**SciPaaS: a Python-based execution platform for rapidly deploying scientific applications to the cloud**

Wesley H. Brewer, Ph.D.

Fluid Physics International

P.O. Box 667606

Houston, TX 77266 USA

**ABSTRACT**

SciPaaS is an execution platform middleware designed to make it easy for scientists to rapidly deploy their scientific applications (apps) to the cloud. It provides all the necessary infrastructure for running typical IXP (Input-eXecute-Plot) style apps, including: web interface, post-processing and plotting capabilities, job scheduling, real-time monitoring of running jobs, and even a file/case manager. In this paper, first the system architecture is described and then is demonstrated for a typical scientific application, Mendel’s Accountant—a forward-time population genetics simulation model. Finally, implementation issues with running on Amazon EC2, OpenShift, and Google Compute Engine (GCE) are discussed.

**1. INTRODUCTION**

With the promise of on-demand computing access, cloud computing has become an invaluable resource for computational scientists. The only problem is that most computational scientists that I know have little knowledge about the cloud, and much less about how to possible get their app running in the cloud. The concept of SciPaaS is that a scientist could easily create a zip archive of their code, upload it to the cloud, and SciPaaS would manage all the infrastructure for them, including the input interface, job scheduling, plotting, etc. leaving them the main responsibility of just writing good code that takes some numerical inputs and produces some numerical outputs.

There have been a number of software packages written over the past few years to address the need of being able to run scientific applications in the cloud. Wu et al. (2010) developed a scientific application framework based on OpenSocial gadgets. Unfortunately, the code is not open source, so cannot be freely downloaded and used. Krishnan et al. (2010) developed Opal2, a toolkit basically which can be used to wrap scientific applications and expose them as web services. Opal2 also provides plugin integration with EC2 and Hadoop. Opal2 provides much of the backend infrastructure for running applications, but relies on other software such as Kepler for pre-processing, and other codes for post-processing. Essentially, there was no package I could find that I could simply upload my app and start to run on the cloud. Furthermore, many of the codes available have become quite large and sophisticated, and have rather steep learning curves.

**2. SYSTEM ARCHITECTURE**

The concept for SciPaaS basically came from identifying the common reusable components in many IXP style software systems, such as:

* Interface design
* Scheduling
* Plotting system
* In-process monitoring
* Parallel subsystem

Moreover, by considering a number of similar type software, we can identify some design goals as follows. SciPaaS should:

* Automatically build a web interface
* Manage job execution, scheduling, and monitoring
* Monitor simulation as it is running
* Provide plotting library interface
* Handle multiple users
* Provide a file/case manager
* Easily run on Amazon EC2, Google App Engine (GAE), Google Compute Engine (GCE), or RedHat OpenShift.

To meet these design goals, the following approach was used:

* Use MVT python-based web framework
* Simple DB-based scheduler
* Deploy to free PaaS provider: Heroku, OpenShift, GAE
* Build to support standard scientific apps patterns

After considering a number of alternative languages, such as Java and Ruby, Python was chosen because it seems to have one of the largest scientific computing communities, also because there are numerous open-source web application frameworks available, and finally because many of the cloud PAAS providers support Python-based applications (e.g. Google App Engine and Heroku). Furthermore, in the design of SciPaaS, a convention-over-configuration philosophy was implemented, such that a number of the files (i.e. the binary executable, the default input deck filename, the name of the view template, etc.) all use the name appname. Need table here to show file extensions??

**3. DISCUSSION**

Now we demonstrate these features for a typical scientific application called Mendel’s Account (Sanford et al., 2007), an advanced numerical simulation program for modeling genetic changes over time. The simulation engine of Mendel’s Accountant was developed in Fortran 90 because of its ability to do numerical computation very efficiently.

**Upload app to cloud**. A zipfile containing a default input file and binary of the application can be uploaded to the ScipPaaS. The upload process unzips the file to the appropriate locations, reads the default input deck and then creates an HTML template file views folder named the same as the application. The next section explains how the interface is generated from the input deck.

**Auto-interface generation**. SciPaaS can be used to automatically generate an HTML interface given an input deck. Since Mendel uses Fortran 90, the input deck is provided in namelist format. In the future, SciPaaS should easily be able to handle numerous file formats by using Python’s ConfigParser library. In Figure ??, we show a portion of the Mendel input deck, and then the HTML template file that SciPaaS automatically generates.

**Web framework**. The core of SciPaaS is based on a micro-web framework called Bottle ([bottlepy.org](http://www.bottlepy.org)). This was chosen over a full stack framework to keep the design simple with no external dependencies. Bottle uses a Django-like MVT (model-view-template) architecture. The main purpose of the web framework is to map URL routes to python methods.

SQLite3 is used as the primary database system to manage information about currently installed applications, manage users, and also manage information about plotting. Figure ?? shows the general system architecture of the SciPaaS web application framework, which uses an Model-View-Template (MVT) architecture. The views folder essentially contains templates rendered by Bottle’s template method in a simple way. For example, to render a plot, we can use a simple command such as:

return template('plot', params)

Here, plot refers to the plot.tpl file in the views folder and params contain a Python dictionary of some parameters about the app, the case to plot, and the user.

**Executing the simulation engine**. There are several possible ways to spawn the simulation engine from within Python. One option is to use the subprocess module, which supports either a call method or a pipe. Another method for spawning the engine is to simply to use the system call from the os module. The import point is that the job must be launched in the background so that SciPaaS can continue to handle requests. Currently, a system call is used; however, in the future, a more robust multi-threaded solution should be implemented. The standard output stdout of the simulation is redirected to the file *appname*.out (e.g. mendel.out in the current example).

**Monitoring the simulation**. Once the simulation has been launched, SciPaaS automatically redirects to the monitor view. The monitor view is essentially a jQuery AJAX call which repeatedly calls a method called tail, which retrieves the last 40 lines of the output file every second.

**Plotting the data**. At any time during the simulation or after running, the user can plot data from the simulation. As shown in Figure ??, when the user clicks the plot button, a list of defined plots is returned. The user can then choose to plot one of the available plots, or can define a new custom plot.

The plotting system is handled using a jQuery library called flot ([www.flotcharts.org](http://www.flotcharts.org)), which they describe as “a pure JavaScript plotting library for jQuery, with a focus on simple usage, attractive looks and interactive features”. In order to prepare the data for plotting, a plot class is defined within SciPaaS. This class contains a method called get\_data, which extracts the necessary data from the files, and returns it in JSON format.

Inputs

Execute

Post-process

P1

P2

P3

P4

Parallel Processing

**Figure 1. Many scientific applications fall under an Input-eXecute-Plot (IXP) design pattern.**

Monitor

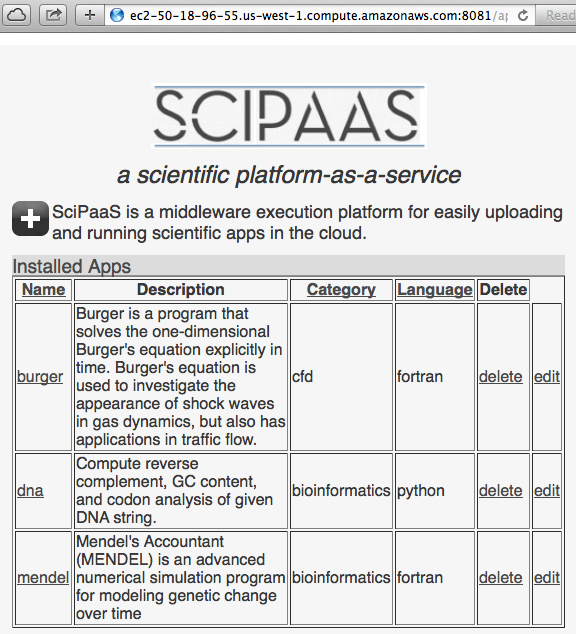


Figure 1. Showing apps view on EC2.

**5. CONCLUSIONS**

A middleware execution platform called SciPaaS was described and demonstrated Mendel’s Accountant, a forward-time population genetics simulator.

Future work includes:

* Supporting numerous input formats via Python’s ConfigParser class.
* Implementing a generalized workflow to be able to handle custom-defined workflows, i.e. non-IXP type workflows.
* Interface to manage virtual machines.
* Interfacing with other web services, such as the Opal2 toolkit
* Supporting parallel execution infrastructure, such as interfacing with MPI and Hadoop.
* Implement multi-threaded approach to spawning jobs.

DataBase

Model

View

Template

URL dispatcher

Brower

Figure 2. MVT architecture.



Figure 3. Monitoring output of simulation which is constantly updated using AJAX approach.

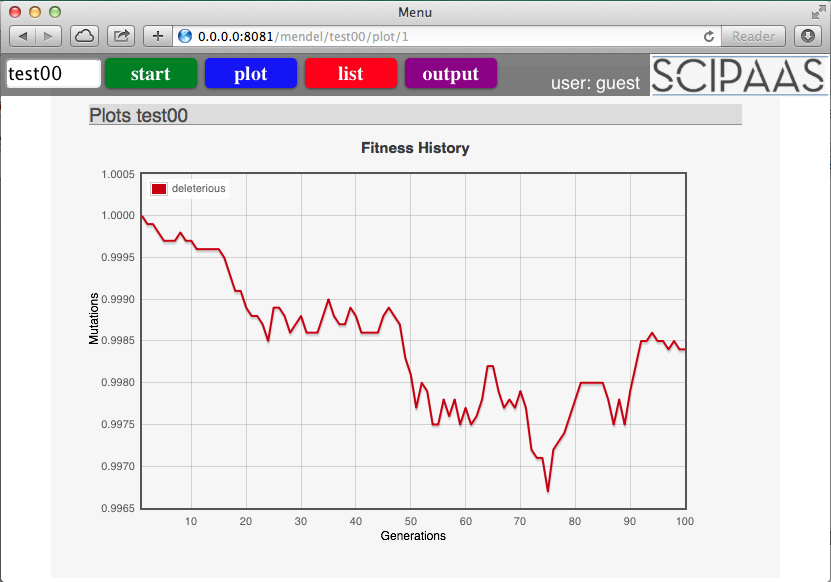


Figure 5. Example of jQuery/JavaScript plot for Mendel.

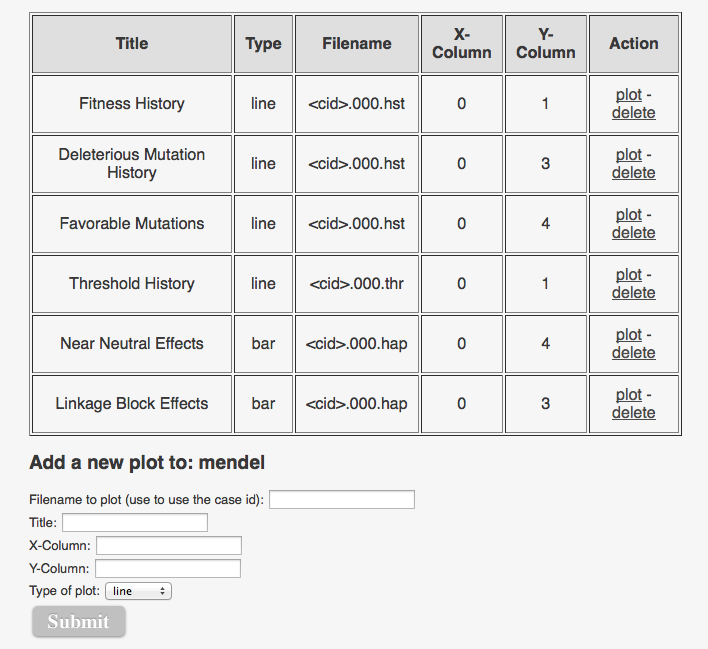


Figure 6. Overview of plotting interface.

**6. REFERENCES**

1. Wu, W., Uram, T., Wilde, M., Herald, M., and Papka, M. “A Web 2.0-Based Scientific Application Framework”, IEEE 2010.
2. Krishnan, S., Clementi, L., Ren, J., Papadopoulos, P., and Li, W. Design and Evalution of Opal2: A Tookit for Scientific Software as a Service. 2010
3. Sanford, J., Baumgardner, J., Brewer, W., Gibson, P., & ReMine, W. (2007). Mendel's Accountant: A biologically realistic forward-time population genetics program. *Scalable Computing: Practice and Experience*, *8*(2).
4. “The Django Web Application Framework” by Zhixiong Hong 2009.3.26. http://www.slideshare.net/fishwarter/the-django-web-application-framework-2-1221391.

Figure 7. SciPaaS converts input deck to HTML input form.

MENDEL input deck

HTML interface

