A Framework for Interacting with Distributed Programs and Data

Steven T. Hackstadt, Christopher Harrop
and Allen D. Malony

CIS-TR-98-02
June 1998

Department of Computer and Information Science
University of Oregon

S. Hackstadt, C. Harrop, and A. Malony, *A Framework for Interacting with Distributed Programs and Data*, Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7), Chicago, IL, USA, July 28-31, 1998. Also available as University of Oregon, Department of Computer and Information Science, Technical Report CIS-TR-98-02, June 1998.

Keywords: parallel tools, distributed arrays, visualization, computational steering, model coupling, runtime data access, runtime program interaction, Fortran 90

A Framework for Interacting with Distributed Programs and Data

Steven T. Hackstadt Christopher W. Harrop Allen D. Malony
Computational Science Institute
Department of Computer and Information Science
University of Oregon, Eugene, OR 97403

Abstract

The Distributed Array Query and Visualization (DAQV) project aims to develop systems and tools that facilitate interacting with distributed programs and data structures. Arrays distributed across the processes of a parallel or distributed application are made available to external clients via well-defined interfaces and protocols. Our design considers the broad issues of language targets, models of interaction, and abstractions for data access, while our implementation attempts to provide a general framework that can be adapted to a range of application scenarios. The paper describes the second generation of DAQV work and places it in the context of the more general distributed array access problem. Current applications and future work are also described.

1 Introduction

This paper describes the second generation of work on the Distributed Array Query and Visualization (DAQV[1]) project. It is intended to be both an update to our previous work and a technical summary of our recent development efforts. We also discuss how our work contributes to the general problem of accessing distributed data on parallel and distributed systems.

DAQV provides one solution to the general problem of providing high-level access to distributed arrays for the purpose of visualization and analysis. It does this by "exposing" the distributed data structures of a parallel (or distributed) program to external tools via interfaces that obviate the need to know about data decompositions, symbol tables, or the number of processes involved or where they are located. The goal is to provide access at a meaningful and portable level -- a level at which the user is able to interpret program data and at which external tools need only know simple, logical structures.

The seminal work on DAQV, by authors Hackstadt and Malony [8,9] and under the auspices of the Parallel Tools (Ptools) Consortium [23], resulted in a publicly available DAQV reference implementation. Ptools' user-oriented process was instrumental in identifying the needs to be addressed by DAQV and refining the scope of the original project. Though the Ptools portion of the project was completed in June 1997, we continue to follow the Ptools model by working closely with scientists to determine the functionality and operation of our tools.

Our current project, referred to as DAQV-II, is funded by Los Alamos National Laboratory (LANL) [20] and is being carried out in conjunction with the Computational Science Institute (CSI) at the University of Oregon [2]. The programming and execution requirements of this user community have driven the evolution and realization of a new operational model for DAQV. For example, a recent project in which our earlier work played a central role addressed the development of a domain-specific environment (DSE) for seismic tomography [3]. That environment is being reimplemented with DAQV-II as a result of the project's evolving needs.

In this context, DAQV-II continues to be the source of interesting research issues and challenging implementation problems. Many of these will be revealed in this paper by describing three aspects of DAQV-II: how it differs from the original reference implementation, how it operates, and how it is used to address the needs of scientists. These topics are covered in sections on design, implementation, and application, respectively. First, though, we briefly describe work related to DAQV-II.

1. "DAQV" is pronounced "dave"; the "Q" is silent.

2 Related work

DAQV is a confluence of several research ideas. Not surprisingly, certain aspects of its design, functionality, and implementation appear in other types of tools. For example, systems like Falcon [5], CUMULVS [7], VASE [14,26], and SCIRun [24] typically provide complete, closed environments for computational steering. At the core of these systems is functionality similar to that provided by DAQV-II, but DAQV-II is unique in that it implements "computational steering middleware," which allows it to support steering functionality without imposing constraints on the system and environment provided to the end user. In this way, DAQV-II also supports a robust, extensible framework for developing runtime interaction and computational steering systems. This style of software development is consistent with our in domain-specific environments [3] for computational science; DAQV-II represents functionality that can be easily integrated with and adapted to domain-specific software. As we describe later, we have coupled DAQV-II with MATLAB to provide a system for runtime simulation-tool interaction with seismic tomography applications [12]. DAQV-II is also unique in that it allows interactions with distributed data; of the steering systems listed above, only CUMULVS directly supports distributed data. With respect to performance and efficiency, domain-specific runtime visualization systems like pV3 [10] are often able to make assumptions and optimizations that obviate the need to directly address the more general distributed data problem.

Because the distribution of parallel data is an important factor in the performance behavior of a program, viewing data and performance information in relation to the distribution assists the user in tuning endeavors. For example, the GDDT tool [19] provides a static depiction of how parallel array data gets allocated on processors under different distribution methods; it also supports an external interface by which runtime information can be collected. In the DAVis tool [16], distribution visualization is combined with dynamic data visualization to understand the effects of decomposition on algorithm operation. Similarly, Kimelman et al. show how a variety of runtime information can be correlated to data distribution to better visualize the execution of HPF programs [17]. And in the IVD tool [15] a data distribution specification provided by the user is used to reconstruct a distributed array that has been saved in partitioned form. DAQV-II could easily provide distribution and runtime information to these tools from a running application. In this way, DAQV-II is unique because it combines distributed program interaction and data access at a level low enough that computational steering, distribution visualization, and other distributed data performance tools could be made to use it.

3 Design

In considering the general problem of interacting with distributed arrays on parallel and distributed systems, our design of DAQV-II was guided by three important considerations: language targets, models of interaction, and abstractions for data access. Decisions made in these areas play a significant role in determining the usability, flexibility, and implementation difficulty of the resulting system.

3.1 Language targets

The choice of language targets for a distributed array interaction system has two important implications. First, the amount of compiler and runtime support offered by a given language for handling distributed data directly impacts the technical difficulty of implementing distributed array access for that language. The second issue is whether providing distributed array access capabilities for a given language will actually be useful to scientists.

Our original implementation of DAQV (referred to hereafter as DAQV-I) targeted High Performance Fortran (HPF) [13]. We relied heavily on the HPF language, compiler, and runtime support in implementing key components (e.g., data access, interprocess communication) of the DAQV-I system. This "language-level" implementation afforded us almost guaranteed portability across different HPF computing environments and minimized our worry about accessing and handling distributed data. On the other hand, DAQV-I functionality was limited to applications written in High Performance Fortran. In our work with CSI scientists, as well as in our other collaborations, there was considerable reluctance in embracing HPF because of its performance and restricted (i.e., data-parallel) programming model. In contrast, there was significant interest in gaining DAQV functionality for Fortran 77 and Fortran 90 applications that use a message passing library such as PVM [4] or MPI [22].

Thus, in DAQV-II, Fortran 90 and MPI became our primary implementation targets. However, we also wanted to avoid being explicitly tied to a given language. Thus, DAQV-II does not actually rely upon Fortran 90 (or MPI) for any key functionality; in contrast to the first version, this makes it easy to retarget to other languages such as Fortran 77, HPF, C, and C++. The primary disadvantage of this generalization is that DAQV-II must now address more fully the general problem of reassembling distributed arrays; we can no longer rely on a HPF-like runtime environment for this support. This presented a tremendous technical challenge that was cleverly avoided in the original work. In DAQV-II, we currently support HPF-like `BLOCK` and `CYCLIC` data distributions over multiple dimensions.

3.2 Models of interaction

Interacting with distributed arrays and parallel programs can follow different models of interaction. In our work, we have considered both client/server and peer-based models. We have found that client/server interactions tend to be easier to understand and implement, while peer-based models are more general and allow a wider range of interaction.

DAQV-I imposed a very strict client/server model of interaction on the tools and applications involved. Tools (e.g., for visualization, debugging, etc.) were clients of the HPF application, which essentially became a distributed data server when linked with the DAQV server library. The model also distinguished between data clients and control clients; only a single control client could connect to a DAQV-I server, while any number of data clients were allowed. This model was simple for users to understand, relatively easy to implement, and avoided synchronization issues among clients. However, the model was too restrictive for addressing emerging requirements like computational steering and simulation coupling [18], which require bidirectional data flow and peer-based interaction.

In response, DAQV-II implements a more general model of interaction and places less emphasis on client/server relationships. The high-level view of DAQV-II is one of tools and applications that interoperate with each other by exchanging data and sharing control of each other's execution. Every component of the environment, whether it is a tool or DAQV-enabled application, is viewed equally. Thus, there is no required distinction between data and control clients; every component is allowed to access the data of other components and to register data (distributed or not) for access by other components. Of course, this does not preclude a strict client/server relationship like that in DAQV-I, if that is all that is required. However, DAQV-II affords many additional system topologies, including peer relationships in which multiple DAQV-enabled applications can read and write each other's data. Such functionality is critical for model coupling, the sharing of control and the exchange of data between multiple simulations at runtime. Execution control is required so that the coupled simulations can be synchronized when necessary; data access is required so that, for example, the intermediate output of one model can be used as the input to some other one.

3.3 Abstractions for data access

The development of effective abstractions for accessing distributed data can have a direct impact on the usability and applicability of a system. Our efforts toward the creation of abstractions for data access have focused on capturing different "perspectives" of distributed data access. In our experimentation, we have portrayed DAQV functionality from two different perspectives: (1) *who* (or what) decides to access data, and (2) *how* the data is to be accessed.

The original motivation for DAQV-I stemmed from a desire for simple, high-level access to distributed data. Users wanted to be able to "get the data" using something as simple as a `PRINT` statement [8]. They also wanted to use existing visualization tools to display the data. This resulted in the development of a *push* model for data access in which the user inserted calls to a `DAQV_PUSH()` procedure in their code (much like `PRINT` statements). When the program was executed, data would be delivered automatically to the appropriate display tool. This very simple model allowed users to do one-time visual data dumps or to create animations of data over time. However, to support a more interactive and flexible approach to array visualization, we also implemented a *pull* model which allowed rudimentary control over program execution and selection of the arrays to be visualized. This control and selection was allowed at explicit yield points compiled into the code and was accessed at runtime through an external interface. For our research in the CSI, however, we learned a valuable lesson from DAQV-I: a push model for data access is unnecessary, and even undesirable, in interactive domain-specific environments.

Push data transfers are fixed because they are compiled into the application. Thus, a push statement in the body of a loop results in a data transfer on every iteration. But for simulations with long, computationally intense loops, a scientist's need to check data is the exception, not the norm. Thus, despite the original Ptools feedback, our scientists desired a low-overhead, interactive means of accessing program data. Having two different modes of access is unnecessarily complicated, especially when external clients can be written to mimic push-like behavior.

As a result, we reformulated and generalized the abstraction for data access in DAQV-II to include both the reading and writing of data (e.g., for computational steering). We have termed the new abstractions simply *probe* (for reading data) and *mutate* (for changing data). We have preserved the notion of applications yielding to DAQV-II to allow data to be repeatedly probed or mutated. However, it is the responsibility of the user to make sure that probe and mutate instrumentation is placed in semantically and scientifically meaningful locations in the code. Finally, DAQV-II is able to restrict access only to in-scope arrays, a feature not supported in DAQV-I.

In summary, the lessons learned from the DAQV-I reference implementation, combined with a shifting focus in

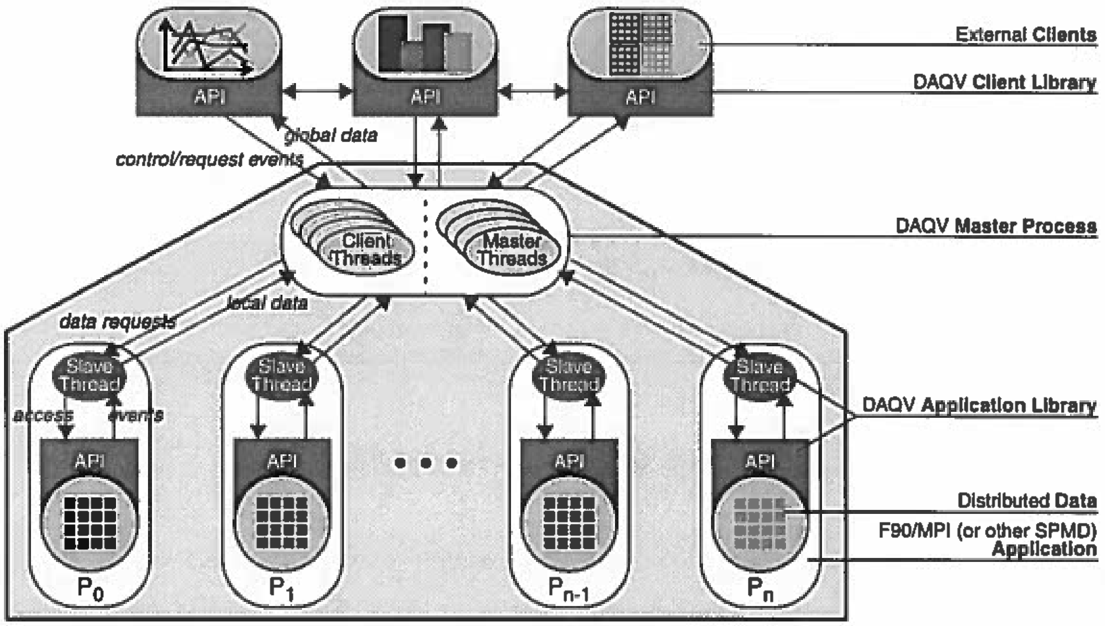


Figure 1: DAQV-II consists of a library for external client tools, a library for application processes with distributed data, several threads, and a master process.

our own research toward domain-specific environments, computational steering, and model coupling, have resulted in fundamental changes in our design for DAQV-II. To that end, our research into the design of a system for distributed array interaction has focused on the important issues of language targets, models of interaction, and data access abstractions.

4 Implementation

The implementation of DAQV-II can be divided into two parts, the client and application libraries. The client library is used by all components (i.e., tools and applications) in a DAQV system. It contains the interfaces and protocols for attaching to and detaching from other tools and/or applications, learning about what data is available, requesting and altering that data, and controlling program execution. The second part, the application library, is used by a parallel or distributed program that wishes to make distributed data available via DAQV-II. The application library contains code that supports communication among the application processes, answers data requests, and reassembles distributed data; it is the core of DAQV-II and will be the focus of our attention in this section. Figure 1 reveals the libraries, threads, and processes that make up the DAQV architecture.

4.1 Application structure

Our primary application targets consist of single-program multiple-data (SPMD) processes written in Fortran 77 or Fortran 90 that may communicate with each other through some message-passing service such as PVM or MPI. We refer to the individual SPMD processes as the application processes. Each application process contains local data that is part of a logical, global array that has been distributed among the application processes. However, the concept of data distribution is likely to be foreign or irrelevant to both the language (e.g., Fortran 90) and the communication system (e.g., MPI). The distribution may manifest itself in certain aspects of the application's algorithms, but we have no means of determining that from the source code. Hence, for our purposes, the "data distribution" exists solely in the programmer's higher-level understanding of the task being solved; therefore, the programmer is our only source for obtaining that information. We are currently developing tools that support collecting this information from the user.

4.2 Interface, threads, and processes

Each application process is linked with a DAQV library that provides a simple procedural interface to the application programmer. The interface allows the programmer to describe how data is distributed and to indicate places in the code where it may be accessed. The library is primarily responsible for initializing other parts of the DAQV-II system and supporting a few data structures. The procedural interface is documented in Table 1.

Procedure and Arguments	Description

DAQV_INIT(
  int *procId,
  int *numProcs
)
	Initializes process procId of an application with numProcs processes.

DAQV_REGISTER(
  void *array,
  int *rank,
  int dim[],
  int *type,
  char *name,
  int distType[],
  int distParam[],
  int *procRank,
  int proc[]
)
	Registers distributed data contained in array, whose rank, dimensions, type, and textual name are also provided; distType[] and distParam[] describe the distribution in each dimension onto an abstract processor grid of rank procRank and with dimensions contained in proc[].

DAQV_PROBE(
  int *probeType
  void *array1
  void *array2
  ...
)
	Fills pending probe (*i.e.*, read) requests for any of the arrays listed; if probeType is yielding, then the call waits for additional requests from clients and does not return until explicitly told.

DAQV_MUTATE(
  int *mutateType
  void *array1
  void *array2
  ...
)
	Same as above, except the arrays listed can be mutated instead of probed.

DAQV_EXIT	Cleans up and terminates DAQV-II operation.

Table 1: The DAQV-II procedural interface allows programmers to make distributed data available to external tools.

When each process initializes DAQV-II, the library creates a slave thread that shadows execution of the application process. The purpose of these threads is to maintain information about available distributed data and to perform accesses to that data when requested. The reason for embodying this functionality in the form of a thread is so that querying the information and accessing the data can be done concurrently with the execution of the application processes, though we do not currently take advantage of this capability. (That is, in our current model, distributed data access is only allowed at those points explicitly specified in the application process.) DAQV-II uses the threading system provided by the Nexus multithreaded communication library [6] to create and manage these slave threads.

The individual slave threads are coordinated by a separate process called the DAQV master. In addition to coordinating the slave threads, the master process is responsible for interacting with the other tools and applications in the DAQV-II environment by acting as the "single point of contact" for the entire set of application processes. The master process spawns additional threads to handle the various requests it receives; because it also uses Nexus, the master process is fully multi-threaded and can take advantage of multiple processors, if available.

DAQV-II expects to be operating in a distributed, heterogeneous environment. Every process involved in DAQV-II -- the application processes, the DAQV master process, and the external tools and applications -- can run on the same machine, different machines, or any combination thereof. Nexus effectively masks any implementational concerns of operating in such an environment. With respect to performance, though, the user may need to consider carefully where the various DAQV components reside. We are planning to integrate performance monitoring support in DAQV-II to aid in this task [25].

4.3 Protocols

The interactions that take place between DAQV-II components occur at two levels: (1) interactions between external tools and an application's master process, and (2) interactions between the master process and the application process slave threads. These interactions can be complex. Figure 2 gives a high-level view of the protocols between external tools, the master process, and application slave threads. In this section, we describe in detail how one part of this protocol, read-only data access (probe), is implemented.

For our example, let us assume the existence of an executing, DAQV-enabled, parallel program that has registered a distributed array, A, with DAQV-II. Let us also assume the existence of a DAQV-enabled client tool capable of sending and receiving the events shown in Figure 2. Further, we assume that this tool has already attached to the application's master process and has received the necessary registration information. The process we are about to describe begins when the user makes a request for some piece of the array A.

The tool uses a procedure provided by the DAQV client library to send a request for the desired array section to the master process. This procedure causes a Nexus remote service request (RSR) to be made to the master process, which responds by adding the request to a queue of pending events. Execution control is returned to the client tool as soon as the master process receives the request; the tool may now periodically query the client library to see if the request has been answered.

Meanwhile, on the application side, the parallel program eventually reaches a call to the procedural interface routine DAQV_PROBE(), which has the array symbol A

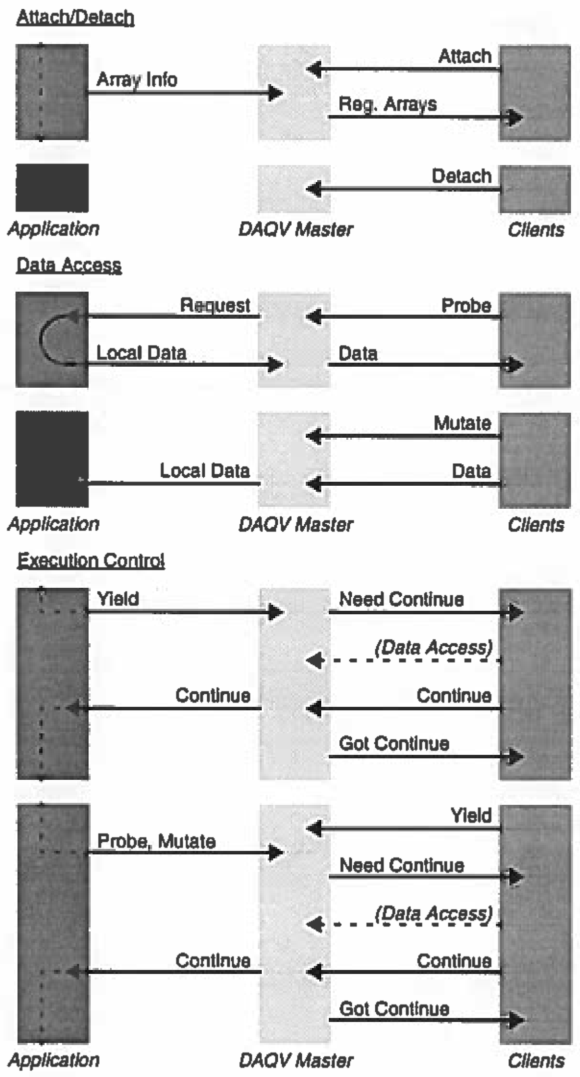


Figure 2: A high-level view of DAQV-II events and protocols for attaching to and detaching from applications, accessing data, and controlling execution.

as one of its arguments. Upon invocation, DAQV_PROBE() transfers execution control to the slave threads. The threads send the argument list of the DAQV_PROBE() call to the master process, which confirms whether this is a valid probe point.

If the call to DAQV_PROBE() is valid, the master searches its event queue for pending probe requests from external tools. Each item on the queue must be cross-referenced with the argument list from the call to DAQV_PROBE() so that only requests for arrays that are actually in scope are answered.

The master forwards each valid request in its event queue to the slave threads. Using distribution information provided by the user when the array was registered, the slave threads determine what part of the array, if any, resides locally.[2] The threads each allocate a buffer and fill it with the appropriate values read from the address space of their associated application process. The slave threads respond to the master process by sending the filled buffers. The master process coordinates receipt of the data from the slaves, assembles the data into a single buffer, and sends it to the external tool that originally requested it.

Similar types of interactions between the master process and slave threads occur when application processes register distributed arrays and when data is mutated. As described above, data access (i.e., probe and mutate) is complicated by having to interpret array distributions and map global data requests from tools into localized requests for the application processes. These routines can be a bottleneck if not implemented with care; we continue to optimize and refine these parts of our implementation.

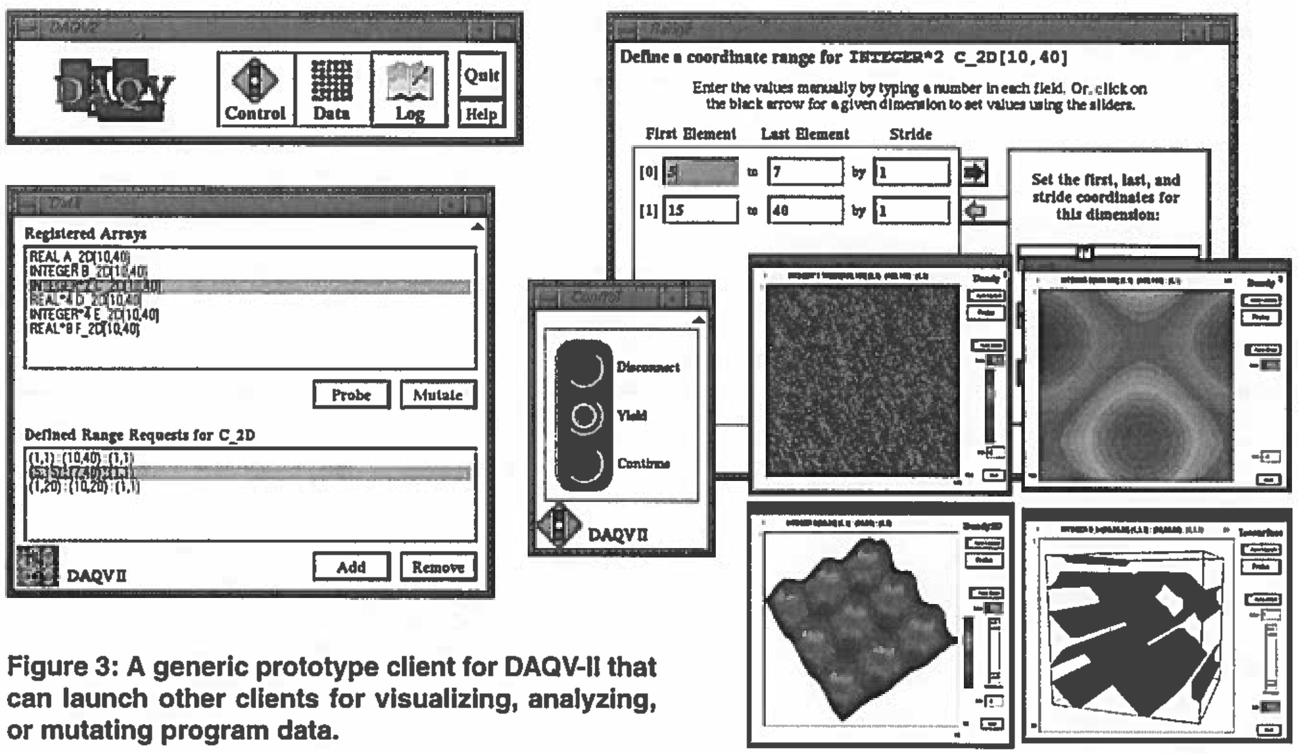
5 Applications

Our current prototype clients for DAQV-II are shown in Figure 3. These clients demonstrate DAQV functionality and are intended to be used as models for other development efforts. As we have mentioned earlier, our interest in supporting the activities of computational scientists is largely responsible for shaping the capabilities of DAQV-II. We plan to use DAQV-II in a number of ways that pertain to these activities. DAQV-I was the cornerstone of a complete program interaction, steering, and visualization environment for a seismic tomography application [3]. However, the performance penalty incurred by using HPF (which DAQV-I required) was too severe, and, as a result, the environment was not used by the scientists.

Now that DAQV-II supports Fortran 90 and improved support for client interaction, we are reimplementing the environment in a new version called TierraLab [11,12]. Built as an extension to MATLAB [21], TierraLab offers powerful, interactive, runtime data analysis and visualization capabilities not previously available to the scientists. The environment is already beginning to be used by geophysicists at the University of Oregon. A TierraLab session with two seismic visualizations created in MATLAB is shown in Figure 4.

The design and implementation of DAQV-II has been driven by the need to interact with distributed data in parallel scientific applications. While we have identified certain priority targets for our work (i.e., SPMD Fortran 90 codes with implicitly distributed data), the DAQV-II

2. Although not discussed here, the determination of where distributed data resides is non-trivial. DAQV-II allows access to any regular sub-section of a multidimensional array distributed in a BLOCK or CYCLIC manner in each dimension.



Registered Arrays

```
REAL A_2D[10,40]
INTEGER B_2D[10,40]
INTEGER*2 C_2D[10,40]
REAL*4 D_2D[10,40]
INTEGER*4 E_2D[10,40]
REAL*8 F_2D[10,40]
```

Defined Range Requests for C_2D

```
(1,1):(10,40):(1,1)
(5,15):(7,40):(1,1)
(1,20):(10,20):(1,1)
```

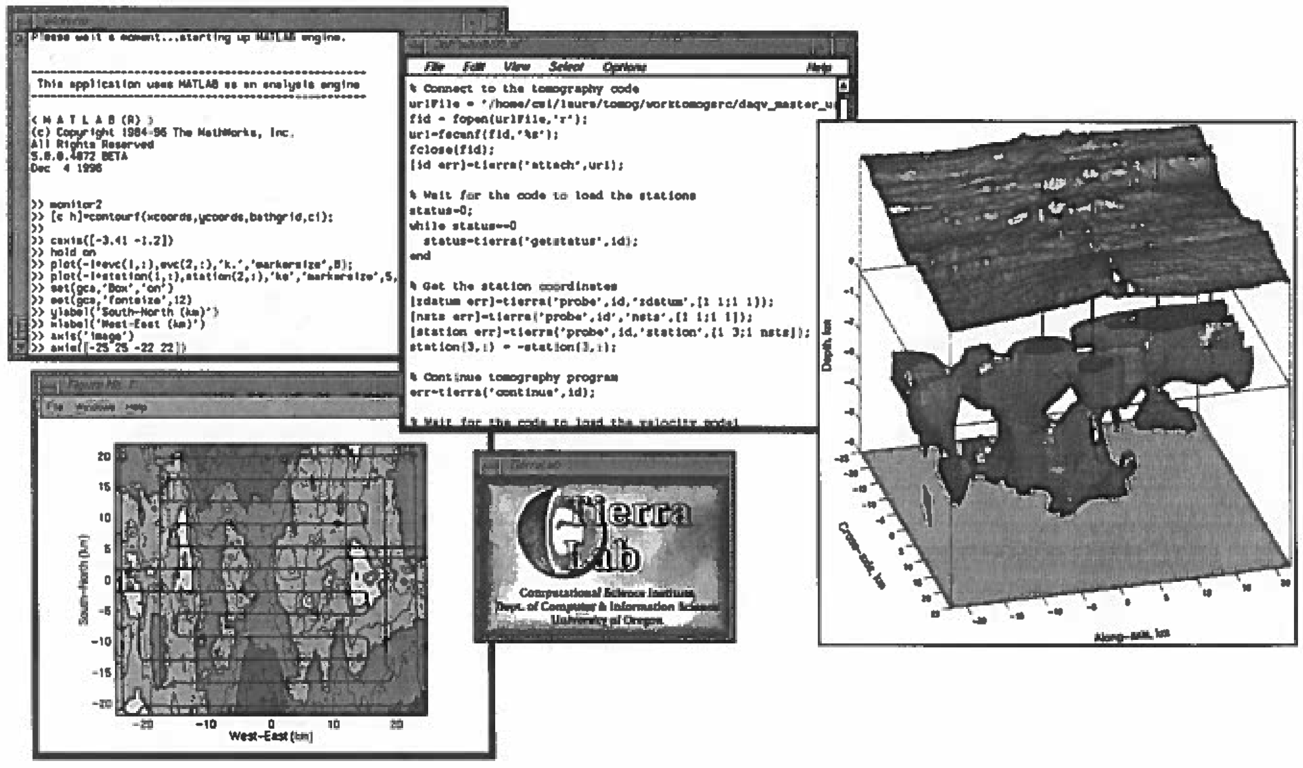
Define a coordinate range for `INTEGER*2 C_2D[10,40]`

Enter the values manually by typing a number in each field. Or, click on the black arrow for a given dimension to set values using the sliders.

	First Element		Last Element		Stride
[0]	5	to	7	by	1
[1]	15	to	40	by	1

Set the first, last, and stride coordinates for this dimension:

Figure 3: A generic prototype client for DAQV-II that can launch other clients for visualizing, analyzing, or mutating program data.



```
Please wait a moment...starting up MATLAB engine.

---------------------------------------------------------
 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
5.0.0.4072 BETA
Dec  4 1996

>> monitor2
>> [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,:),'ko','markersize',5,
>> set(gca,'Box','on')
>> set(gca,'fontsize',12)
>> ylabel('South-North (km)')
>> xlabel('West-East (km)')
>> axis('image')
>> axis([-25 25 -22 22])
```

```
% Connect to the tomography code
urlFile = '/home/csi/laura/tomog/worktomogsrc/daqv_master_u
fid = fopen(urlFile,'r');
url=fscanf(fid,'%s');
fclose(fid);
[id err]=tierra('attach',url);

% Wait for the code to load the stations
status=0;
while status==0
  status=tierra('getstatus',id);
end

% Get the station coordinates
[zdatum err]=tierra('probe',id,'zdatum',[1 1;1 1]);
[nsta err]=tierra('probe',id','nsta',[1 1;1 1]);
[station err]=tierra('probe',id,'station',[1 3;1 nsta]);
station(3,:) = -station(3,:);

% Continue tomography program
err=tierra('continue',id);

% Wait for the code to load the velocity model
```

Figure 4: The TierraLab environment, developed for use by seismologists, integrates DAQV-II functionality into MATLAB's powerful data analysis and display capabilities.

model meets a significantly broader range of requirements. We feel that the functionality of DAQV-II is compelling enough that a more general class of distributed computing problems can also be addressed. In particular, there are problems whose solutions traditionally have not been viewed as involving "distributed data." We are currently exploring two examples of this possible role for DAQV-II.

Consider a system like the Network Weather Service [27] which monitors the performance and utilization of a collection of distributed, heterogeneous machines. Each node maintains an array of metrics that are periodically reported (or collected) by a client responsible for displaying the results. This is fundamentally a distributed data problem. As an experiment, we constructed a simple distributed load monitoring tool written in C and using DAQV-II. As shown in Figure 5, a DAQV-enabled load daemon is executed on each of eight machines in a distributed environment and communicates with a DAQV master process executed somewhere else in the system.

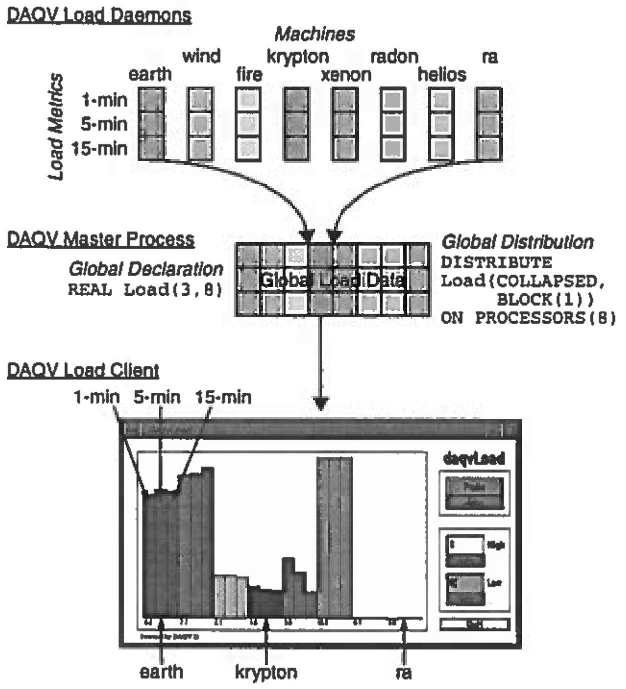


Figure 5: A distributed load monitor implemented in C and using DAQV-II.

Each daemon tracks the 1-minute, 5-minute, and 15-minute load averages in a 3-element array. This array is globally viewed as a two dimensional array distributed across eight machines. A client tool may make requests for the load information across all or some of the machines through the master process via a logical reference to all or part of the global array. These requests, in turn, are converted into requests for the associated data from the appropriate load daemons. In response to its request, the client receives a single, global array of load data, and data is only communicated when it is requested. In this case, we are essentially imposing the notion of a BLOCK data distribution on the distributed load data. The advantage of using DAQV-II is in the simple interface it supports and the ease with which client tools can be developed. (As a comment on DAQV-II's high-level utility, we built this prototype in less than four hours.) In addition, the portability of DAQV-II is limited only by the portability of the Nexus communications library.

We are also applying this technique to other performance metrics and runtime application performance data. For example, we are currently integrating DAQV-II with TAU, Tuning and Analysis Utilities for C++ [1]. Our initial work [25] is focusing on providing client access to TAU's performance callstack, an efficient performance view of a program's execution profile at runtime. This profile shows where time is being spent in a program's code for each thread of execution. The client needs to display this data with respect to individual threads as well as the entire application. The DAQV-II framework will provide access to this data and support the interaction between the display client and the executing program.

6 Conclusion

In summary, DAQV-II is an example of a distributed array interaction framework. In our work, we have experimented with different language targets, models of interaction, and data access abstractions. Our current design and implementation reflects the lessons we have learned through this process as well as our focus on addressing the evolving requirements of high-performance computational science applications. The development of DAQV-II is an ongoing research challenge. Support for data mutation, streaming and parallel data transfers, and optimized distribution mapping algorithms top our list for future work. Through the construction of higher-level domain-specific environments that are based on DAQV-II, we hope to establish its generality and efficacy for providing a range of interactive capabilities with parallel and distributed programs and data. Additional information and documentation on DAQV-II can be found on our web site: <http://www.cs.uoregon.edu/research/paraducks/>.

Acknowledgments

The authors would like to thank Chad Busche (Dept. of Computer and Information Science, Univ. of Oregon) for his substantial work on the implementation of DAQV-II and Laura Magde (Dept. of Geological Sciences, Univ. of Oregon) for providing the visualizations in Figure 4.

References

[1] D. Brown, S. Hackstadt, A. Malony, and B. Mohr, *Program Analysis Environments for Parallel Language Systems: The TAU Environment*, Proc. Workshop on Environments and Tools For Parallel Scientific Computing, SIAM, 1994, pp. 162-171.

[2] Computational Science Institute, Univ. of Oregon, <http://www.csi.uoregon.edu>, January 1998.

[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. Jour. of Supercomputing Applications and High Performance Computing, Vol. 11, No. 3, Fall 1997.

[4] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, *Integrated PVM Framework Supports Heterogeneous Network Computing*, Computer in Physics, Vol. 7, No. 2, April 1993, pp. 166-175.

[5] 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.

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

[7] G. Geist II, J. Kohl, and P. Papadopoulos, *CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications*, Intl. Jour. of Supercomputer Applications and High Performance Computing, Vol. 11, No. 3, 1997, pp. 224-235.

[8] S. Hackstadt and A. Malony, *Distributed Array Query and Visualization for High Performance Fortran*, Proceedings of Euro-Par '96, Lyon, France, August 1996, pp. 55-63.

[9] S. Hackstadt and A. Malony, *DAQV: Distributed Array Query and Visualization Framework*, Jour. of Theoretical Computer Science, special issue on Parallel Computing, Vol. 196, No. 1-2, April 1998, pp. 289-317; also in Proc. Euro-Par '96, Lyon, France, 1996, published as Lecture Notes in Computer Science 1123, L. Bouge, P. Fraiginaud, A. Mignotte, and Y. Robert (eds.), Vol. 1, Springer-Verlag, New York, 1996, pp. 55-63.

[10] R. Haimes, *pV3: A Distributed System for Large-Scale Unsteady CFD Visualization*, Proc. American Institute of Aeronautics and Astronautics, 1994.

[11] C. Harrop and R. Dunn, *Tomographic Imaging Environment for Ridge Research and Analysis (TIERRA)*, Computational Science Institute, Univ. of Oregon, <http://www.csi.uoregon.edu/>, January 1998.

[12] C. Harrop, S. Hackstadt, J. Cuny, A. Malony, and L. Magde, *Supporting Runtime Tool Interaction for Parallel Simulations*, extended abstract, submitted to Supercomputing '98, Orlando, FL, November 1998.

[13] High Performance Fortran Forum, *High Performance Fortran Language Specification, Ver. 1.0*, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice Univ., 1993.

[14] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber, *VASE: The Visualization and Application Steering Environment*, Proc. Supercomputing '93, 1993, pp. 560-569.

[15] A. Karp and M. Hao, *Ad Hoc Visualization of Distributed Arrays*, Technical Report HPL-93-72, Hewlett-Packard, 1993.

[16] A. Kempkes, *Visualization of Multidimensional Distributed Arrays*, Master's Thesis, Research Center Juelich, Germany, 1996.

[17] D. Kimelman, P. Mittal, E. Schonberg, P. Sweeney, K. Wang, D. Zernik, *Visualizing the Execution of High Performance Fortran (HPF) Programs*, Proc. 9th Intl. Parallel Processing Symposium (IPPS), IEEE, 1995, pp. 750-757.

[18] R. Knox, V. Kalb, and E. Levine, A *Problem-Solving Workbench for Interaction Simulation of Ecosystems*, IEEE Computational Science and Engineering, Vol. 4, No. 3, IEEE Computer Society Press. Los Alimitos, CA, 1997, pp. 52-60.

[19] R. Koppler, S. Grabner, and J. Volkert, *Visualization of Distributed Data Structures for HPF-like Languages*, Scientific Programming, special issue High Performance Fortran Comes of Age, Vol. 6, No. 1, 1997, pp. 115-126.

[20] A. Malony, *Developing the Distributed Array Query and Visualization (DAQV) Framework for Domain-Specific Computational Science Environments*, Accelerated Strategic Computing Initiative (ASCI) Level 3 Grant, Univ. of California, Los Alamos Natl. Lab., Contract C70660017-3L, 1997.

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

[22] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, University of Tennessee, Knoxville, TN, 1997. Available at <http://www.mpi-forum.org/>.

[23] Parallel Tools Consortium, <http://www.ptools.org/>, January 1998.

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

[25] S. Shende, S. Hackstadt, and A. Malony, *Dynamic Performance Callstack Sampling: Merging TAU and DAQV-II*, to appear in Proc. of the Fourth International Workshop on Applied Parallel Computing (PARA98), June 14-17, 1998, Umeå, Sweden.

[26] A. Tuchman, D. Jablonowski, G. Cybenko, *Runtime Visualization of Program Data*, Proc. Visualization '91, IEEE, 1991, pp. 255-261.

[27] R. Wolski, N. Spring, and C. Peterson, *Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service*, Proc. Supercomputing '97, San Jose, CA, November 1997.