

Distributed Analysis of Neutron Scattering Experiments on the TeraGrid

ORNL Requisition Item 3400050128-00001
ORNL Purchase Order 4000036006
Period of Performance 1 July 2004 to 30 September 2004

Mike McKerns
mmckerns@caltech.edu

John McCorquodale
mcq@cacr.caltech.edu

Brent Fultz
btf@caltech.edu

*DANSE Group, Caltech
28 May 2005*

Introduction

Neutron scattering research in the United States is poised for a spectacular leap after the Spallation Neutron Source (SNS) is commissioned in 2006. With the power of its source and the performance of its instruments, the SNS will improve measurement efficiencies by typically two orders of magnitude. Analyzing data is an essential but non-trivial step in neutron scattering research. The productivity of the SNS for cutting-edge science will be enhanced considerably by modern computing. It is in the national interest for the SNS to deliver quickly on its promise of scientific advances in our understanding of the structure and dynamics of materials and condensed matter, and the sophistication of its scientific results will be enhanced by the sophistication of its computing resources.

The past decade has seen a spectacular leap in the capabilities of computational science. Dramatic improvements in computing hardware have been matched by dramatic improvements in our understanding of how to compute the properties of materials, molecules, and condensed matter. Many methods, once exotic, have become routine with improvements in computing hardware and software. These methods, such as those of quantum chemistry and condensed matter science, can address many of the same phenomena measured experimentally by neutron scattering research. A number of new scientific opportunities, some discussed below, involve a closer interaction between computational science and neutron scattering science. In the near future, these new directions will therefore track the trends in computing resources that are followed by computational science, although experimental neutron science brings a number of new challenges such as near real-time computing. An important long-term trend, still in its infancy, is distributed computing, which offers new flexibility and power of computing resources.

Over the past several months, we have considered a number of ways where distributed computing could be utilized for the support of neutron science. In particular, we have considered how the evolving capabilities of the pyre system that underlies DANSE (distributed data analysis for neutron scattering experiments), the TeraGrid project funded by the NSF, and a neutron science portal at the SNS could offer new capabilities for neutron science, the types of interactions between these systems that would be plausible and useful, and what opportunities exist for improved designs. These issues are addressed in the present report.

Specification of a High-Level Grid Services Interface Layer (GSL)

Our first goal is to develop a specification of an interface between distributed analysis and a grid computing environment by considering these as forward-looking abstractions. We will conclude that even in the most demanding possible scenarios that existing free software is for the most part sufficient to satisfy our basic needs.

A great deal can be learned about the structure and behavior of component systems by considering them purely in the abstract. The properties these considerations imply about the environment in which a component system must execute are in turn useful for systems-level design. In the case of particular concern here, we will immediately notice conflicts with the TeraGrid execution environment that are intrinsic to all distributed component systems. It is the function of the TeraGrid-specific slice of pyre's Grid Services Light (GSL) to mitigate this conflict.

Nature of Software Components

A collection of software components together with their component framework implement a run-time analogue of module-structured code (modules being linguistic, thus compile-time, constructs). A component is roughly the run-time analogue of a module and a component framework is roughly the run-time analogue of a language's module system, but it is a mistake to take this analogy too far. Modules and components are both codifications of separability and formal mechanisms of composition, with the goals of allowing code reuse and enhancing understandability. However, the similarity ends with this vague abstraction. The qualitative differences of linguistic expression and run-time automation lead to incomparable boundaries of separation and composition formalisms for the two. Like hammers and screwdrivers, components and modules are similar in goal; however, they are very different tools. The situations and ways in which they are used may crosscut in ways which hide their necessarily distinct natures.

The most important feature of component-based programs is that the component system, operating at runtime, composes the application on the fly. This can be in response to user interaction, adaptations driven by the intermediate results of a simulation, or arbitrary application needs. The program directs the component framework to change the program's structure as it runs in dynamic and unpredictable ways.

Various mechanisms can be used to achieve this dynamism. For compiled (C and FORTRAN) programs, dynamic linking is a common tool: calls to the `dlopen`[7] C library functions allow programs to load object code under runtime control and often to have enough control over linker namespace management that disparate code never intended to be combined can be made to run within a single process without large-scale rewriting. Almost any interpreted language can use an "eval" mechanism and a module system to achieve the same thing for code written in that language.

A component may be represented as a tarball or zip file when distributed to users, a collection of files when stored on disk, or as a serialized bytestream when transmitted over a network. Such packings carry metadata used by the component system expressed as idiom: particular interpreted entrypoints must be present, particular symbols must be in the `.o`, or other requirements – these are implementation details of the "well-defined and published interface"[8] that ACM considers the fundamental definition of a component. What makes something a component is only the *abstraction* that it is a unit of functionality combined with a component system that presents that abstraction to a component user. Whereas programs tend to have a one-to-one relationship with files and are thus easy to anthropomorphize, components do not and are thus often difficult to anthropomorphize outside the component framework.

A *distributed runtime environment* includes mechanisms to manage the dynamic complexity of code distribution, composition, execution and communication of a single coordinated "logical application" running on an assortment of network-connected computing resources which may span organizations and involve cases of extreme platform heterogeneity. A *distributed component system* is a specialized slice of the general concept of a distributed runtime environment which limits its concern to the tasks of distribution and composition at the granularity of components. The present work is concerned with extending the DANSE/pyre component system to a distributed context.

What a component system needs in order to achieve this extension follows directly from its goals of distribution and composition. That part of the distributed application which implements the distributed component system must have internal communication during application execution in order to distribute components under runtime direction. Once the distributed component system is running, this communication could easily be serialized components transported over TCP streams. Thus, once the application is running, the most universally available least common denominator of communication is sufficient for the needs of a distributed component system. The complexity lies in the bootstrap process of the distributed component application: before the component system is running, some "grid services" mechanism must exist to deliver a skeleton application to a compute resource, set it to running, and establish basic communication with the whole of the distributed program so that component transport and instantiation can begin.

Supercomputing and the TeraGrid

The TeraGrid[15] is an NSF-sponsored initiative to create an "open scientific discovery infrastructure combining leadership class resources." It is a federation of national supercomputing resources linked by a high-bandwidth wide-area network. Access to TeraGrid resources is simplified by a common, uniform allocation-granting administrative structure.

Neutron diffraction and inelastic scattering, in common with many other experimental techniques used in materials science (e.g. X-ray diffraction, light scattering), uses the measurement of the change in momentum/energy of scattered particles in order to provide information on the positions and motions of atoms (or electrons) in real space. It is never possible to measure complete information in energy/momentum space without errors, and in most all cases, phase information is lost. Although basic data reduction operations are often forward computations following standard procedures, many of the promising types of data analysis methods are inverse problems. We foresee a growing importance for advanced analyses of neutron scattering data that are inverse problems in computing. Depending on the size of the data set, the number of variables and the constraints, an inverse analysis can demand a wide space of possible forward computations. Inverse problems typically require massive computational power and large amounts of memory, and are one of the standard challenges of large-scale computing.

Another challenge suitable for the national supercomputing centers is performing large forward computations, as required in molecular dynamics, Hartree-Fock, and *ab initio* calculations of electronic structure in many quantum chemical and solid state simulations. In existing implementations, these large forward computations involve high data throughput at a rapid rate, coupled with large-scale memory consumption.

The TeraGrid is of notable utility for performing batch-mode supercomputing tasks like these on an unprecedented scale. We can use its resources today to solve a greater quantity and larger scale problems than ever before.

On-Demand Computing and Grid Computing

Beamtime at neutron facilities has a high price per hour, and must be utilized as efficiently as possible. If

high performance computing could help an experimentalist better utilize the beamtime, it is probable that the cost savings from beamtime utilization would far exceed the cost of the computing resources. Examples of this need for computing on demand include the selection of temperatures, pressures, or magnetic fields in parametric studies of structure or dynamics of materials. If a phase transition is understood with a thermodynamic model, for example, seeing the results of the model in near-real time would help guide the specific experimental runs.

There is immediate potential for the application of on-demand computing to experiments we perform today. For example, the optimization of large three-dimensional finite element models using updated experimental data would help an engineering diffraction experiment optimize measurements of stress tensor distributions in the sample. Such a scenario has not yet been realized in practice, but would represent a significant advance to neutron science.

In order to make computations like these useful, results of the computations must be available as quickly as possible. On-demand high-performance computing like this involves jobs that require immediate execution, are comparatively short-lived (perhaps running for only minutes), but are highly parallel and require extremely large amounts of resources. In order to provide this kind of responsiveness, demand-driven compute resources must either sit idle reserved for a single user or they must support a kind of extremely lightweight priority-based preemptive execution in order to fill in this otherwise idle time with useful lower-priority work.

It is vital to note that supercomputing traditionally involves extremely expensive compute resources and the scheduling policies of supercomputers are designed to optimize *that* cost rather than, say, the cost of users' beamtime. Under such an analysis, idle compute cycles are to be minimized. This results in scheduling policies favoring jobs that are long-running, of predictable runtime, and able to be started arbitrarily long in the future wherever they fit best in a schedule with competing jobs. The overhead of preemption in such an analysis merely lowers overall cycle utilization, so supercomputers usually run single jobs to completion with prioritization only during job submission and have turnaround times from job submission to results of days or weeks.

It is easy to see the tension here – computer utilization and response time for on-demand tasks are traditionally at odds in optimization arguments. An effort is underway to create a paradigm shift toward systems supporting demand-driven, location-independent large-scale computation. This effort is called *grid computing*. As purely envisioned[4, 6, 3], a *computational grid* is a language-neutral distributed runtime environment federating separate, geographically dissociated administrative domains. Modern visions of grids are almost necessarily coupled to economic systems as a priority metric to globally optimize utility.

In order for demand-driven high-performance computing to achieve high utilization, the overhead of preemption must be made negligible. The parallel efficiency of programs must also be maximized in order to decrease total runtime. In order to federate geographically dispersed resources, highly dynamic and potentially large long-haul communication costs must influence program partitioning at runtime to optimize program throughput. This requires that programs be composed of medium-grained units of computation, whose communication and distribution can be reasoned about by a runtime scheduler. Demand-driven grid computing, thus, shares much program architecture with distributed component systems. The former is focussed on problems of partitioning, communication and scheduling to optimize resource utilization and the latter is focussed on problems of language design, programming workflow, and programmer/computer interaction. When they arrive, the first successful demand-driven computational grids may be distributed component runtimes.

In an attempt to develop software today for the inevitable future demand-driven grid environment, we have deployed a small inexpensive cluster[9] which gives us the flexibility we need to develop analysis software targetted at experiment-coupled demand-driven operation. The TeraGrid today allows us to develop mechanisms to support batch-mode analysis operation, which will in the long term remain a vital and ideal format for the kinds of large parameter optimization problems the community anticipates will underly many tantalizing novel techniques. We consider it vital and natural to persue *both* these directions.

The grid idea is increasingly vibrant and its evolving vision[5] is accreting a dynamic community of capable minds. It is crucial to realize that the TeraGrid is a supercomputing resource. However, within the lifespan of the SNS we will certainly be performing analysis on a true computational grid. We must plan for the unique characteristics of such a capable environment now, so that the components we develop today remain flexible and adaptable tomorrow.

Grid Services Required for DANSE

As motivated above, the services required to implement a distributed component system are merely inter-process stream communication and a mechanism to bootstrap a skeleton distributed component runtime onto the grid resource and establish communication with the application whole. The mechanisms necessary for bootstrapping are in general referred to as staging and job submission. The necessary and sufficient set of required services is:

- *staging* – this is the action of copying an application binary and its support files from the outside world to a grid or network-attached compute resource in preparation for submitting a job for execution (and in cluster environments subsequently copying to the actual compute resources, often under the control of a batch queuing system).

- *submission* – this is the action of notifying the control mechanism for a grid or network-attached compute resource that your application is staged and ready for execution; the scheduler will invoke the skeleton component runtime when resources are available.
- *stream communication* – once the skeleton is running, it must make initial rendezvous with the application whole (usually in the form of a master control process running potentially in another administrative domain). As components are instantiated over the lifetime of the application, it will also, under the coordination of the master control process, establish communication directly with other compute nodes. Note that stream communication can be implemented on top of a message/datagram system like MPI.

Special care must be taken when crossing administrative boundaries. Particular sites may have site-specific authentication mechanisms that must be negotiated to achieve staging and submission. Further, stream communication may require special treatment or aggregation at the headnode if compute nodes are on a private unrouted network (as is usually the case) – this structure on the TeraGrid is called an *application gateway*[13] and is in common and successful use. Sensitive computations may wish to make use of encrypted channels between administrative domains. Much of this complexity can be mitigated by good abstractions of component communication implemented by the component framework.

Due to the generality of these observations, one can conclude that an administrative boundary (for example between ORNL and network-attached scientists) for a component-available compute resource having the following properties will be sufficient to support *any* distributed component system:

- Interfaces for inbound staging and submission. We will see in the following discussion that the ubiquitous accepted standard `scp` and `ssh` are our preferred mechanism for communication with the headnode of a cluster compute resource.
- Outbound TCP stream establishment for subsequent bidirectional communication is necessary and sufficient (the typical NAT firewall scenario). Inbound TCP communication to compute nodes allows direct communication among individual components, which is preferable from a performance standpoint (this is possible within the TeraGrid).

Detailed Task Completion

(section references are to statement of work)

The grid services required by distributed data analysis (2.1) are staging, submission and stream communication. The DANSE/pyre abstract mechanisms for these services are called Copier, Launcher and Stream, respectively. The grid services that are missing or not fully implemented in CTSS (2.1) include co-scheduling and rapid-turnaround/interactive scheduling as these are incompatible with the cost optimization of large TeraGrid compute resources. These cannot be provided by the addition of software to the TeraGrid; the neutron science community must federate its own existing and planned compute resources for this purpose. Use cases for data analysis and a derivation of derive the requirements for distributed data analysis (2.1) were discussed above as motivation for the discussions of supercomputing and on-demand computing.

GSL (Grid Services Light)

Our second goal is to incorporate the understanding of the previous section into the DANSE/pyre software environment.

DANSE uses the pyre[1] component system, which realizes the benefits of both compiled and interpreted component systems by coupling the “eval” and module systems of the interpreted Python[14] language with C library dynamic object loading to achieve a single runtime environment capable of combining and orchestrating composite applications including Python, C, C++ and FORTRAN code. DANSE/pyre components, then, might include compiled C or FORTRAN object code, Python scripts and other support data all of which is abstracted as an indivisible unit by the component system (and by the programmer using the component).

History and Motivation of GSL

GSL is motivated by the observation that “grid services” are not mature, are overly general or intractably cumbersome, don’t provide checkpointing/state persistence, and are not thus far targetted at on-demand or medium-grained global scheduling of federated resources. We have to provide these things ourselves. In general this is an unsolved problem. In particular circumstances, however, the desired behaviors are simple and tractable, and can be achieved with tractable complexity. We plan, over many iterations of GSL, to avoid the hard problem by iteratively experimenting, generalizing and re-factoring. The first iteration is presented in this report.

Early design planning began 28 January 2004 and development work on a research prototype implementation occurred during the performance period and was supported, in part, by this requisition item. This early design period established goals which endured through the performance period:

- Develop simple abstractions of staging and job submission flexible enough to bootstrap a distributed pyre application on a variety of compute resources, and capable of encapsulating the individual peculiarities of each particular compute resource.
- Formulate a distributed component runtime system for pyre capable of program-controlled component transport, instantiation, configuration and communication in harmony with the established pyre component life-cycle[12].
- Develop convenient prototype implementations for the purpose of refining these abstractions.

Basic Functional Requirements

We performed an informal round-table consideration[11] of the basic functional requirements for distributing pyre components onto cluster-like compute resources. Our consideration included the TeraGrid, at that time limited to the five original sites. Our goal was to choose implementation approaches for the abstract services discussed in the previous pages that would balance the needs to be as simple to implement as possible, as easy to use as possible, and quickly allow experimentation with sample applications on our test cluster.

To support staging and job submission, we require local storage and the ability to copy a file to the headnode, and to log into the cluster headnode and run a script assuming pre-existing installation of arbitrary software. We might assume the pre-existing installation to include basic UNIX utilities like `tar` so that a tarball can be transmitted and extracted to enlarge the set of pre-existing installed software (within our user's home directory). By iterating on this process, we can automatically bring a headnode to the point where it can launch the skeleton pyre distributed component runtime. The distributed component runtime can then abstract arbitrary cluster job submission mechanisms and deploy components onto the compute resource under program control.

The pyre distributed component runtime requires staging and job submission to bootstrap the skeleton application, it requires a dynamic linking facility on the compute nodes and headnode, and it requires arbitrary communication amongst individual components both within a compute resource and across federated compute resources. Communication across resources may involve multiplexing at the headnode if compute nodes are on a private network, and may involve encryption at the headnode if intermediate networks are not trustworthy. Further, across untrusted networks the component runtime may require an inter-component authentication mechanism to prevent application hijacking. While it is possible (and desirable) to build component applications such that no local storage is needed on the compute nodes, we anticipate that many legacy codes that are coerced into components during the DANSE project may require such, especially when the source code is not available.

Design Choices in the DANSE/pyre Component Framework

During discussions about these choices, we considered all environments in which we envision running DANSE components in the 5-year timeframe including on the TeraGrid. We concluded that even in the most demanding possible scenarios that existing free software is for the most part sufficient to satisfy our basic needs during staging and job submission. Most existing scheduler interfaces are easy to invoke once logged in, and the OpenSSH suite provides a ubiquitous and accepted standard for headnode login, authentication, stream encryption and file staging. Summarizing, our implementation choices for the abstract services required are:

- Authentication: `ssh`
- Staging: `scp/ssh`, `cvs` (in turn on top of `scp/ssh`)
- Communication streams: TCP (unencrypted), `ssh` pipes (encrypted), SSL (with XMLRPC)
- Job submission: per-resource interface to scheduling mechanism and policy
- Multiplexing: not implemented in prototype

In the interests of stability and simplicity, and because at the time the TeraGrid standard software “stack” was in a state of flux, we treated the TeraGrid as a collection of disjoint large traditional supercomputers and planned to log in and submit via the same mechanisms (`ssh`) “under the radar” of the evolving grid toolkits. Thus we achieved the benefit of uniformity across our small interactive clusters and the TeraGrid. That in the intervening 18 months the TeraGrid has been repositioned more concretely in the image of disjoint large traditional supercomputers retrospectively supports this approach.

pyre/GSL Implementation Thrusts

GSL/LAUNCHER is the login, authentication and file transport mechanism. It consists of two abstract components: Launcher and Copier. There is one concrete implementation at this time encapsulating `ssh/scp`. This is

sufficient for all TeraGrid systems time as it exists everywhere today and will clearly remain supported in the future. Once a grid middleware implementation passes the test of cultural acceptance and establishes an alternative mainstream approach, it will be easy to abstract it to the Launcher and Copier interfaces.

GSL/INFECT is the staging mechanism. It consists of three subthrusters: detect (remote filesystem detection and report generation), modtools (modification of remote shell environment files), infect (loading and building framework components and packages), plus a single control script that queries the remote system and formulates a strategy for infection with the build procedure. In more detail:

- detect: receives instructions from the control script on optimal procedure for building framework and packages. Copies detect archive to remote filesystem. Detect script stores environment variables, locates and stores paths to specific executables (cvs, gzip, python, etc.). Uses Launcher and Copier based on ssh and scp, system does not depend on pyre; its goal is to deploy pyre.
- modtools: consumes report from detect on remote environment and file locations. possibly creates ssh/config, arranges for appropriate .profile and pyre configuration files in .pyre/. Uses Launcher and Copier, does not depend on pyre existing. When finished, remote filesystem environment is ready for loading and building.
- infect: Assumes modtools has established environment. Creates DV_DIR, TOOLS_DIR and BLD_ROOT if missing. Loads or updates config, templates, pythia. Rebuilds templates and pythia if necessary. Loads or updates desired application packages (GSL, reduction, etc.). Rebuilds desired application packages if necessary. Uses Launcher and Copier. Leaves remote system (login direction) in state ready to run application modules.

GSL/RPC is the intercomponent communication mechanism. There are two implementations, one using the standard XMLRPC and a simpler approach called XML-on-stream. The XMLRPC standard includes use of “web” certificate-based authentication mechanisms and SSL encryption. XML-on-stream merely sends XML-encoded messages over arbitrary streams: either basic TCP sockets or via ssh-encrypted pipes.

The *service factory* is the user-level interface to the pyre distributed component runtime and distributed framework services. The service factory provides a toehold for application migration and control. The service factory is responsible for implementing all complexity associated with encapsulating distribution and unifying remote and local intercomponent communication. The service factory also implements distributed component properties (a pyre term), which is useful as an IPC mechanism that hides streams to create a uniform interface for local and distributed component apps. Distributed properties are a useful communication mechanism in low-bandwidth situations (like initial component parameterization).

The low-level *robustness infrastructure* includes distributed logging, error handling and presentation, distributed exception propagation, failure cleanup and diagnosis aids.

As is necessary for any successful collaboration, all this code is available for detailed study in the public DANSE CVS repository[10].

Experiences with pyre and the TeraGrid

We demonstrated a pyre application running on the TeraGrid at the SC2004 High Performance Computing, Networking and Storage Conference[2] in Pittsburgh, PA during early November 2004. The application included components running a simulation on compute nodes at NCSA feeding data to a visualization engine running on the visualization service at NCSA and saving results and pyre journaling information to the TeraGrid cluster at Caltech. This demonstration was controlled from the Caltech booth on the show floor in Pittsburgh and visualized there with Iris Explorer. Communication during this demonstration was serialized XML objects sent over simple TCP streams.

This demonstration was developed by Sharon Brunett, and provided useful feedback to the development team, from the perspective of a new user. Sharon reported that porting the application to pyre was not difficult, deployment of the relevant pyre services and infrastructure was not difficult, and that the robustness and user friendliness of the pyre code development environment was “much appreciated, as the IA64 TeraGrid nodes were a bit on the finicky side.”

Future of GSL Implementation

The first iteration of GSL design and implementation is concluded. Our next iteration will be a redesign and reimplementing incorporating the added depth of understanding we gained during and after the performance period.

For example, recent work undertaken by Mike McKerns and John McCorquodale, not supported by this requisition item, has demonstrated that all support files for the DANSE/pyre distributed component system skeleton can be linked into a single binary. This achievement renders complicated staging technically unnecessary, and establishes job submission plus stream communication as the minimum sufficient set of grid resources for DANSE/pyre applications. This can be a benefit in extremely inflexible execution environments, and may become the preferred way to deploy the skeleton application.

Additionally, we have identified a need to implement our own kind of lightweight dynamic linking for use on emerging machines like BlueGene that do not support `dlopen` on the compute nodes. We believe this to be achievable; however, realizing this facility will necessarily begin within an experimental development phase.

The time between design iterations offers the most flexibility for refinement of vision, and that time is now. The SNS has promised to provide us with some knowledge of the facility boundary layer that can influence and improve our plans. In an even more appealing scenario, we hope it might be possible to begin testing applications running across the SNS boundary (components running at Caltech and within the SNS communicating through the SNS boundary layer) during this implementation iteration. This will necessarily involve closer communication with SNS system and network architects than has thus far been achieved. We are eager to see this happen.

Detailed Task Completion

(section references are to statement of work)

Concerning implementation strategies for missing Grid services (2.2), we chose not to pursue component communication multiplexing in this iteration of the project, but this will be necessary to consider on the next iteration. There are no other services missing that are crucial to our success that we can solve on the TeraGrid. While coscheduling and interactive scheduling are of course ultimately essential, those remain beyond the immediate scope of the project. Our extension of pyre to use existing Grid services (2.2) utilizes `ssh/scp` which are part of the core grid services on the TeraGrid. A preliminary effort toward building a suite of test harnesses to exercise the various distributed aspects of the system in a controlled way (2.1) has been initiated to test authentication and interaction with daemon services found within the pyre framework. The integration of a test harness is an ongoing process as the distributed services for the pyre framework reach maturity. We are continuing to work toward the construction of a realistic full-application test case (2.2). During the performance period, Tim Kelley worked extensively on related data reduction development in addition to his contribution to GSL. Since the close of the performance period, work has continued on instrument simulation and force constant simulation. Work is currently underway to make an established molecular dynamics code available as a DANSE component. Our goal is to synthesize a composite demonstration application from these components within the next few months.

Interactions with the SNS Neutron Science Portal

Our final directive was to monitor the design and implementation of the SNS Neutron Science Portal so that potential DANSE incompatibilities are identified and repaired as early as possible. The goal is to make DANSE accessible through the SNS Neutron Science Portal (statement of work 2.3).

The SNS Neutron Science Portal is not yet defined in detail, but a number of capabilities can be reasonably assumed. At the most basic level, the Portal would serve as an authentication system for access to SNS computing resources, and as a system to account for the resources used. This is an acceptable capability for the deployment of a distributed computation, if some precautions are taken in its design. For example, a distributed computation may involve a large number of independent processes, and accessing them through a single gateway or server could become an information bottleneck.

At a slightly higher level of service, the Portal could provide a layer of data services, especially access to experimental data and searches through data archives. It would be convenient if a standard query language and search capabilities were selected early, so that developers could anticipate ways that remote users and distributed computations could access the SNS databases. It is also appropriate to consider if the smaller, lightweight metadata from NeXus files would be made available independently of the full data file, which may be rather large. Distributed computations for specific instruments may benefit from detailed queries of the Portal for specific metadata in NeXus files.

At a still higher level of service, distributed computations may benefit from computing resources offered by the SNS through Portal authentication. Because distributed computations are not yet routine, it is more difficult to understand the best way to develop these capabilities. An obvious possibility is that the same system for distributed computations, pyre for example, would be deployed on the SNS resources. This could be done by SNS systems managers, or automatically through the GSL capabilities of pyre. The Portal could activate an independent set of data reduction or analysis computations on SNS resources. These capabilities, once defined, could also be utilized in a distributed computation. This is, of course, new territory, and some experimentation with the optimal approach will likely occur over a number of years.

Adherence To The Proposed Staffing Effort

The development team consisting of Mike McKerns, June Kim and Tim Kelley was involved full time with this effort during the performance period. Michael Aivaizis served as advisor to the development team during the performance period. This effort was supported by this requisition item, as proposed.

Sharon Brunett achieved deployment of pyre applications on the TeraGrid. John McCorquodale served as forensic documentarian in the preparation of this report. These efforts were not supported by this requisition item and occurred outside the performance period.

References

- [1] AIVAZIS, M. *pyre in a Nutshell*. <http://www.cacr.caltech.edu/projects/pyre/docs/ch01.html>.
- [2] (CONFERENCE). *SC2004 High Performance Computing, Networking and Storage Conference*. <http://www.sc-conference.org/sc2004/>.
- [3] CZAJKOWSKI, K., FITZGERALD, S., FOSTER, I., AND KESSELMAN, C. Grid information services for distributed resource sharing, 2001.
- [4] FOSTER, I. The grid: A new infrastructure for 21st century science. *Physics Today* 55, 2, 42–47. <http://www.aip.org/pt/vol-55/iss-2/p42.html>.
- [5] FOSTER, I., AND KESSELMAN, C., Eds. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufman, San Francisco, 2004.
- [6] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications* 15, 3, 200–222. <http://www.globus.org/research/papers/anatomy.pdf>.
- [7] GNU. *dlopen(3) man page*. <http://www.rt.com/man/dlopen.3.html>.
- [8] HOPKINS, J. Component primer. *Communications of the ACM* 43, 10, 27–30. <http://portal.acm.org/citation.cfm?id=352198>.
- [9] MCCORQUODALE, J. *The DANSE Development Cluster*. On the DANSE Wiki. http://wiki.cacr.caltech.edu/danse/index.php/The_DANSE_Development_Cluster.
- [10] MCKERNS, M. *DANSE CVS Repository Documentation*. http://wiki.cacr.caltech.edu/danse/index.php/DANSE_cvs.
- [11] MCKERNS, M., KELLEY, T., KIM, J., AND AIVAZIS, M. *Notes from ARCS Software Meeting, 28 Jan 2004*. http://wiki.cacr.caltech.edu/danse/index.php/Notes_from_ARCS_software_meeting_Jan._28%2C_2004.
- [12] MCKERNS, M., KELLEY, T., KIM, J., AND AIVAZIS, M. *pyre Component Lifecycle*. http://wiki.cacr.caltech.edu/danse/index.php/Notes_from_ARCS_software_meeting_Jan._28%2C_2004#Component_Lifecycle.
- [13] PITTSBURGH SUPERCOMPUTING CENTER. *PSC Application Gateway User Guide*. <http://www.psc.edu/machines/tcs/qsockets.html>.
- [14] VAN ROSSUM, G. *Python Reference Manual*. <http://www.python.org/doc/2.4.1/ref/ref.html>.
- [15] VARIOUS. *TeraGrid Website*. <http://www.teragrid.org/>.