Compute and data management strategies for grid deployment of high throughput protein structure studies

Ian Stokes-Rees
Dept. of Biological Chemistry and Molecular Pharmacology
Harvard Medical School
Boston, MA 02115
ijstokes@hkl.hms.harvard.edu

Piotr Sliz
Dept. of Biological Chemistry and Molecular Pharmacology
Harvard Medical School
Boston, MA 02115
sliz@hkl.hms.harvard.edu

***Abstract*—The study of macromolecular protein structures at an atomic resolution is the source of many data and compute intensive challenges, from simulation, to image processing, to model building. We have developed a general platform for the secure deployment of structural biology computational tasks and workflows into a federated grid which maximizes robustness, ease of use, and performance, while minimizing data movement.**

This platform leverages several existing grid technologies for security and web-based data access, adding protocols for VO, user, task, workflow, and individual job data staging. We present the strategies used to deploy and maintain tens of GB of data and applications to a significant portion of the US Open Science Grid, and the workflow management mechanisms to optimize task execution, both for performance and correctness. Significant observations are made about real operating conditions in a grid environment from automated analysis of hundreds of thousands of jobs over extended periods.

We specifically focus on one novel application which harnesses the capacity of national cyberinfrastructure to dramatically accelerate the process of protein structure determination. This workflow requires $20 - 50$ thousand hours to compute with $1e5$ tasks, requiring tens of GB of input data, and producing commensurate output. We demonstrate the success of our platform through the successful completion of this workflow in half a day using Open Science Grid.

I. Introduction

SBGrid is a consortium of structural and molecular biology researchers from over 120 labs across the US. The consortium supports the distribution of a large package of scientific software (over 100 packages) which are commonly used by the community. With the support of two NSF grants, our lab, which heads the consortium, has been able to do five important things:

1) Introduce the SBGrid community to grid computing using the US national cyberinfrastructure: Open Science Grid (OSG)[1];
2) Deploy key structural biology applications onto OSG from the SBGrid software distribution;
3) Develop novel compute and data intensive workflows that can harness the opportunistic computing capacity made available by OSG;
4) Expand access to grid computing to life science researchers in the New England (Northeast USA) area, leveraging the concentration of hospitals, universities, and researchers in and around Boston, USA;
5) Provide unified web and command-line (shell) access to applications, jobs, and data.

To achieve this, we have developed layered reference models for applications, data, and workflows with the goal of enabling large job set workflows ($1e3$ – $1e6$ atomic jobs), with large data volumes ($>$ 10 GB input and output). While these have primarily been developed in the context one particular novel workflow developed by our lab, the tools and strategy are sufficiently generic to support a range of structural biology workflows with independent tasks, data aggregation, analysis, and web-based access/interaction. This builds on the accumulated experience of grid-enabled parameter sweep environments such as Nimrod/G[2], and the more recently codified map/reduce strategy popularized by Google[3] and Amazon Hadoop[4]. While map/reduce or parameter sweep frameworks provide some foundational tools, they do not address the full set of requirements which arise in an integrated system where a large volume of data is processed through a large number of interrelated steps. This is the domain of *many task computing*[5]. In our particular context, end users are not as concerned about tools to construct and run dynamic workflows – these can be seen as analogous to graphical programming languages and execution managers (e.g. Taverna[6] or Kepler[7]) – but rather the construction of robust static workflows which manage gigabytes of data and thousands of hours of computing.

This paper will outline the application, data, and workflow models, discuss operational issues for deployment of large workflows to Open Science Grid and describe the unified web portal and command line access modes.

II. Application Model

Many applications within structural biology are Fortran based programs (e.g. the CCP4[8] application suite). These have rudimentary interfaces based on command files which

978-1-4244-9705-8/10/$26.00 ©2010 IEEE

are piped through standard input, and output is typically a combination of fixed name output files and results reported on standard output in a verbose, text-based, human readable format. The applications themselves are statically compiled. There are several key goals of the application model:

1) Provide a programmatic interface to applications (API);
2) Unify application interaction between command line; workflow/integrated application, local web portal invocation, local cluster invocation, and grid job invocation;
3) Enable bulk application invocation for parameter sweep, map-reduce processing, and other novel compute intensive uses;
4) Manage application output to improve automated aggregation and analysis;
5) Handle application errors in a more robust way;
6) Improve application configuration (options of command line arguments, environment variables, and configuration files)

Traditionally, cluster computing has been dominated by shell script wrappers around applications. This same tradition has continued with grid computing. Shell scripting provides a fast and portable way to establish a suitable environment (variables, files, directories, command invocation) and as a "glue" language to perform necessary "pre" and "post" operations around application invocation, and application chaining. It does not, however provide rich features suitable for layered application development, error handling, or interfacing from other environments (e.g. a GUI or web sever). Conversely, the level of interaction with our applications of interest is such that starting with some standardized RPC mechanism such as XML-RPC, Web Services or RESTful interfaces also would be inappropriate. To address these various requirements we have standardized on Python as a development language with which we write wrapper modules providing APIs to the underlying applications. These APIs can then be used either for direct application integration, or a further CLI layer can be added which allows a more conventional (and standardized) command-line style access to the underlying application.

To facilitate standardized configuration and the need for system-level "glue" operations, two Python modules have been developed, *shex* and *xconfig*. *shex* provides a single Python module that consolidates and expands on the shell-scripting type operations commonly required when providing cluster or grid application and job wrapper scripts (e.g. commands such as `cat`, `mkdir`, `chmod`, `chown`, `head`, `pushd`, `popd`), and some "helper" features, such as automatic variable interpolation and shell expansion (e.g. `cat('~/.globus/$mycert')`). `xconfig` provides many `ConfigParser`-like features, but also boasts a command line interface for direct interaction with the shell and user environment. It supports both a basic key/value pair style configuration dictionary, and a richer hierarchical configuration model, with API operations to set, merge, or override configuration settings directly or via free-format XML-based configuration files (no schema is asserted on these).

The unified interface allows users to invoke applications on the command line to interactively develop a workflow or computation and then directly use the same command structure (or a trivial translation to a programmatic API call) in their grid job. It is also possible for the web portal to directly invoke the application for "instant" synchronous data processing using the same interfaces. For this facility to be reliable across the heterogeneous grid infrastructure it is necessary to maximally unify the software and execution environment for shell-access users, the web portal service, and remote grid jobs. Given that we are administratively constrained at remote grid sites, it is necessary to model this execution environment maximally after the grid job environment, which provides some standard conventions dictated by Open Science Grid. In particular, OSG provides environment variables which point to site-specific locations for *data* (`OSG_DATA`), *applications* (`OSG_APP`), and *scratch* (`OSG_TMP`). We operate a "gold standard" from our primary file server which provides the *data* and *application* areas. These are then sync'ed to remote grid sites on demand using the *rsync* protocol. Within the local environment, we provide the same or similar environment variables that a user would have at remote sites. Users with specific needs for VO-wide software or data submit a support request which is handled manually by VO administrators.

VO-wide job initiation scripts are used to create a suitable execution environment at remote sites, or to abort if suitable conditions are not met. This includes basic checks for outbound internet connectivity, the ability to fetch data by HTTP from standard ports, available local disk space, availalbe local applications, correctly set environment variables, and pre-staged VO software and data. The standard job initiation procedure also sets up a job working space (sandbox) which is on local disk and stages in any files either over the Internet (via curl to a public or X.509-secured private website), or from the site-local VO storage area on the network file system (possibly extracting files from compressed archives).

The atomic application invocations are then spawned with a user-specified timeout, and four primary output streams: generated files, standard output, standard error, and "results". Standard output from the (wrapped) application provides the normal output, while standard error combines both the application error stream and any errors reported by the application wrapper. Generated files are a by-product of the application or its wrapper and may or may not be retained, depending on the specific configuration of the job. "results" represent a single line summary of the application invocation and contain a minimum of six fixed fields, in addition to any application-specific results:

1) job set name – the same for all jobs in the job set
2) job name – unique for every job in the job set
3) exit status – from the set `OK NO_SOLUTION ERROR SHORT TIMEOUT`
4) timestamp – seconds since the UNIX epoch
5) runtime – in seconds (wall clock)
6) exit code – integer
7) any other application-specific results fields

By using this standard results format and other standard features for configuration files and command line options it facilitates the integration of several applications which can share configuration files, and results processing routines for data analysis and visualization.

III. Data Model

Structural biology relies on several key data sets: protein sequence data (similar to genetic sequence data, but consisting of amino acid sequences which describe the polypeptide chains that compose protein complexes), protein structure data (the 3D description of a protein structure, with coordinates for every atom in the structure), and imaging data (NMR spectra, electron microscopy image stacks, or X-ray crystallography reflections). All of these can be large, extending into 10s of GB of data, and millions of files.

To manage this volume of data in a grid computing environment, we have created data tiers:

- VO-wide – centrally accessible, relatively static, pre-staged, highly curated;
- Project – a common dataset for a particular project, to be used for multiple job sets, but for a time period of a few weeks to months;
- User – user files, which could be scripts, binaries, or data;
- Job set – a data set required for a single job set (workflow invocation);
- Job input – input data required for a single job;
- Job output – data generated by a job that needs to be returned to the user.

VO-wide data is managed by the VO administrators and part of its curation includes documentation for users describing what data is available, the structure and origin of the data, various organizational hierarchies to access sub-sets, and instructions on accessing this data from within grid jobs. The current emphasis is on the provision of protein structure definitions (3D coordinates) which are stored in a text format. This data is conducive to grouping and compressing into archives, and we provide a standard mechanism by which users can request particular entries (files) from a compressed archive that has been pre-staged to a remote grid site. As with applications, the VO-wide data tier is configured on the primary VO grid servers as the gold-standard. The synchronization is done using special grid jobs, which use *rsync*, that run on the storage element at every grid site our VO is authorized to access. Previously a mechanism was in place to attempt this synchronization with every grid job (including those from regular users), however the two main problems which arose from this were i) regular grid jobs and users do not always have permission either to write to the shared file area at the remote grid site or suitable matching file ownership; and ii) the locking mechanism to ensure only one process at a site could perform the update was problematic, especially at large sites where hundreds or thousands of jobs could be running concurrently. To respond to this, synchronization of data to remote grid sites is managed manually by VO administrators whenever necessary.

Project and User tiers allow for structured access to pre-staged data in the `$OSG_DATA/projects/some_project` and `$OSG_DATA/users/some_user` areas. These are pulled from users' grid data areas in `~/secure_html/grid_projects/some_project` and `~/secure_html/grid_data` on the VO grid file server, and synchronized by request through the VO administrators.

The job set tier allows a set of files which are common to a particular job set to be pre-staged in advance of the submission of the actual grid jobs. Currently this is managed in a per-workflow manner, where the class of common files for a particular workflow are well defined. As part of the workflow instantiation, the user specifies the common files which are pre-staged to all sites. Jobs in the job set that constitutes the workflow are then able to retrieve these common files from a predictable site-local location. The system tracks which sites successfully pre-stage the job set files, and this can optionally be used to limit where jobs in the workflow are permitted to run.

Job input files can come from two sources: using the standard "per-job" grid file staging mechanisms; and web-based file fetch via *curl*. The first has the advantage of being the "standard" way to stage files, and therefore reasonably well documented and understood, however adding this burden to the job managment system is also known to cause intermittent problems. The alternative web-based file fetch has the advantage of allowing both public and secure (via X.509-based ACLs) file access, and can leverage web caching, as many grid sites provide a Squid HTTP proxy for content caching (although this is only helpful when the data is accessed by multiple jobs at the same site). The standard VO pre-job setup process at the remote site automatically enables the use of a proxy if one is available. This disadvantage of web-based file fetch of user-controlled data is that users then must manage the data, place it in a web accessible location, possibly configure ACLs appropriately, and tidy the data once it is no longer required.

Job output is currently handled exclusively by the standard grid job stage out mechanisms provided by OSG. In the future options such as accumulating job set results at each site or third party results staging (submit from A, execute on B, results aggregated at C) will be considered, allowing a single operation at the end of the job set to retrieve all results from all sites. The use of HTTP *copy* operations to write files to a particular URL on a suitably configured web server (e.g. by the *htcp* command provided by GridSite[9]) will also be investigated. More details concerning this are covered in the next section.

IV. Workflow Model

We have established a basic workflow model for the management of typical serial structural biology computations, with support for parameter-sweep style job concurrency (i.e. non-communicating, non-synchronized). This builds on the foundation provided by the data and application models which ensure suitable scientific applications are available at remote grid sites

with usable, and (where possible) common interfaces, and that the necessary data sets are also available.

The most basic unit of our workflow is an *atomic job*, the smallest single independent job that can be completed. Typically this will be a single application invocation, which may only require 10s of ms to complete. Users are able to specify timeouts, parameters, and file dependencies at the atomic job level, or using application workflow generators (discussed below) to automate this. Jobs that finish successfully and produce results are marked as finishing in the `OK` state, while jobs that finish but internally detect an error exit in the `ERROR` state. A third possibility exists which is a convenience common exit state used for applications which perform some general data analysis and may not have sufficient information to provide results. In this case, the job exits with the state `NO_SOLUTION`, indicating no results have been produced, but for this scenario none should be expected. This allows job post-processing to adapt accordingly (e.g. don't attempt to aggregate results, or rerun job).

The timeout mechanism is a particularly important aspect of the atomic job, as it provides a backstop to many errors which arise in a grid environment, and can cause a job to stall, run indefinitely, or consume more resource time than the user expects. Jobs which exceed their timeout limit are aborted and marked as finishing in the `TIMEOUT` state. We have used a range of mechanisms to implement this. The most basic basic is a grid-level time limit which results in the unceremonious immediate abort of the running job and precludes any partial results collection or analysis of job state immediately prior to the timeout. A more sophisticated approach uses a watchdog process that is forked with a timelimit and a process handle for the internal application process (or thread). When the application completes, the watchdog is terminated, however if the timeout arrives before the application has completed the watchdog will abort the application. This allows partial results to be collected and better analysis of the job state that led to the timeout being reached. A disadvantage of our watchdog approach was that, under some circumstances, the internal application process could become disconnected from the wrapper and not properly receive termination or abort signals. This could lead to numerous zombied processes which risked system crashes. Refining and improving this timeout facility in the future will be important.

A converse requirement to the "timeout" is the constraint of minimum run time. It is almost always the case that an atomic job that runs for some minimum run time will properly run to completion, but if it has run for less than the minimum some failure has occurred that was not detected prior to initiating the "inner" application or algorithm. Jobs that complete in less than the minimum are recorded as having an exit status of `SHORT`.

For an atomic job to "fit" the common computing resources available within OSG, it should complete in less than 12 hours on a typical compute core (2.0 GHz Intel Xeon), consume less than 1 GB of RAM, have a duty cycle $> 50\%$ (the time it spends processing versus the time it spends idle), predominantly access files on local disk, read and produce less than 2 GB of data, and not require staging more results back to the VO job manager host at a level of 1 MB per minute of execution time (e.g. a 60 minute job could produce 60 MB of data). Most of these constraints are conservative, and it is possible to accept jobs which require double or triple these levels, but it then becomes more important to specify the characteristics in the grid job description to avoid early termination due to violation of remote site resource limits (physical or administratively imposed).

We do not automatically track CPU usage, memory usage or remote job I/O (local storage, network storage, or pure network), although jobs can be instrumented if necessary. As mentioned earlier, each job does track its start time, run time, and exit code as a standard part of its summary results. More details regarding the job characteristics can be found in the `job marker` (also described earlier), or by enabling a higher level of debugging in the remote job wrapper process.

To date, our application workflows, from a grid perspective, have exclusively had three stages: setup, execute, and finalize. The execute stage consists of a "bucket" of atomic jobs, which we call the workflow *job set*. Grid environments generally introduce sufficient job management overhead that it is impractical to have individual jobs at the grid level which require less than 5 or 10 minutes to complete. As such, it is possible that multiple atomic jobs will be grouped into a single *grid job*. In terms of implementation, an atomic job is structured to be executed as a single command with a set of parameters and some stagein and stageout requirements. These are described using the Condor Classad mechanism, with Condor[10] as the batch management system used by OSG for job submission. To produce a grid job from multiple atomic jobs is simply a matter of producing a new "command script" that groups the atomic jobs, and to merge the stagein and stageout requirements. A disadvantage of this approach is the reduced scope for handling atomic job failures in a simple and consistent manner. We have handled this by a job set post-processing stage that checks for unaccounted jobs and resubmits a "retry" job set. Typically one or two such iterations completes all atomic jobs that are possible, and manual review of results will indicate the causes of repeated failure for the remaining jobs.

The actual *workflow* is application specific and is constructed automatically at two levels: the set of atomic jobs, their requirements, and any workflow setup; and the construction of a set of grid jobs, representing the atomic jobs, assembled into a directed acyclic graph suitable for submission to OSG. In particular, we use the Condor DAGMan[11] module for managing jobs assembled into a workflow DAG. We have established some basic tools that facilitate the construction of parameter- or file-sweep style application workflows.

Once a workflow is established and running smoothly, it is typical to suppress (i.e. ignore) all output files and the standard output stream, and return only the standard error and results. When debugging output is off, and there are no excessive errors, these two output sources can be as small as

100 bytes (although 500 is more typical). One requirement of the the application wrapper is that it outputs, to standard error, some clear symbol indicating the completion status of the application. Another is that the job produces a `job marker` that provides a single line with suitable debugging information to identify which user, in which directory, on which host, at which site, and at what time was executing a particular job.

The current protocol for grid jobs is to produce `RESULT:` *<exitstatus>*. When the job returns to the submit system, if one of `OK NO_SOLUTION TIMEOUT` is **not** found (in other words, there is no output, or the exit status is `ERROR` or `SHORT`), then the job is considered to have failed and will be retried up to some retry limit. The failed job output is copied to a unique error directory, with the job completion timestamp, and grouped by site (e.g. `error/tuscany.med.harvard.edu/rw2-1niba1_20100815T03h17`), thus allowing for tracking and post-mortem failure analysis. Successful jobs have their results aggregated with all other results from the job set, and any extraneous output can be purged (although for performance reasons, since `rm` operations can be expensive, this is sometimes left until the job set is completed). An important challenge is to minimize the processing time and additional file I/O imposed by the job post-processing script. With a peak job completion rate $>$ 10Hz we have often managed to overload the job management server with the post-processing tasks of analyzing, moving, or removing the files returned by incoming jobs.

One important aspect of the workflow model we have developed is the ability to review partial job results continuously. By imposing the requirement of a single line job summary result, these can be aggregated as jobs complete and are returned, and analysis or visualization of these partial results provides strong indicators of the job set progress (e.g. the success vs. timeout vs. error rates, or the specific numerical results of parameter sweep applications)

Workflow generators which produce the job descriptions and manage the pre-staging of data also produce two job-specific tools which facilitate job management: custom scripts for submitting the job set, and for checking the status. These automate several important operations, such as checking for a valid certificate, that the job set is not currently running, and that key files exist and are in the correct locations. The submit script creates a record of the submitted job set to a list in the user's home directory which indicates the name, ID, status, and time stamp to facilitate later retrieval of results, or for checking job set progress. The status script caches the verbose results and appends the summary results to a table, enabling historical analysis of job progress. It also creates a copy of the current aggregated results. These can act as a backup if anything happens to the original results.

V. Example Workflow: Protein Structure Determination

The primary workflow we have developed requires approximately 20,000 hours and 100,000 atomic jobs to complete a single iteration. It is used to initiate the process of determining the structure of proteins using X-ray crystallography data through a process called *molecular replacement*, where known protein structure fragments are used as templates to bootstrap the solution of the imaged (and possibly sequenced), but structurally unknown protein. If the template is sufficiently similar to a sufficiently large portion of the unknown protein, then alignment algorithms can do a form of hypothesis testing and model building to identify the actual structure of the unknown protein. In cases where selection of the template protein fragments is not obvious (or where the expected templates have failed to provide a suitable model), we have shown that a brute force search of all known protein fragments can, in certain cases, succeed.

To perform this search, every one of 100,000 known protein domains is considered once. These domains are defined in files stored in archives which are part of the VO-wide data pre-staged to every site. Each atomic job specifies the archive and the file to extract from the archive, along with a set of parameters for the alignment algorithm. The input data is either pre-staged as a common job-set file, or fetched from the web on a per-job basis (but benefiting from HTTP proxy caching for subsequent requests from the same site where this is enabled). Under normal conditions all regular output is immediately discarded (as part of the running job), and only the single line summary results plus standard error status information is retained. This amounts to 100-500 bytes of data between two or three files which needs to be sent back to the submit host.

Results are automatically aggregated as grid jobs complete. When the grid jobs are all done, or on a demand basis, the results are filtered to remove invalid data and sorted by one of the scoring metrics. The results include a key (the equivalent of a foreign key) for each results line which allow constant details regarding the protein template models to be merged into the results table. This filtered, sorted, and augmented results set can then be analyzed by a combination of automated report generators and manually to identify templates suitable for structure determination.

As described earlier, errors are collected and grouped by the site where they occurred. This facilitates failure analysis and resolution (or remote site notification). Per-job output files collect in a single "output" directory.

VI. Operational Experience

Remarkably, almost a decade after "general purpose" grid computing was first seriously advanced and infrastructures supporting this made available (e.g. EDG[12] in Europe or GriPhyN in the US), it is still difficult to successfully run basic computations in a grid environment and manage data I/O. Trying to scale this to $1e5$ jobs for our target workflow has required months of focused effort to understand bottlenecks, failures, and configuration. Our initial hurdles came from interfacing from a web-based submission environment to Condor and OSG via Gridsphere[13] for job submission. Even once we retreated to a primarily command-line based submission

environment, we found it was difficult to reliably construct jobs with appropriate file paths which would be supported on the remote site, and to submit jobs to the grid environment in a way that attempted to allocate jobs appropriately to the available resources. The OSG Match Maker was deployed in mid-2009 and provided an immediate benefit in that it solved the job routing problem, and allocated submitted jobs to the most suitable grid site. This allowed us to run a single job set across a large collection of sites. Then, being exposed to the increased heterogeneity of numerous sites for the same job set meant increased effort catching and recovering from site-specific failures. At this stage we implemented a job log analyzer, inspired by [14], that revealed several interesting patterns in our jobs. First, it became clear that job submission was one rate limiting step. Next, we discovered several sites that were repeatedly evicting (aborting) running jobs, only for them to start again at the same site a few minutes later. Finally, we were able to see the "long tail" effect of a small minority jobs of jobs ($< 1\%$) had on significantly increasing the job set completion time (doubling it or more). We also found that the overhead of OSG in this configuration meant jobs running for less than 15 minutes would introduce a significant processing overhead that would limit our concurrent job capacity to 1500-2500 jobs. We have since transitioned to a "glidein" style job management mechanism[15] which does late binding of jobs to worker nodes and pulls jobs to the remote job site from a local (VO) job pool. Glidein has allowed us to exceed 7000 concurrent jobs due primarily, we believe, to reduced per-job overhead.

VII. Conclusion

A national scale cyberinfrastructure, such as Open Science Grid, can provide opportunities for new techniques in scientific investigation, such as simulation, data analysis, and modelling. Challenges persist around robustness (reliability) and ease of use. By establishing patterns and protocols for data, applications, and workflows we have been able to systematically optimize execution and data management in a grid environment, and continue to add new grid enabled applications for the structural biology community. Progressing web portal interfaces will be a top priority, and has a good precedent for significant impact from the TeraGrid Science Gateways[16] projects.

Acknowledgment

We thank Peter Doherty for grid computing support, Mats Rynge for Matchmaker customizations, Igor Sfiligoi for assistance with glideinWMS, and Steve Timm for operational assistance. This research was done using resources provided by the Open Science Grid, which is supported by the National Science Foundation and the U.S. Department of Energy's Office of Science. The work was supported by National Science Foundation Grant 0639193 (P.S.), and National Institutes of Health Grant P01 GM062580 (to Stephen C. Harrison).

References

[1] R. Pordes *et al.*, "The Open Science Grid," *Journal of Physics: Conference Series*, vol. 78, no. 012057, 2007.

[2] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid," in *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2000)*. IEEE Computer Society Press, 2000, pp. 283–289.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[4] The Apache Project, "Hadoop," in *http://hadoop.apache.org/*, November 2010.

[5] Y. Z. Ioan Raicu, Ian Foster, "Many-task computing for grids and supercomputers," 11 2008.

[6] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, pp. 1067–1100, August 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1148437.1148448

[7] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[8] C. collaboration, "The CCP4 suite: Programs for protein crystallography," *Acta Cryst D*, vol. 50, pp. 760–763, 1994.

[9] A. McNab and Y. Li, "The gridsite web/grid security system," *Journal of Physics: Conference Series*, vol. 216:6, no. 062058, 2010.

[10] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[11] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, *Workflow in Condor*. Springer Press, January 2007.

[12] P. Kunszt, "European DataGrid project : Status and plans," *Nuclear Instruments and Methods in Physical Research, Section A*, vol. 502, pp. 376,381, 2003.

[13] J. Novotny, M. Russell, and O. Wehrens, "Gridsphere: an advanced portal framework," in *Euromicro Conference, 2004. Proceedings. 30th*, September 2004, pp. 412–419.

[14] D. Thain, D. Cieslak, and N. Chawla, "Condor log analyzer," in *http://condorlog.cse.nd.edu*, May 2009.

[15] I. Sfiligoi, D. C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Wurthwein, "The pilot way to grid resources using glideinwms," in *CSIE '09: Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 428–432.

[16] N. Wilkins-Diehr, D. Gannon, G. Klimeck, S. Oster, and S. Pamidighantam, "TeraGrid Science Gateways and their impact on science," *IEEE Computer*, vol. 41, no. 11, pp. 32–41, 2008.