Supporting Runtime Tool Interaction for Parallel Simulations

Christopher W. Harrop
Department of Computer and Information Science
1202 University of Oregon
Eugene, OR. 97403-1202
`harrop@cs.uoregon.edu`
`http://www.cs.uoregon.edu/~harrop`

Steven T. Hackstadt
Department of Computer and Information Science
1202 University of Oregon
Eugene, OR. 97403-1202
`hacks@cs.uoregon.edu`
`http://www.cs.uoregon.edu/~hacks`

Janice E. Cuny
Department of Computer and Information Science
1202 University of Oregon
Eugene, OR. 97403-1202
`cuny@cs.uoregon.edu`
`http://www.cs.uoregon.edu/people/faculty/cuny.html`

Allen D. Malony
Department of Computer and Information Science
1202 University of Oregon
Eugene, OR. 97403-1202
`malony@cs.uoregon.edu`
`http://www.cs.uoregon.edu/people/faculty/malony.html`

Laura S. Magde
Department of Geological Sciences
1272 University of Oregon
Eugene, OR. 97403-1272
`laura@cs.uoregon.edu`

Abstract:

Scientists from many disciplines now routinely use modeling and simulation techniques to study physical and biological phenomena. Advances in high-performance architectures and networking have made it possible to build complex simulations with parallel and distributed interacting components.

Unfortunately, the software needed to support such complex simulations has lagged behind hardware developments. We focus here on one aspect of such support: runtime program interaction. We have developed a runtime interaction framework and we have implemented a specific instance of it for an application in seismic tomography. That instance, called TierraLab, extends the geoscientists' existing (legacy) tomography code with runtime interaction capabilities which they access through a MATLAB interface. The scientist can stop a program, retrieve data, analyze and visualize that data with existing MATLAB routines, modify the data, and resume execution. They can do this all within a familiar MATLAB-like environment without having to be concerned with any of the low-level details of parallel or distributed data distribution. Data distribution is handled transparently by the Distributed Array Query and Visualization (DAQV) system. Our framework allows scientists to construct and maintain their own customized runtime interaction system.

Keywords:
runtime interaction, computational steering, matlab

Introduction

Scientists from many disciplines now routinely use modeling and simulation techniques to study physical and biological phenomena. Advances in high-performance architectures and networking have made it possible to build complex simulations with parallel and distributed interacting components. Unfortunately, the scientific software needed to support the construction of such complex simulations has lagged behind hardware developments. We focus here on one aspect of such support: runtime program interaction.

It is common for simulations to run for many hours, or even days, before producing an output that is analyzed to modify parameters for the next run. For scientists limited to such *post-mortem* analyses, the modeling process can be very time consuming. Runtime interaction can alleviate this somewhat by allowing a scientist to analyze intermediate results to determine whether a computation should be aborted or allowed to continue. In addition, they can dynamically adjust computational parameters to facilitate the exploration of a simulation's parameter space or to improve convergence times. Reducing the time it takes to observe the effects of a parameter change makes it easier for scientists to find trends or patterns in their model's behavior.

Runtime interaction, however, is difficult to deliver. First, because of their computational requirements, many simulations are run on parallel or distributed heterogeneous platforms, making it necessary to synchronize requests and access distributed data structures, activities unfamiliar to many scientists. Second, because many simulations are legacy codes, it is not possible for the runtime interaction system to make assumptions about internal program structures. Third, scientists need a complete understanding of their codes and the tools they use to process their data and thus they insist that their programs remain familiar and undergo little modification.

We have attempted to address these problems in our work on domain-specific environments for the geological sciences. That work first resulted in an environment, called TIERRA (Tomographic Imaging Environment for Ridge Research and Analysis) [3] that provided online visualization and program control capabilities for a seismic tomography application. This new work extends and abstracts that earlier system, providing a framework that can be customized for different applications. We demonstrate that framework with an initial implementation for the seismic application, called TierraLab.

Our framework's design is based on the philosophy that runtime interaction capabilities should be delivered in a way that does not interfere with the scientists' current *modi operandi*. This means that we hide details of parallel and distributed computation, respect and use legacy codes, and have the scientist/programmer actively involved in any necessary annotation or instrumentation of his/her code. As important, we integrate runtime interaction functionality with the scientists' normal data analysis environment so they do not have to re-implement their analysis tools or learn new programming languages, libraries, or paradigms.

TierraLab is an object-oriented runtime interaction system that extends MATLAB [12], an analysis engine familiar to scientists, with commands that provide distributed data access and execution control. In the next section, we discuss the design of our framework and its implementation in TierraLab. In Section 3, we demonstrate its utility using a seismic tomography application. In Section 4, we discuss related work, and in the final section, we present our conclusions and future work.

2. Design and Implementation

TierraLab's architecture consists of three components: a user interface, an analysis engine, and an interaction engine. Figure 1 depicts these components and their relationships. The lightly shaded boxes represent the components of the framework and their interiors show the details of the TierraLab implementation. We use the Nexus multithreaded runtime system [5] to communicate between components. The TierraLab analysis engine consists of a MATLAB process and a MATLAB MEX file [13], called `tierra.mex`. The interaction engine uses the DAQV [7] client library to send interaction commands to a tomography application that has been instrumented with DAQV routines from the DAQV application library. Later in the paper, we refer to the 10 labeled steps depicted in the figure.

Each of the TierraLab components is designed as a class hierarchy where an abstract base class defines the component's interfaces and functionality. An individual component is created by implementing a subclass of the appropriate base class. TierraLab's object oriented architecture provides a plug-and-play framework for delivering runtime interaction capabilities and online data analysis. Different implementations of the components can be combined to provide customized runtime interaction functionality to satisfy different types of users and situations. For example, the analysis engine component could be targeted for IDL [17] or Mathematica [18] instead of MATLAB. Similarly, different types of user interface components could be developed to meet the requirements of different users.

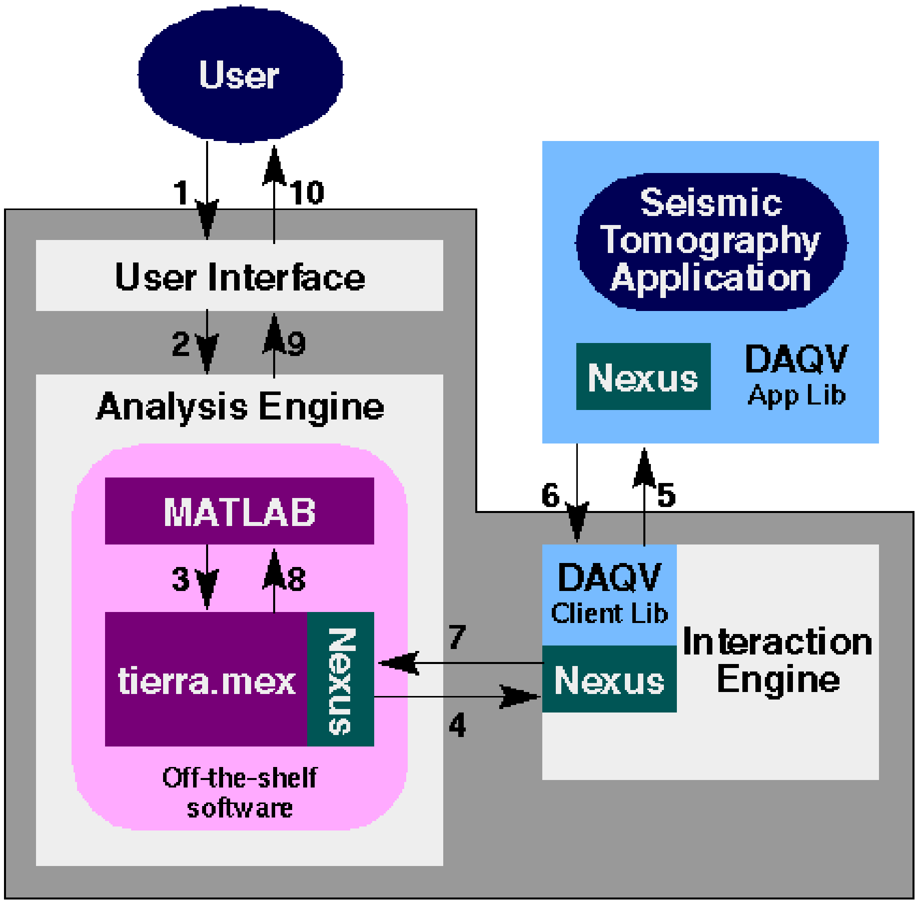

Figure 1. System Architecture

2.1 User Interface

The user-interface component handles all interactions with the user. It runs in a separate thread so that user interaction can be overlapped with computation. When the interface receives input from the user (step 1), it builds a command object and signals the analysis engine that a command is ready for evaluation (step 2). When the analysis engine is done executing the command, it signals the user interface (step 9). At this point, the command object may contain output that needs to be displayed to the user. The user interface displays the output (step 10) and then waits for more input. The signals used by the user interface and analysis engine are implemented using Pthread condition variables. The steps described above apply to all types of interfaces, but the details of how commands are received from the user and how output is displayed are specific to the implementation of the individual interfaces.

Since we used MATLAB for our initial analysis engine, we chose to design an interface subclass that replicates its command-line user interface. For example, the following excerpt from a TierraLab session shows commands for connecting to a running application, in this case a seismic tomography application named `hpt_77`. The ">>" is the command-line prompt.

```
----------------------------------------------------
 This application uses MATLAB as an analysis engine
----------------------------------------------------

< M A T L A B (R) >
(c) Copyright 1984-96 The MathWorks, Inc.
All Rights Reserved
5.1.0.421
May 25 1997


>> % Connect to the tomography code
>> urlFile = '/research/power/harrop/tierra/hpt_77/src/daqv_master.
>> fid = fopen(urlFile,'r');
>> url=fscanf(fid,'%s');
>> fclose(fid);
>> [id err]=tierra('attach',url);
>>
```

We are considering designing a subclass that implements a graphical user interface, but this is not a high priority because our users have not expressed much interest in it and are, so far, satisfied with the command-line interface.

2.2 Analysis Engine

The analysis engine must provide support for monitoring, analyzing, visualizing, and exploring program data online. Our approach accomplishes this by "wrapping" familiar, commercial, off-the-shelf (COTS) analysis packages. TierraLab, as mentioned above, uses MATLAB. We use the analysis engine not only to provide data analysis support, but also as the vehicle for runtime interaction. This makes runtime interaction capabilities palatable and easily accessible to scientists without forcing them to learn anything new.

An abstract analysis engine base class provides all TierraLab analysis engines with the functionality to evaluate commands and to interact with the user interface and interaction engine components. The actual implementation of this functionality is dependent on which off-the-shelf package it wraps. TierraLab's current analysis engine wraps MATLAB because it is heavily used by our community of scientists. The use of MATLAB has several advantages: it is familiar to the scientists; it is extensible, allowing scientists to create their own data analysis tools in its matrix-based language or through interfaces to C and Fortran; it is interactive, so scientists can build new analysis tools on-the-fly while interacting with a simulation; and MATLAB's language features, when extended with interaction commands, allow scientists to write interaction scripts. The primary disadvantage of MATLAB is performance; as expected, its interpretive environment is not as efficient as compiled code. However, this problem is minimized by the availability of C and Fortran interfaces which can accommodate computationally intensive tools and reserve use of the interpreter for less performance-critical analyses. Another problem with MATLAB is that it is a sequential data analysis engine, meaning it can process only one command at a time. This problem could be alleviated by incorporating MATLAB extensions like those in MultiMATLAB [14].

When TierraLab is launched, the analysis engine starts a MATLAB process using MATLAB's Engine interface [13] which consists of a two-way UNIX pipe and a set of high-level routines for communicating across it. Thus the MATLAB process does not need to execute on the same machine as the TierraLab process, although performance may be affected. The MATLAB analysis engine interfaces with TierraLab's interaction engine through a MATLAB MEX routine called `tierra`. That routine uses the Nexus multithreaded runtime system to send interaction requests to the interaction engine. Nexus is initialized for use by the MEX routine immediately after the MATLAB engine starts up, and before the user is allowed to enter commands. A persistent block of memory in the MEX routine is used to preserve the Nexus environment between invocations of the `tierra` command. When the analysis engine receives a signal from the user interface indicating that a command is ready for evaluation (Figure 1, step 2), the command is retrieved and passed to MATLAB for evaluation via the `engEvalString` engine interface routine. MATLAB commands are executed by the MATLAB engine and immediately return. However, the `tierra` interaction command results in additional steps.

When a user issues an interaction request, MATLAB invokes the `tierra` MEX routine (step 3), which checks the command for errors and then sends the command and its arguments, via a Nexus Remote Service Request (RSR), to the interaction engine (step 4). When the interaction command is complete, the results of the command are received by the MEX routine via another Nexus RSR (step 7). At this point the results are copied into the memory locations of the appropriate return arguments of the `tierra` command (step 8). When the command is finished, the output, if any, is copied into a buffer in the command object, and the user interface is signaled (step 9). The analysis engine then waits for its next command.

2.3 Interaction Engine

It is difficult to provide access to data structures that have been distributed across multiple heterogeneous processors. Our approach was to use the Distributed Array Query and Visualization (DAQV) system, which was specifically designed to support tool access to application data arrays during execution, even if that data has been distributed across multiple processors. We extended its functionality to include write access. The enhanced DAQV provides external tools with a view of a distributed array as a logical global array that can be accessed selectively via high-level array references. It allows multiple, remote tools to connect to an application and coordinate control over the program's execution; a more complete description of DAQV's design, implementation, and use can be found in [7].

The purpose of the interaction engine is to enhance the scientists' working environment with command extensions for interacting with their computations. Using DAQV allows us to hide data and code distribution details from the scientists, but the application must first be appropriately annotated with instrumentation. The annotations register variables and specify locations for read/write access to them. Once the annotation is complete, the scientists can invoke interaction with the application using familiar MATLAB syntax, including MATLAB scripts and programs. The results of interaction commands are stored in MATLAB's workspace, making extracted data readily available as input to other analyses. Conversely, the output of MATLAB operations can be used to modify the simulation's data.

Given our plug-and-play model, it is reasonable to consider alternative implementations of the interaction engine. Most obvious would be the use of debugger stubs and a software layer that connects MATLAB to a debugger (e.g., dbx). This approach would support robust program control and completely arbitrary data access. However, it has some serious disadvantages. First, it would shift the responsibility of multiple process coordination and distributed data handling onto our users. Our users know (and care) little about the parallelization of their applications and would be at a loss to deal with data distribution issues. DAQV hides these low level details, allowing tools to interact with the application and data structures at a semantic level known to users. Second, within a given source code file, debugger instrumentation is pervasive; instrumentation is selective only if the programmer is willing to subdivide (further) an application into multiple source code files. While it must currently be inserted manually, DAQV instrumentation is completely selective and can be applied only where needed regardless of source code structure. Third, debugger instrumentation has a detrimental impact on application performance, and in order to turn off the instrumentation, the application must be recompiled. Under DAQV, instrumentation can be enabled and disabled at runtime. When instrumentation is disabled, DAQV instrumentation remains in the program, but only incurs overhead comparable to that of a procedure call (about 3.5 microseconds), making the overall application impact negligible and allowing DAQV instrumentation to become part of the permanent application source. (A full accounting of DAQV performance can be found in [8].) Finally, having completely arbitrary execution control and data access may seem desirable, but explicit consideration of where and what data is to be accessed ensures that the data is accessed at points where it is both scientifically and semantically meaningful.

The enhanced DAQV model and command set were designed in conjunction with TierraLab's interaction engine, so their user commands largely coincide. TierraLab provides the following:

- **Attach**: Establish a connection to a running application;
- **Detach**: Terminate an existing connection to an application;
- **Yield**: Send a request to stop an application at the next opportunity;
- **Continue**: Send a request to resume execution of an application;
- **GetStatus**: Query the current execution status of an application;
- **Probe**: Send a request to read the contents of a variable in an application;
- **Mutate**: Send a request to overwrite the contents of a variable in an application;
- **GetVars**: Query the names of application variables available for reading and/or writing.

These commands encapsulate a great deal of complexity and functionality. For example, **Yield** and **Continue** deal with thread synchronization across heterogeneous processors, but the scientist does not need to know anything about threads to use the commands. **Probe** and **Mutate** commands must figure out where chunks of distributed data live and transport them between processes running on heterogeneous machines, but the user does not even have to know the data is distributed. DAQV thus greatly simplifies the incorporation of program interaction capabilities into remote tools.

As a DAQV client, the interaction engine communicates with DAQV-annotated applications through a separate process called the DAQV master. The master orchestrates the synchronization of application processes and the collection of distributed data. Although users can issue interaction requests at any time, interaction with an application can only take place at certain times, as governed by the placement of DAQV instrumentation. For example, when a user issues a **Probe** request, the TierraLab interaction engine calls the DAQV client library to send the request to the appropriate application's DAQV master process. The DAQV master puts the request into a queue, but does not process the request until the application reaches a `DAQV_PROBE` statement listing the requested variable as one of its arguments. Some interaction commands, such as **GetStatus**, only require communication with the DAQV master process and therefore return immediately.

In TierraLab, the DAQV interaction engine remains idle until it receives an interaction request from the `tierra` MEX routine via a Nexus RSR (Figure 1, step 4). The request specifies which command to perform and provides any arguments that are required for processing it. After examining the request message, the DAQV engine calls the appropriate method for handling the request. There is exactly one method, inherited from the interaction engine abstract base class, that implements each interaction command. The DAQV interaction engine implements these methods by making the appropriate calls to the DAQV client library (step 5). Once the requested interaction has taken place (step 6), the results are sent back to the `tierra` MEX file via another Nexus RSR (step 7). The interaction engine remains idle again until the next interaction request is received.

3. Application

Our efforts in domain-specific environments [3] and runtime interaction [7] have been in close collaboration with geophysicists that use seismic tomography to study the formation and structure of mid-ocean ridges [11]. This technique, similar to medical CAT scans, uses recorded arrival times of acoustic energy from explosions or earthquakes to construct models of the Earth's velocity structure. The geophysicists' seismic tomography code is a computationally intensive, parallel Fortran 90 program that runs on an SGI Power Challenge. This code takes between 2 and 8 hours to calculate a single iteration, and hundreds of iterations are required to search the model parameter space.

The geoscientists have developed a large collection of MATLAB programs for *post-mortem* visualization and analysis of the data produced by their tomography code. They wanted to execute some of these programs online. For example, one of their programs produces a visualization of the geometry of the seismic experiment which could be used early in a run to spot fatal errors before wasting hours of computation.

```
% Connect to the tomography code
urlFile = '/home/csi/laura/tomog/worktomogsrc/daqv_master_url';
fid = fopen(urlFile,'r');
url=fscanf(fid,'%s');
fclose(fid);
[id err]=tierra('attach',url);
:
% Get the station coordinates
[zdatum err]=tierra('probe',id,'zdatum',[1 1;1 1]);
[nsts err]=tierra('probe',id','nsts',[1 1;1 1]);
[station err]=tierra('probe',id,'station',[1 3;1 nsts]);
station(3,:) = -station(3,:);
:
% Wait for the code to load the bathymetry
status=0;
while status==0
  status=tierra('getstatus',id);
end

% Get the bathymetry data
[bath err]=tierra('probe',id,'zhang',[1 nx;1 ny]);

% Continue tomography program
err=tierra('continue',id);
:
% Plot top-down view of bathymetry and sources and stations
figure(1);
hold off;
[c h]=contourf(xcoords,ycoords,bathgrid,ci);
caxis([-3.41 -1.2])
hold on
plot(-1*evc(1,:),evc(2,:),'k.','markersize',8);
plot(-1*station(1,:),station(2,:),'ks','markersize',5,'linewidth',5);
```

Figure 2. Script

The first time our runtime interaction system was used, we provided the scientist with some initial DAQV instrumentation, and then supervised her as she took the necessary steps to produce a visualization of the experimental geometry described above. She added DAQV instrumentation to her tomography code, added `tierra` commands to an existing MATLAB program, and then used our system to run the script and display the figure online. Although this first trial run encountered a few small problems, we found that the scientist was able to learn and use our system in a remarkably short period of time.

Although she had no prior experience with, or knowledge of, TierraLab the entire process, starting from the instrumentation and instruction we provided, took only 3 hours. Most of this time was spent in the instrumentation phase. The scientist had little difficulty determining which instrumentation calls to insert. However, determining where to put the instrumentation, learning the arguments of the DAQV registration call, and compiling the code to use DAQV were more problematic. These difficulties rapidly decreased as she became more experienced. The process of adding interaction commands to her MATLAB program to read its input data from the executing program took only about 15 minutes. After her MATLAB program was augmented with interaction commands (excerpts shown in Figure 2), it was executed, and produced the plot in Figure 3. Although our system continues to evolve, our preliminary feedback is very encouraging.

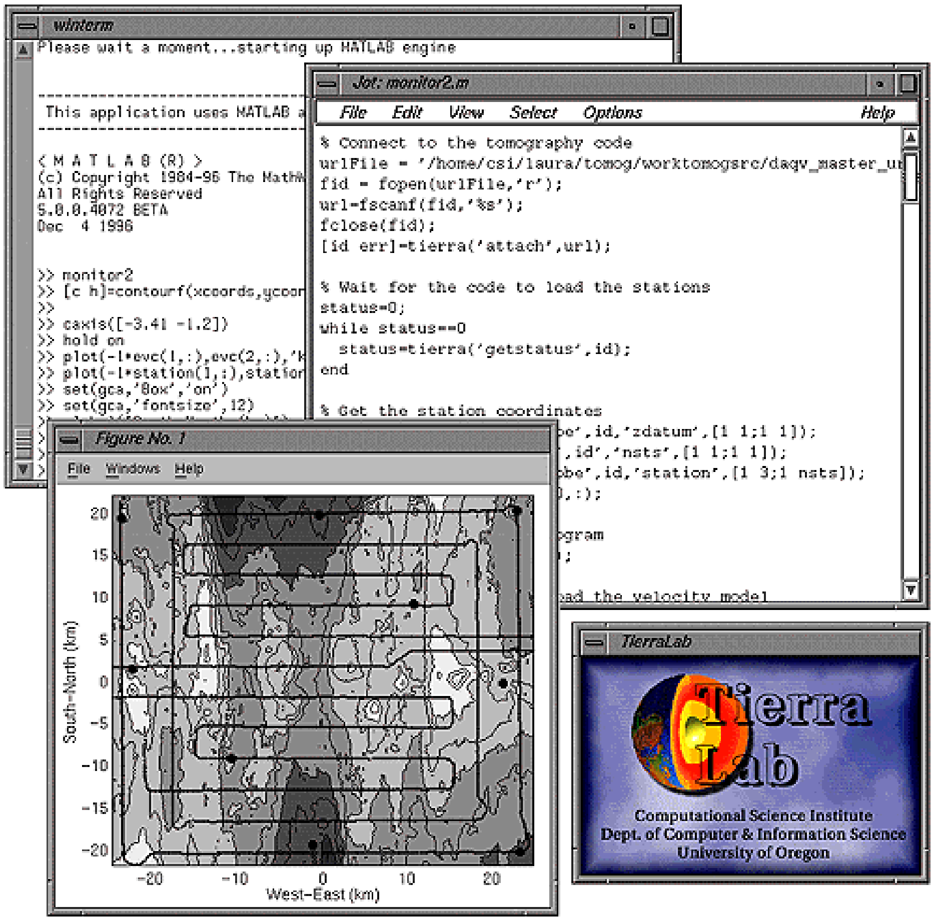

Figure 3. Experimental Geometry of the Seismic Experiment

4. Related Work

Several computational steering systems have been developed [3,4,6,9,10,15,16]. Some of them have focused on support for new applications which do not impose as many restrictions as legacy codes do. SCIRun [15], for example, uses an object-oriented data flow approach to support runtime interaction for new applications in computational medicine. However, it does not easily support pre-existing applications because code must be encapsulated within a SCIRun module. For applications where such modification is intolerable, SCIRun is unusable; our system specifically targets such applications. Some systems support legacy code. Some systems, such as VASE [9], also attempt to support legacy code. However, VASE only allows interaction to take place at the interfaces between steerable modules that are defined by instrumentation. Cumulvs [6] is a system that, in addition to providing access to distributed data at runtime, also supports application checkpointing. Finally, the PAWS [1] project does not directly support computational steering, however, it allows multiple parallel applications to connect with each other at runtime and efficiently share distributed data structures. However, none of these systems provide a general interactive, extensible, programmable mechanism for analyzing program data online. Our system attempts to fill this gap.

5. Conclusions and Future Work

The design of our system was guided by user-oriented requirements. Our strict adherence to those requirements has resulted in a system that allows scientists to construct and maintain their own customized runtime interaction systems. A critical factor in achieving this result was our integration of an existing commercial data analysis package with state-of-the-art runtime interaction functionality. It is clear that this combination is very effective for delivering runtime interaction capabilities to scientists. Our use of MATLAB dramatically increased the scientists' willingness to test our system during its development. The first time the scientists used the system, it took only a few hours and a little supervision to instrument their code, add interaction commands to a pre-existing MATLAB program, and view program data online. This quick prototyping was possible because the scientists did not have to learn new programming languages, libraries, or paradigms, or reimplement their analysis and visualization tools.

Since it is impossible for a single data analysis package to serve all the scientists' needs, we plan to expand our framework by supporting additional analysis engines such as Mathematica and IDL. Users will then be able to interact and share data among all these tools and their applications simultaneously. In principle, there is no limit to how many applications and tools can be coupled together using our framework. We also plan to build an instrumentation assistant and to improve the implementation of our interaction engine by making use of improved communications technologies, such as the Tulip runtime system [2], that require less copying of data. Additional information on our work can be found on our web site: <http://www.csi.uoregon.edu/>.

Acknowledgements

This work was supported in part by the following grants from the National Science Foundation: STI-9413532, CCR-9023256, ASC-9522531, S0155A-02. The research was done within the Computational Science Institute at the University of Oregon http://www.csi.uoregon.edu/.

We would like to thank Chad Busche of the Department of Computer and Information Science, University of Oregon, for his contributions in providing invaluable DAQV software support. We would also like to thank Robert Dunn of the Department of Geological Sciences, University of Oregon, for contributing the seismic tomography source code.

References

[1] P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski, **Efficient Coupling of Parallel Applications Using PAWS**, Proc. of the 7th IEEE Symposium on High-Performance Distributed Computing (HPDC-7), pp. 215-222, 1998.

[2] P. Beckman and D. Gannon, **Tulip: A Portable Run-Time System for Object Parallel Systems**, Proceedings of the 10th International Parallel Processing Symposium, (Aug 1996).

[3] J. Cuny, R. Dunn, S. Hackstadt, C. Harrop, H. Hersey, A. Malony, and D. Toomey, **Building Domain-Specific Environments for Computational Science: A Case Study in Seismic Tomography**, Intl. Journal of Supercomputing Applications and High Performance Computing, Vol. 11, No. 3, pp. 179-196, 1997.

[4] G. Eisenhauer, W. Gu, K. Schwan, and N. Mallavarupu, **Falcon--Toward Interactive Parallel Programs: The On-line Steering of a Molecular Dynamics Application**, Proc. Third Intl. Symposium on High-Performance Distributed Computing (HPDC-3), San Francisco, August 1994.

[5] I. Foster, C. Kesselman, and S. Tuecke, **The Nexus Approach to Integrating Multithreading and Communication**, Journal of Parallel and Distributed Computing, Vol. 37, No. 1, August 1996, pp. 70-82.

[6] G. Geist II, J. Koh., and P. Papadopoulos, **CUMULVS: Providing Fault**

Tolerance, Visualization, and Steering of Parallel Applications, Intl. Journal of Supercomputing Applications and High Performance Computing, Vol. 11, No. 3, pp. 224-235, 1997.

[7] S. Hackstadt, C. Harrop, and A. Malony, **A Framework for Interacting with Distributed Programs and Data**, Proc. of the 7th IEEE Symposium on High-Performance Distributed Computing (HPDC-7), pp. 206-214, 1998.

[8] S. Hackstadt and A. Malony, **Performance of a System for Interacting with Parallel Applications**, International Journal of Parallel and Distributed Systems and Networks, special issue on Measurement of Program and System Performance, to appear.

[9] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber, **VASE: The Visualization and Application Steering Environment**, Proc. Supercomputing ’93, pp. 560-569.

[10] R. van Liere and J. van Wijk, **CSE: A Modular Architecture for Computational Steering**, Technical Report CS-R9615, CWI, Centrum voor Wiskunde en Informatica, 1996.

[11] L. S. Magde, A. H. Barclay, D. R. Toomey, R. S. Detrick, and J. A. Collins, **Crustal Magma Plumbing Within a Segment of the Mid-Atlantic Ridge, 35N,** submitted to Nature, 1998.

[12] Math Works, The, **MATLAB: High Performance Numeric Computation and Visualization Software Reference Guide**, Natick, MA. 1992.

[13] Math Works, The, **MATLAB: High Performance Numeric Computation and Visualization Software External Interface Guide**, Natick, MA. 1992.

[14] V. Menon and A. Trefethen, **MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing**, Proceedings of Supercomputing ’97 (SC97).

[15] S. Parker, D. Weinstein, and C. Johnson, **The SCIRun Computational Steering Software System**, Modern Software Tools in Scientific Computing, E. Arge, A. Bruaset, and H. Langtangen, eds., Birkhauser Press, 1997.

[16] S. Rathmayer and M. Lenke, **A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications**, Proc. IPPS ’97, 1997.

[17] Research Systems Inc., **Using IDL**, March 1997 Edition

[18] S. Wolfram, **Mathematica: A System for Doing Mathematics by Computer**, Second Edition, Addison-Wesley Publishing Company, Inc. 1991.