Fourth IEEE International Conference on eScience

Services Oriented Architecture for Managing Workflows of Avian Flu Grid

Luca Clementi, Sriram Krishnan, Wesley Goodman, Jingyuan Ren, Wilfred W. Li, Peter W. Arzberger
National Biomedical Computation Resource, University of California, San Diego
{clem, sriram, wgoodman, jren, wilfred}@sdsc.edu, parzberger@ucsd.edu

Guillaume Vareille, Sargis Dallakyan, Michel F. Sanner
The Molecular Graphics Lab, The Scripps Research Institute
{vareille, sargis, sanner}@scripps.edu

Abstract

The Avian Flu Grid is a virtual organization dedicated to the discovery of novel inhibitors for the pandemic avian flu threat, leveraging grid technologies and computational resources provided by PRAGMA and its partners. In this context it is essential to adopt tools which increase the productivity of the computational scientists without advanced training on grid technologies. To reduce the learning curve, we have augmented tools that most domain scientists are already familiar with, and hidden the complexity of the underlaying infrastructure, through automatic user interface generation and workflow support, beyond the standard command line based approach. Here we describe the current state of the infrastructure deployed and how it has facilitated the training of new researchers in the drug discovery area. We provide details on Vision, a visual programming environment used to define scientific workflows, and Opal, an automatic Web service wrappers for scientific applications on Grid resources, and how they are integrated with other established back-end technologies.

1 Introduction

PRAGMA [27] (Pacific Rim Application and Grid Middleware Assembly) was formed in 2002 to establish sustained collaborations for advancing the use of grid technologies in applications among a community of investigators working with leading institutions around the Pacific Rim. The Avian Flu Grid (AFG) [1] is a research project and a virtual organization (VO) of the PRAGMA grid that leverages the computational tools and resources contributed by members of the PRAGMA community to develop a global infrastructure for the analysis of avian flu as an infectious agent and a pandemic threat.

To discover new candidate drugs, scientists need to evaluate large libraries of compounds against suitable avian flu virus targets. This process, commonly referred to as Virtual Screening [15], has very high computational requirement depending on the number of compounds. AutoDock is one of the most suitable tool for this purpose [20]. It is a suite of automated docking tools designed to predict how small molecules, such as substrates or drug candidates, bind to a receptor of known 3D structure. It is composed of a set of command line based applications that have to be executed in a particular order. Each of these commands requires several input files from either previous steps or user inputs, and produces output files for subsequent steps.

The process of setting up AutoDock experiments could be very complex for beginners. Even a trained biologist would have problems remembering all the necessary steps without making any mistakes. Figure 1 shows a simplified version of the workflow in an Autodock virtual screening experiment, where rectangles represent executable applications (Autogrid, Autodock, and PrepareGPF) and arrows represent the input and output files in the workflow.

Learning this process can be a time consuming task, even for a scientific programmer. Research scientists would rather not deal with the complexity of accessing distributed computational resources across several institutions [27]. For these reasons we have adopted several grid middleware components: (i) to hide complexity of the underlying computational infrastructure, and (ii) to simplify usage of this complex scientific application by means of a more intuitive graphical environment, and preferably one that the users are already familiar with. To address the first issue we have leveraged the Opal toolkit which enables wrapping scientific applications as Web services, while encapsulating standard grid security mechanisms, (meta)schedulers, and state management for jobs. For the second problem, we have enhanced several existing Problem Solving Environments (PSEs) such as the Python Molecular Viewer and AutoDockTools [5] with distributed computing capability.

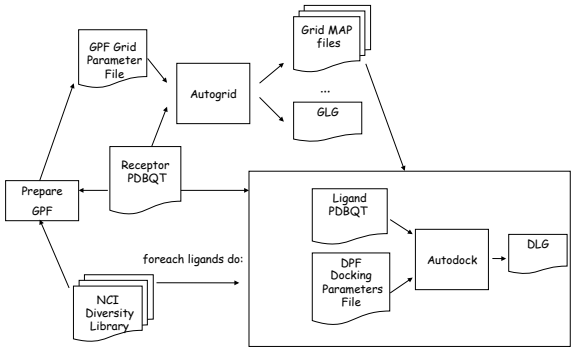

Figure 1. Virtual screening for the Avian Flu Grid

However PSEs can only perform a predefined set of tasks. For greater flexibility and better code reuse, we have begun to add support for visual programming environments (VPEs), which allow the creation of scientific pipelines, e.g., Vision [24], Kepler [12], and Taverna [9], to name a few.

In the following sections we discuss how we have used these tools to help AFG researchers in pursuing their scientific goals, providing them with remote computation resources and simplifying the usage of their command line applications through automatically generated graphical interfaces. In particular in Section 2, we introduce our main tools, Vision and Opal. In Section 3 we present first the implementation details of the work done to integrate these two tools and in the second part we discuss some real scientific use cases. We discuss our experiences and future work in Section 4, and present out conclusions in Section 5.

2 The Avian Flu Grid Infrastructure

In the next two sections we describe the two generic tools, Vision and Opal, that we have adopted in order to support AFG scientists in discovering novel candidates for drug development.

2.1 The Vision Environment

Vision (originally called ViPEr) [24] has been developed as a component-based, application-domain agnostic, cross-platform visual programming environment. It is designed as an extension to the Python programming language thus providing a fully fledged, high level, object-oriented, interpreted language for the interactive manipulation of data and/or modification of the application. Some of the key features of Vision are described below:

1. **Framework:** It is designed as a set of libraries of computational nodes providing specific functionality. These libraries are loaded dynamically to reduce the footprint of Vision in memory, and also to avoid cluttering the interface. Vision network nodes are Python objects that provide input ports and output ports. Each library can define its own data types, with data type validation. A node can only run if all required ports present valid data, and triggers may be used to further refine the execution order of workflows. Introspection tools allow the user to inspect data on input ports and output ports in real time.

2. **Node:** Each node contains a compute function with a set of parameters, configured by the user before running the workflow. The difference between a parameter and an input port is that parameters do not change during the execution of the workflow and do not need to be piped by some other nodes. Vision gives also the possibility to change a parameter into an input port. Hence, when developing a node, it is not necessary to decide whether an input field for the computing function should be a parameter or a port. A node can be designed interactively in Vision using the node editor (as seen in Figure 2). All modifications made to a node or a network are saved in the network description file. This file contains Python code that recreates the network when executed. Since the node's compute function is written in the Python programming language, it can be inspected and modified interactively.

3. **MacroNodes:** Vision also supports the concept of "MacroNodes", which appear as a simple node in a network, but encapsulate a sub-network. MacroNodes can exist inside MacroNodes, thus allowing a hierarchical visual representation of an algorithm, and facilitating code reuse within Vision. In addition, it allows the workflow diagram to be free of details of the subnetworks, greatly facilitating the design and implementation process by different users.

4. **Execution:** When a node is scheduled for execution, it collects the data on its input ports and passes it to the node's compute function. This function typically imports Python code from some other Python package to operate on the data and produce a result, which is then sent to the node's output port. This important design features makes Vision "just another" user interface to functionality available to any program running

a Python interpreter, thus promoting code re-use and inter-operability.

5. **Libraries:** The standard library comprises of nodes exposing Python keywords (print, eval, setattr and call methods), along with nodes enabling the generation of simple input data. Other available libraries are the 3-D Visualization library based on the DejaVu component, the SymServ library which provide geometric transformations, the MolKit library to manipulate molecule data, the Python Image Library (PIL) to manipulate images. The wslib for web services is described later. Figure 2 shows a sample Vision workflow.

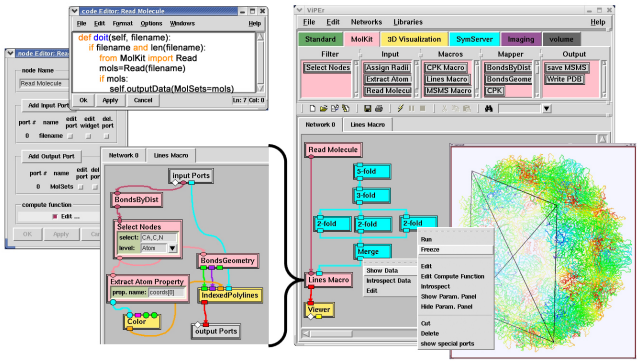

Figure 2. A molecular visualization application built using Vision – a network used to display a viral capsid is shown, the sub-network embedded in the Lines Macro is shown as an inset

Several similar tools have originated from different scientific communities to support the design and execution of workflows, such as Kepler, Taverna, Vistrails, SCIRun, and AVS, each with a different focus. For a more comprehensive comparison, please refer to [11]. In Table 1 we drew a simple comparison of some of these tools, whose features often change rapidly depending on requirements of their intended user communities.

The choice of a tool is often decided upon the available features, the stability of software, the user base, and the programming language. Our choice of Vision is motivated by its advanced support for volume rendering, 3-D visualization of proteins and other molecules, integration with open source electrostatics codes, and a strong user base in the AutoDock and AutoDockTools community. In addition, the simple yet powerful programming paradigm of Python has made development of new Vision libraries straight forward. By adopting a service oriented architecture, users have even less to worry about which tools to use except for their personal preferences.

Workflow Tool	Parallel Execution of Tasks	Programming Language	Notable Features
Vision	no	Python	Advanced visualization, web services support
Kepler	yes	Java	Ontology based annotation
Taverna	no	Java	Web services and provenance
Vistrails	no	Python	Data provenance and version tracking

Table 1. Workflow tools comparison

2.2 The Opal Toolkit

Programming and using distributed grid resources is quite a challenging task. A Service Oriented Architecture (SOA) can address this problem by providing platform-independent, language-neutral service interfaces that hide the complexity of the implementations while providing a well-defined and high-performance Quality of Service (QoS). As outlined in [16], SOA systems for scientists should be designed focusing on application-level services in a domain oriented fashion, providing reusable interfaces, and lowering the deployment cost.

The Opal toolkit [22] is such a simple toolkit for wrapping scientific applications as Web services. Application developers are expected to write a simple XML-based application configuration. The application configuration contains information about the scientific application, such as application name, binary location, and application metadata (e.g. usage information). Using a simple Apache Ant task, the Opal toolkit can deploy the application as a Web service into a container based on Apache Tomcat and Axis. Once the application is deployed as a Web service, clients can access this service programmatically using its Web Service Description Language (WSDL) description. The WSDL API provides operations for job launch (which accepts command-line arguments and input files as its parameters), querying status, and retrieving outputs. Furthermore, it provides an API for retrieving application metadata. WSDL savvy users could write their own clients to access Opal based applications. However, not every scientific user is capable of or even interested in writing Web ser-

vice clients. Hence, several interfaces have been provided for the end-users. Apart from command-line clients written in languages like Java, Python and Perl available from the Opal Sourceforge site. Examples of web interfaces for Opal based services are available through the GridSphere portal environment [2], such as the MEME Portlet in My WorkSphere [18]. In addition, Opal based interfaces via Rich Internet Applications (RIA) such as Gemstone [14], and Problem Solving Environments (PSE) such as the Python Molecular Viewer (PMV), AutoDockTools [5] and Continuity [19] have also been provided.

Although a detailed description of the Opal toolkit is beyond the scope of this paper (and can be found in [22]), the salient features of Opal are as follows:

1. **Scheduling and Cluster Management:** Since different sites use different schedulers such as the Sun Grid Engine, Condor [7], etc., Opal provides a uniform way to access them via standard APIs such as Globus GRAM [8] and DRMAA [21]. The schedulers are simply specified using a properties file, and Opal configures itself appropriately to access the schedulers via the appropriate parameters. In addition, we are currently able to submit jobs to the CSF4 metascheduler [25] as described in [17]. Leveraging its WSRF interface [13], application based scheduling and transparent data staging, we have been able to transparently couple it with Opal to dispatch jobs on PRAGMA resources.

2. **Data management and Persistence:** The Opal toolkit manages the job data for a particular run on behalf of the user. Every job is run in a separate workspace so that multiple jobs from different users can be run concurrently. Furthermore, information about job status and outputs is persisted in a database to provide fault-tolerance for the services.

3. **Security:** Opal services can be configured to use transport-level GSI-based [10] security so as to restrict access to only authorized users via standard Grid security mechanisms.

4. **Standardized WSDL API:** All Opal services use a standard WSDL so that they can be accessed in a uniform manner by different clients. We choose not to generate application specific WSDLs to avoid regeneration of client-side stubs and rewriting of clients for every application.

5. **Argument Description and Dynamic Interface Generation:** Opal services can be optionally configured with an XML specification for the command-line arguments. This specification consists of a description for flags, tagged and untagged parameters, and grouping of arguments. Flags are not ordered and are usually represented with a character prefixed with a dash and they activate a functionality in the application (e.g. -verbose). Tagged Parameters are usually formed by a prefix and input value (e.g. -input <filename>) - they can appear in any order. Untagged parameters are not prefixed, and hence their order is relevant. The argument description is mainly useful for two purposes - for validation of command-line arguments passed to the application, and for automatic interface generation. A detailed description of the argument description, and automatic generation of Web forms from the description is provided in [17]. A complete life cycle of this process is shown in Figure 3. We will discuss dynamic interface generation in the context of the Vision environment in Section 3.1.

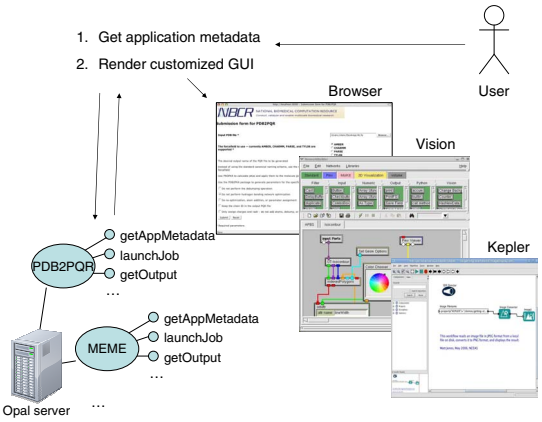

Figure 3. Service invocation via the Opal GUI

The Opal toolkit is freely available from SourceForge, and uses an Open Source BSD-style license [6].

3 Implementation Details and Use Cases

3.1 Vision and Opal Services

As we described in Section 2.1, the Vision environment was mainly designed to run workflows on the users' workstations. We have now augmented Vision to access distributed resources via the Opal toolkit. To do so, we have developed the "wslib" library, which we describe in this section.

Users can now load remote Web services by loading the wslib library. This library contains a set of generic Vision nodes for accessing Web services, and a set of service-specific nodes that are generated on the fly. The generic

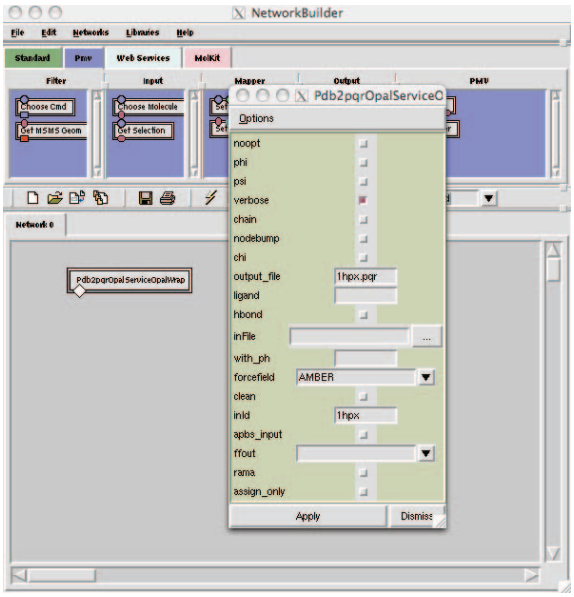

Figure 4. Automatic interface generation for the PDB2PQR service in Vision

nodes of interest are:

1. **Load WS:** This node is responsible for looking up a server for a list of Web services deployed using Opal, and creating one application specific node for every service it finds. This node gets a list of services hosted on the remote server from the Apache Axis Servlet, parses the list, and generates the necessary code for the service nodes. For instance, Figure 4 shows a list of service nodes that have been automatically generated for services being hosted on the ws.nbcr.net server. These nodes can now be dragged over to the network editor, and used to access applications running on remote resources.

2. **Download:** This node is useful for downloading outputs from an Opal service after the execution is complete.

3. **Web Browser:** This node pops up a Web browser to access results for a remote Opal job run.

When the Opal service node is dragged on to the network editor, it fetches the service metadata which consists of the flags, tagged and untagged parameters for the command-line arguments as described in Section 2.2. From the Opal metadata, it automatically generates a Python class to access the remote service. Every entry in the Opal metadata is mapped to a Vision widget as follows:

1. Flags are displayed as check boxes,

2. Tagged and untagged parameters are rendered as check boxes if their type is BOOL, file input fields (with a file browser) if their type is FILE, drop down lists if their type is a list of mutually exclusive values, and input text fields for all other cases.

Furthermore, a description of the parameter is displayed if the mouse is pointed over a particular widget. When such a node is run, it marshals the command-line arguments using the input form and the metadata information and launches a remote application via the Opal Web services API, using the ZSI (Zolera SOAP Infrastructure) [23] client-side libraries. Once it has finished execution, the "Download" or "Web Browser" nodes can be used to access the results.

With the advent of multi- and many-core CPUs and the constant increase in computational power of personal workstations, scientific users might prefer to execute some of their computational tasks locally in order to avoid network latencies or waiting in the queues. Opal nodes can be configured to use either remote or local binaries. When a Opal node is created by Vision, it searches an executable file on the client's computer with the same name as the one published by the remote service. Otherwise the user can also input this value manually. Finally a check box in the properties of the node allows the user to choose whether the computation should be performed locally or remotely.

Figure 3 shows an example of an automatically generated interface for an Opal service inside Vision. Flags "noopt", "phi", "psi", "verbose", "chain", "nodebump", "chi", "hbond", "clean", "apbs_input", "rama", and "assign_only" are rendered as check-boxes. Tagged parameter "inFile" is rendered as a file browser, while "forcefield" and "ffout" are rendered as drop down lists since they are associated with a set of mutually exclusive values. All other parameters are rendered as text boxes.

3.2 Use Cases

Vision and Opal are used in practice for molecular visualization and virtual screening workflows. In particular, Vision can use Opal to run jobs on remote clusters and interoperate seamlessly with the 3D visualization capabilities provided by the Python Molecular Viewer (PMV).

3.2.1 Protein Docking using AutoDockTools

AutoDockTools [5] is a graphical front end for setting up, launching and analyzing AutoDock runs. One benefit of ADT is that it can set up and run all AutoDock applications from a graphical user interface. In addition, ADT uses Vision networks or python modules to analyze and display molecular surfaces, secondary structures, and visualize molecular dynamics trajectories.

Typically ADT is useful to set up docking experiments and execute on users' workstations. We have augmented ADT to leverage Grid resources using the Opal toolkit. ADT uses Python stubs generate by ZSI [23] to access remote Opal-based AutoGrid and AutoDock services. Users can specify the URL for the services, and first run AutoGrid to set up the Grid parameter files. Outputs from AutoGrid are then passed to the AutoDock service, which then runs the docking simulations. In this case, the flow of control is hard-coded within ADT itself. For training purposes, we have developed a workflow in Vision that performs the same steps as ADT. The visual representation of the tasks involved in the simulation as a network of interconnected nodes help new users to learn the logical steps of an AudoDock simulation. We have used it in training the undergraduate students in the PRIME project [4], a NSF funded activity which supports undergraduate students to conduct research abroad.

3.2.2 Virtual Screening within the Vision environment

In a typical scenario of virtual screening using AutoDock, scientists have to adapt complex shell scripts to code the workflow presented in Figure 5, where they submit jobs on a cluster to search against each one of the compounds present in the library. To increase code reuse, and to leverage distributed resources, we provide a Vision based workflow environment for the workflow. We deployed a set of Opal services to wrap the most computational intensive steps, and use a Vision network to coordinate these services in proper order. Figure 5 shows: ADT user interface which is invoked from our workflow to set up a grid box for the docking experiment; the Vision workflow editor can be used to change simulation parameters; and a html browser which is opened at the end of the simulation to browse the results remotely.

3.3 Deployment Scenarios and Performance Evaluation

The Vision network may be used to access individual clusters or distributed resources in the PRAGMA grid. For example, We have deployed Opal on individual clusters such as Kryptonite, a NBCR production cluster, running the Rocks cluster distribution (http://www.rocksclusters.org), with a Gigabit Ethernet network. The batch queuing system is the Sun Grid Engine. All the client requests are sent to the login node of the cluster where an Apache web server (http://httpd.apache.org/) redirect them to the Tomcat server where Opal is running. Opal is configured to submit its jobs to the local scheduler, so that they get treated exactly like standard Unix user. Currently the virtual screening is performed against the NCI Diversity Set [3] of almost two thousands compounds, and more libraries are supported as

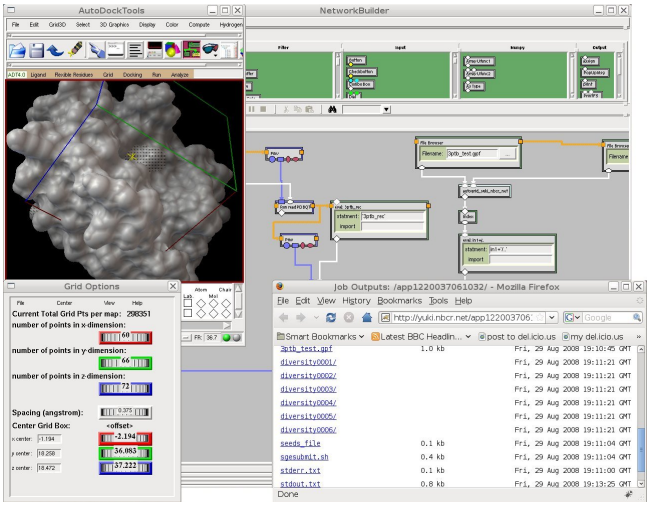

Figure 5. Using Vision remote capabilities to perform virtual screening

required. We have also used Opal-CSF4 from a command line client [26] or a web interface described earlier [17] to access distributed resources.

There is some overhead in using Vision and Opal, but negligible when the tasks are running on the order of minutes or more. For example, we created a Vision workflow to simply perform a remote AutoDock submission to detect this overhead. We set up an Opal instance on one of our developmental cluster with duo Intel Xeon CPU's at 3.06GHz, gigabit Ethernet network and 2 gigabyte of RAM per node. Opal was set up not to use the batch scheduler but to fork the process on the local system, additionally the AutoDock executable was wrapped inside a bash script created to log execution times. Vision was running on a Intel Core 2 Duo 2.40GHz with a gigabit Ethernet network adapter and 2 gigabyte of RAM. The two systems were located into two different buildings interconnected by the campus gigabit Ethernet network.

We performed 4 sets of experiments, each one consisting of ten executions, and we observed the average of the execution times. The first set was executed running an empty bash script in order to evaluate the total execution time of Vision and Opal. The other tree experiments were performed changing AutoDock configuration parameters in order to increase its execution time from 27.81 second to 36.38 and to 44.53. Essentially, the difference between the workflow execution time and the Autodock execution time indicated that the overhead introduced by Vision and Opal remains constant against the total execution time varing between 3 to 4 seconds.

We also explored how Opal-Vision overhead is impacted

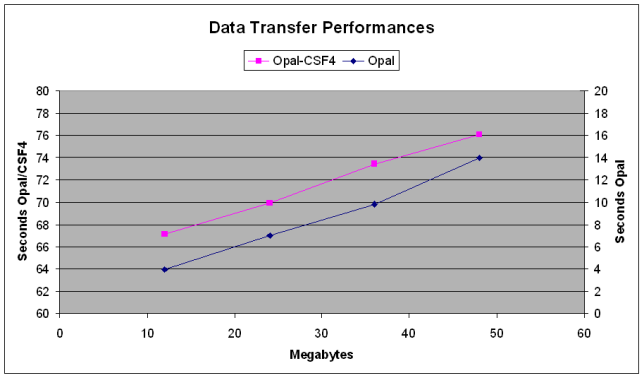

Figure 6. Execution time with growing input data set

by the amount of data transferred by performing a set of experiments invoking an empty script (i.e. with no execution time) with different input data size. We ran 4 simulations with 12 files in each one but we increased the total transfer size from 12 to 48 Megabyte. Additionally we also set up an instance of Opal configured to submit jobs to CSF4 metascheduler [25]. In this case the input data is also moved from the Opal server to the execution host by means of GridFTP which adds an additional transfer time cost. Figure 6 shows how the execution time grew linearly with the size of the input data set in both cases. CSF4 introduced about a min of overhead to the test workflow, from the graph it is also possible to notice how CSF4 data staging does not impact sensibly the final execution time. Therefore, when adding a metascheduler to dispatch jobs, one needs to minimize the amount of data transferred when possible.

These preliminary experimental results suggest that the most important factor when analyzing the overhead introduced by our middleware, is the input data size and the number of jobs submitted. Hence it is important that when designing service oriented infrastructure the granularity of the deployed services is established in such a way that data transfer and number of submissions are minimized.

From the users' perspective, e.g., PRIME students, it typically takes between two to four hours of training to get them confident with the screening process. Then at least two more hours are required to teach how to access our Unix clusters (ssh, scp, bash scripting, etc.). Using Vision and Opal we have been able to eliminate completely the second part and reduce the time to learn the first.

4 On-Going and Future Work

Currently, we are working on a new release of Opal that supports a plug-in architecture for the various backends submission systems. This would allow the users to use other metaschedulers such as GridWay or Nimrod, in addition to CSF4. Additionally, the use of Hibernate (http://www.hibernate.org) will increase the number of supported databases for state management. User data sources may include common protocols such as GridFTP, FTP, in addition to HTTP.

In [22], we discussed a bioinformatic workflow using Kepler and Opal. However, the downside to that approach was that we had to write application specific actors for the scientific applications that we were interested in. The automatic interface generation approach used in Vision would obviate the need to write such application specific actors. We are currently developing a generic Opal actor for Kepler [12] which first retrieves the metadata about the command-line arguments from the Opal service, and then automatically adds parameters and input and output ports to itself.

5 Conclusions

Grid computing provides an enormous opportunity to scientific end-users to use distributed computing and storage resources to enable novel science hitherto impossible using traditional computing mechanisms. In this paper, we described the workflow tool called Vision, a component-based, application-domain agnostic, cross-platform visual programming environment, and how it has been augmented to leverage Opal based web services using Grid resources. Further enhancements to such tools would help scientists perform their tasks in a more intuitive way without the hassle of access grid resources. Advances in visual programming environments may one day enable non-programmers to intuitively and interactively build scientific workflows and visualize their results, without having to write code or understand the details of the back-end infrastructure.

The authors would like to acknowledge support from the NIH P41 RR 08605 to NBCR; TATRC W81XWH-07-2-0014 to PWA, the Gordon and Betty Moore Foundation grant for the CAMERA project, and the NSF grants INT-0314015 and OCI-0627026 for the PRAGMA project.

References

[1] Avian Flu Grid. http://avianflugrid.pragma-grid.net.

[2] Gridsphere Portal Framework. http://www.gridsphere.org.

[3] NCI - Diversity Set Information. http://dtp.nci.nih.gov/branches/dscb/diversity_explanation.html.

[4] Pacific RIM undergraduate Experience. http://prime.ucsd.edu/.

[5] Python-based software development at MGL - AutoDockTools, PMV, and Vision. http://mgltools.scripps.edu/.

[6] The Opal Toolkit. http://nbcr.net/software/opal/.

[7] J. Basney, M. Livny, and T. Tannenbaum. High Throughput Computing with Condor. In *HPCU news*, volume 1(2), June 1997.

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *IPPS/SPDP 98, Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[9] T. Oinn et al. Taverna: A tool for the composition and enactment of bioinformatics workflows. In *Bioinformatics Journal*, volume 20(17), 2004.

[10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *ACM Conference on Computers and Security*, 1998.

[11] GC Fox and D. Gannon. Workflow in Grid Systems. *Concurrency and Computation: Practice and Experience*, 18(10):1009–1019, 2006.

[12] I. Altintas et al. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, 2004.

[13] K. Czajkowski et al. WS-Resource Framework. http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf, 2004.

[14] K. K. Baldridge et al. GEMSTONE: Grid-Enabled Molecular Sciences through Online Networked Environments. In *Life Sciences Grid Workshop (Satellite of Grid Asia)*, 2005.

[15] W.P. Klters, M.T. Stahl, and M.A. Murcko. Virtual screening-an overview. *Drug Discovery today*, 3(4):160–178, 1998.

[16] S. Krishnan and K. Bhatia. SOAs for Scientific Applications: Experiences and Challenges. *e-Science and Grid Computing, IEEE International Conference on*, pages 160–169, 2007.

[17] L. Clementi, et al. Providing Dynamic Virtualized Access to Grid Resources via the Web 2.0 Paradigm. *International Workshop on Grid Computing Environments*, 2007.

[18] W.W. Li, S. Krishnan, K. Mueller, K. Ichikawa, S. Date, S. Dallakyan, M. Sanner, C. Misleh, Z. Ding, X. Wei, et al. Building cyberinfrastructure for bioinformatics using service oriented architecture. *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops Singapore*, pages 39–46.

[19] AD McCulloch. Continuity 6.0: Continuum Modeling for Bioengineering and Physiology.

[20] Garrett M. Morris, David S. Goodsell, Robert S. Halliday, Ruth Huey, William E. Hart, Richard K. Belew, and Arthur J. Olson. Automated docking using a lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry*, 19(14):1639–1662, 1998.

[21] H. Rajic, R. Brobst, W. Chan, F. Ferstl, J. Gardiner, A. Haas, B. Nitzberg, and J. Tollefsrud. Distributed Resource Management Application API Specification 1.0. *Grid Forum Document GFD*, 22, 2004.

[22] S. Krishnan et al. Opal: Simple Web Services Wrappers for Scientific Applications. In *IEEE International Conference on Web Services*, 2006.

[23] R. Salz and C. Blunck. ZSI: The Zolera Soap Infrastructure.

[24] M.F. Sanner, D. Stoffler, and A.J. Olson. ViPEr, a Visual Programming Environment for Python. *Proceedings of the 10 thInternational Python Conference*, pages 4–7, 2002.

[25] X. Wei, Z. Ding, S. Yuan, C. Hou, and H. Li. CSF4: A WSRF Compliant Meta-Scheduler. *International Conference*, 2006.

[26] Z. Ding, X. Wei, O. Tatebe, P.M. Papadopoulos, P.W. Arzberger, and W.W. Li. *Cyberinfrastructure for biomedical applications: metascheduling as essential component for pervasive computing*. Nova Science, 2009.

[27] C. Zheng, D. Abramson, P. Arzberger, S. Ayyub, C. Enticott, S. Garic, M. J. Katz, J.-H. Kwak, B. Sung Lee, P. M. Papadopoulos, S. Phatanapherom, S. Sriprayoonsakul, Y. Tanaka, Y. Tanimura, O. Tatebe, and P. Uthayopas. The pragma testbed - building a multi-application international grid. *International Symposium on Cluster Computing and the Grid*, 2:57, 2006.