Overview of the DANSE architecture

Paul Kienzle[†‡] and Przemek Klosowski[‡]
†University of Maryland ‡NIST Center for Neutron Research

The DANSE project is a cooperative effort to build next generation neutron scattering software. The goal is to provide an extensible framework to integrate new and existing analysis software. To achieve this we are designing a dataflow architecture with computational components that can be distributed across different nodes of a network. End users and instrument scientists will be able to define new components and combine existing components in new ways as their data analysis needs change. The component cores can be implemented in almost any language, but a small Python wrapper must be provided for communication with the framework. DANSE subprojects are designing components for use in SANS, reflectometry, diffractometry, inelastic scattering, and engineering diffraction, and providing a user interface suitable for researchers who are not experts in the field of neutron scattering. We will describe the DANSE framework in detail with examples from reflectometry.

Work supported by the NSF grant DMR-0412074 and by the NIST SIMA grant *Infrastructure for neutron reflectivity component integration.*

1. Introduction

Most neutron scattering software in common use today exists as standalone applications. The level of sharing is at the level of programs. As programs are shared, people introduce changes they need for their own particular science, possibly passing the changes back to the original author, possibly not, and possibly sharing the code further on. Modifications to the code are not easily shared. Extensions to Joe's version of Julie's modification of Kevin's code can't easily be merged with Frank's version of Kevin's code. Even though the code is fully in the open, we are not receiving the benefits of open source. Improvements to the code are not available to everyone. The DANSE project hopes to change this situation by providing a framework for the development of neutron scattering software which encourages the development of reusable code.

2. DANSE

DANSE is a collaborative project to develop software for the Spallation Neutron Source (SNS). The DANSE project is organized around a core framework and five scientific disciplines: diffraction, engineering diffraction, reflectometry, SANS and inelastic scattering. The software will be open source and available for other institutions worldwide. All subprojects will be developing code for existing instruments in addition to the new SNS instruments.

A growing set of design documents is taking shape on the DANSE wiki:

http://wiki.cacr.caltech.edu/danse

In particular, the DANSE white paper from last year discusses much of what we talk about here in more detail:

http://arcs.cacr.caltech.edu/arcs/danse/docs/danse.pdf

DANSE is a layered architecture, with the following layers:

1. Computational kernels
2. Python interpreter
3. Component framework
4. Visual programming environment
5. User interface
6. Interface extensions

In the following sections we will describe these layers. Keep in mind that the DANSE design is a work in progress, and the details will change as we understand more of the issues.

2.1 Computational kernels

The bottom most layer is composed of independent computational kernels. These kernels will be routines in C, C++ or Fortran. Core DANSE kernels must compile on all compilers and architectures that DANSE supports, including Mac OS X, Linux, and Windows platforms and perhaps high-end systems available on the TeraGRID. Supporting more languages and architectures will be possible if necessary, but it may complicate the build environment. Users will be able to create kernels in other languages so long as they can compile them for all architectures that they need.

The Neutron Science Software Initiative (NeSSI) is an inter-laboratory project to share software. Representing fundamental computations of interest to neutron scattering, the computational kernels will be developed in conjunction with the NeSSI initiative, and usable outside the DANSE framework. The NeSSI is much broader scope than DANSE, encompassing as it does instrument control, data management, experiment proposals, electronic collaboration, user authentication, data reduction and analysis. The DANSE project will help organize existing code and provide new code for data reduction and analysis.

The computational kernels must be modular, with a clear definition of inputs and outputs. They should not rely on internal or external state (in C, static or global variables; in Fortran, SAVE variables or COMMON blocks). Any information that needs to be communicated between calls to the code must be passed through the inputs and outputs. Without modularity, components can only be used for one thing at a time. For example, consider min_x g(x) where g(x) = min_t f(x,t). If the minimization function uses global variables then it cannot be called during the evaluation of g. We have encountered this problem in the Octave numerical environment, which uses minpack for its minimization. Similarly, if the modeling code uses state, then users cannot simultaneously fit multiple data sets. State is also a problem for threaded programming, though the current design of DANSE does not require kernels to be thread safe.

The computational kernels must be thoroughly tested and documented. As the basis for neutron science around the world the code must be correct with its limitations clearly specified. Kernels should perform correct uncertainty analysis. Numerically stable algorithms should be used where possible, and the limits of stability clearly documented. Uncertainty in the inputs should be propagated correctly to the outputs. Additionally, the software must be robust. When code is running on a distant machines across the net as part of

a larger computation, simply halting the process on encountering an error is not an option. Instead errors need to be propagated up from the kernel into the next software layer. The tests must verify that the routines work within tolerance for good input, and fail gracefully for bad input. Regression tests must be included for each bug that is fixed. If the bug got reported, that means the code is on a common code path, and future revisions of the module should not break it.

Writing correct computational kernels is key to the success of DANSE. If the software is not reliable, scientific users will avoid it no matter what its other benefits might be. Adapting existing software will not be trivial. For example, the NCNR analysis software was based on old FORTRAN code with common blocks throughout. To support simultaneous fitting, we were forced to rewrite it.

DANSE does not provide parallel programming for free. What it does provide is a build environment which knows how to use MPI and PVM libraries and a runtime environment which controls the computation for us. To get the benefit of fine-grained parallel processing in our computational kernels we will still need code based on MPI or PVM, either written by somebody else (as it has been for many problems in linear algebra) or written by ourselves.

2.2 Python interpreter

The computational kernels will be wrapped with a Python interface. Python is an powerful interpreted language which is easy to learn. Python frees the user from memory management, making robust code easier to write. This is key to rapid application development, and to the exploration of new algorithms. Python already has code for implementing unit test suites, which would otherwise be difficult to write in C or FORTRAN. The DANSE component framework is implemented in Python.

Computational kernels can be implemented directly in Python. Facilities for scientific programming are available from the Python environment, such as matrix manipulations, plotting, etc. For more sophisticated functionality, direct calls can be made to more complete scientific environments such as Octave (a Matlab-like language) or R (a statistics package implementing the S language). Bindings already exist for Matlab and IDL, but these will only be available to users with licenses for the software. Legacy command line applications can be wrapped with Python as well, assuming that binaries exist for the desired architectures, by controlling standard input and output from Python.

We expect many user extensions will require some Python code to modify the inputs and outputs of the computational kernels, though with enough basic components, all extensions could be written directly in the visual programming environment.

2.3 Component framework

The Python functions and methods exposing the computational kernels will be wrapped as components in the Pyre framework (see http://www.cacr.caltech.edu/projects/pyre). Each component is an object with properties and methods. The properties have default values which can be set by name when the component is created and changed any time. Rather than hard coding references between components, the framework provides for runtime resolution of components based on component name. By making the name of the referenced component a property, the user can control which component is referenced from outside.

As an example, we created four components:

1. reader, with property filename, has method read() which opens the filename, reads the contents and returns it as a list of numbers.
2. generator, with property n, has method read() which generates and returns a list of n random numbers.
3. writer, with property filename, has method write(list) which opens the filename and writes the list of numbers.
4. plotter, with property style, has method write(list) which creates a plot and adds the list of numbers as a data line with style 'points', 'lines' or 'lines+points'.

The main program resolves the components source (with default 'reader') and sink (with default 'writer'), calls sink.write(source.read()) and returns. From the command line, the user can set things like source=reader and source.filename=sample.txt to override the defaults. Components can equally be manipulated from a Python shell, giving the user a lot of flexibility to dynamically reconfigure components.

The Pyre framework allows components to be distributed over the network, providing coarse-grained parallelism. For example, it should be possible to set up a SETI@home style parallel programming system in which an executive process distributes pieces of the computation to a number of processors and collects the results. The framework handles the details of transferring data between various machines, possibly with different architectures. This will be easier than programming such a system directly in Python, even with its rich set of networking primitives, and much easier than programming it directly in C.

2.4 Visual programming environment

The Pyre framework will be accessible from a visual programming environment (VPE). By using a restricted component interface, the Pyre components can be wired together in a dataflow diagram which can then be translated into a component network. The VPE will provide some control over how the components are distributed across multiple systems. Early prototypes of DANSE used ViPEr (now called Vision) as the VPE (see http://www.scripps.edu/~sanner/python/viper/index.html). The DANSE project will continue to work with the Vision developers to create a programming system usable with minimal knowledge of programming. With a rich set of components, and a convenient way to organize them, users should be able to do much of their work without having to learn Python.

Each component has parameters that must be provided, usually from the metadata in the files, but with the possibility for the user to override it. The VPE will provide input boxes allowing the user to enter parameter values, and display boxes for producing graphs, etc. Parameter defaults may be based on the outputs of other network components, but the user should be able to override them. We want flexibility without sacrificing convenience.

2.5 User interface

A graphical user interface (GUI) sits on top of a set of component networks, providing a simple view of data analysis and reduction. The DANSE architecture will provide tools to easy the creation of scientific user interfaces such as high-level interactive graphing widgets. In keeping with the spirit of flexibility of the underlying architecture, the GUI will be represented by a tree structure. Much like the XUL engine which underlines the Mozilla browser, nodes can be added and removed from the tree dynamically, and the positions of the

surrounding widgets will automatically adapt. This allows the interface to be dynamically configurable, even in a running application.

End user applications can be fully functioning members of DANSE. The early prototypes can already send data to ISAW, Matlab and IDL. DAVE, the NCNR data analysis package which is written IDL, will need some modifications so that it can call into and be called from the DANSE framework. Work is underway to define what those are.

2.6 Interface extensions

The user interface will provide hooks for extensibility. Without modifying the application, users will be able to redirect data from the through their own component network and back into the application. A registry will record the set of extensions the user has for the application. These extensions can be shared amongst users without involving the central application programmers.

A suitably modified independent application such as DAVE can use this mechanism to cooperate as a full member of the DANSE framework. For each compliant DAVE window, the user will be able to query what data is available for reading and writing, and know what parts of the interface is extensible. For example, if a dialog contains a graph and a list of buttons and a message bar, the user will be able to add a new button which selects a data line and a range from the graph and send it to a DANSE component. This component could sum the range of data it receives, and send it back to the message bar in the application. Now the dialog supports peak integration without the DAVE programmers doing any work. A more sophisticated extension would create its own DANSE window to store a set of peak locations and areas, with buttons to clear it or save it to a file.

3. DANSE reflectometry

Reflectometry stands to benefit in several ways from DANSE. By providing extensibility, modeling will be more flexible. By harnessing distributed computing, fitting can go faster. By sharing resources with other neutron scattering software developments, reflectometry software will be more capable.

The basic model for a reflectometry experiment consists of a series of layers with diffusion between the layers to create a scattering length density (SLD) profile. The model quickly becomes more complicated however, as samples can have repeating layers for example. The nature of the diffusion changes from sample to sample as well so users want to be able to choose a guassian model or a hyperbolic tangent model, or even their own model. For systems with no clear layer structure, users may want to define a SLD profile based on some theory in the literature. Many times users are comparing multiple samples, with the same basic structure in each sample and constraints tying parameters from one model to the other. Because we cannot anticipate in advance the broad range of models that are useful, reflectometry software needs to be extensiible.

Fitting reflectometry models is expensive. The reflectivity calculation itself is time consuming (up to 1s per evaluation on a 400 Mhz Pentium), but worse is that the fit is highly nonlinear, with many local minima. In order to find a reasonable fit, many function evaluations are required. With simultaneous fitting the process is even slower. The reflectometry calculation is embarassingly parallel, so we can hope to see close to linear speedup as we add processors to the problem. DANSE will allow users to harness distributed

computing without having to sort out the details of compiling and running parallel programs by themselves. The reflectometry subproject will be contributing distributed global optimization components to the DANSE framework.

By combining our resources DANSE can make richer software for all subprojects. For example, SANS and reflectometry want to fit biologically relevant models. To do this we need software to express, view and manipulate models, such as a lipid bilayer on top of a silicon substrate which researchers want to model in terms of bond angles for example. The two subprojects only need separate code to convert the 3D models to an expected signal, but can share the rest.

4. Reflectometry Survey

As part of the design work for DANSE we sent a survey to 140 reflectometry experts. Our goals were to announce the DANSE reflectometry effort, find out what software is in use by the community, describe our goals and request additional features, and assess the programming resources available within the community.

We received 31 responses (22%). The responses were balanced according to region (15 from North America, 12 from Europe and 4 from Australia and Japan) and type of user (17 facilities users from 10 labs and 14 university users from 12 universities).

Of the 31 respondents, 16 use the most popular package (Parrat32 from BENSC), 7 use each of the next most popular packages (AFIT from Oxford and Reflfit from NIST) and 4 or fewer use each of the remaining 22 packages. Unfortunely, none of the packages are good enough. Only 14 respondents use one package exclusively, and only 4 of those use the most popular package. Another 14 respondents use 2-3 packages, and 3 respondents use 4 or more packages.

When asked why they use a particular package, 13 responded "flexible modeling", 10 responded "friendly interface", 7 responded "simultaneous fitting", 5 responded "polarized neutrons". This was an open set response, so a number of other reasons were given by 1-2 respondents. In summary, people want power and flexibility, but they want it with a pretty face.

We also asked a question about software skills. 20 of the 31 respondents admitted to programming at least occasionally. Of these, 11 program regularly. Some of them even maintain their own reflectometry software because the software available elsewhere does not meet their needs. Writing high quality end user software is difficult. Working scientists for the most part do not have the time or the inclination to polish their software and make it available to community at large. If we provide a framework for developing software, with tools already in place for addressing the user interface issues, we believe scientists will willingly extend it with the necessary code to do their research.

5. Discussion

While still in the design stages, it is clear that the primary hurdles DANSE must overcome are not technical. Yes there are many technical issues which have not yet been resolved, but there is no feeling that they are insurmountable. Packaging is going to be a big task as it always is for complex, multi-platform products with many users

Open source software is a powerful development methodology. With the modern internet and teleconferencing capabilities it is easy to create transnational groups working together toward a common goal. Certain technologies are key to a successful project such as a central software repository with version control, an e-mail list to discuss the issues as they come up, a web site for project news, documentation and distribution, and an issue tracker to report bugs and requests. Many of these features are provided for free on sites such as sourceforge.net and savannah.gnu.org.

Besides organizing tools you also need people. Many successful open source projects are developed by 2 or 3 core developers and a handful of volunteers. Octave for example, has implemented a complete numerical environment with 1 core developer over 10 years and many, many contributors. The R project implementation of the S statistical environment lists 16 core developers from Europe, the US and New Zealand, and there are many more contributors.

The most successful open source projects are responsive to their user base. That is, when a user wants to contribute an extension back to the community, someone close to the core of the project will assist them in the process. Much of this involves helping with coding problems. Part is reviewing code for style and correctness, and making sure it is documented and tested. Some will be directing them to talk to other users who are working on similar code. The goal is to nurture productive members of the community. They will be happier not waiting on somebody else with different priorities to fix their problem, and we will be happier concentrating on our priorities rather than their problems.

As shown by the reflectometry survey, we have a community of active programmers, many shared needs and no good software currently fulfilling those needs. The NSF grant is providing resources for the less glamorous job of developing the DANSE infrastructure and organizing the community. The enthusiasm with which the survey was received leads me to believe that with the infrastructure in place, DANSE reflectometry will be an open source success.