Workflow Management Through Cobalt

Gregor von Laszewski[1,2], Christopher Grubbs[3], Matthew Bone[3], and David Angulo[4]

[1] University of Chicago, Computation Institute, Research Institutes Building
#402, 5640 S. Ellis Ave., Chicago, IL 60637

[2] Argonne National Laboratory, Argonne National Laboratory,
9700 S. Cass Ave., Argonne, IL 60439

[3] Loyola University Chicago, Department of Computer Science,
Lewis Towers, Suite 416, Water Tower Campus, Chicago, Illinois 60611, USA,

[4] 4 DePaul University, School of Computer Science, Telecommunications and Information Systems,
243 South Wabash Ave, Chicago, Illinois 60604, USA

Contents

Abstract

Workflow management is an important part of scientific experiments. A common pattern that scientists are using is based on repetitive job execution on a variety of different systems, and managing such job execution is necessary for large-scale scientific workflows. The workflow system should also be client-based and able to handle multiple security contexts to allow researchers to take advantage of a diverse array of systems. We have developed, based on the Java Commodity Grid Kit (CoG Kit), a sophisticated and extensible workflow system that seamlessly integrates with the queueing system Cobalt through the advanced features provided by the CoG Kit.

1 Introduction: Queueing systems and workflows

Parallel computing systems offer scientists incredibly powerful tools for analyzing and processing information. These supercomputers benefit researchers in many diverse fields, from medicine to physics to social sciences.

In order to take advantage of parallel system capabilities, scientists must be able to schedule their jobs with a high degree of confidence, knowing that the jobs will run, the results will be saved, and any errors will be handled. Use of these systems is not, of course, limited to one user; therefore the systems must be able to reliably handle large amounts of tasks from many different users.

This is where queueing system software comes in. Queueing software handles job execution on parallel systems, scheduling the jobs and allocating resources accordingly. While the simplest queueing systems would simply execute jobs on a first-come-first-serve basis, other factors, like the projected running time or number of nodes required, may come into play in more complex systems. By maintaining a user job queue, the queueing software automates the scheduling process such that the system's resources are utilized to their fullest extent. It also provides an interface for monitoring the current system queue and retrieving information about a job, such as its location in the cluster or grid, how many nodes it is using, how long it has been running, and so on. Some queueing systems also allow server-side specification of dependencies, or workflows, so that users can string together tasks which depend on prior tasks.

There are many queueing systems available, in both commercial and open-source implementations. Commercial queueing software packages include Platform's Load Sharing Facility (LSF) [9], IBM's Tivoli Workload Scheduler [17], and Altair Engineering's PBS Pro [11]. An open-source implementation of PBS, called OpenPBS, also exists [10]. Other open-source packages providing job scheduling functionality include the Globus Toolkit [6], Condor [5], Sun's GridEngine [12], and Cobalt [2], used on Argonne National Laboratory's BGL cluster system. Furthermore, many portals exist which streamline access to queueing software. TeraGrid's portal is one example [16]. The TeraGrid Portal is a web-based system, allowing the grid's users to manage their projects, receive important system-related information, and access documentation, all from within their browsers.

As was mentioned before, some queueing systems such as LSF provide workflow functionality; however, such workflows are necessarily tied to one resource. A workflow submitted to one grid system will only be able to utilize the resources provided by that grid system. The addition of a client-side workflow implementation, however, allows access to numerous resources. Through a client-side workflow, a user may access both grid and non-grid systems, working across several different security contexts.

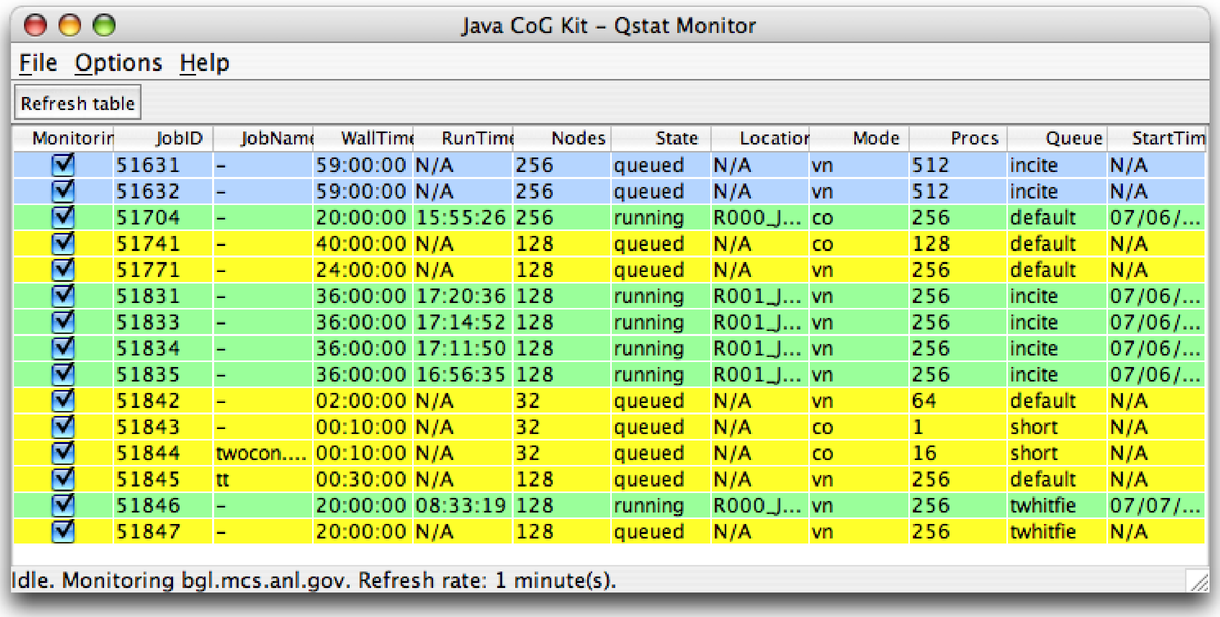
A particularly flexible client-based workflow engine called Karajan is provided in the Java CoG Kit [7], an open-source set of grid tools. This paper will focus on the integration of Karajan with the aforementioned queueing system Cobalt. We chose Cobalt primarily because it is used on BGL at Argonne National Laboratory, where our research is based. Furthermore, because BGL is not a grid system, we wanted to demonstrate that tools normally applied to grid systems could also be applied to non-grid parallel systems. This will demonstrate Karajan's flexibility in creating robust workflow solutions; the techniques used to integrate Cobalt with Karajan can be employed for other queueing systems as well, as we will show through our additional integration with PBS systems.

Before we go into details about Karajan, a brief introduction to Cobalt will be useful.

2 Cobalt: System software for parallel machines

Cobalt is used for handling jobs on BGL, the 1024 node, 2048 processor IBM BlueGene/L system at Argonne National Laboratory. BGL is used primarily for scientific computing, application porting, system software development and scaling studies [1]. Following a "smaller and simpler is better" philosophy, the Cobalt software package trades on feature-richness for agility [3]. While its core implementation is comprised of less than 4000 lines of code, mostly in Python, its component-based architecture makes it a highly adaptable and useful research platform [3]. Researchers can readily add new components or rewrite existing ones.

Users may login to BGL via SSH using public key authentication. The cqsub command submits a job to the queue. Like the qsub command found on other queueing systems, cqsub takes a variety of command line arguments, including the desired execution time, the amount of nodes requested, the path to the executable, and so on. Cqsub also takes some specialized arguments, including one for dynamic kernel selection. This feature, which is only found in Cobalt, allows users to test experimental kernels and conduct system software research [3]. It is special features like this that motivated our integration of a specialized Cobalt submission



Monitorir	JobID	JobNamı	WallTimı	RunTimı	Nodes	State	Locatior	Mode	Procs	Queue	StartTim
✓	51631	-	59:00:00	N/A	256	queued	N/A	vn	512	incite	N/A
✓	51632	-	59:00:00	N/A	256	queued	N/A	vn	512	incite	N/A
✓	51704	-	20:00:00	15:55:26	256	running	R000_J...	co	256	default	07/06/...
✓	51741	-	40:00:00	N/A	128	queued	N/A	co	128	default	N/A
✓	51771	-	24:00:00	N/A	128	queued	N/A	vn	256	default	N/A
✓	51831	-	36:00:00	17:20:36	128	running	R001_J...	vn	256	incite	07/06/...
✓	51833	-	36:00:00	17:14:52	128	running	R001_J...	vn	256	incite	07/06/...
✓	51834	-	36:00:00	17:11:50	128	running	R001_J...	vn	256	incite	07/06/...
✓	51835	-	36:00:00	16:56:35	128	running	R001_J...	vn	256	incite	07/06/...
✓	51842	-	02:00:00	N/A	32	queued	N/A	vn	64	default	N/A
✓	51843	-	00:10:00	N/A	32	queued	N/A	co	1	short	N/A
✓	51844	twocon....	00:10:00	N/A	32	queued	N/A	co	16	short	N/A
✓	51845	tt	00:30:00	N/A	128	queued	N/A	vn	256	default	N/A
✓	51846	-	20:00:00	08:33:19	128	running	R000_J...	vn	256	twhitfie	07/07/...
✓	51847	-	20:00:00	N/A	128	queued	N/A	vn	256	twhitfie	N/A

Idle. Monitoring bgl.mcs.anl.gov. Refresh rate: 1 minute(s).

Figure 1: sample caption

library with the Karajan workflow engine.

3 Karajan: A workflow scripting language

Karajan is a parallel scripting language which uses a declarative concurrency approach to parallel programming [4]. It is a dual-syntax language, supporting both a native syntax and an XML-based format. For the purposes of this paper, we will use the XML syntax, although there is no underlying difference between the two forms. Basic sequential workflows in Karajan are more or less lists of tasks; the user simply creates the task elements in the proper execution order. To execute tasks in parallel, the user places the tasks in question within a `<parallel>` element. Karajan also provides a `<sequential>` element, in case the user requires certain tasks within a parallel element to execute sequentially.

In addition to its parallel and sequential workflow components, Karajan provides elements for remote task execution, file transfers, a variety of data structures, logical operators, variables, access to GUI forms, Java object and method bindings, and more. Karajan contains a general task execution framework which employs the CoG Kit's task/provider model. The task/provider model provides a consistent programmatic interface while the underlying provider implemenation (which may be GT2, GT4, SSH, etc.) changes dynamically. This approach is useful in many ways, affording developers a high-level syntax for specifying tasks while leaving actual submission and execution details to the CoG Kit. It also allows developers a simple way to add new protocols without having to change the task implementation; they need only write a new provider. One downside, however, is that it does not allow for special features specific to a certain queueing system, such as Cobalt's kernel profile selection options.

In order to utilize Cobalt's features within the Karajan framework, we employ Karajan's Java binding functionality, which allows limited interfacing with Java classes, objects, and methods. Through the Java bindings, we are able to invoke Java code which logs into a Cobalt system, submits a job, and monitors its status. This approach allows us to bypass the CoG Kit abstraction layer while still affording the same workflow capabilities of Karajan's task/provider model. *Java Binding in Karajan Objects can be instantiated in a Karajan script with the `<java:new>` element, provided that the class is present in the CoG Kit's classpath.

The instantiated object's methods can be accessed with the `<java:invokeMethod>` element; static methods may also be accessed with this element by either setting its "static" attribute to true or by specifying a class name. Creating these bindings can be an tedious task, especially if many different methods are involved. To automate this process we created the Java2Karajan tool.

3.1 Java2Karajan

The Java2Karajan program takes a compiled Java class file and, using the Java reflection API, constructs a Karajan Java binding. The approach is similar to that taken by SWIG [14], where libraries implemented in a lower level language, in this case Java, are bound and translated to a higher level language, in this case Karajan. Here is an example of an object and method binding generated by Java2Karajan:

```
<element arguments='''' name=''example:initialize''>
    <global name=''_obj''>
        <java:new classname=''org.cogkit.q.Example''/>
    </global>
</element>

<element arguments=''arg0, arg1'' name=''example:myMethod''>
    <java:invokeMethod method=''myMethod'' object=''{_obj}''
                       types=''int, java.lang.String''>
        <argument value=''{arg0}''/>
        <argument value=''{arg1}''/>
  </java:invokeMethod>
</element>
```

Here we have a binding for instantiating an Example object, as well as a binding for that object's method `myMethod()` which takes an int and a String as parameters. If a script imports this Java binding, it can, for example, instantiate an Example object with the element `<example:initialize/>`, then call myMethod(4, "hello") with the element `<example:myMethod arg0=''4'' arg1=''hello''/>`. If a method returns a value, that value can be placed within a Karajan variable and used elsewhere in the script. * Integrating Cobalt job submission with Karajan workflows In order to take advantage of Java2Karajan for our purposes, we had to first create a set of Java classes for submitting and monitoring the status of jobs on a Cobalt queueing system. The basic sequence of events in a Cobalt submission is as follows:

SSH Initialization → SSH Authentication → Open SSH session channel → Execution of cqsub, returning the job's ID → Close channel

We created a CobaltSubmitter object which utilizes the SSH connection, authentication, and command execution functionality found in the open-source J2SSH SSHTools libraries [13]. First, we instantiate a CobaltSubmitter object, which takes as parameters the hostname, the username, the port, and the path to the user's private key. In order to avoid a situation where the user would need to enter their passphrase in cleartext, we used a function in the CoG Kit which, upon execution of the Karajan script, masks the passphrase console input with random characters. Assuming all of the authentication information is correct, and that connection and authentication is successful, we are left with an initialized CobaltSubmitter object, through which we can submit jobs using its `submit()` method.

Any command line argument which can be passed to Cobalt's cqsub command can be given as a parameter in the CobaltSubmitter's `submit()` method. This method formulates a cqsub command based on the given parameters and executes it on the server. Cobalt, after adding the job to its queue, responds with a job ID number, which the CobaltSubmitter's `submit()` method returns as a string.

These two processes, initialization and submission, are essentially all we need to submit jobs to a Cobalt system. However, the CobaltSubmitter object would be useless for workflow purposes without a third function: job status monitoring. Simply submitting jobs in parallel does not preserve dependencies; what we need to do is not only submit the job, but continually monitor its state, moving on to subsequent jobs in the workflow only when the job has finished executing.

Cobalt provides the command cqstat for queue monitoring purposes. The CobaltSubmitter object has a method `status()` which takes the id of a job as a parameter, and executes the cqstat command for that particular job, returning a string representating the job's state. If the cqstat command returns a blank table, that means that the job in question has completed execution and left the queue, so `status()` returns the string "finished."

Having written this code in Java, we can now use our Java2Karajan tool to generate our bindings for us. Once we do that, one further step remains: defining a `<cobalt:execute>` Karajan element, which combines the job submission and job monitoring methods provided by the CobaltSubmitter bindings into a single task element. This element will invoke the `submit()` method, placing the returned job ID string in a variable. We then enter a loop, utilizing Karajan's `<while>` element, in which we repeatedly invoke `status()` using that returned job ID string as a parameter, exiting the loop when the method returns "finished." Of course, we wait an arbitrary amount of time between invocations of `status()` to prevent undue stress on the server.

It should be noted that although polling is usually regarded as a dirty word, it is necessary for our purposes, as we can provide no callback function to inform us of the job's status. While polling is not an ideal solution, the flexibility afforded by our approach makes it worth the relatively crude implementation.

Having created the `<cobalt:execute>` element, we now have a simple, convenient way to submit Cobalt jobs from a Karajan workflow. Here is a simple abbreviated example, in which two jobs are executed in parallel, and a third is executed afterward:

```
<cobalt:initialize host=''bgl.mcs.anl.gov'' user=''user''
                   port=''22'' privatekeypath=''/home/user/.ssh/id_rsa''/>
  <parallel>
      <cobalt:execute walltime=''30'' nodes=''32'' path=''~/bin/task1'' .../>

      <cobalt:execute walltime=''60'' nodes=''64'' path=''~/bin/task2'' .../>
  </parallel>

<cobalt:execute walltime=''45'' nodes=''32'' path=''~/bin/task3'' .../>
```

4 Qstat Monitor: observing the workflow

As we developed the Cobalt submission classes, we concurrently developed the Qstat Monitor, which is a graphical component displaying the current state of a parallel system. It provides the user with a clean, intuitive interface to a cluster or grid system. Essentially, it parses the output of a qstat command into a Swing JTable. The table is color-coded (running jobs are green, queued ones are yellow) and allows users to customize which fields they would like to view. Jobs which have left the queue remain in the Qstat Monitor's table, colored blue, providing the user with a record of completed jobs.

The monitor, in addition to being of general use for users of BGL and other parallel systems, was specifically relevant for the Cobalt workflow research, because it allowed us to observe the queue's status throughout the various stages of the workflow execution. We can check if a given job has begun execution or if it is still waiting, and we can monitor other system activity; for example, perhaps the system is particularly backlogged with other users' jobs, which will slow down our workflow execution.

The Qstat Monitor also offers job submission functionality, using the same Cobalt submission classes used in the Karajan workflow integration. Through a dialog box, the user can fill in the relevant information for the job (wall time, amount of nodes, etc.) and submit the job directly through the Qstat Monitor program.

5 Implications and possibilities

Although our integration of a custom queueing system library with Karajan is fairly basic, it has significant implications for researchers interested in maximizing their usage of diverse parallel systems. While Karajan previously supported a heterogeneous workflow, allowing multiple systems and security contexts, we now have the ability to fine-tune our workflows to take advantage of the unique features provided by specific queueing systems. This approach is by no means limited to Cobalt.

5.1 Extensibility: adding PBS system support

As a demonstration of the extensibility of our approach, we developed a set of PBS submission classes which can be integrated with a Karajan workflow in the same way as the Cobalt classes. PBS is a widely used batch queueing system; here at Argonne, it is used on the UC/ANL TeraGrid [15]. We use the same SSH methods as the Cobalt system for these classes. What changes is the structure of the qsub command and the arguments available to the user. Having implemented the methods to generate the qsub command, and with the features provided by our Java2Karajan tool, creating `<pbs:initialize>` and `<pbs:execute>` elements is a fairly straightforward task. This makes possible a Karajan workflow combining Cobalt and PBS elements, as well as Karajan's existing task submission elements, and potentially any other system for which a library is available.

Additionally, we implemented PBS functionality in the Qstat Monitor to demonstrate that component's extensibility. The Qstat Monitor can switch between the two queueing system modes, providing features specific to each, while the underlying SSH interaction mechanism remains the same.

5.2 Further research: graphical workflows in the Qstat Monitor

One possibility for further research in this area might be the integration of the Karajan workflow with the graphical Qstat Monitor component. Instead of defining workflows in Karajan XML, the Qstat Monitor component could be expanded to allow the graphical creation of workflows, which could then be passed to the Karajan engine. This would combine the customizability offered by the custom queueing system submission libraries and the convenient graphical interface provided by the Qstat Monitor.

It may also be possible to integrate the queueing system-specific features described in this paper with the existing Karajan GUI application [8].

6 Conclusion

We have demonstrated an effective way to extend Karajan to take advantage of the unique features provided the Cobalt queueing system, further improving Karajan's client-based heterogeneous workflow capabilities. In doing so, we have also demonstrated a general methodology of augmenting Karajan through its Java binding functionality. Furthermore, our Karajan implementation works without installing any grid software, and authenticates through standard SSH public key authentication. As part of the CoG Kit, this research expands the commodity vision, in which we integrate grid resources with non-grid resources.

7 Availability

The Java CoG Kit is available through its homepage at http://wiki.cogkit.org. A Java Web Start release of the Qstat Monitor is available at `http://wiki.cogkit.org/index.php/Java_CoG_Kit_Qstat`.

Code referred to in this paper can be found in the qstat section of the CoG Kit's Subversion repository, viewable at `http://svn.sourceforge.net/viewvc/cogkit/trunk/five/qstat/`. Instructions for downloading the repository to Eclipse, using the Maven build system, can be found at `http://wiki.cogkit.org/index.php/MavenRepository`.

Acknowledgments

The REU was supported by NSF REU site grant 0353989. The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No.

W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References

[1] Argonne National Laboratory BG/L System. Web Page. Available from: `http://www.bgl.mcs.anl.gov/`.

[2] Cobalt: System Software for Parallel Machines. Web Page. Available from: `http://www-unix.mcs.anl.gov/cobalt/`.

[3] Cobalt: An Open Source Platform for HPC System Software Research. Web Page. Available from: `http://www-unix.mcs.anl.gov/cobalt/Cobalt-epcc-10-05.pdf`.

[4] Java CoG Kit Karajan Workflow Reference Manual. Web Page. Available from: `http://wiki.cogkit.org/index.php/Java_CoG_Kit_Karajan_Workflow_Reference_Manual`.

[5] Condor: High Throughput Computing. Web Page. Available from: `http://www.cs.wisc.edu/condor/`.

[6] The Globus Toolkit. Web Page. Available from: `http://www.globus.org`.

[7] Java Commodity Grid (CoG) Kit. Web Page. Available from: `http://www.cogkit.org`.

[8] Java CoG Kit - Karajan GUI. Web Page. Available from: `http://www.cogkit.org/release/4_1_2//webstart/#karajan-cog-workflow-gui`.

[9] Load Sharing Facility. Web Page, Platform Computing, Inc. Available from: `http://www.platform.com/`.

[10] Portable Batch System. Web Page, Veridian Systems. Available from: `http://www.openpbs.org/`.

[11] Altair PBS Professional. Web Page, Altair Engineering. Available from: `http://www.altair.com/software/pbspro.htm`.

[12] gridengine: Home. Available from: `http://gridengine.sunsource.net`.

[13] J2SSH SSHTools. Web Page. Available from: `http://sourceforge.net/projects/sshtools`.

[14] Simplified Wrapper and Interface Generator (SWIG). Web Page. Available from: `http://www.swig.org/`.

[15] TeraGrid. Web Page, 2001. Available from: `http://www.teragrid.org/`.

[16] TeraGrid Portal. Web Page. Available from: `http://www.teragrid.org/userinfo/portal.php`.

[17] IBM Tivoli Workload Scheduler. Web Page. Available from: `http://www-306.ibm.com/software/sysmgmt/products/support/IBMTivoliWorkloadScheduler.html`.