SPECIAL ISSUE PAPER

HUBzero and Pegasus: integrating scientific workflows into science gateways

Michael McLennan¹, Steven Clark¹, Ewa Deelman²,*†, Mats Rynge², Karan Vahi², Frank McKenna³, Derrick Kearney¹ and Carol Song⁴

¹HUBzero Platform for Scientific Collaboration, Purdue University, West Lafayette, IN, USA
²Information Sciences Institute, University of Southern California, Marina del Rey, CA, USA
³University of California, Berkeley, CA, USA
⁴Purdue University, West Lafayette, IN, USA

 SUMMARY

In this paper, we described the benefits and the challenges of integrating existing scientific workflow technologies into science gateways. Scientific workflow managers are powerful tools for handling large computational tasks. Domain scientists find it difficult to create new workflows, so many tasks that could benefit from workflow automation are often avoided or performed by hand. Two technologies have come together to bring the benefits of workflow to the masses. The Pegasus Workflow Management System can manage workflows comprised of millions of tasks, all the while recording data about the execution and intermediate results so that the provenance of the final result is clear. The HUBzero platform for scientific collaboration provides a venue for building and delivering tools to researchers and educators. With the press of a button, these tools can launch Pegasus workflows on national computing infrastructures and bring results back for plotting and visualization. As a result, the combination of Pegasus and HUBzero is bringing high-throughput computing to a much wider audience. Copyright © 2014 John Wiley & Sons, Ltd.

Received 8 November 2013; Revised 11 February 2014; Accepted 11 February 2014

KEY WORDS: computation; workflow; collaboratories; user interfaces; automation

1. INTRODUCTION

Computation has brought about a revolution in science. As software was created to analyze large amounts of data, researchers found they could gather even more data, which necessitated even more software and more computing power. Today, many analysis tasks require not just a single program or computer but dozens of different programs running across thousands of computational nodes. Handling any one of those tasks requires a complex orchestration of moving data to appropriate nodes, finding or staging the executables, starting jobs, handling data flow dependencies, registering data products, and overcoming job failures.

Over the past decade, several frameworks have been created to support the execution of such large computational tasks. The Pegasus Workflow Management System (Pegasus WMS) [1] and its workflow engine, the directed acyclic graph manager (DAGMan) within HTCondor [2], were built to manage thousands of jobs in a high-throughput computing environment. Taverna [3] graphically connects bioinformatics Web services together into a coherent flow. Kepler [4] also provides

*Correspondence to: Ewa Deelman, Information Sciences Institute, University of Southern California, Marina del Rey, CA, USA.
†E-mail: deelman@isi.edu

Copyright © 2014 John Wiley & Sons, Ltd.
The Pegasus WMS (or Pegasus) manages the execution of scientific workflows on desktops, private clusters, campus clusters, grids, and academic and commercial clouds. It automatically locates the necessary input data and computational resources needed for workflow execution; thus, it allows scientists to specify their workflows at a high-level of abstraction (devoid of resource information) and then maps/plans this description onto the available execution resources. Next, Pegasus reliably executes this plan. As a result, Pegasus workflows are easy to compose and are portable across heterogeneous cyberinfrastructure.

Pegasus is capable of executing workflows on a single resource or across resources. Even a single workflow description can be distributed and executed across the cyberinfrastructure. Pegasus has been used to run workflows ranging from just a few computational tasks up to millions.

In order to support the abstract workflow specifications, which let scientists concentrate on their science rather than on the operational aspects of the cyberinfrastructure, mapping technologies are needed to automatically interpret and map user-defined workflows onto the available resources. The workflow mapping process involves finding the appropriate software, data, and computational resources required for workflow execution. The mapping process can also involve restructuring the workflow to optimize performance and adding transformations for data management and provenance.
information capture. DAGMan, Pegasus’s workflow engine, relies on the resources (compute, storage, and network) defined in the executable workflow to perform the necessary actions. Individual workflow tasks are managed by a task scheduler (HTCondor), which supervises their execution on local and remote resources.

When errors occur, Pegasus tries to recover when possible by retrying tasks, by retrying the entire workflow, by providing workflow-level checkpointing, by remapping portions of the workflow, by trying alternative data sources for staging data, and, when all else fails, by providing a rescue workflow containing a description of only the work that remains to be performed [1]. Thus, it tries what it can to shield the user from errors and intermittent issues with the computing resources. It cleans up storage as the workflow is executed so that data-intensive workflows have enough space to execute on storage-constrained resources [12]. Pegasus keeps track of what has been performed (provenance) including the locations of data used and produced and which software was used with which parameters [13, 14].

3. HUBS FOR SCIENTIFIC COMMUNITIES

3.1. Building a community on nanoHUB.org

In 2002, the US National Science Foundation created the Network for Computational Nanotechnology (NCN), a collection of universities engaged in simulation and modeling for the nanotechnology community [15]. NCN established a web presence at nanoHUB.org and offered an array of simulation and modeling tools, along with seminars, tutorials, courses, and other supporting materials. Over the years, nanoHUB.org evolved into a software platform supporting collaboration in private group areas, discussion forums, ratings and reviews, and, most importantly, deployment of new simulation tools via its unique middleware. In 2007, the underlying software for nanoHUB.org was spun off as a separate project, the HUBzero® Platform for Scientific Collaboration [16]. Since then, it has been used to create similar hubs for many other scientific areas—all supported by the same underlying middleware and HUBzero’s content management system, which has been specifically designed for many different types of scientific content.

nanoHUB grew from 10 tools and 1000 users in 2002 to more than 330 tools and 300,000 users today [17]. User contributions were key to that growth. Each new tool, seminar, or tutorial brought new users to the site, some of whom contributed their own new content. This virtuous cycle was fueled by an open content management system that enabled users to upload and deploy their own content according to their own schedule, with minimal intervention from the nanoHUB team.

Uploading and publishing a single document or PowerPoint presentation is one thing; uploading and deploying a simulation tool is quite another. Tools have potentially hundreds of individual source code files; they must be compiled and tested within the hub environment; and all of this must be performed by untrusted users with minimal intervention. The HUBzero infrastructure was designed to support this activity. Researchers upload their code into private project areas with a Subversion [18] repository for source code control; they compile, test, and modify their code within a secure execution container managed by OpenVZ [19] running on a cloud of execution hosts; they approve tools for deployment via a Web-based content management system. All of this enables their code to be separated from the system and from other users and yet deployed in a secure and scalable manner.

End users find a tool on a hub and press a button to launch it. The tool is launched in a Linux/X11 environment within a secure container on a ‘cloud’ of execution hosts, and the graphical output from the session is piped through the Web to the user’s browser via Virtual Network Computing (VNC) [20]. The result is a live, interactive simulation tool running in a secure environment, capable of running workflows on national computing infrastructure, and accessible to any user via a Web browser without any special software installation.

There are many other gateway frameworks, including the Distributed Application Runtime Environment [21], Open Grid Computing Environments [22], and WS-PGRADE/gUSE [5], but none provide direct access to interactive tools and scientific visualization. This is a distinct advantage of the HUBzero platform.
There are many different users that work together within each hub community, and their interactions are mediated by the HUBzero platform. A hub owner creates a new hub and engages others in the community to seed the site with initial content. Users browse the site and take advantage of interesting content, perhaps by watching a seminar or running a simulation tool. Users also contribute their own content to the site, including new workflows and simulation tools. Hub administrators help to compile and stage new simulation tools and ultimately approve all content before it is released.

3.2. NEES.org and OpenSees laboratory

Other hubs leverage the same HUBzero infrastructure to support different toolsets for their own community. In 2009, the National Science Foundation (NSF) George E. Brown Network for Earthquake Engineering Simulation (NEES) moved operations to Purdue and created a hub for the civil engineering community. Today, NEES.org offers more than 70 simulation and data analysis tools used to understand the damage caused by earthquakes and improve building design. One of these tools leverages an open source code, the Open System for Earthquake Engineering Simulation (OpenSees) [23], to provide a collection of utilities for structural and geotechnical engineers.

One of the utilities within this OpenSees Laboratory tool [24] is the Moment Frame Earthquake Reliability Analysis. This tool is intended to demonstrate to engineers the importance of uncertainty in their models and the need to use probabilistic methods when determining important response measures. The tool prompts the user for the material properties including probability distributions, building layout, and then a list of earthquake ground motion records. On submission, the tool automatically builds and executes a Pegasus workflow. The workflow manages several hundred to tens of thousands of separate OpenSees jobs on the OSG. Together, these jobs perform a reliability analysis of the building frame under seismic loads.

Users access this capability through a simple interface via a Web browser, but the underlying infrastructure is much more sophisticated, as shown in Figure 1. The GUI for the OpenSees Laboratory tool runs on one of the execution hosts managed by the HUBzero middleware on NEES.org. The interface runs under X11 within a secure OpenVZ container, and an image of that session is projected over the network back to the user’s Web browser via VNC, as described earlier. When the user presses a button to launch the analysis, the tool, running within the secure container, executes a HUBzero command called ‘submit’ to launch the Pegasus workflow. The submit command (described in detail in Section 4) connects to a job submission server running outside the container, which launches Pegasus, which in turn launches jobs on OSG and monitors their progress. Pegasus reports progress back to the submit server, which sends progress updates back to the GUI, ultimately reaching the user via the Web browser. Once the analysis is complete, Pegasus transfers data from OSG back to the hub, the submit command returns control to the GUI, and the interface loads the results and presents plots to the user.

The resulting plots show the variation in roof displacement, interstory drift, and base shear given the uncertainty in the materials for each of the earthquake motions specified. A single dot plotted in red for
each earthquake ground motion shows the engineer what would happen if only the mean value had been used for the material properties for that earthquake. Other dots on the plot show the variation in performance, as shown in the output displayed in Figure 1. The advantage of using Pegasus and OSG for such a workflow is that Pegasus recognizes that most of the computation can be performed in parallel, and the OSG permits concurrent job submission and execution. As a consequence, the resulting time spent obtaining these results can be a few minutes or hours instead of many hours, weeks, and even months that would be required if the results were obtained from a series of sequential analyses.

To date, OpenSees Laboratory has served 1186 simulation users from the US, China, Canada, Italy, Japan, and other countries all over the world. All of them are driving OpenSees through a GUI, perhaps without even knowing if and when they are launching complex workflows.

3.3. DiaGrid.org and various applications

DiaGrid is a high-throughput computing resource utilizing the HTCondor system, with more than 50,000 cores from machines at Purdue University and nine other campuses. Since its inception in 2005, DiaGrid has handled more than 100,000,000 simulation jobs.

In 2011, Purdue established a hub at DiaGrid.org to provide more widespread access to the underlying DiaGrid execution pool. DiaGrid.org hosts a growing number of tools that use Pegasus workflows to simplify large computational tasks.

One such tool, called BLASTer [25], can be used to run Basic Local Alignment Search Tool (BLAST) [26], a bioinformatics tool for analyzing DNA sequence data. Users upload a query file containing one or more nucleotide sequences and search against standard databases from the National Center for Biotechnology Information (NCBI) or against their own custom databases. The run for a single sequence may take anywhere from 10 s to 30 min. But many researchers bundle thousands of searches into a single request. Performed linearly, such a search might take several days or weeks of computation. BLASTer automatically divides the query file into chunks and creates a Pegasus workflow to distribute the search over hundreds of DiaGrid nodes, so the end-to-end execution time can be shortened from weeks to a matter of hours. Shared storage allows the NCBI standard databases, updated regularly, to be available to all users without requiring file transfer, as shown in Figure 2, thereby further reducing the time to results. So far, 125 users have accessed this tool, and in the process, they have run more than 1,300,000 individual BLAST jobs.

Another tool, called Single Particle Cryo-EM 3-D Reconstruction, is based on the image processing routines in EMAN/EMAN2 [27]. This tool takes a series of transmission electron microscopy images of a nanoparticle or biological macromolecule frozen within a medium, and it applies image processing routines to the various viewpoints to reconstruct the 3-dimensional (3D) shape of the particle. This requires a complex workflow with 24 processing tasks, most of which can be managed automatically by Pegasus. In the first step, particles are selected via an automated method and manually screened by the user. The whole data set is then split into even and odd halves, and each half is independently processed. In each half, initial models are built using different methods, and

Figure 2. The BLASTer tool on DiaGrid.org manages the execution of Basic Local Alignment Search Tool (BLAST) programs on DiaGrid. National Center for Biotechnology Information (NCBI) sequence databases are kept up to date on shared storage, so they do not need to be transferred for each run.

results are compared to find candidate particles with consistent positions and orientations. Particles are further refined for scale, astigmatism, and focus parameters. Results from the two halves are combined to compute a Fourier Shell Correlation curve and determine a resolution where that curve becomes worse than some threshold. The results from the two halves are then merged to compute a 3D density map from the combined dataset. The final map is then sharpened and normalized. This entire process requires significant computational power. For example, a high-resolution refinement of the final result using 1000 to 10,000 images (each 400×400 to 1000×1000 pixels) typically requires hundreds of cores and many hours from start to finish.

The Cryo-EM tool shows the real power of DiaGrid—not only to deliver cycles but also the ability to combine that computational power with interactive tools and intermediate visualizations. Throughout the simulation process, this tool helps the user upload and organize the various transmission electron microscopy images for an experiment and draw boxes on the images to make particle selections. It computes an initial contrast transfer function used for image analysis and allows the user to view and tweak that function. It then launches the Pegasus workflow, monitors progress, and creates visualizations of the final results. So far, 100 users have accessed this tool and performed more than 1800 3D reconstructions.

DiaGrid.org also offers a tool for statistical analysis scripts written in the R programming language [28]. It is quite common for researchers to run a particular script over a wide range of input parameters to explore the parameter space or to perform a sensitivity analysis. The SubmitR tool [29] takes a script, data files, and a specification of parameter values, and automatically builds and executes the Pegasus workflow to run a large number of R jobs in parallel on DiaGrid. Again, this appeals to the domain scientist who is quite familiar with R, but knows nothing about workflows, HTCondor, or DiaGrid. So far, 50 users have accessed this tool and performed more than 41,000 simulation runs.

4. CREATING SCIENTIFIC WORKFLOWS

4.1. Job submission via ‘submit’

The integration of Pegasus and HUBzero certainly makes it easier not only for end users to launch workflows but also makes it easier for tool developers to build and test new workflows. Each hub comes with one tool called a Workspace, which is a full-featured Linux desktop accessible via the Web. Tool developers use this to upload, compile, and test their code before deploying a tool. Experienced researchers also use this to get past the constraints of a graphical interface—to edit files, write simple scripts, and dispatch computational jobs.

HUBzero provides a command called ‘submit’, which is used within the Workspace to dispatch jobs, as shown in Figure 3. In this example, the command line ‘spice3 –b circuit’ represents the research code that is being executed. Electrical engineers will recognize this as the SPICE3F4 program created back in the 1970’s at UC Berkeley and still in use today [30]. But the approach we describe would apply to any such research code amenable to command line execution. The command spice3 is the executable, the flag –b means to execute in batch mode, and the file circuit contains the netlist representing the electrical circuit being simulated.

(a) spice3 –b circuit
(b) submit –v DiaGrid spice3 –b circuit
(c) submit –p @res=100,1k,10k spice3 –b @:circuit
(d) submit –p @res=100,1k,10k –p @cap=1u,10u,100u spice3 –b @:circuit
(e) submit –d indata.csv spice3 –b @:circuit
(f) submit –p @num=1:100 spice3 –b circuit@num
(g) submit –p @file=glob:circuits* spice3 –b @file
(h) submit pegasus-plan --dax myworkflow.dax

Figure 3. HUBzero’s submit command makes it easy to send jobs off to remote sites and includes support for parameter sweeps and Pegasus workflows.
When invoked directly on the command line as in Figure 3(a), the \texttt{spice3} program runs locally within the execution host managing the tool session (Figure 1). However, prefixing the usual command line with \texttt{submit} will send the job off to remote execution venues, such as DiaGrid or the OSG. The `-v DiaGrid' arguments shown in Figure 3(b) request that the job be sent to DiaGrid. The \texttt{submit} command will automatically transfer files such as \texttt{circuit} that are required for simulation. It will speak the appropriate protocol—currently Condor, Condor-G, Load Leveler, LSF, SGE, PBS, or SLURM—to queue the job. And, it will monitor progress and transfer results back to the execution host once the run is complete. It runs all jobs with a common credential owned by the hub, so individual users need not have an account or any credentials for the remote venue. The hub tracks all jobs, including those run on remote venues, so that if a security incident arises, a complete report can be generated for any affected venue. The same accounting is used to produce monthly reports of the overall simulation usage for each hub.

4.2. Simple workflows: parameter sweeps

The \texttt{submit} command can also manage parameter sweeps with a very simple command line, as shown in Figure 3(c). The -p flag indicates that the next argument is a parameter specification. In this case, the parameter \texttt{@@res} is taking three values: 100, 1 k, and 10 k. The @: prefix before the \texttt{circuit} file indicates that this file should be processed with parameter substitutions. In other words, the \texttt{submit} command treats “\texttt{circuit}” as a template file and substitutes all occurrences of \texttt{@@res} with a particular parameter value. In this case, it produces three separate input files—one for each of the parameter values. Then, it automatically builds a Pegasus workflow to manage the jobs and dispatches the workflow off to an appropriate venue. Because the optional -v argument was not specified in this case, the \texttt{submit} command will automatically select a venue that supports Pegasus job submission and data transfer interfaces, such as DiaGrid or OSG.

A simulation run may have multiple parameters. In Figure 3(d), the parameter \texttt{@@res} is swept through three values, and the parameter \texttt{@@cap} is swept through three values. The \texttt{submit} command creates nine different input files representing all possible combinations of these two parameters, builds a Pegasus workflow to manage the jobs, and then dispatches the workflow to an appropriate venue.

Instead of specifying input parameters on the command line, the values for all combinations of parameters can be stored in a comma-separated value file. In Figure 3(e), the arguments `-d \texttt{indata.csv}' indicate that parameter values should be loaded from the file \texttt{indata.csv}. The first line should contain the parameter names for the columns, such as `@@res, @@cap' in this example. Each of the remaining lines represents the parameter values for a single job, such as `100,1u' or `10 k,100u' for this example. Passing parameters in this way is useful when the desired values are not a simple combination of values, but perhaps a series of values generated by a genetic optimization algorithm, or by a Latin Hypercube sampler.

Instead of relying on template files and parameter substitutions, the user may generate a series of separate input files. For example, suppose a user has 100 different circuit files to evaluate. The syntax shown in Figure 3(f) supports this scenario. The arguments `-p @@num=1:100' define a parameter \texttt{@@num} that takes on integer values from 1 to 100. In this case, the circuit input file is specified as \texttt{circuit@@num}, so the \texttt{submit} command builds a Pegasus workflow with 100 different command lines, substituting the value of \texttt{@@num} for each job to produce commands like \texttt{spice3 -b circuit1}, \texttt{spice3 -b circuit2}, and so forth, referencing the series of numbered files that the user created by hand for this run.

Input files can also be recognized using glob-style pattern matching. The arguments `-p @@file=glob:circuit*' in Figure 3(g) define a parameter \texttt{@@file} that takes on all values matching the glob-style pattern \texttt{circuit*}. With this syntax, the files could be distinguished by different numbers, different letters, or other naming conventions, but all files that \texttt{submit} can find matching \texttt{circuit*} will be simulated as separate cases, with \texttt{@@file} representing the whole name of each matching file, such as \texttt{circuit1}, \texttt{circuit-2b}, \texttt{circuitXYZ}, etc.

4.3. Customized workflows in Pegasus

More experienced users can learn how to create their own customized Pegasus workflows. Each workflow is expressed as a high-level directed acyclic graph (DAG) called a directed acyclic graph...
in XML (DAX) description. Each node in the DAX represents a job in the workflow, and the edges between nodes represent data files that flow from one job to the next. The DAX is similar to the HTCondor DAGMan file [31] but at a higher level of abstraction so that it can be targeted to different execution environments.

Each DAX is ultimately expressed as an XML file. Writing that file by hand, however, can be tedious and impractical. Many developers prefer to express their workflow as a small program written in Python, Java, or Perl. Pegasus provides Application Programming Interfaces (APIs) in these languages to help construct the needed XML. The workflow designer needs to build up the nodes and edges of a DAX object, and the API is used to write out the object in XML format. Running the program, therefore, produces a DAX file expressing the entire workflow, suitable for submission.

The submit command described earlier is also used to launch Pegasus workflows, as shown in Figure 3(h). In this case, the programs being launched (such as `spice3`) do not appear on the command line; instead, they are encoded within the DAX description of the workflow, which in this example is a file named `myworkflow.dax`. The submit command invokes another program called `pegasus-plan`, which reads the DAX, ‘compiles’ the workflow for a specific execution venue, then launches the workflow and manages execution.

For example, the Moment Frame Earthquake Reliability Analysis tool, discussed in Section 3.B, uses both the Python API provided by Pegasus and the submit command. At the launch of the simulation by the user, the tool invokes a shell script that first invokes a Python script (`opensees-dax.py`) to create the XML file and then invokes the submit command using the `pegasus-plan` program and the XML file to run the workflow on the OSG, as follows:

```
#!/bin/sh
python opensees-dax.py --nMat=$1 --nMotion=$2 > dax.xml
submit pegasus-plan --dax dax.xml
```

The workflow compilation or ‘planning’ step is the strength of the Pegasus approach. It optimizes the workflow by removing any branches where partial results are in place and where dependencies are already satisfied. It also augments the workflow, adding tasks to stage data files in/out of remote resources, and to clean up unnecessary results after execution. And, it can partition the workflow and cluster short-running jobs together, so that they run more efficiently on a single compute node. It does all of this at a high-level, freeing the user from having to request or identify specific resources. During execution, Pegasus dispatches jobs, monitors progress, and automatically retries failing jobs so that sporadic failures do not derail the entire workflow. If a workflow does fail (for example, if the venue goes down, or if a particular job triggers a bug in the code), then it can be corrected and restarted by the user. Pegasus will recognize partial results and pick up where it left off to complete the workflow.

5. BUILDING AND DEPLOYING APPLICATIONS

5.1. Graphical user interfaces and the Rappture toolkit

Researchers can build their own GUIs on top of the basic submit capability, so that other (perhaps less experienced) users can access a simplified interface and launch powerful workflows for targeted applications, as described earlier in Section 3. Such tools can be created using MATLAB®, Java, Qt, GTK+, wxWidgets, or any other toolkit that runs under Linux/X11. For example, the interfaces for BLASTER and SubmitR were both created using Java and Python, respectively.

HUBzero’s Rappture toolkit provides an easy way to create such graphical interfaces. Rappture comes preinstalled within the Workspace tool and includes an interactive ‘builder’ application for new tools. Researchers specify the inputs and outputs for each new tool by dragging objects from a palette of available controls and by dropping them into an interface specification tree. Rappture supports more than 20 objects, including Boolean values, integers, real numbers, choice options, images, x–y curves, finite element meshes, scalar/vector fields, molecules, and even sequences of
objects over time or some other quantity. A sequence of images over time, for example, acts like a movie.

Each object is given a label, description, a default value, and other attributes, such as units of measure for numeric values. The builder provides a preview of the resulting interface and generates errors and warnings if any elements are missing or incomplete. The builder saves each interface in an XML description file, and it can also generate a skeleton for the main program of the tool in a variety of programming languages, including C/C++, Fortran, MATLAB, Octave, Java, Python, Perl, R, Ruby, and Tcl/Tk. The researcher modifies the body of the skeleton code to include the core of the tool, then runs the ‘rappture’ command to produce the graphical interface.

The graphical interface acts as an intermediary for the user, gathering input values, launching simulation runs, and plotting results. Remote jobs and workflow submissions are handled via something like the C language system() function, which forks and executes a separate shell process to handle a command line string with the submit command shown earlier in Figure 3. Scripting languages are particularly well suited for this task, so it is quite common to build an interface tool in Python, for example, which calls submit to dispatch a complex workflow built from a series of C/C++ and Fortran programs.

Rappture also includes a regression testing tool for ongoing verification. When the first version of a tool is complete, output from a series of runs can be gathered together to form a test suite. From then on, as the tool is modified to fix bugs and implement new features, the Rappture regression testing tool will run the latest version through all cases defined in the test suite, and compare the latest output to the original (golden) output. If the latest tool deviates from the expected behavior, Rappture highlights the failed test and plots the differences. The tool developer can then examine the difference and decide whether a bug has been introduced, or whether the latest code is supposed to behave differently. If the difference is expected, output from the latest run can be made the new golden standard for all future tests. This process keeps unintended differences from creeping into a simulation code and greatly improves tool quality.

Most tool developers think of Rappture as a graphical interface builder, but it is also a data exchange standard. The XML specification used to describe the inputs and outputs of a tool are akin to a Web service description. Indeed, Rappture tools can be executed without the graphical interface simply by launching the underlying program with an appropriate XML-based input file and by extracting results from the XML-based output file. That is precisely how the Rappture regression tester is implemented. Right now, Rappture tools run within secure OpenVZ containers, and the only way to interact with the tool is through the graphical interface. In the future, the HUBzero platform will be extended to support Web service access for all Rappture-based tools, so that the analysis capabilities of published tools can be leveraged by external Web sites and Web technologies, including WebGL-based visualization tools and mobile apps.

5.2. Tool publication process

HUBzero includes its own powerful content management system for uploading and deploying many different types of content, including seminars, tutorials, teaching materials, and, most importantly, computational tools. Any researcher within a particular hub community can click on the Upload link, fill out a form describing their new tool, and obtain immediate access to a private project area complete with a Subversion source code repository and a wiki area for project notes. The researcher can launch the Workspace tool, check out the latest source code, and compile and test within the workspace environment. The researcher can build Pegasus workflows and test them right at the command line by typing the command shown earlier in Figure 3(h). Once that part is working, the researcher can create a graphical interface using Rappture or any other toolkit and embed the same submit command within the code of their new tool.

Once the tool is working properly, the researcher can visit the page representing their tool contribution and click a link saying, ‘My code is committed, working, and ready to be installed’. This signals the hub administrators to check out the latest code, compile it, and install it in the official/apps directory under a subdirectory with the tool name and its revision number from the source code control system.
Once a tool is installed, the researcher receives a message to begin testing. Clicking a *Launch Tool* button brings up a preview of the final tool, exactly as other users of the hub would see it. Once the tool has been verified, the researcher can click a link saying, ‘*My tool is working properly. I approve it*’.

That prompts the hub administrators to take one last look, to verify that the tool is indeed functional and has an adequate description page, and then move the tool to the ‘published’ state, where it becomes available to other users according to the restrictions set by the tool authors. A tool can be open to the world, or protected by export control, or accessible only by a certain group of users. If a tool is published as Open Source, an archive of the source code is presented for download alongside the *Launch Tool* button, and the terms of the Open Source license are clearly displayed on the tool page. All tools have a unique digital object identifier, so they can be cited within academic publications.

### 5.3. Ongoing tool support

The publication of a tool is not the end of the story. Other users may encounter a bug in the tool and file a support ticket on the hub. Any ticket filed on a live tool session page is automatically routed to the tool authors and becomes their responsibility to fix. The authors receive email about the ticket, and they can see it listed on the *My Tickets* module within their *My Account* dashboard page. They can update the ticket to communicate with the user, check out the code within a *Workspace*, commit a fix, and close the ticket. When the code is ready for another release, the authors click a link saying, ‘*I’ve made changes Please install the latest code for testing and approval*’. This takes the tool back to the ‘updated’ state, causing hub administrators to stage the latest version, letting the tool authors test and approve the latest version, leading to an updated tool publication.

Each tool also has its own question/answer forum for community discussion. Another user might not understand the physics within the tool and post a question. One of the tool authors, or perhaps another user, might follow up with an answer. Researchers can have threaded discussions within the question/answer area, and they earn points for participation. The points can be redeemed for merchandise on some hubs, traded for other favors, or used as bragging rights.

End users may like a tool, but may have requests for improvement, such as support for extra model parameters or material types. They can post such ideas on the wish list for each tool. Other users can vote the ideas up or down. The tool authors can evaluate each wish on a scale of importance and effort, so that wishes can be sorted in order of priority. Ideas that are deemed important and low effort bubble up to the top of the list, while those that are unimportant and high effort sink to the bottom. When any wish is granted, the original user is notified, and the tool author may earn points for fulfilling the request.

Tool authors also earn points each month for the tool itself, depending on how the tool is valued by the community. Users can submit their own ratings of tools on a scale of 1 to 5 stars, with comments to support their viewpoint. Other users can appreciate helpful reviews by pressing a ‘thumbs up’ button. Community reviews are combined with other factors to compute an overall quality score called the ‘ranking’ on a scale of 0 to 10. Tools with good reviews, strong usage, and very few outstanding issues have a ranking of 10, and tool authors each earn that many points every month as an ongoing royalty for that tool. Tools with poor reviews, low usage, and many unresolved bugs have a ranking closer to 0 and therefore earn very few (or 0) points.

### 6. INTEGRATION CHALLENGES AND RESULTING DEVELOPMENT OPPORTUNITIES

As in any integration work, the HUBzero/Pegasus integration required a set of small code changes. For example, functionality was added to enable better feedback and workflow progress information in the user interface and to provide workflow metrics to the HUBzero metrics framework about how the workflow ran. Furthermore, the Pegasus team used the integration work to collect requirements, and as a driver for some major data management changes in Pegasus. When the integration effort started, the most recent Pegasus release was version 3. The data management in that version was still based on a model of high-performance computing (HPC) systems, with assumptions such as a shared parallel file system being mounted on the compute nodes used for the workflow. In this
model, the resource (the actual cluster) had to be chosen up-front by either the hub or the end user. This works well for MPI-type of workflows that are targeting specific HPC resources. In the case of high-throughput workloads, which are the more common workloads in hubs, and for which it does not matter where the jobs run, selecting a resource was not a very user-friendly solution. The user would not have enough information about the resources to make an informed decision. At any given moment, a particular resource might be heavily loaded or broken, while other venues might be working fine and provide a much better resource for the workload. A more user-friendly method would be to let the jobs run anywhere and across any available resources.

The requirements of handling high-throughput workloads and running those workloads across a set of resources simultaneously with late binding lead to the release of Pegasus 4.0. In addition to the already existing HPC shared file system model, two new data management approaches were added. One of the new approaches was to use HTCondor I/O for file transfers, and another approach was to use a shared staging storage element (such a GridFTP server [35] or Amazon S3 server [36]) for storing intermediate data products. This enabled workflows to span multiple compute resources with the data coordinated at a central point.

6.1. Details of the new Pegasus data management models

When creating a workflow, the user first creates catalogs that contain information about the data files and transformations used by the workflow (replica catalog and transformation catalog) as well as information about the available computing resources (site catalog). Then the user provides an abstract representation of their workflow as a DAG. Pegasus Mapper, also known as the Pegasus Planner, takes in this abstract workflow (composed of tasks) and generates an executable workflow (composed of jobs) based on information about available compute resources. During this mapping process, Pegasus determines where the jobs and data should be placed. Pegasus queries a replica catalog to discover the locations of the input files performing replica selection in case of multiple file locations. It then adds data stage-in and stage-out jobs to the executable workflow, which are responsible for staging in the input data for the workflow from the input storage site and staging out the output data to the output storage site. The advantage of adding separate data stage-in and stage-out jobs to the executable workflow is twofold:

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

The Mapper can also add cleanup jobs [32, 33] to the workflow that delete intermediate files when they are no longer required, and it can remove jobs from the workflow for which outputs already exist (data reuse).

Figure 4 illustrates the data flow for a simple two-task workflow on a single HPC cluster with a shared file system. In this case, file F.in is transferred by the data stage-in job to the shared file system of the cluster. Task T1 starts on node C1, reads the input file F.in from the shared file system, and writes the intermediate file F.int to the shared file system. Task T2 then starts up on node C2, reads F.in (created by T1) from the shared file system, and writes its output file, F.out, to the shared file system. Pegasus then transfers F.out to the output site. This approach worked particularly well in the traditional supercomputing environment such as XSEDE with fast parallel file systems. However, this static binding of jobs and data at mapping time made it hard to support late binding of jobs, where the execution location is determined at runtime according to the resource’s policy and configuration.

With the new data management models, the Mapper still adds data management jobs for the executable workflow but these jobs are not tied to an execution site. Instead, the data is transferred to a data staging site associated with the compute site. The data staging site is defined as the intermediate data storage site used to manage data for a workflow. It is used to store the input and all intermediate data generated during a workflow execution. The data staging site may or may not
be the same as the cluster where the jobs execute. We also introduced Pegasus Lite [37], a semi-
autonomous lightweight execution engine for Pegasus tasks (with or without dependencies), running
as part of the compute job, on remote worker nodes and in cloud virtual machines. Pegasus Lite
receives a set of tasks to manage on the remote resource and is responsible for managing their
execution and data needs.

The notion of a separate data staging site and the introduction of Pegasus Lite enabled workflows to
span multiple compute resources. Today, most of the hub workflows are using a simple site catalog
describing either DiaGrid or OSG as the target compute resource, and no jobs are bound to
particular resources of those grids.

Another improvement in the Pegasus 4.X series, but which is not yet used much in the hub
environments, is to leverage the fact that in the OSG it is common for the compute sites to have
Squid caches [38] available to the jobs. This meant that workflows were able to stage data to the
staging storage element using Storage Resource Manager (SRM) [34] or GridFTP [35], and stage
data out over HTTP. This allowed the Pegasus Lite wrapped compute jobs to automatically use the
Squid caching mechanism provided by the sites, when pulling in data to the worker nodes over
HTTP. In order to support this, we redesigned the site catalog format to support different interfaces/
protocols for accessing the same staging storage element. A sample site catalog for OSG execution
is illustrated in Figure 5. The data flow for the hub workflows executing on OSG using a shared
staging storage element with the new approach is illustrated in Figure 6. In this setup, the data
stage-in jobs are added by the Pegasus Mapper to the executable workflow, to retrieve the input data

Figure 4. Workflow execution on a single Open Science Grid (OSG) high-performance computing (HPC)
cluster.

Figure 5. Pegasus workflows on Open Science Grid (OSG) with staging storage element.
Data Flow for Hub Workflows through Pegasus on OSG with GlideinWMS and Staging Storage Element

Figure 6. Hub workflows through Pegasus on Open Science Grid (OSG) with staging storage element.

from multiple input storage sites and store them on a staging storage element. For an application using the Rappture interface, users upload the input files to the hub workflow host using the Rappture web frontend. In case of OSG, the staging storage element is usually an SRM server. The SRM server serves as the data staging site for the workflows. The compute jobs in the workflow are dynamically matched with available worker nodes at runtime by the OSG provisioning tools [39, 40]. The Pegasus Lite instances that start up on the worker nodes retrieve the input data from the staging storage element using the HTTP frontend, execute the compute tasks, and push the output files back to the staging storage element using SRM interface. The stage-out jobs added by the Pegasus Mapper transfer the output files for the workflow back to the hub workflow host, to be presented through the Rappture/submit interfaces in the hub. Because most of the compute resources on OSG do not have a large shared storage capability, the use of a separate staging storage element allows us to execute large big data workflows on OSG.

For executing larger workflows, we wanted to leverage Pegasus capability to add cleanup nodes [32, 33] to the executable workflow. When enabled, the in-place cleanup jobs reduce the peak storage requirements of a workflow on the staging server. Data on the compute nodes is cleaned up by Pegasus Lite. However, with the new data staging approaches, the workflows used for the integration highlighted two issues:

1. In order to compute where in the underlying graph a cleanup node is added, a reachability analysis is calculated for each file in the workflow. When developing the OpenSees workflows, it was discovered that the implementation took a long time, and as a result, the overall planning process took a long time.
2. It was discovered that for the large workflows with the addition of the cleanup nodes, the workflow walltime increased drastically. This was because of the fact that the algorithm is aggressive and, in the worst case, can add a cleanup node for each compute job in the workflow.

Improvements were made in Pegasus 4.2 to address the aforementioned issues. The first issue was addressed by improving the underlying implementation of the algorithm. The second issue was
addressed by clustering the cleanup nodes added per level of the workflow. When clustering, we take care to cluster together cleanup jobs for a single staging site. This reduces the number of cleanup jobs added for the workflow and at the same time does ensure that cleanup happens as the workflow is executing.

6.2. Workflow status reporting to users via hub interface

Another integration challenge was how to provide better feedback to the user about the execution of the workflow. The interface to Pegasus is mostly command line tools, and even though there are tools like pegasus-status (which provides a progress report), the question was how to provide the information back through the tool to the end user. In particular, we ran into a problem where planning a workflow was taking a long time—on the order of 30 min! During this time, there was no feedback to the user that the system was actually doing anything. We are still working on a solution to provide better output from the pegasus-plan operation, but once the workflow is planned and is running, the hub regularly runs pegasus-status and provides feedback to the user. Once the workflow has finished, the pegasus-statistics and pegasus-analyzer commands are run automatically to provide the user with summary statistics and, if something failed, a report on what failed. Improving these integration points is an ongoing effort for the HUBzero and Pegasus developers.

Some integration issues required just simple code changes. One example is how a report generated by the pegasus-statistics tool was modified to better match the data needed inside HUBzero. The changes included adding a few columns to the job/host breakdown and providing the data in comma-separated value format so that the HUBzero framework could easily import the data at the end of a workflow run. While HUBzero collects detailed provenance information about all of the jobs executed, it is used primarily for summary statistics and not yet available for users to trace the course of their own research. That remains as a challenge for future development.

7. CONCLUSION

The integration of Pegasus into the HUBzero framework has brought the power of automated workflows to many more users. HUBzero workflow developers can leverage the data management improvements in Pegasus 4, to more easily hand off workflows to the underlying infrastructure and worry less about where the jobs will be running and how the data transfers will take place.

Expert users can launch a Workspace within a hub and gain immediate access to Pegasus and computational resources—without having to install or setup the Pegasus software, without having to apply for Grid credentials, and without having to learn complex Grid protocols. A simple submit command dispatches any single job or complex workflow off to remote facilities for execution. Expert users can experiment with such commands, use them in research activities, and wrap them into GUIs that are deployed as new tools for the community.

Without knowing a thing about workflows, high-throughput computing, campus, or national cyberinfrastructure, others in the community can access such tools, enter their own input parameters, and launch a complex workflow with the press of a button. If the users have questions, they can post them for the tool authors and the rest of the community. They can post ideas about improving the tool, and submit bug reports when a tool fails or produces incorrect results. All of these capabilities are provided not just for one community or one site but for 50+ sites built on the HUBzero platform. This improvement is bringing workflow automation and high-throughput computing to thousands of users around the world who will benefit from it without ever realizing the complexity.

ACKNOWLEDGEMENTS

This work was supported by the US National Science Foundation through grants CBET-0941302, CMMI-0927178, OCI-1148515, and OCI-0943705. The Pegasus work was funded by the National Science Foundation under the OCI SDCI program grant #0722019 and OCI SI2-SSI program grant #1148515. The development of HUBzero was funded by Purdue University and the National Science Foundation awards EEC-0228390, EEC-0634750, OCI-0438246, OCI-0721680, OCI-0944665, and IIS-1017837. DiaGrid...
was supported in part by the NSF TeraGrid Resource Partner award (#0503992) as a TeraGrid and XSEDE high-throughput computing resource from 2007–2013. The authors wish to thank Dr. Wen Jiang, Department of Biological Sciences, Purdue University, for developing the Cryo-EM tool and making it available on the DiaGrid.

REFERENCES


