Active Code Generation and Visual Feedback for Scientific Workflows using Tigres

Ryan A. Rodriguez
Lawrence Berkeley National Lab
Berkeley, CA
Email: {ryanrodriguez}@lbl.gov

***Abstract*—**

The Tigres project is a next generation scientific workflow API designed with ease of composition, scalability and light-weight execution in mind. This paper will focus on the development of an early stage user interface that aims to accelerate the prototyping of workflows by providing real-time visual feedback during workflow construction along with a code generation feature. Most programmers recognize that an IDE is necessary, but not sufficient for effective workflow design. The bridge between the graphical and code generating aspects of the interface will be an active notepad application inspired by Google's real-time search bar. Using a minimal command set, a user can enter and modify workflows in the active notepad with instant graphical feedback. The result will be an interface that feels like an IDE, but provides the option of graphical interaction and compositional streamlining.

I. Introduction

Researchers in computationally intensive sciences are faced with analyzing rapidly expanding data sets on increasingly parallelized machines. These datasets have grown so large that it is no longer practical for partner institutions to download and analyze this information locally. This paradigm poses great challenges for scientists developing the complex workflows necessary to operate on this data, particularly those working on data-intensive research, known as 'e-Sciences'. Scientists building workflows for data intensive analysis currently have one of two options. They can choose to write their own scripts in a programming language of their choice, or they can employ a specialized workflow tool like Taverna [1], Pegasus [2], or Kepler [3]. A popular approach to dealing with data-intensive tasks has been to follow the MapReduce pattern for parallelization, merging, and splitting tasks - sometimes referred to as 'Scatter-Gather.' At present, writing scientific workflows using a MapReduce model with a Hadoop implementation can require a significant amount of programming expertise and custom code. Some difficulty arises in scientific workflow implementations stemming from the difference in the nature of typical MapReduce tasks in comparison to scientific functions. Additionally, writing scripts from scratch comes with a considerable amount of overhead due to the complexity of provenance tracking, parallel fault tolerance and monitoring features. Currently there is a significant separation between workflow implementations in programming languages, and those built using specialized GUI workflow tools. In light of these challenges, a tool that can simplify the implementation of common computing patterns while offering a high degree of interactivity is highly desirable.

The goal of the Tigres project is to develop a next generation scientific workflow tool that addresses these concerns. Following the examples of the MapReduce model, in particular the Apache Hadoop implementation, Tigres is designed to be a highly portable and lightweight API to radically simplify the deployment of scientific workflows to diverse resource environments. Workflows in Tigres can be composed with the flexibility and ease of standard bearer technologies like C and Python while eliminating the overhead of writing custom workflow scripts. To use Tigres, a user simply needs to import the Tigres library into their project and compose their workflow using Tigres Templates. These templates represent the most commonly used computational patterns: Sequence, Parallel, Split and Merge. These templates are used to organize Tigres Tasks, the atomic unit of execution. Each Task object holds a function or executable, input values and input types. In constructing a Workflow using these Tigres types, a user has done everything that they need to invoke the advanced monitoring features of Tigres. Provenance tracking, fault-tolerance, execution-state and visualization tools become artifacts of Tigres' use, not overhead. This model allows analyses created on a desktop to be scaled and executed seamlessly on a diverse set of resources.

While the importance of effective workflow tools is widely recognized, the process of API design has thus far been colored by designer bias and preconception. The 'User Centered Design' model has proven to be an effective tool for successful API development because of its persistent focus on user wants and needs. Currently, the Tigres project is focusing on a number of use cases from the biological sciences to cosmology. These diverse user groups offer a unique opportunity to retrieve useful feedback from highly domain specific sciences in structured review processes referred to as 'Scientist Centered Design' or SCD. User input is particularly valuable in the early stages of development to catch confusing features before they become deeply embedded in the system. Throughout the SCD review process an API gets distilled into a tool that mets user needs without sacrificing usability.

Despite high demand for capable workflow tools, existing applications have failed to gain widespread acceptance due to clunky interfaces, high costs of adoption and loose focus on the user. At its core, Tigres endeavors to be as minimally intrusive as possible. A researcher using Tigres in the IDE of their choice should find that this experience is indistinguishable from using their IDE on any other type of project. This is desirable on the one hand because users have expressed their discontent with confusing features and opaque interfaces. On

the other hand, many users still appreciate the interaction that a well designed UI can provide. An effective interface has the potential to eliminate much of the menial coding involved in workflow composition and provide valuable visual feedback without feeling intrusive and cumbersome.

The focus of this paper is the design and implementation of an interface developed to engage users and accelerate workflow development. This work aims to be a proof of concept for a future integrated environment where users can move seamlessly between graphical and source code domains. Modifying a directed graph could provide an equivalent workflow representation in C or Python instantly, while modifications to the workflow in source code are reflected in a visual representation just as quickly.

II. Related Work

The demand for applications which address issues facing e-Science has grown dramatically in recent years. Several workflow tools have been developed to address this demand. Discovery Net [4] was one of the first tools available to coordinate the execution of remote services using the web. It provides a means to coordinate workflows between data owners and analysis servers that may be far-flung geographically. The system also allowed for visual coding through a drag and drop interface. Although the Discovery Net project was quick to recognize the need for workflow coordination services, their XML based language known as 'Discovery Markup Language' in addition to its domain specific nature and high cost of integration into existing projects did not allow for widespread adoption. Taverna Workbench [1] is an open source tool that has emerged with a graphical focus; using a limited drag and drop functionality, users can construct a workflow with representations for inputs, outputs and various services. The application facilitates the building, execution, and collaboration of workflows through web services. The interface provides a services tab to associate graphical blocks with analytical functions, and a standard project hierarchy navigator. Coined as an 'enactment engine,' Taverna's integration with popular programming languages is difficult. Additionally, users quickly find themselves lost in a maze of drop down windows.

Kepler [3] is another open source GUI based workflow tool being developed as a joint project between multiple University of California campuses. It offers support for various grid services and has the ability to execute workflows from a command line or graphical interface. Like Taverna, however, Kepler's main drawback is its loose association with C or Python. Instead, it uses a host of pre made components for statistical analysis.

Triana's [5] take on workflow development was to formulate a 'pluggable architecture' that can integrate into existing systems. It comes with a variety of built in tools for signal analysis as well as image manipulation. Triana has basic error detection, but no support for handling errors during parallel failures during a join stage. Again, the drawback of this tool is the user is confined to using a set of pre-made functions. If a custom function is desired, it must be specified in a wizard provided by the application.

The concept for Pegasus [2] is interesting in its divergence from the others. Instead of the typical graphical focus, Pegasus' focus is on being able to execute on a wide variety of distributed environments including clusters, grids and clouds. Pegasus has the ability to map elements of a given workflow to available resources on the target environment. Pegasus offers a GUI, but goes so far as to include a disclaimer on their site which reads, "It is a demonstration tool and can be used to see how simple DAXes look like. However, to generate your workflows, we recommend to use the various APIs we provide." The broad class of computational patterns (DAXes) that Pegasus supports makes the API powerful, but non-trivial to learn or use.

Within these projects, there is little support for automatic scaling to fit growing datasets. Further, the problems of fault tolerance and dynamic workflow execution remain. Focus on exotic computation patterns or specialized syntax often makes for confusing APIs and limited usability.

III. Visualizations

In spite of the shortcomings of many graphical interfaces for workflow tools, visualizations are often the first step to understanding the semantics of a workflow. If an API is sufficiently difficult to learn or chronometrically expensive to adopt, users may tend to compose their workflows entirely in a graphical interface. Although the Tigres API aims to be exceptional in its ease of use and ability to quickly integrate into existing projects, graphical representation will still be an essential, if peripheral, tool for composition and execution monitoring.

To this end, a tool called Graphviz was employed for the initial production of workflow execution graphics. Graphviz is an open source tool built on top of several popular graphing engines including 'dot', 'neato', and 'sfdp'. 'Dot' is a tool built by ATT for representing complex networks. It provides edge drawing and placement algorithms that allow dot files to be specified quite simply. Additionally, 'dot' offers a wide variety of sub graphing and rank specifications for exhaustively specified graphs. The tool can output into a number of common image formats including SVG, making manipulation of 'dot' files possible in web pages. Currently, 'dot' graphs take the form of static output images. There is little *direct* support for graphical editing, though this is an area of interest for ATT engineers.

Work on Tigres visualizations began with manual specifications of workflows in the 'dot' language. Individual code snippets for each Tigres template (Sequence, Parallel, Merge and Split) were implemented, and the underlying structure was analyzed in order to produce an algorithm that could take tigress workflows of arbitrary size and convert its constituent templates into a 'dot' graphic using Graphviz. Once this work was complete, it was possible to perform additional subgraphing and clustering to render workflows in an intuitive and easy to read format. (put some pictures here?)

In order to make this visualization unobtrusive to the user, this functionality is built into the monitoring capabilities of Tigres. In other words, generation of workflow execution graphs is another artifact of using Tigres types to construct a workflow. Initially, this functionality has been limited to working on executions from memory. Future iterations of the Tigres graph module can work from log files which track

Tigres execution on computing systems in real-time. Each task holds information pertaining to its current execution state, allowing for changing visuals depending on the current status of a task.

While direct support for interactive graphics in 'dot' is not possible, web browsers offer the advantage of advanced visual capacity without additional software tools. The first choice for cross platform compatibility and visual flare was the D3 [6] library in javascript. Other JavaScript libraries that have been considered for retooling to fit Tigres visualizations include JS Plumb [7], GraphDracula [8], and Arbor JS' HalfViz [9] extension of the dot language, which is renderable in web pages. This system was initially attractive in its similar, but scaled down functionality to 'dot', but still suffers from the inability to edit.

D3's focus is on data-driven documents; by binding visual elements in a web page to data elements it is possible to create dynamic graphs and animations. Additional motivation for implementing visualization and manipulation functionality in a browser was to conform to Tigres' goal being easily adoptable. It's expected that many users would prefer to stick to their editor or IDE of choice, so having this functionality in a web browser instead of a standalone application would be less intrusive to the user. Additionally, due to the dynamic nature of the tool it would be well suited to operate on changing data sets.

Several visualization schemes were used as test cases for D3. Most familiar is the 'weighted graph' which uses a charged particle and spring constant model to control the placement and motion of nodes in the graph. Altough this model is visually interesting and fun to interact with, it suffers from a 'hairball' even with large charge repulsion between nodes for graphs of non-trivial size. To address this concern, a 'hive' visualization paradigm was explored. Hives use axes to organize nodes and draw edges based on user defined metrics. A secondary motivation for this style of visualization was the difficulties posed when representing complex dependency networks in a workflow. The 'hairball' effect when representing graphs in 'dot' and D3 were compounded when edges showing their dependencies were added. In a hive format, these 'hairballs' took an orderly and visually pleasing form. (put a hive picture here?) Although hive graphs solve the problem of data dependency visualization, they are not so easily understood in the context of execution graphs.

The D3 library offers a robust solution for web-based graphics. In many cases, however, it can tend toward the experimental or quixotic. Other avenues include the use of the xDot [10] python library, which allows for rapid dot visualization without a web browser. Interactivity is still limited, but the possibility of onboard viewing and editing may prove worth the additional development. In subsequent research a method for making dot files rendered in SVG draggable and somewhat modifiable was discovered. At present this seems to be the most lucrative way forward, since it would require the least development to bring it to full functionality. This would include, but not be limited to including graph cluster scaling and node creation.This method relies upon the dot jQuery plugin [11] for interactivity.

IV. Active Console

In the process of visualization research, many hurdles involved with implementation specifics were cleared. This, however, did not address the ultimate goal of creating an interactive workflow tool that could be executed from code running inside of an existing C or Python project. Existing tools for manipulating graphs have a certain duality to them. There exists a clear distinction between the graphical and code domains inherent in workflows. Because Tigres follows an API model, it differs from many other tools in that Tigres is executed inside of a workflow rather than the other way around. This paradigm implies a much deeper relationship between coding and graphing. Therefore, it is undesirable at this point to develop a monolithic Java tool for describing workflows graphically.

The need for Tigres to maintain tight integration with code while adding a visual layer implies a tool that can be executed within a workflow using libraries that are native to the given programming language - in this case Python. Such a tool provides several distinct advantages for our purposes. First, while workflow composition in Tigres is relatively simple, specifying large workflows can become tedious. An interface that could generate large chunks of Python code with a minimal instruction set is, therefore, highly desirable. Second, in addition to code generation, this graph could assist in the prototyping of workflows by offering a 'dry-run' visualization. This is advantageous because current implementations of the monitoring module only operate during execution. This requires that methods, executables, inputs and types all be specified in advance. By interfacing directly with the active notepad, the use of Tigres types for graph generation is bypassed and no call to execution is necessary. Lastly, the major advantage of using such an interface would be its ability tie together the code generation and visual aspects of the tool. For example, a user could quickly specify a workflow using the active notepad application using the reduced command set. This would result in generation of the code necessary to implement this workflow in tigress along with a visualization of the workflow execution. Referring to the output of the real-time visualization, a user and his or her collaborators could quickly identify problems with the workflow or modify their algorithm by physically manipulating nodes in a drag and drop style. The result of such an implementation would be a tool that ties together the visual and coding domains seamlessly, with the choice of being executed as a simple Tkinter and or GTK application in python, or serve as the basis for a dedicated editor where a user could quickly move between domains.

The idea of a workflow tool that could emphasize the code/graphic relationship has been inspired by several code editors and program interfaces that give realtime feedback to users writing code. Mathematica allows users to write a brief snippet of code and execute it within the editor. The output of such an execution is displayed in-line with the code and can take the form of a graphical or textual output. In order to improve the feel and response of the application, cues were taken from the Google real-time search bar. With google's new search bar, users can see responses to their queries as they type. This is desirable for our purposes because it removes the need for specific execution commands, and results in an interface that feels less like a command line.

Several open source editors built entirely in Python currently exist which could be extended to include the desired functionality, along with modern features of a python editor like smart indenting, code highlighting and so on. For our purpose, the idea is to get a working prototype to test user response and it is therefore not necessary to offer such functionality just yet. Instead, this tool uses a built in python library, TkInter. TkInter offers a fairly extensive GUI construction toolset, and comes standard with every python installation. The functionality of this package is sufficiently broad to produce the desired interface with an active notepad, and code generation output area. Getting 'dot' output to display in a TkInter window is not a straightforward task, however. To accomplish this, a library called XDot is used which offers some functionality for interactivity by offering node and edge highlighting features. Implemented using GTK, another GUI library for Python, the 'dot' viewer window features a smart zoom for honing in on specific pieces of workflow execution. Although this library does not currently offer the ability to modify graph by dragging and dropping nodes, It holds considerable promise for future development of more interactive features.

With the three main UI components in place, constructing the desired 'active console' was a matter of implementation details. The first step was to build a TkInter window that could accept text in a notepad style. In order to embed the real-time functionalities of the editor, some mechanism for parsing upon editor input was needed. By interfacing with the TkInter action listeners, it was possible to execute a parsing script on upon every key press.

In order to accept the minimized command set that the editor works with, it was necessary to build the framework to detect when a user had specified a template, and furthermore, to find out if they were ready to generate code output or quit the editor. The use of a fully feature parser was considered for this purpose, but with a minimal command set consisting of six commands - sequence, parallel, split, merge, generate and exit - It was found to be much more practical to employ regular expressions. For speed considerations, since a total of six regular expressions would need to be detected upon each key press, a multithreaded solution was implemented. Upon receiving a key press, all 6 regular expressions were assigned to a function, each being the target of its own thread. Because TkInter windows are generally considered to not be thread safe, a function that pipes input to the window was created to assuage these concerns. After a key press, and a call to all 6 regular expression processes is made, each of the functions puts all of its match items onto a queue, where the regular expression 'exit' and 'generate' get their own queue for special considerations. After a join stage, each of the tasks in the queue are placed into a list and sorted based upon their index tuple to accurately represent the execution order of the given workflow. This process occurs continually, unless a match in the exit/generate queue s found, in which case the program quits, or a call to the code generation module, dubbed 'console', is made. A secondary function of the keyPress action listener inherent to the TkInter window is the production of 'dot' graphs. With the appropriate algorithms in place from the visualization stage, this functionality was a matter of extending the functions which converted a workflow of Tigres types to a graph to also accept the minimal command set of the interface. In order to output this graph in it's own window, XDot is employed to call 'dot's graphing algorithms.

V. Conclusion

The purpose of this research was to investigate the possibility of an interface that provides visual feedback and workflow composition capability while remaining minimally intrusive. For the initial test cases, ATT's Dot language was employed to output graphs generated by the execution of Tigres Workflows. To extend this capability, various possibilities for making graphs interactive and editable were explored. D3 has proven to be a very capable library for the construction of dynamic and interactive graphs. However, the challenge of editing and interpreting these edits in D3 remains. Additionally, users may find the flashy nature of the library distracting. Additional research has uncovered a method of applying a jQuery plugin to SVG output of Dot files which may hold the greatest promise moving forward.

Through careful and honest analysis of the existing graphical workflow tools, the need for a more direct connection between graphic and programming domains was discovered. By employing built in Python libraries to build GUI elements which can be executed alongside existing workflows, the focus on the IDE remains paramount while the interface is available on demand. This paradigm leverages the flexibility and feature set available while coding in a modern IDE while preserving the visual capability of other workflow tools.

Acknowledgment

This work was funded by the Office of Science of the U.S. Department of Energy under Contract Number DE-AC02-05CH11231.

Additionally, this work was supported in part by TRUST, Team for Research in Ubiquitous Secure Technology, which receives support from the National Science Foundation (NSF award number CCF-0424422).

References

[1] T. Oinn *et al.*, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

[2] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, and S. Mock, "Kepler: An extensible system for design and execution of scientific workflows," in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pp. 423–424, 2004.

[4] S. AlSairafi, F.-S. Emmanouil, M. Ghanem, N. Giannadakis, Y. Guo, D. Kalaitzopoulos, M. Osmond, A. Rowe, J. Syed, and P. Wendel, "The design of discovery net: Towards open grid services for knowledge discovery," *International Journal of High Performance Computing Applications*, vol. 17, no. 3, pp. 297–315, 2003.

[5] I. Taylor, M. Shields, I. Wang, and A. Harrison, "Visual grid workflow in Triana," *Journal of Grid Computing*, vol. 3, no. 34, pp. 153–169, 2005.

[6] "D3 data driven documents," 2013.

[7] "jsPlumb api," 2013.

[8] "Graph dracula website," 2012.

[9] "Arbor js halfviz website," 2012.

[10] "xdot website," 2013.

[11] "dot jquery plugin website," 2013.

[12] P. Nguyen and M. Halem, "A mapreduce workflow system for architecting scientific data intensive applications," in *Proceeding of the 2nd international workshop on Software engineering for cloud computing*, SECLOUD '11, (New York, NY, USA), pp. 57–63, ACM, 2011.

[13] E. Dede, M. Govindaraju, D. Gunter, and L. Ramakrishnan, "Riding the elephant: managing ensembles with hadoop," in *Proceedings of the 2011 ACM international workshop on Many task computing on grids and supercomputers*, pp. 49–58, 2011.

[14] "Tigres website," 2013.

[15] "Graphviz website," 2006.

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," tech. rep., EECS Department, University of California, Berkeley, Dec 2006.