ELSEVIER Theoretical Computer Science 196 (1998) 289–317

Theoretical Computer Science

DAQV: Distributed Array Query and Visualization Framework

Steven T. Hackstadt*, Allen D. Malony*

Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA

Abstract

This paper describes the design and implementation of the distributed array query and visualization (DAQV) framework, a project sponsored by the Parallel Tools Consortium. DAQV's reference implementation targets High Performance Fortran (HPF) and leverages the HPF language, compiler, and runtime system to address the general problem of providing high-level access to distributed data structures. DAQV supports a framework in which visualization and analysis clients connect to a distributed array server (i.e., the HPF application with DAQV control) for program-level access to array values. Implementing key components of DAQV in the parallel language (e.g., HPF) itself has led to a robust solution in which clients do not need to know how data is distributed. The framework is highly portable to different computing environments as well as other languages where distributed data is involved. © 1998—Elsevier Science B.V. All rights reserved

Keywords: Distributed data; Fortran; Visualization; Parallel tool; Client/server tool framework

1. Introduction

In recent years, parallel processing has evolved from a rather esoteric technology for achieving high-performance computing on high-end, expensive machines to a more mainstream and wider-spread technology for delivering parallel computing capability (not necessarily for high performance) across a range of machine platforms. In part, this evolution has been the result of more powerful microprocessors offering performance levels where speedups from modest parallelism provide acceptable computational capability, e.g., 200 Mflops performance on current generation four processor servers is not unrealistic. This evolution has also been a result of improved systems infrastructure for composing parallel computing environments, e.g., PVM and MPI allow networks of workstations to act as virtual parallel machines. The impact of such hardware and systems infrastructure (in general, the filtering down of parallel systems technology to workstations environments) is to make parallel computing more available.

* Corresponding authors. E-mail: hacks@cs.uoregon.edu., malony@cs.uoregon.edu.

Unfortunately, availability does not imply ease of use. Hence, there has been an increased emphasis on parallel programming environments, including parallel language systems and tools for performance analysis, debugging, and visualization. Research work in these areas has had its share of successes and failures. We argue that there are two reasons for the failures. First, parallel programming tools are often designed without the needs of users in mind [7]. Tools can be complex and hard to understand because of the intricacies of the parallel system or, sometimes, in spite of it. That is, although a tool may be solving a difficult parallel analysis problem, its utility will ultimately depend on how well it can be applied by the user to a specific problem. The second reason comes from a perspective of parallel computing's role in the scientist's problem-solving environment. In the past, computation's dominance in time and cost requirements has, we suggest, allowed tool designers to adopt a parallel system-centered view. However, the scientist's environment has always been more heterogeneous, combining data gathering (static and dynamic), data analysis (manual and computational), data management (large and complex, real time and archival), and data visualization (still and animated, composed and real time). Many parallel tools, though, focus only on the analysis of the parallel computation (e.g., for performance and debugging). The point is that as parallel computing technology evolves (e.g., faster processors), so might the nature of the problem-solving bottlenecks (e.g., from computationally intensive to data management or visualization intensive). Tool technology must be able to follow the changing focus and target the prevailing needs.

It is our contention, then, that parallel systems are evolving to operate as components of a larger heterogeneous environment supporting scientific problem solving, and that tools should then be designed with the broader requirements of the environment in mind. This implies that issues such as tool integration, distributed operation, and portability must be dealt with up-front, promoting designs based on open interfaces, client/servers models, standard data formats, and commonly accepted APIs. This can be easier said than done. In particular, tools that must function as part of the parallel computation (e.g., parallel debugging tools) are non-trivial to implement; the additional requirements of distributed operation and environment-wide integration do not facilitate matters. However, one advantage of taking a heterogeneous (rather than centrist) view towards tool design is that technologies accommodating that view can be utilized in tool implementation (e.g., MPI for data communication). This affords the possibility of high-level problem-solving tools built upon common environment infrastructure used to abstract system-specific details.

In this paper, we describe a framework for distributed array query and visualization (DAQV) in HPF programs. One of the unique features of DAQV (pronounced like "dave") is that it was designed with the philosophy discussed above. The system was originally a tool project effort under the auspices of the Parallel Tools (Ptools) Consortium [24], and as such, its proposed functionality was subject to critical reviews by general consortium members, especially users. For example, the decision to target HPF was a by-product of the evaluation process. The motivating concerns for DAQV forced a unique design and implementation strategy to provide the high-level heterogeneous operations that we envision. The result is a portable framework that can interoperate with other tools via open interfaces, and that can be extended by users as their array query and visualization needs require. The next section discusses the motivation behind the DAQV project. We then present related work and contrast it with DAQV's functionality and implementation. Then, after discussing DAQV's design, functionality, and implementation, we give examples of its application and portability. We end with a discussion of future work.

2. Motivation

During the execution of a parallel program it is often necessary, for various reasons, for the user to inspect the values of parallel data structures that are decomposed across the processing nodes of a parallel machine. For correctness debugging in a sequential environment, the ability to observe program variables to determine coding errors is a common user requirement; debugging with parallel data objects is no different. Similarly, for interacting with parallel applications (e.g., for computational steering), it may be necessary to access and analyze distributed data at runtime. Also, performance data is typically collected in a distributed manner on each processing node, requiring some mechanism to retrieve the information if it is important for the user to do so during execution.

Any tool that addresses a particular user need (whether it be debugging, performance analysis, or application interaction) where distributed data access capabilities are required has either to implement these capabilities itself or rely on some other (open) infrastructure to provide them. In the former case, there are two problems. First, tool implementations tend to become specific to low-level system details. The user-level functionality of the tool may be provided, but platform dependencies may severely hinder its portability. Second, the tool implements only what is necessary for its purposes and abstractions of distributed data access tend not to be developed. This has the effects of limiting tool extension, restricting the ability of other tools to apply the distributed data access support, and leading to inconsistent functionality across tools. The common anecdote from users that "print" is the only good tool is indicative of these two problems: tools for parallel systems are often complex, not integrated, and vary widely across machines. "Print" is the only thing that they can rely on.

The case of relying on some other infrastructure for distributed data access suffers because no such infrastructure currently exists. One might think that "print" provides suitable functionality. But the problem is really the level of the tool's interface to the infrastructure; "print" would require significant scaffolding to give it a high-level interface to an external tool. Similarly, many parallel debuggers are not always aware of how the parallel data structures are distributed. Whereas they provide low-level mechanisms to get the data, the higher-level semantics about what the data is (i.e., the type and characteristics of the distributed data object) are generally not utilized in the infrastructure and are not apparent in its external interface. However, it is exactly these semantics that are helpful in developing open frameworks and portable tools.

The DAQV project highlights the dilemma between the two alternatives above that often face tool designers. Begun as a project sponsored by the Parallel Tools Consortium, DAQV set out to address the user requirement of being able to access and visualize distributed array data coming from a parallel program. Initially, it placed more emphasis on the environment for array interaction (query specifications, array operations, visualization types, etc.) than on the underlying infrastructure for interfacing with the running program. However, based on feedback from the Ptools community [24], it quickly became clear that DAQV's mission was too broad. The most interesting feedback was from users who helped refine the focus of the project towards creating a robust and well-defined infrastructure rather than a multitude of user-interface features (as tool designers tend to do). Users made it clear that the most important contribution that the project could make was to provide a technology that lets them "get the data" (a high-level "print" capability, if you will) and use other tools to "work with the data". The goal, then, was to develop interfaces for both (1) low-level extraction of data from the program, and (2) the higher-level request/delivery of data to an external client (e.g., for visualization). Users also acknowledged the importance of HPF as a potential execution target for the DAQV project.

The result is the DAQV tool as described in this paper. We present this background because we feel that the Ptools project evaluation process was instrumental in forcing the design to seriously consider user's needs. This gave DAQV its focus, but how DAQV is built is what makes it unique. DAQV is characteristic not of a tool necessarily, but of a framework that facilitates the development of software for a particular problem area. A *framework* may generally consist of a set of components with well-defined functions and relationships for constructing the software; templates that can be used to customize functionality or create new components, and user and system interface definitions and modules that allow the developed software to interoperate with other system tools. A framework may also raise the level of developer (user) interaction to hide implementational or operational details, while providing robust backend infrastructure support for targeting multiple platforms. The DAQV framework has all of these features, embodying the requirements of the distributed array access problem in its design and function while keeping flexible and open how the resultant tool is created for a target system. In this way, the DAQV framework truly supports higher-level semantics of data access, and allows extensions both in tool functionality and enhanced data analysis capabilities.

3. Related work

We view the DAQV work as a confluence of research ideas that have come from the fields of parallel programming languages, parallel tools, scientific visualization, and distributed computing over the last five years. Indeed, in some sense, DAQV is an amalgamation of these ideas in a form that is tempered by strong user requirements and is targeted to the HPF language domain. Below, we review the related research from four perspectives of DAQV: language-level parallel tools, distributed data visualization, client–server tool models, and program interaction.

There has been a strong advocation in the last few years for parallel tools to be more integrated in parallel language systems and to support the language semantics in their operation. Because a language system abstracts the parallel execution model to hide low-level system operation, it is important for users of program analysis, debugging, and performance tools to be able to work with program objects at the language level as they seek to understand program structure, behavior, and performance. For instance, the research work integrating Pablo with the Fortran D compiler [1] and Tau with the pC++ compiler [4] demonstrates the importance of providing a high-level semantic context. This is also clearly seen in the Prism [28] environment for the Connection Machine systems which is, perhaps, the best example of the ease of use that can come from an integrated tool system. DAQV clearly follows in this spirit as it bases its entire functionality and operation on the data distribution semantics of the HPF language model. But DAQV goes one step further and actually uses the language system itself for part of its implementation. This was also a feature of Breezy, a forerunner of DAQV that provided high-level program interaction for the pC++ system [5].

One of the key programming abstractions found in parallel language systems is data parallelism – the parallel operation on data that has been distributed across the processing nodes of a machine. Because distribution of parallel data is an important factor in the performance behavior of a program, viewing the data and performance information in relation to the distribution aids the user in tuning endeavors. The GDDT tool [21] provides a static depiction of how parallel array data gets allocated on processors under different distribution methods and also supports an external interface by which runtime information can be collected. In the DAVis tool [18], distribution visualization is combined with dynamic data visualization to understand the effects of decomposition on algorithm operation. Kimelman et al. show how a variety of runtime information can be correlated to data distribution to better visualize the execution of HPF programs [19]. Similarly, DAQV could provide distribution information to clients for augmenting views of data structure and operation. But there is another use of distribution information, and that is to reconstruct the partitioned data into its logical shape. The IVD tool [17] uses a data distribution specification provided by the user to reconstruct a distributed data array that has been saved in partitioned form. In DAQV's case, this reconstruction is done, in essence, by the compiled HPF code using the implicit distribution information passed to the array access function.

The increasing importance of portability and extensibility in parallel tools has evoked designs following client/server models. The Panorama debugger [22] demonstrates how the concept of interoperating modules can lead to increased functionality and generality in debugging systems. The p2d2 debugger [6] extends this concept considerably in proposing a full client/server debugging framework with comprehensive abstractions of operating system, language, library interfaces, and protocols for distributed object interaction. DAQV is clearly adopting the client/server approach for similar reasons, but in contrast to these two particular tools, the functionality is at a higher level, affording the possibility of layering DAQV on top of systems like Panorama and p2d2.

The final perspective is one of dynamic program interaction. There has been a growing interest in runtime visualization of parallel program and computational steering. Implementing such support raises interesting systems implementation issues as well as user issues. On the one hand, runtime visualization for a particular application domain might be able to utilize domain knowledge to implement a system meeting certain performance constraints. The pV3 system [12] is a good example. However, such specialized implementations may be limited when considering the general runtime visualization problem. DAQV, in many respects, is a direct descendant of the Vista research [29] since it embodies many of the same design goals: client/server operation, automated data access support, runtime operation, structured interaction. The improvement DAQV offers is in language-level implementation to increase portability. We believe that this will also improve DAQV's ability for computational steering. Although Vista was extended for interactive steering in the VASE tool [9], the steering operations were still very much dependent on the target implementation. Supporting steering at the application language level is a more robust, general solution and is something we have begun to investigate in DAQV.

4. Design

DAQV attempts to address the general problem of providing high-level access to parallel, distributed arrays for the purpose of visualization and analysis. It does this by "exposing" the distributed data structures of a parallel program to external tools via interfaces that obviate the need to know about data decompositions, symbol tables, or the number of processors involved. 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 logical structures. To this end, DAQV has three primary design objectives:

- A logical, global view of parallel, distributed data.
- A simple, many-to-one client/server model of communication.
- A portable parallel program interaction model.

Given these design objectives, how should they be addressed in the development of a DAQV tool? In one scenario, DAQV could just be a reference design specification that new tools built from scratch for different target platforms would try to incorporate. Although certain user requirements might be captured in formal operational and interface semantics, unanticipated functionality may be difficult to retrofit to the different tool versions and still maintain a consistent design model. Alternatively, a DAQV framework could be defined that provides techniques to accommodate flexibility in functionality, but provide a common infrastructure for platform retargetability. In our tools research, we have found that techniques that support:

- *programmability*: the ability to program a tool's operation to capture domain abstractions and capabilities;
- *extensibility*: the ability to add new functionality to a tool's (or toolset's) repertoire consistent with its operational model; and
- *interoperability*: the ability for tools to work together through well-defined programming and communications interfaces;

are important for building effective tool frameworks. Below, we describe how the design objectives are met in DAQV, as a result of the framework approach.

4.1. Global view of data

In a language such as high-performance fortran (HPF), the programmer views distributed arrays at a global level, which means the programmer may perform operations on whole arrays and refer to array elements with respect to the entire array, as opposed to some local piece on a particular processor. In other words, HPF supports a *global name space* [15, 20]. Clearly, a programmer cannot completely disregard the issue of data distribution, though, and expect the best performance. Distribution directives allow the programmer to use their knowledge about the application and advise the compiler on the best way to distribute data. However, concern for data distribution does not affect how the programmer references the data. HPF syntax insulates the user from ever dealing with an array in a distributed manner, thereby supporting a logical, global view of data. For similar reasons it is important that DAQV supports a logical, global perspective when interacting with the user (through external tools).

4.2. Client/server model

DAQV is a software infrastructure that enables runtime visualization and analysis of distributed arrays. It is not a stand-alone application or tool that performs these tasks itself; rather, it interoperates with either external tools built specifically for use with DAQV, or existing tools that have been retargeted or extended to interact with DAQV. The goal is to incorporate the existing visualization and analysis tools. The feedback we received from the Ptools user group during design discussions was clear: they did not need another fancy, self-contained visualization tool. They just wanted improved facilities for querying and extracting distributed arrays such that their existing tools could be easily used with this system. To this end, we logically view the entire HPF program (not individual processes or processors) as a *distributed array server* to which these external client tools connect and then interact with the program and its data. At issue, though, is how to make HPF's single-program, multiple-data (SPMD) program execution [20] appear as a single, coherent distributed array server, and how to simplify as much as possible the requirements placed on DAQV clients.

4.3. Portability

Another goal in the design of DAQV was to minimize the degree to which DAQV is dependent on a particular HPF compiler. By targeting the HPF language in the first place, machine portability is inherent to the extent that the HPF compiler is. However, portability across HPF compilers is also important. DAQV primarily accomplishes this in three ways: key components of DAQV are implemented in HPF, compiler and runtime systems are utilized, and compiler-dependent code is minimized and isolated.

The first two items in the above list will be discussed in more detail in Section 6.3. Compiler-dependent code is limited to two areas: the yield point procedural interface (see Section 6.1) and the array access argument-passing interface (see Section 6.3). The amount of code that is compiler-specific is small, so isolating it is very easy.

In addition, we claim that the DAQV *model* and *architecture* of parallel program interaction is also portable. This model will be described in more detail in the next section, and the topic of portability will be revisited in Section 7.2.

5. Functionality

This section will attempt to describe, in general terms, the functionality of DAQV. As a framework, DAQV addresses concerns from both tool users and tool developers. Two different operational models, called *push* and *pull*, are supported. These two models differ in the degree of interactivity available with the HPF program at runtime. This section will describe these models, how they differ from one another, and how they address the concerns of users and developers.

5.1. The push model

The *push model* forms the basis of DAQV and constitutes the simplest and least intrusive way to access distributed arrays from an external tool, or *data client*. The push model is implemented by inserting simple DAQV subroutine calls into the HPF source code. These calls allow the programmer to

- select the DAQV model (i.e., push or pull) to be used in the program,
- register distributed arrays with DAQV,
- set parameters for communicating with data clients,
- make DAQV connect with data clients, and
- send the data values of a distributed array to the data client.

The functionality of the push model is the practical solution to the feedback from the Ptools user group. That is, with minimal additions to the HPF source code, users can extract the data values of distributed arrays and visualize them with other tools, never having to worry about array reconstruction. The push model can be used to spot-check the state of an array or to create animations of data values over the iterations of a loop. Multiple arrays can be pushed out of the program to multiple data clients. More details about how the push model is used will be presented in Section 7.1.

5.2. The pull model

The push model is adequate if the programmer knows exactly which arrays they wish to visualize and when they want to view them. However, to support a more interactive and flexible approach to array visualization, DAQV implements the *pull model* which allows program execution to be controlled and arrays to be selected for visualization through an external interface. An example of such an interface that uses a debugger metaphor will be presented in Section 7.1.

The pull model allows the programmer to repeatedly run an HPF code for a period of time and then extract data values from the distributed arrays of interest. Two types of clients are used in the pull model. Data clients process data values from arrays just as they did in the push model. In fact, any data client that works in the push model also works in the pull model because the pull model is layered on top of the push model. In addition, though, the pull model requires a *control client* to direct program execution and to configure and initiate array transfers to data clients.

Both models support open interfaces for data transfer, allowing new data clients to be developed. The pull model also supports an open control interface through which new DAQV control clients can interact with the program. But perhaps more interestingly, DAQV functionality can be incorporated into other tools (e.g., a debugger) through this simple interface.

The primary conceptual difference between the push and pull models is where the decision to extract an array originates. The names "push" and "pull" are meant to reflect this difference in perspective. In the push model, the HPF program itself "pushes" data out, while in the pull model, an external client reaches in and "pulls" data out. However, the implementation of these very different conceptual models is built upon a common infrastructure supported by DAQV. This will be a large part of the discussion of DAQV's implementation.

6. Implementation

Our goal with DAQV is to facilitate simple and useful conceptual models for distributed array collection and extraction, and to implement those models in a high-level, portable manner. The first of these goals (the design and functionality of the push and pull models) has already been discussed in the previous sections. This section explains how we have met the implementation goals. It also discusses the software mechanisms and requirements we have developed for DAQV in four areas: procedural interface, client/server interface, underlying mechanisms, and system requirements.

Table 1
The procedural interface between HPF and DAQV is a small set of subroutines that the user, a preprocessor, or a compiler inserts into the HPF source code

Type	Subroutine	Arguments	Model	Description
Setup	`daqv_mode`	mode	Push Pull	Sets the DAQV operational model (push, pull, or off) to be used by this program
	`daqv_register`	array_id, name, symbol, type, rank, dim1, dim2, ...	Push Pull	Gives DAQV a handle on a distributed array for later access
	`daqv_config_push`	array_id, port, size	Push	Establishes communication parameters between DAQV and a data client for a registered array
Access	`daqv_push`	array_id	Push	Causes DAQV to send the values of a registered array to the appropriate data client
Control	`daqv_yield`		Pull	Causes HPF program to yield execution control to DAQV
	`daqv_pull_enable`	port	Pull	Activates yield points; control client will be notified at next yield point
	`daqv_pull_disable`		Pull	Deactivates yield points; the control client will not be notified at future yield points

6.1. Procedural interface

DAQV's core requirement is to support interaction with HPF programs. In part, this interaction is similar to what might be provided by a HPF debugger. One would want the ability to set breakpoints in the program where distributed array data can be accessed in some manner. However, a HPF debugger may not provide an interface for distributed array query based on HPF semantics, relying instead on low-level support for gathering array data on different processing nodes. This system dependency defeats DAQV's goal of portability, making it too reliant on the target compiler or machine. Instead, we chose to implement key components of DAQV as HPF subroutines, allowing array access and other functions to employ the HPF compiler and runtime system automatically. In this respect, we view DAQV as a language-level tool design and implementation.

The high-level operation of DAQV demands a different method for interacting with the HPF program than what a HPF debugger can provide. Our solution is to implement DAQV as a library that is linked with the HPF object file, creating a procedural interface between the HPF program and the distributed array server component. A small set of seven subroutines, described in Table 1, handles initialization, registration of arrays, configuration of data clients, and data extraction.

Currently, the DAQV interface subroutines are inserted manually by the programmer. If the compiler supports it, DAQV routines could be inserted automatically. For

Table 2
The DAQV event protocol extends functionality for program control, communication configuration, and array extraction to clients

Event	Type	Description
YIELD	Server-to-client	Informs control client that program has yielded
CONTINUE	Client-to-server	Instruct DAQV to let the HPF program continue execution
CONFIG_PULL	Client-to-server	Set data client communication parameters for a specified array
PULL	Client-to-server	Instruct DAQV to send data values from a specified array to the appropriate data client
GET_REG_ARRAYS	Client-to-server	Request a list of registered arrays from DAQV
STATUS	Client-to-server Server-to-client	Send current status and request status from other

example, PGI's HPF compiler, *pghpf*, automatically inserts line- and/or function-level tracing routines into the HPF source code [26]. By linking in the DAQV library in place of the default tracing routines, DAQV uses the calls to these routines as *yield points*, places in the program where distributed data can be accessed.

6.2. *Client/server interface*

DAQV assumes that the HPF implementation exhibits a SPMD execution model with several separate but identical processes each operating on small pieces of larger arrays. Because DAQV is a library, it should respect the execution semantics of the HPF program. However, since DAQV supports a client/server interface it is not possible for all HPF processes to behave exactly the same when the distributed array server is executing; in particular, one process must be identified as the DAQV "manager process" and play the role of the communication server. In general, any operation dealing with communication or data buffering is executed only by the DAQV manager. DAQV provides a compiler-specific process identification macro to determine who is the manager.

Conceptually, the client/server interface supported by DAQV exists between the entire HPF program (i.e., all SPMD processes representing the HPF program) and external data and control clients. From an implementation viewpoint, however, only the DAQV manager process operates as the communications server; the other HPF "worker" processes cooperate with the manager to effect DAQV operations. DAQV uses a bidirectional event protocol (described in Table 2) between the HPF server and the data clients, in both the push and pull models. In the push model, data clients respond to array data transmissions with a confirmation that signals DAQV to let the HPF code continue executing (i.e., the data client's reception of the data is synchronous with program execution). The pull model uses a more sophisticated event protocol to interact with the control client. Events that come into DAQV are received only by the manager HPF process; some mechanism for informing the other HPF processes about events is needed. Section 6.3 will discuss in detail how event sharing is accomplished between the manager and worker processes. We continue here by describing the DAQV protocols and client requirements.

6.2.1. Event protocol

DAQV's event protocol, as described in Table 2, consists of a small set of textual, list-based events that are primarily used to interact with control clients. The `YIELD` event is sent from the server to the control client whenever the HPF program encounters a `daqv_yield()` procedure call. The event contains information about where the program is stopped (e.g., line number, function name) as well as a list of registered arrays and their descriptions. After sending this event, the distributed array server waits for a reply. When the server receives a `CONTINUE` event, execution control is transferred back to the HPF program. But before sending the `CONTINUE` event, the control client can issue several other events: `CONFIG_PULL` instructs DAQV how to set the data client communication parameters for a particular array; `PULL` instructs DAQV to send data from a specified array to the appropriate data client; and `GET_REG_ARRAYS` requests a list of registered arrays from the server.

6.2.2. Data protocol

The data protocol used by DAQV is made up of two components: raw data values from distributed arrays are prefaced by "metadata" which describes the values. In particular, the metadata contains the ID of the array being sent, the textual name of the array, its rank and dimensions, and the data format that the raw values are in. Eventually, the metadata will be extended to provide semantic information about how the data values being sent relate to the original distributed array. Metadata could indicate, for example, if an array was sampled down before being sent, or if the data sent to a client actually represents the difference of two program arrays.

6.2.3. Client requirements

DAQV is an open framework that is intended to work with a variety of different data clients. Efforts have been made to allow existing visualization and analysis tools to be easily ported for use with DAQV. A data client has very few requirements. First, the current implementation uses only sockets for communication. A client must be able to respond over a socket with a simple text-based confirmation event after receiving data. Second, a client must be able to parse the data being sent to it by DAQV. Currently, we have developed three data clients. The first of these clients, Dandy, displays 2D arrays as a color-mapped grid and was easily ported from the pC++ Tau tools [4]. Next, the Viz visualization environment [14] has been extended to support interaction with DAQV. Dandy and Viz will be described in more detail in Section 7.1. The third client functions as a module within the IRIS Explorer visualization environment. The module reads DAQV data from a socket and returns an Explorer "lattice" data structure which can be further manipulated and/or visualized in the environment. The

```
      SUBROUTINE DAQV_PUSH_2D_INT(A,M1,M2)
      INTEGER M1, M2, I, J
      INTEGER A(1:M1,1:M2)
CHPF$ INHERIT A
      INTEGER ITMP(NUMBER_OF_PROCESSORS())
CHPF$ DISTRIBUTE ITMP(CYCLIC)
      DO I = 1,M1
         DO J = 1,M2
            ITMP(1) = A(I,J)
            CALL DAQV_BUFFER_INT(ITMP)
         END DO
      END DO
END SUBROUTINE DAQV_PUSH_2D_INT
```

Fig. 1. An array access function written in HPF for two-dimensional arrays of integers.

use of scientific visualization packages is further discussed in Section 7.3 and examples can be seen in Figs. 7 and 8. Work on other new clients is ongoing.

Control clients must be more sophisticated than data clients in terms of the events that they must handle. They must accept and/or read the events listed in Table 2. A C-library has been developed for supporting event processing; support for other languages (e.g., Tcl/Tk) is under consideration. Furthermore, a control client usually interacts with the user. As part of our work, we have created command-line and graphical control client interfaces (see Section 7.1). More importantly, though, we see the DAQV pull model and the specification of control events as establishing a framework for other tool developers. In particular, it provides a simple interface that can be incorporated into other parallel tools to gain high-level access to distributed data in a flexible and well-defined manner.

6.3. Underlying mechanisms

We have reached a point where a detailed discussion of two critical yet subtle components of DAQV is possible. The majority of the high-level concepts, functionality, and even implementation of DAQV that have been presented thus far, rely on two "mechanisms" implemented in DAQV. These two mechanisms, *array access* and *event sharing*, play a critical role in DAQV's operation. Furthermore, they are the components of the framework that are implemented in HPF itself, allowing DAQV to build on HPF's high-level language system and semantics for array access. Our implementation is innovative and deserves explanation.

6.3.1. Array access functions

Because an HPF compiler can generate code automatically to access distributed data, DAQV can implement its access capabilties in the HPF language itself. Thus, DAQV provides a library of *array access functions* written in HPF that accept arrays of different type and rank. The current library is extendible to allow for new array structures and operations. Fig. 1 shows an example of an access function for a two-dimensional array of integers.

```
   SUBROUTINE DAQV_EVENT_SHARE(CMD_SUM)
   INTEGER CMD_SUM
   INTEGER CMD(NUMBER_OF_PROCESSORS())
CHPF$ DISTRIBUTE CMD(CYCLIC)
   DO I=1,NUMBER_OF_PROCESSORS()
      CMD(I) = CMD_SUM
   END DO
   CMD_SUM = SUM(CMD)
   END SUBROUTINE DAQV_EVENT_SHARE
```

Fig. 2. A simple routine written in HPF for event sharing among HPF processes.

The decision to extract an array can come from two places: the HPF program in push mode (via daqv_push()) or the control client (via a PULL event). Both of these actions ultimately result in the same internal DAQV code being invoked. Currently, DAQV determines the appropriate access function to use based on array rank and element type, though we plan to extend the control client event protocol to allow runtime selection of array access functions. It then creates the calling context for the access routine and invokes it. An array's dimensions are passed in as arguments, and the subroutine inherits the distribution for the incoming array. Compiler-specific optimizations can be built into the access function. In Fig. 1, e.g., the array ITMP is used to reduce a data broadcast to a point-to-point communication on each iteration.

6.3.2. *Event sharing*

While DAQV presents the appearance of a unified distributed array server to external clients, the manager process is really the one performing the communication. From a SPMD control point of view, this raises an interesting question: what are the other worker processes doing during this time? An example should clarify this.

Consider the pull model, and suppose each HPF process has encountered a daqv_yield() call. Each process separately transfers control to the DAQV portion of its executable. The manager process immediately notifies the control client that the HPF program has yielded to DAQV and then waits for a reply. Meanwhile, the other processes determine that they are indeed workers and enter into an event sharing loop (to be explained). Eventually, the manager receives a request from the control client, say, to pull an array. However, since all HPF processes must participate in the call to the array access function (since it is implemented in HPF), the other processes must be notified of this request and make the "same call" as the manager to the access function (as required by HPF's SPMD execution semantics). This is done with a control mechanism called *event sharing.*

To accomplish this, each worker process calls the daqv_event_share() routine shown in Fig. 2 to get the event information from the manager. This allows them to update their DAQV event state to be consistent with the manager's and to function the same as the manager does based on the events received from the control client. As seen, DAQV implements this routine using the HPF language.

Each worker process enters this routine with the value of CMD_SUM set to zero (note that CMD_SUM is passed by reference); the manager process, however, calls the routine with an integer value (representing an event code, parameter, or other information) that is to be shared with the other processes. The routine distributes one element of the CMD array to each process (workers and manager) on each processor. Each process initializes its element of the array to the value with which it entered the procedure. The global sum reduction is performed, and the result is assigned back to CMD_SUM (as a procedure side-effect). Because only the manager's initial CMD_SUM value is non-zero, the new value of CMD_SUM (in every process) at the end of the reduction is this value, as we want. Now each process can return to the DAQV code to perform together whatever operation is required based on the value of CMD_SUM.

This is how event sharing is implemented in DAQV to remain consistent with HPF execution semantics. It is probably not the obvious solution when first faced with the problem of sharing information among several processes, but in the context of DAQV, it is both an elegant and very portable one.

6.4. *System requirements*

With a primary goal being portability, DAQV does not require any modifications to a HPF compiler, however certain assumptions have been made in the DAQV reference implementation. Currently, DAQV assumes that HPF data parallelism is achieved with a SPMD execution model. (In many respects this is a more difficult problem when compared to, say, a multithreaded implementation.) An HPF program must be able to invoke C routines, and C routines so invoked must be able to call back to HPF. The process of array registration requires knowledge about the transformations applied to function and subroutine arguments by the parallel language compiler. However, the information required is minimal and so far has not required vendors to divulge proprietary information. DAQV can optionally take advantage of compiler-specific tracing/profiling support, though this is not required by the reference implementation. Finally, in two instances (array access functions and event sharing), DAQV requires HPF distribution directives to be carried out by the compiler, i.e., DAQV will not work if the directives are ignored. With these minimal requirements and assumptions, DAQV is able to achieve a high degree of portability across machines and compilers, as well as other languages.

7. Results and evaluation

Ultimately, the efficacy of tools for parallel scientific computing must be evaluated with respect to how well they aid the problem solving process. Clearly, our intent with DAQV is to produce a tool that addresses a user requirement for parallel program interaction with distributed data structures and, thus, benefits the problem solving application where this requirement is present. However, as a framework, DAQV also seeks to improve how a DAQV tool instance is created, including how it can be ported to different problem-solving platforms, how it can be extended to add new capabilities, and how it can interoperate with other tool components. Evaluation of these goals should also be based on experiences. In this section, we demonstrate through real examples how the DAQV framework has been instrumental in producing effective tools for scientific applications, how it has allowed the DAQV model to be ported to different computational platforms, and how it has enabled the inclusion of client tools in the problem solving environment.

7.1. Applications

One means of validating the DAQV framework, as proposed above, is to evaluate its use in specific application instances. In this section, we first explain a simple application to illustrate how DAQV concepts are used. Then, we describe how DAQV has been applied to a larger, scientific application.

7.1.1. Laplace heat equation

To illustrate how to use the current implementation of DAQV, we will consider a HPF program that implements a finite-difference method for solving the Laplace heat equation iteratively [13]. Once begun, the code executes a five-point stencil operation on the sample surface until a steady state is reached to within some tolerance. DAQV is used in push mode to visualize the heat flow through the two-dimensional surface at each iteration of the main loop.

The sequence of images in Fig. 3 was generated by the Dandy data client (Section 6.2). The displays show the progression toward a steady state in the two-dimensional array representing the surface to which a uniform heat source has been applied. These displays were created using DAQV's push model. Under this scheme, when the program is executed and reaches the call to DAQV's `daqv_config_push()` routine, the Dandy client connects to the HPF/DAQV program. At this point, execution resumes and the animation of the data values begins. At each loop iteration, the new array values are sent to Dandy each time `daqv_push()` is called. The Dandy interface (not shown in Fig. 3) allows the user to pause/resume the animation and redraw the display. Dandy automatically determines the range of the data values and maps them onto a fixed colormap. However, if the viewer wishes to fix the color range across several iterations, the automatic scaling feature can be disabled, allowing minimum and maximum values to be set manually. This feature can also be used to identify outliers or to identify values beyond a certain threshold. For example, Fig. 3(b) shows values above the specified range as white. As the algorithm nears convergence, these values move into the fixed data range, as seen in Fig. 3(c).

The same application can also be used with DAQV's pull model. Fig. 4 shows several windows from a DAQV session. A prototype control client interface (lower, in back) supports several DAQV features. A source code browser allows the viewer to control program execution by simply double-clicking on the line to which the code should run next. Alternately, the user may use the controls in the upper left portion of the control client to specify a number of yield points (steps) to skip before reporting back.

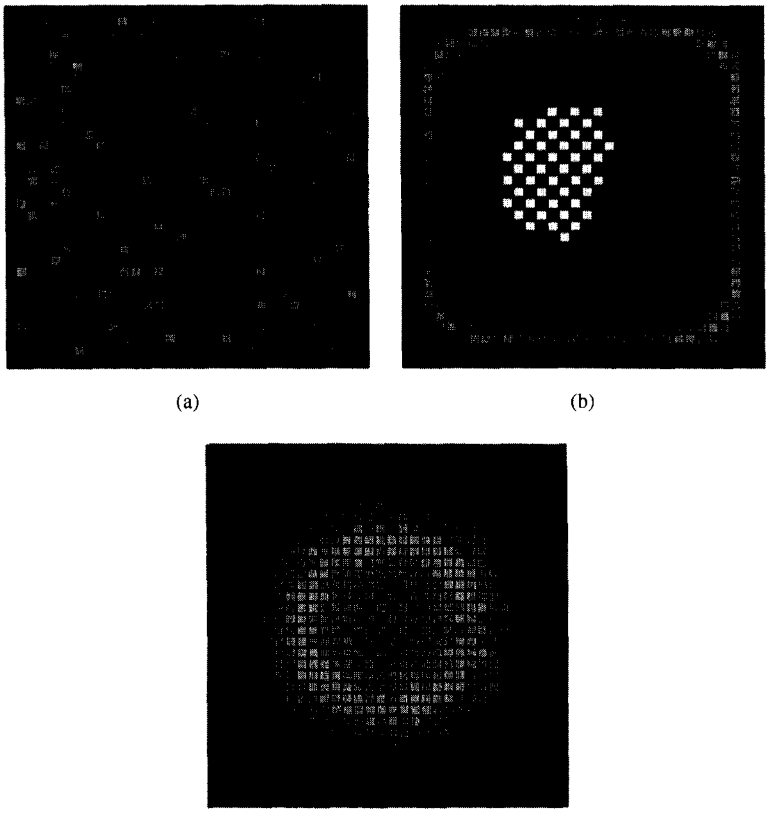
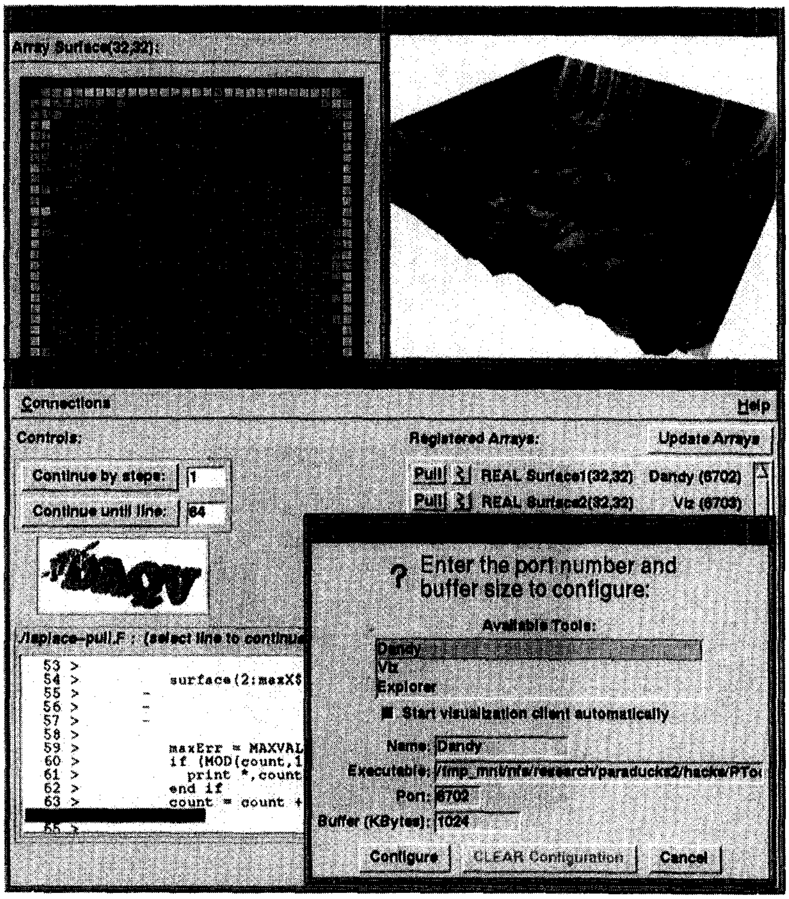


Fig. 3. Convergence of the finite method for solving the Laplace Heat Equation can be seen by visualizing the two-dimensional array representing the heated surface.

In the upper right portion of the control client (partially obscured) is a list of arrays in the program that have been registered with DAQV. In this case, `Surface`, a 32×32 real-valued array, has been registered twice so that it may be viewed with two different data clients. The button immediately to the left of each registered array entry allows the user to select a data client for visualizing that array. The pull configuration dialog (lower right) presents a list of preconfigured tools (specifiable in a resource file) in a list box, but the user may also provide their own parameters explicitly. Once an array is configured, the "Pull" button next to its entry in the registered arrays list is enabled. This button causes a PULL event to be sent to the server and makes DAQV send the requested array to the appropriate data client. The first registered array, `Surface1`, has been mapped to the Dandy client in the upper left part of the screen. Similarly, `Surface2` is being sent to a more complex, three-dimensional display built in the Viz visualization environment builder [14].



Continue by steps: 1
Continue until line: 64

Pull	REAL Surface1(32,32)	Dandy (6702)
Pull	REAL Surface2(32,32)	Viz (6703)

./laplace-pull.F : (select line to continu

```
          surface(2:maxX$
     -
     -
     -

          maxErr = MAXVAL
          if (MOD(count,1
            print *,count
          end if
          count = count +
```

Enter the port number and buffer size to configure:

Available Tools:
Dandy
Viz
Explorer

■ Start visualization client automatically

Name: Dandy
Executable: /tmp_mnt/nfs/research/paraducks2/hacks/PTo
Port: 6702
Buffer (KBytes): 1024

Fig. 4. A DAQV session with a control client. A simple two-dimensional data client and a three-dimensional data client both represent the 2D array from the Laplace Heat Equation code.

7.1.2. *Seismic tomography*

As part of a recent effort to build a domain-specific environment for seismic tomography [8], the DAQV framework was used to both extract and modify simulation data and control the program at runtime. As an integral part of the interactive steering and visualization environment, DAQV provided access to the data needed by marine seismologists studying the formation and structure of volcanic mid-ocean ridges.

Seismic tomography is used to construct 3D images from seismic wave information recorded by sea-floor seismometers. The technique, similar to that used in medical CAT scans, uses recorded arrival times of acoustic energy from explosive sources or earthquakes. Energy from the sources propagates downward into the Earth and is then refracted back to the surface by the positive velocity gradient of the crust and mantle. The travel time of a seismic wave from its source to a receiver depends on the velocities sampled by the wave along its path. The computation of velocity structure from travel times is achieved by a search of the parameter space, with each evaluation requiring first a forward propagation of ray paths and then an inverse calculation.

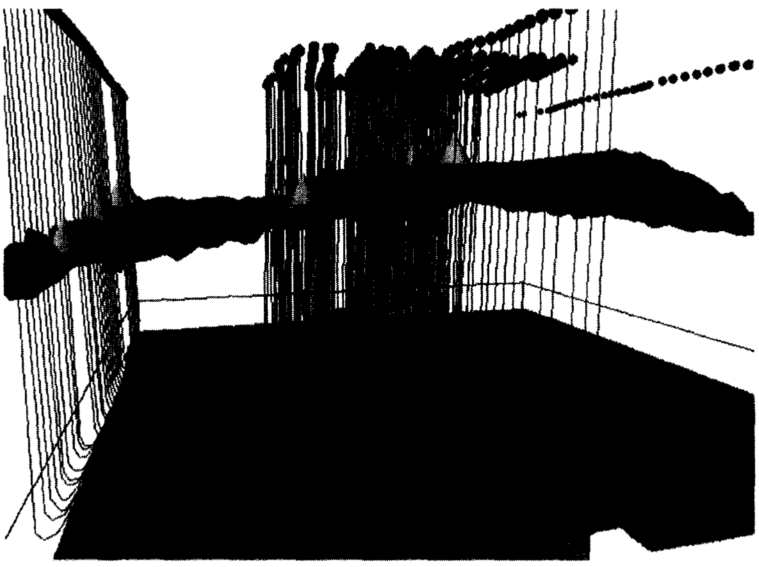
The basic computation as described, however, is only part of a much larger analysis process that can take several months. The full analysis requires extensive, domain-specific input from a geoscientist. In a sense, the scientist "steers" the generation of the final model, examining intermediate results to check the validity of the inputs, establishing parameter values, optimizing the calculation, testing hypotheses, and determining the robustness of the final model. The goal of our environment was to support this total process.

To create appropriate support tools, we broke the overall problem-solving process for seismic tomography into a series of seven steps and generated a list of domain-specific requirements for each of those steps. The primary requirement was a major improvement in performance. This was initially achieved by a trivial parallelization of the algorithm and execution on an 8-processor SGI Power Challenge. Our analysis also yielded two requirements which DAQV addressed or helped to address: (1) interactive, on-line visualization; and (2) computational steering. (For more details on each step and its associated requirements, see [8].)

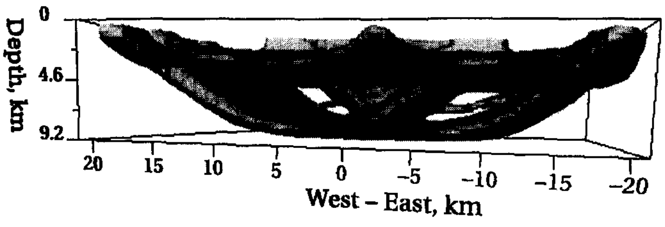
Interestingly enough, the need for these capabilities was as much a result of the performance improvements gained by the parallelization of the seismology application as it was the scientists desire for a more robust and flexible analysis environment. In the original environment, a "run" of the application consisted of a set of iterations that generated a single model refinement. At the end of the run, model data was dumped to a file, reformatted, and imported into a basic 2D visualization tool for analysis. The seismologists adjusted the constraint parameters of the model for the next run. When new parameters were determined, the application was re-executed. This process was acceptable to the scientists because of the long computation time (about 2 hours); on-line analysis would result in significant waiting time for the scientists. However, in the parallelized version, only 6 minutes were required to generate a new model. Suddenly, a tighter coupling between model generation and model evaluation was possible (and desired) – a coupling that did not require all the separate steps of saving, reformatting, importing, and analyzing the data, and then restarting the application. This shift in the computational analysis bottleneck from model generation to model evaluation demanded that the evaluation phase be better supported by the environment. That is, a solution to "close the loop" of the evaluation phase was needed.

The keys to achieving this goal were (1) avoiding writing model data to files and then reformatting it before visualization and analysis, (2) allowing the application to remain in an executing state while model evaluation occurs, and (3) allowing new model parameters to be communicated back to the (still) executing program for the next set of iterations. The DAQV framework was well-suited to these needs. Note, however, that the third item listed above is not something directly supported by the DAQV framework previously described.

DAQV's simple procedural interface made it easy for the seismologists to access model data at runtime and pass it to cooperating tools for analysis. Much of the



(a)

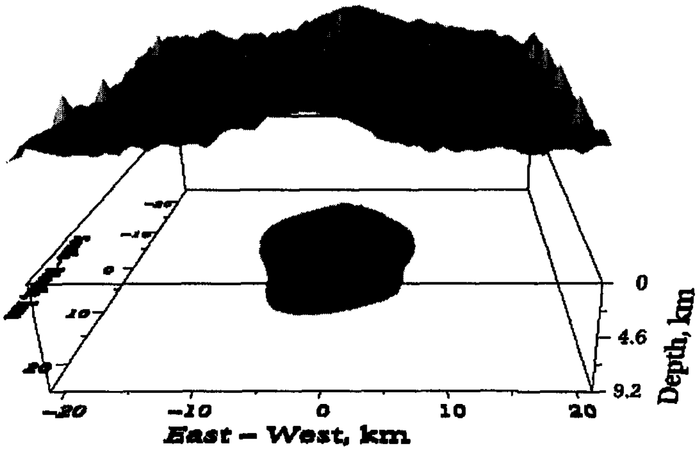


(b)

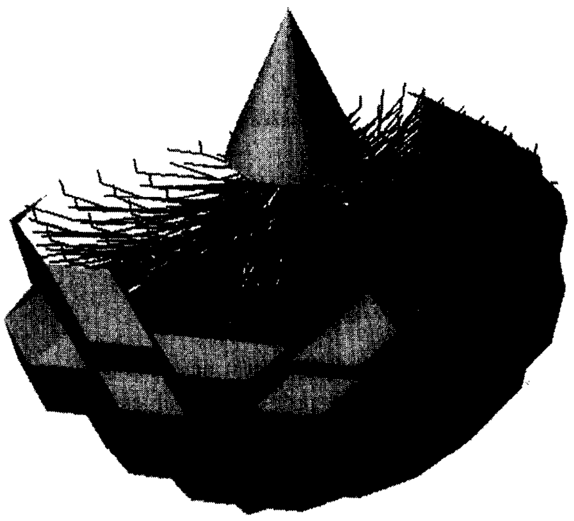
Fig. 5. Example visualizations from the seismic tomography application environment made possible by DAQV; (a) allows carved regions of the velocity model to be compared to the individual ray paths; (b) an isosurface reveals undersampled regions of the forward space model; (c) reveals a previously undiscovered magma chamber beneath the rise; and (d) shows the computation of the travel time wavefront from a seismic receiver.

model analysis was supported by sophisticated 3D visualizations created in the Viz environment [14] mentioned earlier. Viz supports rapid visualization prototyping in a system that can be extended with application-specific visualization abstractions. Viz was used to build a visualization environment for the seismic tomography application that acted as both the control client and several data clients. Any data that Viz used to create visualizations was obtained or simply delivered through the DAQV request and delivery protocols. Some examples of the visualizations made possible by DAQV are shown in Fig. 5, while elements of the control and data client user interfaces are shown in Fig. 6.

Getting data out of the seismic tomography application only closed the model evaluation loop halfway; there was still a need to communicate the new parameter values



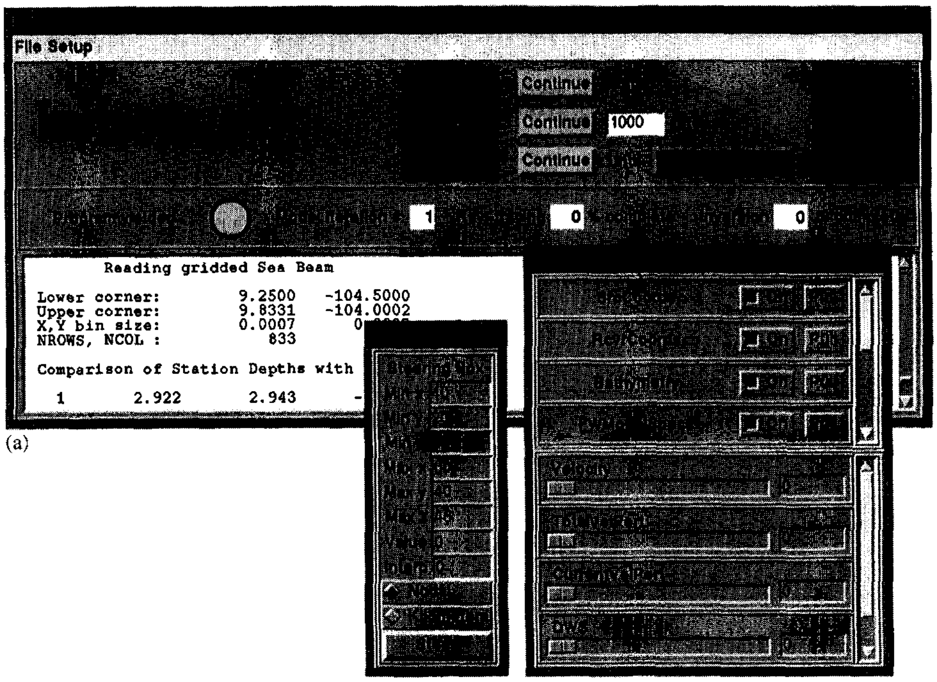
(c)



(d)

Fig. 5. (Continued)

back to the running program for the next set of iterations. These values are derived by the seismologist and the interactive visualizations provided by Viz. What was needed was a way to send values back to the application, preferably by the same mechanisms that were used to get data in the first place (i.e., the DAQV event interface). The extensible nature of DAQV's framework made it very simple to incorporate the ability to alter HPF program variables based on events received from the external control client. So, at the end of the evaluation phase and when new constraint parameters were



```
        Reading gridded Sea Beam

Lower corner:            9.2500    -104.5000
Upper corner:            9.8331    -104.0002
X,Y bin size:            0.0007        0
NROWS, NCOL :               833

Comparison of Station Depths with

  1         2.922         2.943       -
```

Fig. 6. User interfaces of DAQV control, data, and steering clients for the seismic tomography application. The main control panel (a) is used to start, stop, and continue execution of the program. The steering window (b) allows the scientist to adjust the model constraint parameters. The data client window (c) is used to control the visualization of program data.

established, the Viz control client simply sent a new DAQV "steer" event to the server to effect the steering operation on the program variables.

Similarly, while the DAQV model proposes the use of just one instrumentation model (push or pull) in a single execution, the seismographers desired access to both instrumentation models during a single program execution. That is, there were certain HPF arrays that the programmers wanted to send out on a regular basis (push model), and there were some that they only wanted under certain circumstances (pull model). The DAQV framework was easily adapted to accommodate this requirement as well.

Using DAQV in this manner, the graphical user interfaces of the control client (Fig. 6) allow the seismologist to specify input data, constraint parameters, and the run-time options needed to launch, start, stop, and continue the execution of the program. The Viz-based data clients let the seismologist turn on and off specific visualization features to observe the model space, ray path calculations, and travel-time fields. In this way, the DAQV framework facilitates a tight coupling between the executing application and those analytical processes of the seismologists that rely on parallel program interaction and data visualization.

7.2. Framework portability

As the primary derivative of a Ptools project, the DAQV reference implementation is required to function on two platforms. However, the definition of "platform" as it pertains to DAQV is somewhat ambiguous. For most projects, a platform refers to a specific brand of machine. For DAQV, though, that level of portability is trivially achieved since we code to the HPF language, not the specific HPF compiler. Thus, e.g., in the case of DAQV using *pghpf*, we met the Ptools portability requirement simply by demonstrating DAQV running on an SGI Power Challenge (or workstation) and a Sun workstation. Instead, for DAQV we identified three degrees of portability: machine, compiler, and language. *Machine portability* assumes use of a given compiler for a given language and refers to DAQV's ability to be used on different computer hardware; *compiler portability* assumes only a given language and refers to DAQV's ability to be used with different compilation systems; and *language portability* refers to DAQV's applicability to other language or programming model targets. As discussed earlier, we view portability as a means of validating our notion of a framework. We have successfully demonstrated DAQV's portability in all three degrees.

7.2.1. Machine portability

DAQV was developed using The Portland Group's *pghpf* compiler [25] on SGI workstations and Power Challenges. With no changes, DAQV will also work on Solaris workstations. With only minor modifications, DAQV was recently ported by the Cornell Theory Center to run with *pghpf* on an IBM SP2. Initial tests on porting DAQV to *pghpf* on a Cray T3D also look very promising.

7.2.2. Compiler portability

DAQV has been ported with varying degrees of success to two other HPF compilers. First, Cornell Theory Center attempted a port of DAQV to IBM's *xlhpf* compiler that was only partially successful because of limitations in the compiler. In particular, *xlhpf* did not support HPF's `INHERIT` directive, which, as alluded to in Sections 6.3 and 6.4, is necessary in the array access functions. The implication of this was that the implementation of the array access functions was unable to leverage HPF semantics and the *xlhpf* runtime support to carry out data collection. Second, at a recent conference we also discovered that DAQV worked without any modifications to the public domain ADAPTOR/HPFIT compilation system [2,3]. Future compiler targets may include Applied Parallel Research (APR), DEC, and the D System/dHPF compiler [1].

7.2.3. Language portability

So far, two attempts at moving DAQV beyond HPF have been made and both have been largely successful. First, DAQV was successfully ported to a Fortran90/message passing code at Los Alamos National Laboratory (LANL) [11]. A primary distinction

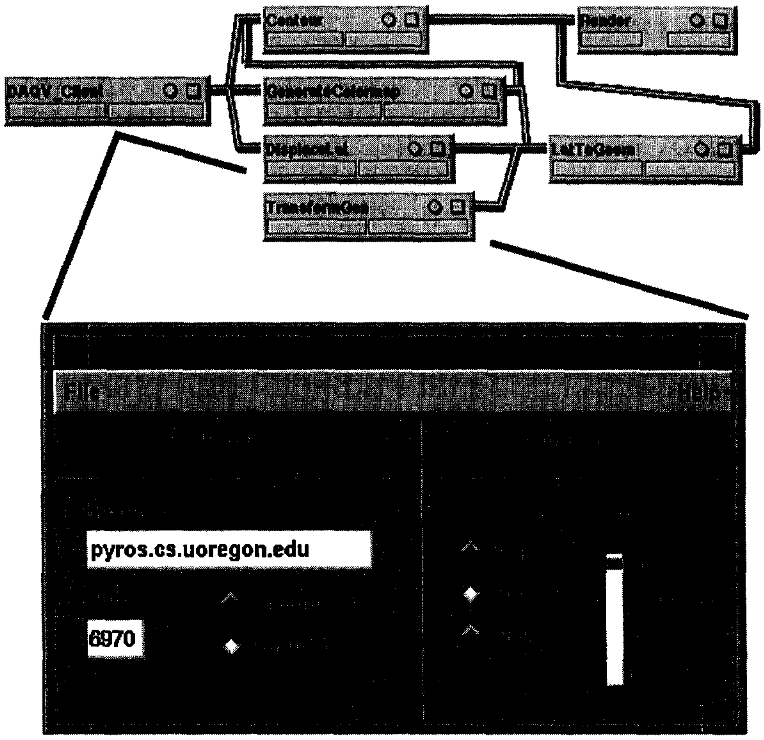
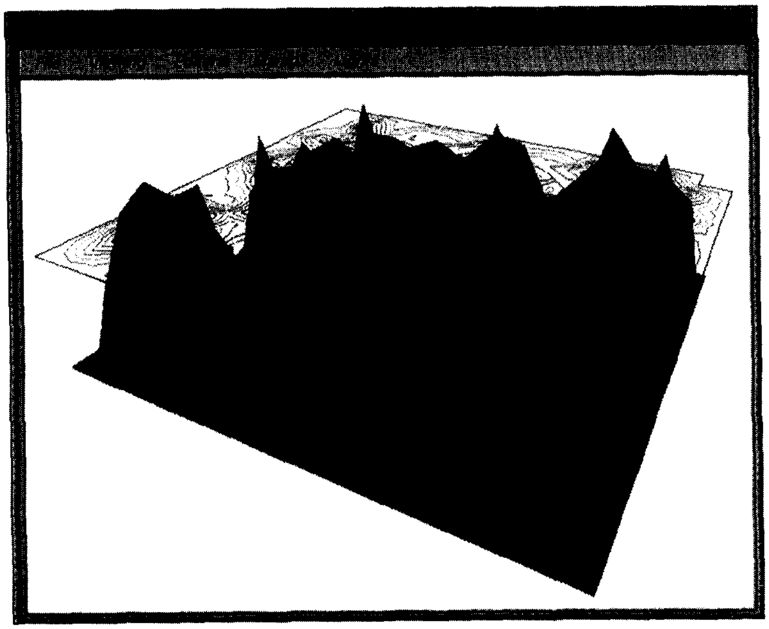


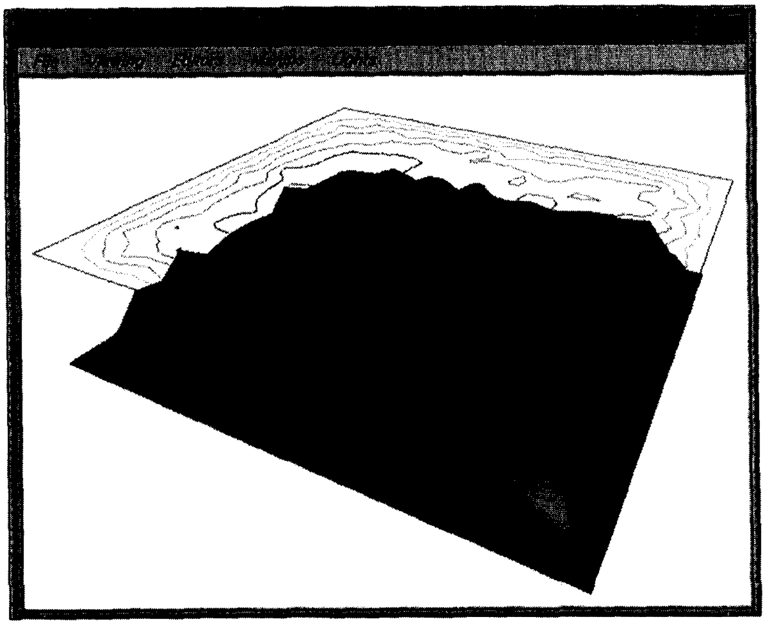
Fig. 7. DAQV functionality can be extended to scientific visualization packages like IRIS Explorer by creating a module within the environment that acts as a DAQV data client. Connecting to a DAQV server is accomplished by specifying a hostname and TCP/IP port number. Basic controls for updating the visualization are also provided. The module outputs data in the primitive format of the application environment allowing the data to be manipulated and displayed by other modules in the environment.

between HPF and this programming model is the lack of a global address space. That is, each node only knows about its local elements, and access to remote values is achieved through explicit message passing. This complicates our ability to provide general array access functions. The approach that was taken in this case was to create application-specific array access functions that understand the data decomposition and substitute them for the default ones in the DAQV library. A general port of DAQV to this model, currently in progress, will support commonly used distributions and simultaneously support user-defined access functions. The second language target was the POOMA/C++ application framework under development at LANL [27]. In this case, robust language semantics allowed us to create generic access functions for POOMA `Field` objects analogous to those used with HPF. While a full port to POOMA has not been completed, the concept has been proven and demonstrated [10].

We believe that the DAQV model (and even parts of its implementation) can be applied in many situations where distributed data is involved. For instance, in the pC++ project, we implemented early versions of the DAQV design [5]. We now intend to retarget the DAQV reference code to the HPC++ system [16]. More generally, we

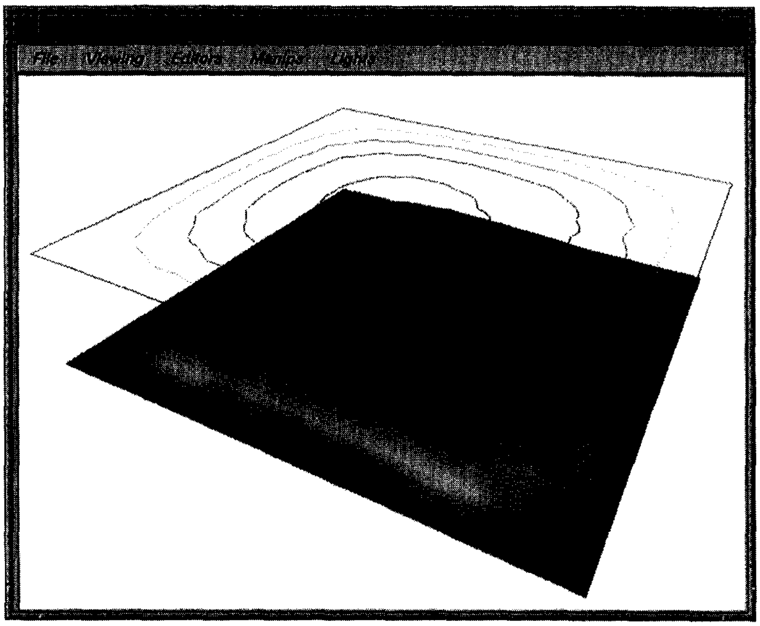


(a)



(b)

Fig. 8. Displays created within IRIS Explorer can take advantage of existing, advanced scientific visualization capabilities. For example, the visualization program in Fig. 7 superimposes contour lines over a 3D displaced grid surface to create the displays in (a), (b), and (c). The data source is the Laplace Heat Equation program discussed in Section 7.1 and shown in Figs. 3 and 4.



(c)

Fig. 8. (Continued)

see no intellectual difficulties with porting DAQV to SPMD environments where some runtime means (language or library) for accessing distributed data exists that can be merged with the model (e.g., SPMD parallel array libraries, such as P++, built using PVM or MPI).

7.3. Data client portability

DAQV also exhibits another, orthogonal notion of portability with respect to its data clients. A DAQV data client is portable in that it need not be tied to a specific DAQV implementation. The DAQV data protocol defines a standard interface through which array data is formatted and delivered to external clients regardless of the system (e.g., HPF, Fortran90, or C++) from which the data originates. In its most general form, a DAQV data client need only understand this data format.[1] Currently, this format is a simple row-major ordering of the data prefaced by a small amount of metadata (as described in Section 6.2). In the future, DAQV will include support for standard data formats like HDF to extend even further the interoperability of the DAQV framework.

[1] Data clients need not be completely generic; they may be as application-specific and/or integrated into a programming/analysis environment as desired. The seismic tomography environment is a good example of this approach. A data client can take either approach depending on its intended use.

A good example of data client portability is the IRIS Explorer module described in Section 6.2, and shown in Figs. 7 and 8. Instances of this module could be used to collect data from any DAQV implementation. An environment like IRIS Explorer would easily support collecting and visualizing data from multiple DAQV applications, even if those applications were implemented in different languages and running on different machines.

8. Conclusions and future work

In its current form, DAQV should be regarded as a reference implementation that demonstrates functionality, specifies components and their system requirements, and defines APIs and transport protocols. In fact, being the primary derivative of a Ptools project, the reference implementation serves as a prototype for vendors to evaluate and potentially adopt.

If the Ptools reference platform were to be adopted by a HPF compiler vendor, there are some possible opportunities for improvement. In particular, DAQV lends itself nicely to compiler/preprocessor support for instrumentation. DAQV already utilizes to some extent automatic instrumentation in the PGI compiler, but one can easily imagine more sophisticated front-end support for selecting arrays for access (automatically generating registration, push, and yield code), enabling and disabling DAQV operation (generating mode, enable, and disable code), and providing information concerning when arrays are in and out of scope. Another improvement that an HPF compiler vendor could provide is in the (possibly automatic) generation of access functions for distributed arrays. Coding these by hand is not always easy nor does it always lead to efficient execution. With their sophisticated program analysis infrastructure, HPF compilers could perhaps provide more support to the programmer in this regard. Finally, we have not discussed DAQV's ability to provide information about the distribution of arrays, concentrating more on the support for array data access instead. The HPF standard does provide the ability to get this information through intrinsic procedures, and DAQV could trivially incorporate the request, collection, and delivery of this information into its event and/or data protocols.

As demonstrated in the seismic tomography application, the DAQV framework does allow distributed data and program variables to be altered as well as accessed. The ability to extend the event protocol and include appropriate access functions makes this capability possible; we intend to do so. As a result, DAQV could be applied to application scenarios where changing runtime array data or program control variables is beneficial. We believe that the benefit of the high-level, semantic-based query provided by DAQV will allow sophisticated client tools to be developed for interaction with the runtime data. In particular, we are currently working on extending the client-side infrastructure so as to facilitate the binding of distributed array and program variables, as provided by DAQV, with interactive 3D visualizations of the array data and program control.

Availability

The DAQV reference implementation, ported implementations, documentation, and more detailed information can be found on our web pages at http://www.cs.uoregon.edu/~hacks/research/daqv/ or on the Parallel Tools Consortium web pages at http://www.ptools.org/.

Acknowledgements

The authors would like to thank the Parallel Tools Consortium, Cornell Theory Center, Los Alamos National Laboratory, The Portland Group, Inc., the Computational Science Institute at the Univ. of Oregon, and the members of the ParaDucks research group at the University of Oregon.

References

[1] V. Adve et al., An integrated compilation and performance analysis environment for data-parallel programs, in: Proc. Supercomputing '95, ACM, New York, 1995.
[2] T. Brandes, ADAPTOR programmer's guide, version 4.0, Report ADAPTOR 3, German National Research Center for Information Technology (GMD), 1996.
[3] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. Mignot, F. Desprez, J. Roman, HPFIT: a set of integrated tools for the parallelization of applications using high performance fortran: Part I: HPFIT and the TransTOOL environment, in: Proc. 3rd Workshop on Environments and Tools for Parallel Scientific Computing, SIAM, Philadelphia, 1996.
[4] D. Brown, S. Hackstadt, A. Malony, B. Mohr, Program analysis environments for parallel language systems: the TAU environment, in: Proc. Workshop on Environments and Tools For Parallel Scientific Computing, SIAM, Philadelphia, 1994, pp. 162–171.
[5] D. Brown, A. Malony, B. Mohr, Language based parallel program interaction: the breezy approach, in: Proc. Internat High Performance Computing Conference (HiPC'95), Tata McGraw-Hill, New Delhi, 1995.
[6] D. Cheng, R. Hood, A portable debugger for parallel and distributed programs, in: Proc. Supercomputing '94, IEEE, New York, 1994, pp. 723–732.
[7] C. Cook, C. Pancake, What users need in parallel tool support: survey results and analysis, in: Proc. Scalable High Performance Computing Conference, IEEE, New York, 1994, pp. 40–47.
[8] J. Cuny, R. Dunn, S. Hackstadt, C. Harrop, H. Hersey, A. Malony, D. Toomey, Building domain-specific environments for computational science: a case study in seismic tomography, Internat. J. Supercomp. Applications and High Performance Computing 11 (3) (1997), Fall 97.
[9] R. Haber et al., A distributed environment for runtime visualization and application steering in computational mechanics, CSRD Tech. Report 1235, University of Illinois at Urbana-Champaigne, 1992.
[10] S. Hackstadt, DAQV/Pooma Porting Notes, Department of Computer and Information Science, University of Oregon, 1996. Available at http://www.cs.uoregon.edu/~hacks/research/daqv/updates/.
[11] S. Hackstadt, LANL Visit Work Summary, Department of Computer and Information Science, University of Oregon, 1996. Available at http://www.cs.uoregon.edu/~hacks/research/daqv/updates/.
[12] R. Haimes, pV3: A distributed system for large-scale unsteady CFD visualization, in: Proc. American Institute of Aeronautics and Astronautics, 1994.
[13] P.B. Hansen, Studies in Computational Science: Parallel Programming Paradigms, Prentice-Hall, Englewood Cliffs, NJ, 1995.
[14] H. Hersey, S. Hackstadt, L. Hansen, A. Malony, Viz: a visualization programming system, Tech. Report CIS-TR-96-05, Department of Computer and Information Science, University of Oregon, 1996.

[15] High Performance Fortran Forum, High Performance Fortran Language Specification, Version 1.0, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1993.
[16] High Performance C++ Working Group, HPC++ Whitepapers and Draft Working Documents, Supercomputing '95 Workshop, 1996. Available at http://www.extreme.indiana.edu/hpc++/.
[17] A. Karp, M. Hao, Ad hoc visualization of distributed arrays, Tech. Report HPL-93-72, Hewlett-Packard, 1993.
[18] A. Kempkes, Visualization of multidimensional distributed arrays, excerpts from Master's Thesis, Research Center Juelich, Germany, 1996.
[19] D. Kimelman, P. Mittal, E. Schonberg, P. Sweeney, K. Wang, D. Zernik, Visualizing the execution of high performance fortran (HPF) programs, in: Proc. 9th Internat Parallel Processing Symp. (IPPS), IEEE, New York, 1995, pp. 750–757.
[20] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., M. Zosel, The High Performance Fortran Handbook, MIT Press, Cambridge, MA, 1994.
[21] R. Koppler, S. Grabner, J. Volkert, Visualization of distributed data structures for HPF-like languages, Scientific Programming (special issue High Performance Fortran Comes of Age) 6 (1) (1997) 115–126.
[22] J. May, F. Berman, Panorama: a portable, extensible parallel debugger, in: Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, ACM, New York, 1993, pp. 96–106.
[23] Parallel Tools Consortium (Ptools) DAQV Working Group, Ptools Meeting Summary: Distributed Array Query and Visualization (DAQV) Project, Department of Computer and Information Science, University of Oregon, 1995. Available at http://www.cs.uoregon.edu/~hacks/research/daqv/.
[24] Parallel Tools (Ptools) Consortium, 1996. Available at http://www.ptools.org/.
[25] The Portland Group, Inc., PGHPF User's Guide, Portland Group, Wilsonville, OR, 1995.
[26] The Portland Group, Inc., PGHPF Profiler User's Guide, Portland Group, Wilsonville, OR, 1995.
[27] J. Reynders et al., POOMA, in: G. Wilson, P. Lu (Eds.), Parallel Programming Using C++, MIT Press, Cambridge, MA, 1996.
[28] Thinking Machines Corp., CM-5 Technical Summary, Thinking Machines Corp., Cambridge, MA, 1992.
[29] A. Tuchman, D. Jablonowski, G. Cybenko, Runtime visualization of program data, in: Proc. Visualization '91, IEEE, New York, 1991, pp. 255–261.