOWLEDGEMENTS

We would like to thank the XSEDE user support staff and especially Matt McKenzie at NICS for helping us setup and debug the CyberShake workflows on Kraken.

Pegasus WMS is funded by the National Science Foundation under the OCI SDCI program, grant OCI-0722019. CorralWMS is funded by the National Science Foundation under the OCI-0943725 grant.

This research was supported by the Southern California Earthquake Center. SCEC is funded by NSF Cooperative Agreement EAR-0529922 and USGS Cooperative Agreement 07HQAG0008. The SCEC contribution number for this paper is 1636.

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

## 7. REFERENCES

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

[2] Augonnet, C. et al. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*. 23, 2 (2011), 187–198.

[3] Bosilca, G. et al. 2012. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Comput.* 38, 1-2 (Jan. 2012), 37–51.

[4] Bui, P. et al. 2011. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. *Workshop on Python for High Performance and Scientific Computing*. (2011).

[5] Callaghan, S. et al. 2011. Metrics for heterogeneous scientific workflows: A case study of an earthquake science application. *International Journal of High Performance Computing Applications*. 25, 3 (2011), 274 – 285.

[6] Casajus, A. et al. 2010. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*. 219, 6 (Apr. 2010).

[7] Chervenak, A.L. et al. 2009. The Globus Replica Location Service: Design and Experience. *IEEE Trans. Parallel Distrib. Syst.* 20, 9 (Sep. 2009), 1260–1272.

[8] Condor DAGMan (Directed Acyclic Graph Manager): *http://research.cs.wisc.edu/condor/dagman/*.

[9] Cosnard, M. et al. 2004. Compact DAG representation and its symbolic scheduling. *J. Parallel Distrib. Comput.* 64, 8 (2004), 921–935.

[10] Deelman, E. et al. 2005. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*. 13, (2005), 219–237.

[11] Fahringer, T. et al. 2005. ASKALON: a tool set for cluster and Grid computing: Research Articles. *Concurr. Comput. : Pract. Exper.* 17, 2-4 (2005), 143–169.

[12] Frey, J. et al. 2001. Condor-G: a computation management agent for multi-institutional grids. *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on* (2001), 55–63.

[13] Goux, J.-P. et al. 2000. An enabling framework for master-worker applications on the Computational Grid. *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on* (2000), 43–50.

[14] Graves, R. et al. 2011. CyberShake: A Physics-Based Seismic Hazard Model for Southern California. *Pure and Applied Geophysics*. 168, 3 (Mar. 2011), 367–381.

[15] Juve, G. et al. 2010. Experiences with resource provisioning for scientific workflows using Corral. *Sci. Program.* 18, 2 (2010), 77–92.

[16] Maeno, T. 2008. PanDA: distributed production and distributed analysis system for ATLAS. *Journal of Physics: Conference Series*. 119, 6 (2008), 062036.

[17] Moscicki, J.T. 2003. DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data. *Nuclear Science Symposium Conference Record, 2003 IEEE* (Oct. 2003), 1617 – 1620 Vol.3.

[18] Oinn, T. et al. 2006. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*. 18, 10 (2006), 1067–1100.

[19] Perez, J.M. et al. 2008. A dependency-aware task-based programming environment for multi-core architectures. *Cluster Computing, 2008 IEEE International Conference on* (Oct. 2008), 142 –151.

[20] Raicu, I. et al. 2007. Falkon: a Fast and Light-weight tasK executiON framework. (2007).

[21] Rynge, M. et al. 2011. Experiences Using GlideinWMS and the Corral Frontend across Cyberinfrastructures. *E-Science (e-Science), 2011 IEEE 7th International Conference on* (Dec. 2011), 311 –318.

[22] Sfiligoi, I. et al. 2009. The Pilot Way to Grid Resources Using glideinWMS. *Computer Science and Information Engineering, 2009 WRI World Congress on* (2009), 428–432.

[23] Singh, G. et al. 2008. Workflow task clustering for best effort systems with Pegasus. *Mardi Gras Conference'08* (2008), -1–1.

[24] Taylor, I. et al. 2005. Visual Grid Workflow in Triana. *Journal of Grid Computing*. 3, 3 (Sep. 2005), 153–169.

[25] Zhao, H. and Sakellariou, R. 2007. Advance reservation policies for workflows. *Proceedings of the 12th international conference on Job scheduling strategies for parallel processing* (Saint-Malo, France, 2007), 47–67.