Distributed Array Query and Visualization for High Performance Fortran

Steven T. Hackstadt and Allen D. Malony
Department of Computer and Information Science
University of Oregon, Eugene, OR 97403
Phone: 541.346.4408 / Fax: 541.346.5373
{hacks,malony}@cs.uoregon.edu

Abstract

This paper describes the design and implementation of the Distributed Array Query and Visualization (DAQV) system for High Performance Fortran, a project sponsored by the Parallel Tools Consortium. DAQV's implementation 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 HPF itself has led to a robust and portable solution in which clients do not need to know how the data is distributed.

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; for example, 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; for example, 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.

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 would argue that there are two reasons for this. First, parallel programming tools are often designed without the needs of the users in mind [5]. Tools can be complex and hard to understand because of the complexities 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 changes (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 upfront, 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; requiring distributed operation and environment-wide integration in addition does 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 leveraged in tool implementation (e.g., MPI for data communication). This affords the possibility of high-level problem-solving tools built upon common environment support that abstracts system-specific details.

In this paper, we describe a tool (more appropriately, a framework) for distributed array query and visualization (DAQV) in HPF programs. One of the unique features of DAQV is that it was designed with the philosophy discussed above. The tool was originally offered as a project effort to be sponsored by the Parallel Tools (PTOOLS) Consortium, 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 of DAQV as we envision it. The result is a tool that is portable with HPF (both across system architectures and compilers), that can interoperate with other tools via an open interface, 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 contrasts it with DAQV's functionality and implementation approach. Then, after discussing DAQV's design, functionality, and implementation, we give examples of its application. 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 good, 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 leverage 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. In a similar vein, 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 (PTOOLS) [21], 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 the underlying infrastructure for interfacing with the running program. However, based on feedback from the PTOOLS community [22], it quickly became clear that DAQV's mission was too broad and that it should be more focussed. The most interesting feedback was from users who helped refine the focus of the project to creating a robust and well-defined infrastructure rather than a multitude of user interface features (as tool designers might 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) and use other tools to "work with the data." The goal then should be 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). To this end, users also acknowledged the importance of HPF as the primary 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 integrated with the HPF language is what makes it unique. This will be discussed at length. One final comment is that DAQV is characteristic not of a tool necessarily, but of a framework or meta-tool. It provides the intended capabilities in a component form that can be utilized in other tools that may want to do something with the data that DAQV provides; a visualization tool is but one example. The DAQV framework truly intends to support higher-level semantics of the data, allowing extensions both in tool functionality (via new connected tool modules) and data access 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 frameworks.

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 [2] 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 [3].

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 [17] provides a static depiction of how parallel array data gets allocated on processors under different distribution methods. In the DAVis tool [14], 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 [15]. 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 [13] 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 extendability in parallel tools has evoked designs following client/server models. The Panorama debugger [20] demonstrates how the concept of interoperating modules can lead to increased functionality and generality in debugging systems. The p2d2 debugger [4] 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 which actually affords 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 [8] 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 decendant of the Vista research [26] 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 [7], 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 intend 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. By this definition, DAQV is somewhat different from many of the systems and tools discussed in the previous section. DAQV attempts is to "expose" the distributed data structures of a parallel program to external tools -- but to do so in a way that does not require external tools to have any knowledge about data decompositions, symbol tables, or the number of processors involved. The goal, then, 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 characteristics:

- A logical, global view of parallel, distributed data
- A simple,many-to-one client/server model
- Portability from HPF and compiler-independence

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 [16]. A programmer cannot completely disregard the issue of data distribution 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. Thus, HPF supports a logical, global view of data. For this reason, DAQV supports a similar perspective when interacting with the user (through external tools). The implementation of this design requirement will be discussed in Section 6.

4.2 Client/Server Model

DAQV is a software infrastructure that enables runtime visualization and analysis of distributed arrays. It is not a standalone 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 in this design is to leverage 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 [16] appear as a single, coherent distributed array server, and how to simplify as much as possible the requirements placed on DAQV clients. These issues will be discussed in more detail in Section 6.

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 HPF in the first place, machine portability is inherent to the extent that a given 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; compiler-dependent code is minimized and isolated.

The first two items in the list above 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 summary, the design of DAQV has been motivated by a range of factors, including input from the PTOOLS user group, the need to support semantic-based access to distributed arrays, and a desire to maximize DAQV's portability.

5 Functionality

This section will attempt to describe, in general terms, the functionality of DAQV. In many ways, DAQV targets two different groups of people: tool users and tool developers. It does this by supporting two different operational models, called push and pull. These two models differ in the degree of interactivity available with the HPF program at runtime. After presenting some common terminology used in discussing DAQV, this section will describe these models and how they differ from one another.

5.1 Terminology

distributed array server: an executing HPF program that was compiled with the DAQV library.

data client: an external tool, connected to the distributed array server, that visualizes (or otherwise analyzes) data.

control client: an external tool (interface), connected to the distributed array server, that allows the user to (1) control the execution of the HPF program (under the control of the distributed array server), (2) associate data clients with specific arrays, and (3) send array data values to data clients.

yield point: a point in a HPF program at which execution control is transferred to the control client via the distributed array server.

registering an array: the process by which the distributed array server gets information about an array so values can be collected and sent to a data client at a later time.

pushing an array: the act of sending data from the distributed array server to a data client when the decision to do so is made from within the HPF program (the distributed array server) itself.

pulling an array: the act of sending data from the distributed array server to a data client when the decision to do so is made by the control client (e.g., via the user).

5.2 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 (1) indicate the DAQV model (i.e., push or pull) to be used in the program, (2) register distributed arrays with DAQV, (3) set parameters for communicating with a data client, (4) make DAQV connect with a data client, and (5) 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.

5.3 The Pull Model

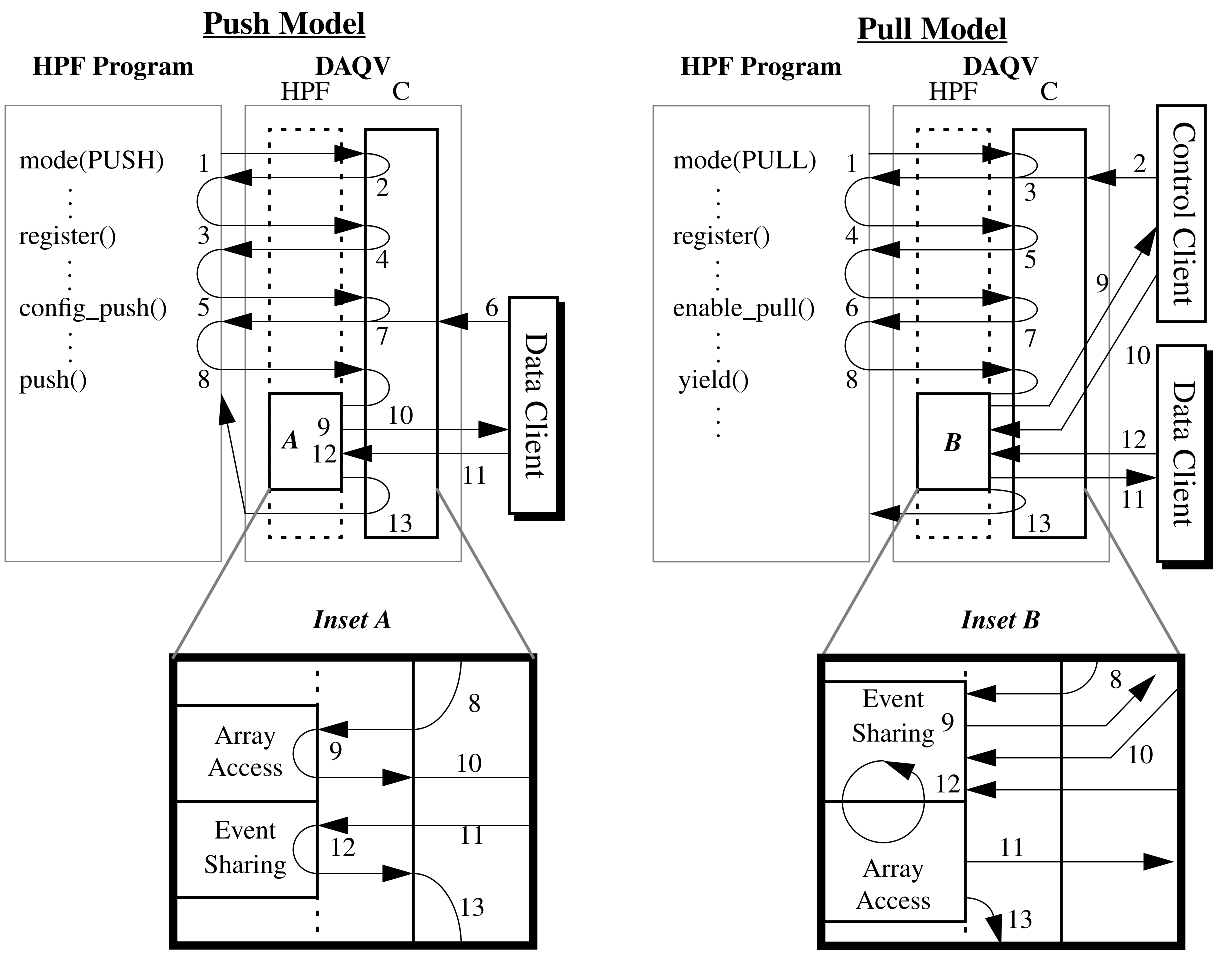
The push model is adequate if the programmer knows exactly which arrays they wish to visualize and when they want to view them. However, to support a more exploratory and flexible approach to array visualization, DAQV implements the *pull model* which supports controlling program execution and selecting arrays for visualization through an external interface. An example of such an interface that uses a debugger metaphor will be presented in Section 7.

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.

Thus, 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 "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 in the next section.

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 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 we have developed for DAQV in three areas: procedural interface, client/server interface, and underlying mechanisms. As a reference for this section, Figure 1 shows the path of execution and control exhibited by HPF programs using DAQV. The execution paths are presented with respect to the procedural interface supported by DAQV.



(1-2) Set DAQV mode to PUSH.
(3-4) Register a distributed array that will be pushed out of the program later.
(5) Set communication parameters for a registered array.
(6) User starts up data clients for each configured array; data clients make socket connection.
(7) HPF program resumes after client connects.
(8) Push transfers control to DAQV to handle data transfer.
(9) The appropriate HPF array access function is invoked.
(10) Data is buffered and sent to the data client.
(11) The data client acknowledges receipt of all data.
(12) All processes are informed of acknowledgement.
(13) Control transfers back to HPF program.

(1) Set DAQV mode to PULL.
(2) Control client connects to DAQV.
(3) HPF program resumes after control client connects.
(4-5) Register a distributed array that will be pulled out of the program later.
(6-7) Activate subsequent yield points.
(8) HPF program yields control to DAQV and enters event processing loop.
(9) Control client is notified that HPF program has yielded.
(10) Control client responds with events.
(11) Control client may request that a particular array be pulled; array access function is invoked; data is buffered and sent.
(12) The data client acknowledges receipt of all data.
(13) The control client allows the HPF program to continue; control transfers back to the HPF program.

Figure 1. *The flow of execution control in DAQV changes between the HPF program, the parts of DAQV implemented in C, and the parts of DAQV implemented in HPF. The diagrams and accompanying text trace the execution flow through the push and pull models with respect to DAQV's procedural interface*

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 leverage the HPF compiler and runtime system automatically. In this respect, we view DAQV as a HPF 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 subroutines, shown in Figure 2, is there to handle initialization, registration of arrays, configuration of data clients, and data extraction. Figure 2 describes each routine with its name, the arguments required, the DAQV model (push or pull) under which the routine is used, and a brief description of the routine.

Type	Subroutine	Arguments	Model	Description
Setup	daqv_mode	mode	both	Sets the DAQV operational model (push, pull, or off) to be used by this program
	daqv_register	array_id[a], name, symbol, type, rank, dim1, dim2, ...	both	Gives DAQV a handle on a distributed array for later access
	daqv_config_push	array_id, port, size	push[b]	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[c]		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 subsequent yield points

a. The argument array_id, an array ID number used by DAQV, is modified by daqv_register().
b. In the pull model, data client configuration is intended to take place via the control client. However, calling daqv_config_push() in the pull model will still accomplish the desired results.
c. daqv_yield() creates a manual yield point; DAQV uses compiler-specific tracing routines, if available, for more robust (e.g., line- or function-level) yield point instrumentation.

Figure 2. *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.*

Currently, the DAQV interface subroutines are inserted manually by the programmer. If the compiler supports it, DAQV routines could be inserted automatically. For example, PGI's HPF compiler, pghpf, automatically inserts line- and/or function-level tracing routines into the HPF source code [24]. By linking in DAQV in place of the default tracing routines, DAQV uses these as yield points.

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.[1] Because DAQV is a library, it must respect the execution semantics of the HPF program. However, because 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

1. Some modifications to DAQV would be necessary for a multi-threaded HPF implementation.

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 process identification macro to determine who is the manager.

6.1.1 Setting the Mode

The daqv_mode() call to DAQV establishes the DAQV operational model: push or pull. It also can be used to enable and disable DAQV operation. In the future, we will allow the user to provide the mode type as a program command line argument.

6.1.2 Registering Arrays

To access distributed arrays, DAQV could have been developed to work with a HPF runtime system. However, this approach jeopardizes DAQV portability across HPF compilers because runtime systems are not necessarily compatible. Instead, DAQV must implement array access in a standard, portable manner. We decided to use the HPF language itself for this purpose. Most of DAQV is written in C, but it uses HPF routines to access distributed data. Why? Because the HPF compiler can be employed to perform all of the low-level runtime operations to get the data, probably more efficiently and correctly than what we could develop. Furthermore, this solution is standard between HPF compilers.

The only problem limiting the portability of this technique is that each compiler has its own "calling context" for functions and subroutines. A compiler is at liberty to modify arguments or change the number of arguments in a subroutine call during compilation. The purpose of the daqv_register() routine is to let DAQV determine -- at runtime -- exactly what arguments are required to invoke an appropriate HPF array access function at a later time. Daqv_register() reads the arguments passed into it and stores them in an internal data structure. When the array values are to be extracted, DAQV reconstructs the calling context (from the stored parameters) for the appropriate HPF array access function. If the registered argument values for a distributed array become inconsistent for some reason during execution (e.g., perhaps from a redistribution of the array), the array need only be re-registered.

The arguments to daqv_register() have several purposes. Array access functions take an array symbol (which, in HPF, is likely to be a pointer to a structure that describes the array and its distribution) and the sizes of each dimension of the array as arguments. Array type and rank are used by DAQV to determine which access function is appropriate for the array, and the array name is a textual description for the array to be used by data and control clients. The array_id argument is modified by daqv_register() to indicate the DAQV array index for the registered array. Subsequent DAQV calls use this value to refer to the registered array.

Registration is one of the few compiler-specific parts of DAQV. The only knowledge necessary is the compiler's calling convention, or more concretely, what transformations the compiler applies to function and subroutine arguments. However, DAQV does not need to know what the transformations actually represent, so this information can easily be provided by a vendor without threat of exposing proprietary information. DAQV simply figures out the appropriate argument values by reading them off the argument list that comes into daqv_register(), stores them in an internal structure, and then recreates them when it wants to invoke an HPF routine. While currently inserted manually, DAQV registration calls could easily be handled by either a preprocessor or the compiler itself.

6.1.3 Configuring Pushes

Distributed arrays are associated with an external data client to which data values are communicated when the array is pushed or pulled out of the program. The daqv_config_push() routine accomplishes this task. Each array ID corresponds to a single data client configuration. (Note that a single array can be registered multiple times under different IDs to send the same array to multiple data clients.) In the pull model, data client configuration takes place as part of the interactive control (event) protocol. In the push model, the communication parameters must be established at compile time via davq_config_push(). In either case, configuration determines a port number for the TCP socket connection to the client and the buffer size for carrying out the communication. A socket connection from a data client to the DAQV manager HPF process is required before daqv_config_push() transfers control back to the HPF program.

6.2 Client/Server Interface

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 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. Below, we briefly describe the data and event protocols used by DAQV.

6.2.1 Clients 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 the socket with a simple text-based confirmation event after receiving data. Second, a client must be able to parse one of the formats supported by a DAQV formatting library. Currently, we have tested two data clients, both ported with from other projects (see Section 7). Work on several new clients is ongoing, ranging from clients that simply write data values to files or display them in a table, to clients that are modules in a data flow visualization system like IRIS Explorer.

Control clients, on the other hand, require somewhat more sophistication. They have the same requirements as the push model, but in addition, they must handle control events. A C-library has been developed for supporting event processing, and support for other languages (e.g., Tcl/Tk) will be forthcoming. 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). More importantly, though, we see the DAQV pull model and the specification of control events (see below) as establishing a framework for other tool developers. In particular, it provides a simple interface that can be incorporated into other control clients to gain high-level access to distributed data in a flexible and well-defined manner.

6.2.2 Event Protocol

DAQV's event protocol consists of a small set of textual, list-based events that are primarily used to interact with control clients; see Figure 3. The PROG_YIELD event is sent from the server to the control client whenever the HPF program encounters a daqv_yield(). 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, which may be any of the client-to-server events listed in Figure 3. 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; GET_REG_ARRAYS requests a list of registered arrays from the server. The META_DATA event is sent to data clients prior to actual array values.

6.2.3 Data Protocol

The data protocol used by DAQV is made up of two components: a META_DATA event which is part of the event protocol and a raw data event which is in some "standard" data format (e.g., HDF or netCDF). The META_DATA event contains information about the array values that are soon to follow. In particular, it contains the ID of the array being sent, the textual name of the array, its rank and dimensions, the data format that the raw values are in, and information about how the data values being sent relate to the whole array. The last two of these items require further explanation.

6.2.3.1 Formatting Libraries

DAQV abstracts away from a particular data format by calling data formatting functions that are selectable during client configuration. Support for a particular format can be included in DAQV if a formatting library is created. Currently, DAQV supports a straightforward text format that lists array values in row-major order. Work is under way for support of additional formats.

6.2.3.2 Data Contexts

The DAQV design makes it possible to extend the set of array access functions to provide additional operations such as reductions, comparisons, or user-defined calculations before communicating the data to a client. Because of this, we felt that it was necessary to include in the data protocol some means for conveying knowledge about how the data values a client will receive relate to the original array. In some cases, sending out raw data is not enough. Even augmenting values with rank and size information might not be sufficient, especially when complex operations (e.g., a row reduction) may reduce the rank and size of an array before it is sent. Although such operations impose context on the data values that are sent out of DAQV, that context is inevitably lost to the client unless this information can be passed along. The META_DATA event is there for this purpose. By including this event, DAQV is enhancing the framework it hopes to provide for supporting high-level, semantic access to distributed data.

Event	Sub-event	Type	Description
PROG_YIELD		server-to-client	Informs control client that program has yielded control
CONTINUE	UNTIL_LINE	client-to-server	Instruct DAQV to let the HPF program continue to the first yield point past a specified line
	FOR_STEPS	client-to-server	Instruct DAQV to let the HPF program to skip a specified number of yield points
CONFIG_PULL		client-to-server	Set data client communication parameters for a specified array
PULL	ALL	client-to-server	Instruct DAQV to send data values from a specified array to the appropriate data client
	ELEMENT	client-to-server	Instruct DAQV to send a single data value 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
META_DATA		client-to-server	Describe raw data values to follow

Figure 3. *The DAQV event protocol extends functionality for program control, communication configuration, and array extraction to clients.*

6.3 Underlying Mechanisms

We have finally 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 that are implemented in HPF to leverage its high-level language system for array access. This represents an innovative approach to the problem and deserves explanation.

6.3.1 Array Access Functions

DAQV provides a library of array access functions in HPF that have been written for different array types and rank. The current library is extendible to allow for new array structures and operations. Figure 4 show an example of an access function for a two-dimensional array of integers.

```
      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
```

Figure 4. *An array access function written in HPF for two-dimensional arrays of integers.*

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. Most access functions are general in that they handle any array of a given rank and type. Its dimensions are passed in as arguments, and the subroutine inherits the distribution for the incoming array. Efficiencies can be built into the access function. In the example above, the array ITMP is used to reduce a data broadcast to a point-to-point communication on each iteration. This had significant performance improvements in accumulating the array data in the manager for communication to the client.

6.3.2 Event Sharing and Implicit Synchronization

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, the other processes must be notified of this request and make the “same call” as the manager to the access function (as required by SPMD execution semantics). This is done with a manager/worker control mechanism called event sharing.

Worker processes call the daqv_event_share() routine shown in Figure 5 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 it receives. As seen, DAQV implements this routine using the HPF language.

```
      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
```

Figure 5. *A simple routine written in HPF for event sharing among HPF processes.*

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.

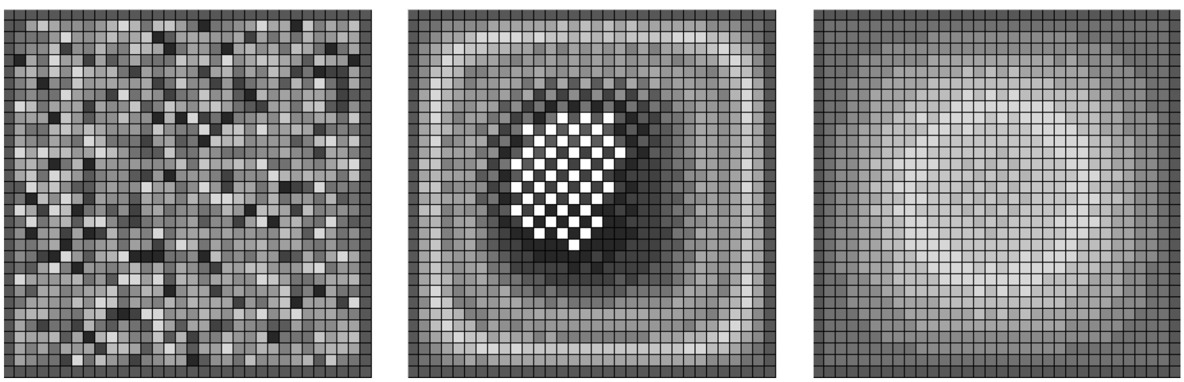
This is how event sharing is implemented in DAQV to remain consistent with HPF execution semantics. It is probably not the obvious solution when first faced with the problem of sharing information among several processes, but in the context of DAQV, it is both an elegant and portable solution.

7 Application: Laplace Heat Equation

To illustrate what the current implementation of DAQV is capable of, we will consider a HPF program that implements a finite difference method for solving the Laplace Heat Equation iteratively [9]. The code continues 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 Figure 6 was generated by a data client called Dandy. Implemented in Tcl/Tk and easily ported from the pC++ Tau Tools [2] to DAQV, 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_config_push(), the Dandy client connects to the HPF 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 and displayed. The Dandy interface (not shown in Figure 6) 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 entered into Dandy. This feature can also be used to identify outliers or to identify values beyond a certain threshold. For example, Figure 6(b) shows values above the specified range as white. As the algorithm nears convergence (Figure 6(c)), these values move into the data range.

The same application can also be used with DAQV's pull model. Figure 7 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. 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 32x32 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 an event to be sent that makes DAQV send the requested array to the appropriate data client. The first registration, 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 [10].



(a) (b) (c)

Figure 6. *Convergence of the finite method for solving the Laplace Heat Equation can be seen by visualizing the two-dimensional array representing the heated surface.*

8 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; in this regard, PTOOLS requires the reference implementation to function on two platforms. The DAQV PTOOLS project finishes in June. By that time, we hope to have tested DAQV on several parallel machine platforms supported by PGI's HPF compiler, including the SGI Power Challenge, the Intel Paragon, and a network of workstations. In addition, we hope to have DAQV working with other HPF compilers; currently, we are discussing this possibility with Applied Parallel Research (APR), DEC, and IBM. Included with the reference implementation will be a few data clients (e.g., the simple Tcl/Tk 2D data viewer shown above and a module to use with Iris Explorer) and a sample control client.

If the PTOOLS reference platform were to be adopted by a HPF compiler vendor, there are some possible opportunities for DAQV 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 generation, possibly auto-

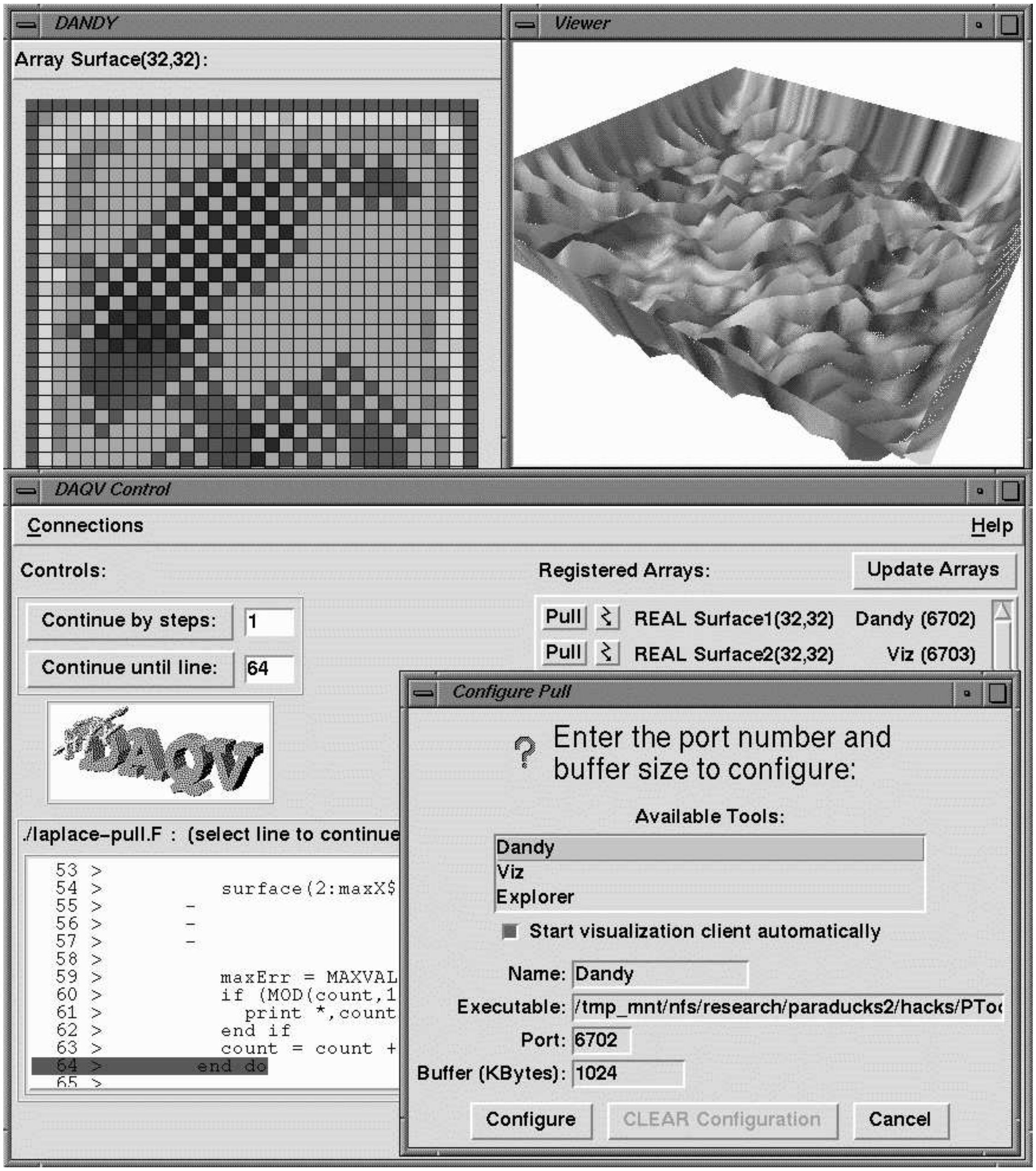


Figure 7. *A DAQV session with a control client, a simple two-dimensional display, and a more sophisticated three-dimensional representation of the 2D array from the Laplace Heat Equation code.*

matically, 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. 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. However, the HPF standard does provide the ability to get this information through intrinsic procedures. Our reason for delaying on support for distribution information has to do with the efforts being undertaken by the Parallel Compiler and Runtime Consortium to provide a distributed data descriptor specification for HPF [6]. Because this specification is intended to be multi-lingual, we hope that the distribution information support could be abstracted to be a general capability of DAQV and not a special capability for HPF. Clearly, adherence to the specification would need to involve the compiler manufacturer.

Beyond the PTOOLS project deliverables and HPF compiler improvements, there are several interesting research directions for the DAQV work. One of the most important short-term goals is to evaluate the benefit of DAQV in a significant scientific application. At the University of Oregon, we are developing an HPF-version of a seismic tomography application for modeling mid-ocean ridges. Our hope is to eventually build a Problem Specific Programming Environment (PSPE) for this application to allow a high level of model interactivity so that one can see, for example, a visualization of the ray path calculations against a three-dimensional image of seismic velocity to better understand the sea floor geology. To provide this visualization at runtime, a framework such as DAQV is required. Basic questions concerning how easily DAQV conforms to the PSPE requirements and how well DAQV performs will be studied.

The DAQV framework does allow distributed data 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 (e.g., for computational steering in the seismic tomography application). 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.

Finally, we believe that the DAQV model (and parts of its implementation perhaps) can be applied in other parallel language systems where distributed data is involved. For instance, in the pC++ project, we have implemented early versions of the DAQV design [3]. We now intend to retarget the DAQV reference code to the HPC++ [11] system. More generally, we 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. We are currently investigating this in the context of SPMD parallel array libraries, such as P++, built using PVM or MPI.

9 Acknowledgments

The authors would like to thank the members of the Parallel Tools Consortium for their feedback and assistance in the design of DAQV. The refining of DAQV's focus and purpose would not have been possible without the help from David Presberg and Craig Lee during Consortium meetings and project breakout sessions. Kurt Windisch, Harold Hersey, and Bernd Mohr are to thank for their work on the data and control client tools shown in this paper. Finally, the assistance and insight provided by Portland Group, Inc. was instrumental in creating DAQV.

10 References

[1] V. Adve, et al., *An Integrated Compilation and Performance Analysis Environment for Data-Parallel Programs*, Supercomputing '95, December 1995.

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

[3] D. Brown, A. Malony, B. Mohr, *Language Based Parallel Program Interaction: The Breezy Approach*, Proc. of International High Performance Computing Conference (HiPC'95), New Delhi, India, December 1995.

[4] D. Cheng and R. Hood, *A Portable Debugger for Parallel and Distributed Programs*, Proc. of IEEE Supercomputing '94, Washington, D.C., November 1994.

[5] C. Cook and C. Pancake, *What Users Need in Parallel Tool Support: Survey Results and Analysis*, Proceedings of the Scalable High Performance Computing Conference, IEEE Computer Society Press, 1994, pp. 40-47.

[6] J. Cowie and T. Haupt, *Common Runtime Support for HPF: High Performance Fortran Distributed Data Descriptor Specification*, Parallel Compiler Runtime Consortium, April 1995, available at URL:http://aldebaran.npac.syr.edu:1955/.

[7] R. Haber, et al.., *A Distributed Environment for Runtime Visualization and Application Steering in Computational Mechanics*, University of Illinois at Urbana-Champaigne, CSRD Technical Report 1235, June 1992.

[8] R. Haimes, *pV3: A Distributed System for Large-Scale Unsteady CFD Visualization*, American Institute of Aeronautics and Astronautics Paper 94-0321, Reno NV, January 1994.

[9] P. B. Hansen, *Studies in Computational Science: Parallel Programming Paradigms*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1995.

[10] H. Hersey and A. Malony, *Viz*, information available at URL:http://www.cs.uoregon.edu/~hersey/viz/, February 1996.

[11] HPC++ Working Group, *HPC++ Whitepapers and Draft Working Documents*, Supercomputing '95 Workshop, available at URL:http://www.extreme.indiana.edu/hpc++/hpc++wp/, February 1996.

[12] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 1.0*, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993.

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

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

[15] D. Kimelman, P. Mittal, E. Schonberg, P. Sweeney, K. Wang, D. Zernik, *Visualizing the Execution of High Performance Fortran (HPF) Programs*, Proceedings of the 9th International Parallel Processing Symposium (IPPS), IEEE Computer Society Press, April 1995, pp. 750-757.

[16] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994.

[17] R. Koppler, S. Grabner, and J. Volkert, *Visualization of Distributed Data Structures for HPF-like Languages*, to appear in Scientific Programming, special issue on implementations of High Performance Fortran, 1995.

[18] A. Malony, J. May, A. Karp, D. Presberg, and V. Schuster, *Distributed Array Query and Visualization*, Parallel Tools Consortium Working Document, January 1995.

[19] J. May and F. Berman, *Creating Views for Debugging Parallel Programs*, Proceedings of the Scalable High Performance Computing Conference, Knoxville, TN, May 1994, pp. 830-844.

[20] J. May and F. Berman, Panorama: *A Portable, Extensible Parallel Debugger*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 1993, pp. 96-106.

[21] Parallel Tools (PTOOLS) Consortium, information available online at URL:http://www.llnl.gov/ptools/, February 1996.

[22] Parallel Tools Consortium (PTOOLS) DAQV Working Group, *PTOOLS Meeting Summary: Distributed Array Query and Visualization (DAQV) Project*, available online at URL:http://www.cs.uoregon.edu/~hacks/research/daqv/, May 1995.

[23] The Portland Group, Inc., *PGHPF User's Guide*, Wilsonville, OR, 1995.

[24] The Portland Group, Inc., *PGHPF Profiler User's Guide*, Wilsonville, OR, 1995.

[25] A. Tuchman, D. Jablonowski, G. Cybenko, A *System for Remote Data Visualization*, University of Illinois at Urbana-Champaigne, CSRD Technical Report 1067, June 1991.

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

[27] T. Wagner and R. Bergeron, *A Model and a System for Data-Parallel Program Visualization*, Proceedings of IEEE Visualization '95, Atlanta, GA, October 1995, pp. 224-231.

[28] Thinking Machines Corp., *Prism*, available at URL:http://www.think.com/tmhtml/ProdServ/Products/prism.html, February 1996.