Performance of a System for Interacting with Parallel Applications

Allen D. Malony, Steven T. Hackstadt

Abstract— **A variety of systems have been developed for interacting with parallel programs for purposes of debugging, monitoring, visualization, and computational steering. In addition to addressing different functional objectives, these systems have different non-functional characteristics that, we believe, are equally important for a user to know. Clearly, for most users performance is an most important non-fuctional requirement of a program interaction system. However, characterizing performance for an interaction system for parallel programs is particularly challenging, especially in asynchronous, distributed environments. In this paper, we present a comprehensive performance analysis of the DAQV system. DAQV has been successfully applied in runtime data visualization, online performance monitoring, and computational steering environments. However, DAQV's future success depends significantly on application context and requirements. By giving a full accounting of DAQV performance, we aim to provide application and environment developers with valuable information about DAQV's potential benefits before an integration effort takes place. As DAQV's designers, this in-depth performance analysis has led to new insights, resulting in higher performing designs.**

Keywords— **Parallel tools, performance analysis, distributed arrays, runtime data access, runtime program interaction, computational steering.**

I. Introduction

Applications that run on multiple processors, whether on a single parallel machine or across distributed machines, may benefit greatly from software support that offers runtime program interaction capabilities for purposes of on-line data access, monitoring, visualization, or adaptive control. Indeed, several systems for program interaction have been developed that provide such support in various forms, including the DAQV system that we discuss in this paper. However, in addition to certain functional needs that program interaction systems may address, developers of parallel or distributed applications are likely also to be concerned with the impact these systems have on their application's performance. The consideration of interaction capabilities that the developer desires versus the choices made during implementation of an interaction system often involve a tradeoff between functional and non-functional requirements, performance being the primary example of the latter.

If functional and non-functional tradeoffs are to be made effectively, it is imperative that the performance of a program interaction system be well understood. There are obvious direct influences on performance, such as the size of data queried from a running program and transferred elsewhere for analysis or visualization; larger data sizes incur larger access delays. There are also more subtle indirect factors that tend to reflect design decisions about how the interaction system operates and the extent to which it can take advantage of high performance computing and networking support. For instance, the choice of a client-server system model to support flexible remote interaction will nevertheless constrain interaction performance because of inter-process communication. However, program interaction does not take place in isolation, and the overall "quality" of the integrated environment (application plus interaction system) will be a consequence of combined factors. Certainly, the characteristics of the application's algorithms and the execution platform are important determinants of what it is possible to achieve. But if the interaction system is mismatched with the application or platform, a poor performing environment is likely to result.

Simply put, not only must designers of program interaction systems address the functional requirements of their target applications, but the non-functional, quality aspects should also be investigated, both to validate well-engineered solutions and to provide application users with valuable information about the potential benefits before an integration effort takes place. In this paper, we give a comprehensive accounting of the performance of a program interaction system designed for accessing application data distributed across multiple processors. We conducted the performance study with respect to the functional use of the system in a bottom-up manner, varying system and environmental (application and platform) factors to broadly characterize performance behavior. This posed several performance experimentation challenges unique to interaction systems, relating to the number of measurable entities, the types of requests made, and the frequency of interaction. Our approach segments the functionality of the system so that the different dimensions of the problem can be studied and factors isolated. The work has led to new engineering insights, resulting in higher performing designs, as well as new techniques for system application.

The following section discusses our interaction system in the context of the general program interaction problem; the functional requirements we targeted are specifically identified. The design and implementation of the system is presented in Section §III; in particular, we hightlight here functional vs. performance concerns. Section §IV describes the performance experiments we conducted and our experimental apparatus. The performance results are then analyzed in detail. We conclude the paper with a discussion of the "quality" of the program interaction system as represented by its performance capabilities, and we look towards future work.

Department of Computer and Information Science, Computational Science Institute, University of Oregon, Eugene, OR 97403, Email: {malony,hacks}@cs.uoregon.edu.

II. Program Interaction

Merriam-Webster's Collegiate Dictionary, tenth edition, defines **interaction** as *mutual or reciprocal action or influence*. This definition indirectly implies the existence of two or more interacting entities and it suggests points in time when interaction takes place. In our case, one of the entities is a parallel or distributed program in execution, and the others are what we might call *interactors*. Interacting with a program involves three general aspects:

- interaction with program state
- interaction with program operation
- interaction with program execution

Program state is defined as the values of program variables, application data, and process state (program counter, stack, etc.). Program state is dynamic, changing as the execution proceeds. Program state interaction involves access to program state, either to read or to modify the state. Interaction with program operation involves controlling or altering program behavior. This can be done by modifying the control flow, explicitly requesting certain program operations be performed, or directly changing program code. Program execution interaction involves how the program and the interactors coordinate program state and operation interactions. Synchronous and asynchronous are two general types of execution interaction.

Program interaction systems exhibit these general, "functional" interaction aspects in different ways depending on the interaction purpose and target environment. Systems that we regard as program interaction systems usually are not referred to as such, being named more for their purpose and application. Perhaps the best known example is a sequential breakpoint debugger. In the GDB debugger [1], for example, program execution interaction is provided through instrumented breakpoints and synchronous command processing. Access to symbol table and runtime information make it possible to query most, if not all, of the program state, and to modify code and execute program routines outside the normal program flow. However, sequential debugging is a simple matter compared to the complex interactions that arise when debugging parallel or distributed programs [2][3][4]. These complexities are a result of distributed state, parallel operation, and multiple threads of execution. The *Prism* debugger [5] demonstrated effective techniques for accessing distributed program state and presenting it in a combined logical view to the user. The integration of event- and state-based debugging in Ariadne [6] showed how parallel program operation could be controlled and altered to display logical, consistent states; Pangaea's [7] use of execution replay to debug nondeterministic heterogeneous applications is similar in concept. The *p2d2* debugger [8] implemented portable mechanisms for coordinating multiple processes on different processors, providing the user with a uniform debugging interface.

Other program interaction systems encounter similar complexities when addressing functional requirements, but also stress different non-functional concerns in their design and implementation. The OMIS [9] online monitoring system for parallel and distributed programs provides an API that addresses a broad range of these concerns for debugging, performance analysis, data visualization, and program management, including problems of real-time operation, large data communicaton, and adaptive feedback. Paradyn [10] supports real-time interaction with a parallel program for purposes of performance analysis. Based on a technique called dynamic instrumenation, measurement code can be inserted to gather data for runtime viewing or decision support. The Autopilot [11] and Active Harmony [12] systems extends such an interaction model with mechanisms for adaptive feedback of performance control. In both ParaDyn and Autopilot, data and control interactions must be of low intrusion so that the performance of the application as a whole is not severely affected. However, challenges arise when the execution platform is more distributed and the performance tools themselves are remote, requiring interaction that is more costly.

Online visualization [13][14] shares the real-time issues as well as the problem of large data transfer. Here, interactions involve application-level data semantics with the desired result being an application-specific visualization for certain program variables; usually the application data is further processed before being displayed. Unfortunately, large, distributed data may need to be accessed to create the visualization. Hence, the ability to maximize communication performance is critical for online visualization environments to be effective. This is especially true for computational steering systems [15][16][17][18] where interaction for program adaption and online visualization are combined. Steering systems must provide support for application-level state and operation to be accessed by the user or online tools for purposes of assessment and control. With some optimization of the application's execution being the goal, the performance of the computational steering system is of critical concern. Systems for coupling application models [19] push non-functional interaction requirements even further since they intend to offer an alternative for developing large-scale, high-performance parallel programs.

Both functional and non-functional requirements drive the design and development of interaction systems for parallel and distributed programs. Different sytems target different problems and the systems above are clearly better known for what set of functional features they provide. However, these systems do share common program interaction aspects, and non-functional features, although also problem-determined, are manifest across different areas. Performance, for example, is a universal concern, but the requirement for performance is based on criteria from the problem domain and the system implementation environment. By providing a full characterization of the performance of the DAQV system, we hope to provide a basis for validating the acceptability of the system for problems that it was intended.

III. Design and Implementation

DAQV[1] is a system that lets *tools* interact with parallel applications. In particular, using DAQV, tools can access application data arrays during execution, even if the data is distributed across multiple processors or machines on which the parallel application runs[2]. DAQV was designed to meet four main functional objectives:

- *Read/write access to program arrays*: Tools must have both read and write access to program data arrays. Reading an array involves a transfer of array data to the tools, whereas writing an array updates its contents using data received from the tools.
- *Distributed data interaction*: Program array interaction should allow for the case where the array is distributed across multiple processes of the program which are running on multiple processors of the execution platform.
- *High-level interface to program arrays*: Tools should be able to interact with the program via logical array references without the need to know on which processes the program data arrays are located or how they are physically stored.
- *Multiple, remote tools*: Tools should not be required to run on the same machine(s) used by the parallel program (i.e., the tools can be *remote* from the program). More than one tool should be able to interact with the program at the same time and these tools can be remote from each other.
- *Coordinated execution*: Execution interaction should be coordinated between the program and the tools to allow for both synchronous and asynchronous control.

These requirements leave many issues unresolved, such as the choice of parallel computing model, the languages and operating systems used, the structure and distribution of arrays, and the type of machine environment where the parallel program will run. Such issues define the scope of functionality of an interaction system and they are resolved through early design decisions regarding the functional model. In the case of DAQV, these design decisions included:

- *Heterogeneous, distributed cluster of Unix-based multiprocessor computers*: DAQV is targeted at a metacomputing environment characterized by a heterogeneous collection of networked parallel machines which may be geographically distributed. The reliance on Unix is determined primarily from the dependencies of the chosen communications subsystem.
- *SPMD parallel computing model*: The parallel program is assumed to be written in SPMD style. The program is composed of multiple processes, each running the same program code (this is not a hard constraint) and communicating via messages.
- *Parallel arrays with HPF-style distribution*: Under the SPMD assumption, any program array to be accessed is assumed to be a parallel array, distributed across the parallel processes[3]. Parallel array interaction semantics are used to access all arrays. Array distribution will be in the manner of HPF [20].
- *Array consistency*: It is the responsibility of the application programmer to ensure that program data is in a consistent state when it is made available for access, either synchronously or asynchronously.
- *Availability and requests are independent*: The application determines which arrays are available to be accessed at any time, but the tools decide what data will actually be requested.

A. Architecture Model

The remote separation of tools from the parallel program and the need to access program data on different processors necessitated a distributed architecture for the DAQV interaction system. We chose a client-server architecture as the basis for the design primarily because of the natural view of the program as a parallel "data server." However, DAQV's view of coordinated execution separates execution control decisions from data access. Both the application and the tool participate in controlling the execution to ensure state consistency and to perform data access. This is done through the concept of *yielding* program execution. A yielding application is suspended pending an explicit control signal to continue. While suspended, application data remains unchanged and can be repeatedly accessed. This differs from *non-yielding* access in which execution continues immediately after any waiting requests have been satisfied. Applications provide access and tools make requests, but both can cause the program to enter a yielding state. The tools do this by converting a non-yielding access in the application to a yielding one. The cooperation between the application and the tools required us to extend the client-server architecture to include a separate "execution server" component; as explained below, we refer to this server as the "master".

Figure 1 shows the DAQV system architecture. DAQV consists of three components:

- application data access interface
- tool (client) request interface
- request management and execution control

The application data access interface is a library used by a parallel program to make its distributed data available to tools; Table I describes the interface. The `DAQV_REGISTER` procedure allows the programmer to describe what data is accessible and how it is distributed. The `DAQV_PROBE` and `DAQV_MUTATE` procedures are used at places in the code where arrays may be accessed and define how this access capability is synchronized with program operation. On the tool side, the request interface is a library that allows a regular section of a registered program array to be read (probed) or written (mutated). Tools must first "attach"

[1] DAQV stands for Distributed Array Query and Visualization. Pronounced "DAVE," the "Q" is silent.

[2] From now on, "multiple processors" is meant to include the case of applications running distributively on multiple machines.

[3] Note that this does not preclude the case of an array distributed "across" a single processor.

to DAQV and obtain array registration information. The request interface also allows the tools to alter execution control.

The role of the request management and execution control component of DAQV is to implement probe and mutate requests, and to coordinate the execution of the parallel program with tool interactions. It consists of three parts: a *slave* part, a *master* part, and a *client* part. When each application process executes `DAQV_INIT`, the access library creates a slave thread that shadows application process execution. The purpose of these threads is to maintain information about available distributed data and to perform accesses to that data when requested. The reason for embodying this functionality in the form of a thread is so that querying the information and accessing the data can be done concurrently with the execution of the application processes, if desired. The individual slave threads are coordinated by a separate process called the DAQV master. In addition to coordinating the slave threads, the master process is responsible for interacting with the tools in the DAQV environment by acting as the "single point of contact" for request management and execution control. The master process spawns additional threads to service the various requests it receives, in particular, to handle data transfers between itself and the application slaves and tools. The client part of the request management and execution control component runs as a separate thread in the tool process. Its responsibility is to communicate requests to and receive events from the master, and to participate in data transfers on behalf of the client tool.

B. Array Access

The access and request API's insulate the program and tools from the underlying DAQV runtime system (i.e., request management and execution control) by implementing a high-level array access model. From the point of view of a tool, the request interface gives access to registered program arrays. Consider the registered array `A[4][4]`. A tool may access any regular, dense subsection of a registered array, say `A[2:3][2:3]`. DAQV regards program arrays as being distributed across the processes of an SPMD program. As such, each process contains only a portion of the entire "global" array. What portion each process contains depends on the distribution of the array onto a virtual processor (process) grid. We provide HPF-style distributions of `BLOCK` and `CYCLIC` in each array dimension. Global array structure, type, and distribution are supplied to DAQV by the `DAQV_REGISTER` procedure. If, for example, `A` is distributed `BLOCK(2), BLOCK(4)` on a 2x1 processor grid, DAQV can determine that elements of `A` are distributed across two processes in the manner shown in Figure 2. Furthermore, DAQV assumes that the "local" array parts are contiguously stored. Thus, when given the tool request to access `A[2:3][2:3]`, DAQV can determine on which processes the array elements are located and where they are stored with the process-local data. These elements must be accessed from the application processes and packaged together in a 2x2 sub-array that is returned to the tool; see Figure 2. The complexity of array access supported by DAQV is not only in understanding data distribution, but also includes mapping elements from their local storage to their storage position in the global sub-array.

C. Protocols

DAQV's runtime system defines protocols to control the slave, master, and client operations needed to realize array registration, tool attachment, and array access. Figure 3 shows the interactions that occur within the DAQV architecture for these different functions. These interactions can become quite complex, especially during probe and mutate array access. However, it is important when designing experiments to consider non-functional performance aspects of DAQV to be clear about the actions that can take place in protocol execution. For brevity, we will describe the protocol for servicing a yielding probe.

Consider a DAQV-enabled parallel program that has registered a distributed array, `A` (as before), and a DAQV-enabled tool that wants to probe `A`'s values. Once this tool successfully attaches to the DAQV master process it gets information about registered arrays. At this point, it sends a request for probing `A[2:3][2:3]` to the master process, which responds by adding the request to a queue of pending events. Execution control is returned to the tool as soon as it tells the client thread to send the request to the master process. The tool may now periodically query to see if the request has been satisfied.

Meanwhile, on the application side, the parallel program eventually reaches a call to the procedural interface routine `DAQV_PROBE`, listing `A` as one of its arguments to indicate it is in scope and available for read access at this point in the code. Upon invocation, `DAQV_PROBE` transfers execution control to the slave threads (one per program process). The slaves send the probe argument list to the master process for confirmation. If confirmed, the master searches its event queue for pending probe requests from external tools. Each item on the queue must be cross-referenced with the array argument list so that only requests for arrays that are actually in scope are answered.

The master forwards each valid request in its event queue to the slave threads to be filled. Using array distribution information, the slave threads determine what part of the array, if any, resides locally. The threads each allocate a buffer and fill it with the appropriate values read from the address space of their associated application process. The slave threads respond to the master process by sending the filled buffers. Each response to the master from a slave causes a new master thread to be created to receive the data and load it into the buffer representing the global array sub-section requested by a client. The master coordinates among these threads until all data is received for a request. This buffer is then sent to the external tool that originally requested it. Each tool request and buffer delivery results in the creation of a separate master thread. The slave threads, master threads, and the master process work together in this way until all requests are satisfied. When a tool finally issues a continue, the master communicates this to the slaves and the application continues execution.

D. Implementation

The DAQV architecture is flexible enough to support the functional objective of remote tool interaction with a SPMD parallel program running in a distributed, heterogeneous environment. Every process involved in DAQV – the application processes, the DAQV master process, and the tools – can run on the same machine or different machines, in any combination. This is made possible by the use of the Nexus multithreaded communication library [21]. Although other alternatives were available for the communication subsystem, Nexus offered integrated threading support which we thought would allow us to address more effectively certain performance concerns. For instance, when the application, master, and tool processes run on the same parallel machine, it may be possible to have parallel, simultaneous data transfers between slave, master, and client threads. One of DAQV's non-functional goals was to have a single implementation that performed well across different system environments.

Access and request API's have been developed for the Fortran-based languages, with MPI considered as the program's message communication library. It is important to note, though, that DAQV does not rely upon these languages (or MPI) for any key functionality, making it easy to retarget to other languages such as C and C++.

IV. Experimental Design

Our performance experiments attempt to categorize broadly the performance of the DAQV system by focusing on three areas. First, because DAQV supports interaction with executing applications, an important concern is how much overhead an application can expect to experience by using DAQV. Second, DAQV's task of assembling distributed data from multiple application processes and delivering that data to multiple client tools increases its potential as a bottleneck in the program interaction process. For this reason, we examine the performance of several aspects of the DAQV system itself, including a comprehensive system profile, the effects of parallelism, and the performance of our data access algorithms. Finally, we examine the more complex interactions that arise under certain use scenarios between application processes, the DAQV master process, and connected tools. This section describes our experiments in each of these areas in detail. We begin with a more general description of the experimental environment, our measurement technique, and the parameters we varied.

A. Computational Environment

All of our tests were carried out on a cluster of Silicon Graphics, Inc. shared-memory multiprocessors (SMP) (SGI Power Challenge and Power Onyx boxes) connected by 100 Mb/s ethernet; this has also been our target development environment. In the work reported here, though, we often isolated our experiments to a single SMP box to eliminate communication bottlenecks between machines and to better measure different performance behaviors exhibited by DAQV. Unless otherwise noted, all tests were conducted on a Power Onyx with eight 194 MHz MIPS R10000 processors, 1 GB of 4-way interleaved main memory, and 2 MB of secondary unified instruction/data cache per processor. During our experiments, no competing production jobs were allowed to execute on the Power Onyx. However, our use of the machine was not exclusive; several users were logged in running shells and other interactive processes.

For all tests, the application processes used were written in Fortran90. The DAQV application access library, master process, and tool request library are all written in C. The client tool used in our tests was designed specifically to support our experimentation and to reduce common client overheads such as graphics display and user input. However, the client did use the standard DAQV client library upon which other clients have been built.

B. Measurement Technique

Our measurements were made using the IRIX routine `clock_gettime`, which allowed us to access the nanosecond-resolution free-running hardware counter on the Power Onyx. Using this counter allowed us to make accurate measurements across multiple threads and processes running on the same SMP. We made use of two different timing techniques: intervals and timestamps. In some cases, we were only interested in generating the elapsed time over an interval of code (e.g., for measuring overhead). In other cases, we were interested in generating a series of time-stamped events across multiple threads and processes. We built a small timing library with C and Fortran interfaces that supported both of these measurement techniques. The overhead of the interval routines that actually perturbed our measurements was very small (i.e., less than 5 microseconds) compared to most of the measurements reported here. However, to measure intervals across threads and processes, timestamps had to be recorded and stored, and then matched after the experiment was completed. Our current method for taking timestamps adds the overhead of a system call (`fprintf`) to the measurement. This overhead was determined and subtracted from the appropriate values reported here. The primary disadvantage of using the free-running hardware counter was that it complicated the creation of time-stamped events on multiple machines; interval timing across machines was still possible, though.

C. Experimental Parameters

Our experimental methodology involved numerous parameters that are best organized into the categories shown in Table II. At the highest level, we could control the relative location of execution among all DAQV components (slaves, master, and clients). The relation between each pair of components was classified as either local (the components were running on the same machine) or remote (the components were running on different machines). Because of the interactions in the DAQV model, the only pairings of interest were slaves/master and master/clients. Within the application slave processes, we could control numerous factors, including the rank and size of arrays registered with DAQV, data distribution type and parameters, the number of processors across which the data was distributed, the duration of the application's core "computational loop," and whether DAQV was operating in a multithreaded or single-threaded manner. In the clients requesting data and controlling slave execution, we varied the number of them connected to the master and the number of simultaneous requests they made, and the frequency of those requests. Missing from these categories is the DAQV master process. The reason for this is that the performance of the DAQV master process is ultimately affected by most of the parameters just described. For example, increasing the number of slave processes imposes additional management responsibilities on the master; changing the data distribution can affect how efficiently the master fills client requests; and increasing the number of simultaneous data requests issued by clients makes more work for the master.

D. Experiments Performed

As described above, our experiments were broken into three main categories: application, the DAQV system, and use scenarios. Our application experiments focused primarily on measuring the overhead incurred by applications using DAQV. First, it is important that applications with DAQV instrumentation have the opportunity to run at a speed comparable to an uninstrumented version when DAQV functionality is not needed; application programmers should not be forced to maintain two versions of their code. Second, when DAQV is being used, application programmers should be aware of the overheads associated with it. We examined the overheads associated with both slave-master and slave-master-client communication. These experiments were performed by measuring the elapsed time of the DAQV calls in a Fortran program by bracketing the calls with timing routines. Particular emphasis was placed on the `DAQV_PROBE` call since it is the most frequently invoked call during the execution of an application.

With respect to the internals of the DAQV system itself, we further categorize our experiments into two additional groups: system profile and data access. First, in the system profile experiments we attempted to quantify the execution time for the various phases of DAQV operation, providing insight on the minimal overhead imposed by DAQV when in full operation. To minimize the perturbation of our results, separate experiments were conducted for each phase of the system. The measurements collected from each experiment were then merged to create an overall profile, or time line, of DAQV operation. Second, the goal of studying our data access algorithms was both to quantify the performance of those algorithms and to show the benefits of running the slave and master processes on multi-processor machines. We performed numerous experiments in these areas. In one experiment, we examined the time for the master to request and receive data back from the slave processes. We varied the size and shape of the array being requested, the number of slave processes used, and whether the master and slaves were running on the same machine or not. In another series of experiments, we explored the behavior of our data access algorithm. The main experiment in this category used an application with a core computational loop whose duration could be varied. At the end of each iteration of this loop, each slave called `DAQV_PROBE` to allow access to a fixed-size array. We varied the number of clients trying to access that data. We created a time-stamped trace of the key events in requesting, accessing, and delivering the data. From the event trace, we computed interval times and were also able to identify the chronological relationships between the multiple threads and processes involved, allowing us to identify, for instance, the amount of overlapped communication and computation achievable by DAQV.

E. Data Collection Methodology

For those experiments where interval measurements were required, we typically performed several trials of each experiment. Furthermore, each trial consisted of several individual measurements. For example, suppose we wanted to measure the data access time for a probe request. We would execute a test application which contained a single, yielding `DAQV_PROBE` call. Our client would then make, say, 50 data requests, and record the time for the master to fill each request. After this, all processes would terminate. This sequence describes a single trial consisting of 50 measurements. We conducted up to four trials for each experiment. For each trial, we threw out the first measurement, as it was usually an outlier when compared to the other readings (probably as a result of operating system and/or cache behavior). At this point, the readings were merged and sorted. Depending on the experiment, additional statistical or numerical analyses were then performed.

An alternate interval timing measurement approach that we chose not to use attempted to minimize the timing perturbation by bracketing a sequence of calls (e.g., `DAQV_PROBE`) with the timing routines. Then, the average behavior was determined by dividing the measured time by the total number of calls in the sequence. Unfortunately, this technique makes it impossible to capture "best case" performance. Since our measurements typically contained many outliers (e.g., caused by context switches, cache effects, and thread scheduling), this technique would not yield meaningful results without more sophisticated statistical analyses that erradicated the effects of those outliers.

For event trace experiments, it is more difficult to coalesce multiple trials of an experiment in order to improve the statistical quality of the data. Each event trace represents a unique execution, and we were typically interested in the exact ordering of events during that execution. Consequently, our results from these experiments serve to demonstrate specific capabilities of the system. We may derive timing information from them, but it is not possible to determine where such measurements fall in the performance spectrum. In some cases, though, the derived measurements can be compared against results derived from other (interval timing) experiments.

V. Performance Analysis

A. Access API Overhead

To use DAQV, an application programmer must instrument his code with the DAQV access routines shown in Table I. If the overhead of calling these routines is minimal, the programmer may decide to leave the instrumentation in when DAQV functionality is disabled. The DAQV system can be disable by setting a user environment variable. The minimum observed overhead in microseconds for calling the access routines when DAQV is disabled is given in Table III. The `DAQV_INIT` call takes 6.9 $\mu secs$ to check the environment variable, but it will only be called once in the program. `DAQV_REGISTER` and `DAQV_PROBE` times are for subroutine calls with a single array. Obviously, these routines may be called multiple times in the program, but the overheads are less than 3.5 $\mu secs$. To put these times into perspective, they are comparable to the overhead of a procedure call, making the overall application impact of disabled DAQV instrumentation negligible.

When DAQV is enabled, all routines interact with the master process in some way, causing the DAQV protocols to come into effect. Here, it is important to set a baseline of the impact of the protocols on performance. Table III shows these baseline overheads for the access API routines, except for `DAQV_PROBE` which we will address shortly. Obviously, the overheads are affected by where the master resides: locally on the same parallel machine as the application, or remotely on a different machine. The main contributors to the overhead during initialization are slave thread creation and Nexus startup[4]. The difference between local and remote overhead is wrapped up in Nexus operation and we were unable to analyze it further. Clearly, remote slave synchronization with the master process over a network adds to the overhead, but this should be comparable to that observed in `DAQV_REGISTER`. Array registration takes longer than in the enabled case because a 450-500 byte structure giving a complete description of the array, including distribution information, gets created and transferred to the master process. The `DAQV_EXIT` overheads may appear excessively large, but this is mainly due to a one-second wait each slave performs to guarantee termination of all slave threads.

The `DAQV_PROBE` and `DAQV_MUTATE` routines can cause data transfers to occur that add to the overall time for routine execution. To establish the baseline for these, we performed experiments that did not involve data transfer (i.e., there is no array request to handle). There are other parameters, however, that affect DAQV operation in this case. In particular, if the probe/mutate is non-yielding, there is no communication with attached tools. In contrast, a yielding probe/mutate requires an explicit continue from one of the attached tools. We show the overhead for these cases in Table IV, with application, master, and tool processes running in different locations; note, probe and mutate cases are equivalent so only probe is shown.

Non-yielding probe operations take minimally 1.465 and 2.872 milliseconds when the master process is local and remote, respectively. Two communication events are involved (see Figure 3): the slaves notifying the master of a probe access, and the master telling the slaves to proceed. As we will see shortly, most of this time is due to Nexus interprocess communication and thread creation. Between the local and remote cases, this time is influenced by network delay. Note, the Nexus library we used for the SGI machine at the time did not support shared memory communication between processes running on the same machine.

For yielding probes, we have different configurations to consider because the client tools are now involved; here we use just one tool. When all processes are on the same machine, overhead increases by 2.263 milliseconds compared to the non-yielding case due to the additional master-client communication. When the communication is remote, additional network overhead results, as seen; in the cases shown, tool locality is with respect to the master. However, not all additional overhead is due to the network communication. For instance, the tool uses a query routine in the request API to check to see if a continue is needed. But this routine is non-blocking and the tool effectively polls for the event to arrive, repeatedly checking a conditional variable under lock to do. Unfortunately, this locking can compete with the client thread which handles the event and sets the condition for which the tool is waiting. Evaluating these types of asynchronous, distributed interactions is extremely difficult, especially without instrumenting the Nexus communication and thread library directly.

B. Probe Profile

Despite these difficulties, we felt it was important to separate DAQV performance artifacts from those that depend on the underlying communications infrastructure. For this purpose, we designed experiments to measure events in the application and master processes during the operation of a non-yielding probe. Figure 4 shows the events in the application and master processes that we measured. By comparing times for different event intervals (e.g., B-F measures the time for a slave to notify the master process) we can determine the cost of each step in the interaction. Again, we are focusing on establishing a performance baseline; no data is being sent, there is only one application process, and it is running on the same machine as the master.

Figure 5 shows timing results for event pairs in blocks with the minimum time value reported. Each event pair measurement was conducted separately from the rest. The first column clearly shows the largest performance costs to be those that involve interprocess communication between the application and the master (B-D, G-I). However, there is a 100 to 200 microsecond cost for thread related synchronization in both the slave and master (E-F, J-K). This performance depends on the implementation of Nexus

[4]For experiments reported in Table III, the application was run with one process only.

thread synchronization. Note, once the master process is notified of the probe it can check for pending requests and determine that the non-yielding probe can continue in 6 microseconds (F-G). In fact, all the other code sections (A-B, D-E, I-J, K-L) relate to managing probe access requests and generate very little overhead.

Moving to the second column, we isolate the two intervals in which Nexus communication is contained (B-F, G-K). The main conclusion is that it costs about 650 microseconds to communicate a probe access to the master, and the same for the master to communicate a continue response to the slave. Of the time, approximately 73% is in Nexus operations to deliver a probe to the master or a continue to the slave. The last column shows the performance for the entire probe access, from the beginning of the `DAQV_PROBE` call until its termination (A-L). The 1,444 microseconds value corresponds to what we measured in the non-yielding, all-local case in Table IV. This cost represents the minimum overhead for invoking DAQV in the application to provide for tool data access.

C. Data Access – Single Process

The performance of data access in the DAQV system will be determined by three factors: distribution processing, buffer construction, and data communication. We performed a series of experiments to understand the relative influences of these factors. Of particular interest to us was the effect of parallelism on the overall performance of data access in the application and master process. Because data communication will ultimately limit performance when tools are remote from the application, we wanted to demonstrate that parallelism could reduce distribution processing and buffer construction to relatively insignificant levels in this case. On the other hand, when processes are local with respect to each other, higher bandwidth communication should result and we would like parallelism to allow DAQV to access the full potential of the memory system of the multiprocessor machine.

To set baseline performance as before, we begin with data access experiments for a single application process. We isolate the measurement interval from when the master sends the tool request to the slave to when the master tells the slave to continue; a tool probe request is assumed to be waiting. Array size is varied from 100 to 1,000,000 elements, with different ranks, 1D to 3D (N, NxN, and $NxNxN$ structure), and the master process is running local and remote to the application. Figure 6 shows results on a log-log plot.

The first remark to make is that there is a clear performance difference between local and remote, primarily due to network communication. Interprocess communication (i.e., TCP/IP) is used in both cases, but when the slave and master are on the same SGI machine, data flows down to the IP level of the TCP/IP stack, then gets diverted to the loopback interface at the network driver level; no hardware driver software is invoked and the physical 100 Mb/s network is not used. Second, for a single process, the results show that rank is relatively unimportant. The reason for this is that all the data is stored in a contiguous space in the application and in the master client buffer. Thus, buffers get filled with just a single memory copy, since the entire array, in this case, is being requested[5]. Third, the inflection point at 10,000 elements is explained by the fact that below this point the fixed DAQV protocol overhead has a greater effect on performance, whereas after, buffer copying and data communication delays are more significant, growing linearly with increasing array size.

D. Data Access – Multiple Processes

Our initial investigation with multiple processes looked at the effect of parallel threads on buffer copying and data communication. In the results shown in Figure 7, we fixed the array size at 500x500 (250,000 elements, 1,000,000 bytes total) and used a BLOCK distribution in each dimension. The abstract processor grid was varied, changing the number of processes used and the assignment of array elements. Local and remote measurements were performed using the same timing interval as in the single process experiments.

It should first be noted that a separate thread is dynamically created in the master to handle communication with a slave thread (one per slave) and to write data received to the client buffer. Thus, if there are four processes for the application, implying four slaves, four master threads will be active in communication operations; the communication operations will be concurrent. In the local case, all eight threads (slave plus master) will be running together on the same machine, whereas in the remote case, the master process and threads are on a different machine, but network communication results.

When there are multiple slaves, we hope to see a performance improvement overall. In Figure 7 we see this improvement, particularly with respect to the local case. When slave and master are local, data access, communication, and buffer filling each can occur in parallel. We see good speedups up to four application processes (four slave threads and four master threads) and then a leveling off with eight processes. The reason, we suspect, has to do with two factors: memory system effects in the SGI machine and scheduling of multiple threads. In the remote case, the performance of concurrent communications can suffer from sequentialization due to sharing of the 100 Mb/s network interface. Any performance improvement in this case comes from parallel slave data access and parallel client buffer filling[6]. We see the performance improvement in the remote curves up to four processes with no improvements beyond, most likely due to master thread scheduling on the remote machine.

Figure 7 shows curves for different processor grids. In general, the assignment of array data to processes changes with different grids, affecting the characteristcs of data ac-

[5] Note, there will be some performances effect of rank when subsections of an array are accessed because in ranks greater than 1, elements accessed may not be contiguous.

[6] Because of the communications sequentialization, the client buffer filling is pipelined.

cess and client buffer filling. Since we are transfering the entire array in this experiment, data access is done in each slave via a single memory copy to the Nexus communication buffer. However, during client buffer filling, multiple memory copies may occur. We see minor performance variation between grids of the same number of processors with a maximum of 8% difference between the 2x4 and 4x2 case.

D.1 Data Access Algorithm

We cannot tell directly from Figure 7 the relative effects of parallelism in data access, communication, and client buffer filling. Moreover, these effects change depending on array size and distribution, since different aspects of the machine will be affected. We sought to isolate the effects while comparing "best case" and "worst case" access scenarios. Best case access occurs when each slave can copy data to a Nexus communication buffer in a single `nexus_put_TYPE` operation[7]. Worst case access occurs when each element must be copied separately, losing the advantage of large data moves. Our experiments selected array sizes from 10^4, 10^5, 10^6, and 10^7 elements for best and worst cases, varying the number of application processes from 1 to 8. We present speedup results for all application and DAQV processes running on the same 8-processor SGI machine.

We first focussed on the performance improvements in data access due to parallelism. Ideally, when slaves receive the request sent from the master they will begin data access simultaneously; this is a reasonable assumption in our experiment. Figures 9 and 8 show data access speedups for best and worst cases, respectively. In the best case, we observe the performance effects of the SGI memory system when each slave performs a single memory copy. The superlinear speedups in the 10^5 curves in both cases is very likely due to cache effects (i.e., caches misses occurring in the one process case and less with more processes). In the 10^4 case, all data elements fit in cache for any number of processes and ideal speedup is seen. When a large amount of data is accessed in the best case, performance depicts how well the memory system can accommodate the requests. The peaking of speedup at 4 processes suggests that the memory system suffers saturation when 8 processes are used. In the worst case, we get near ideal speedup because the rate of hardware memory copy operations is less due to the overhead of calling `nexus_put_TYPE` multiple times.

D.2 Full Probe Path

When we measure the full probe path, the speedup results show performance degradation cause by interprocess communication; see Figures 11 and 10. Taking the worst case first (Figure 11, speedups are still observed because the data access component represents a significant portion of the time; roughly 50% in most experiments. We did not expect ideal speedups because of overheads in Nexus communication. The best case allows us to observe the effects of these overheads more directly. In Figure 10, speedups do not rise above 1.6 and, in fact, slowdowns occur for 10^4 and 10^5 (8 processors) array sizes. There are two reasons for this behavior. Additional overhead is incurred for invoking TCP/IP communication. This overhead is fixed and independent of array size and, thus, represents a larger percentage of the total time for smaller arrays. This accounts mostly for the slowdown seen.

The overhead is less significant with larger data accesses, but contention in communication begins to intrude. Based on the above discussion of TCP/IP implementation in the SGI machine, we suspect that communication between processes on the same machine are pipelined due to possible synchronization in TCP/IP stack processing. More seriously, however, is the fact that Nexus currently performs socket reading and writing single threaded. Although it provides a socket for each *pair* of communicating processes (e.g., each slave has a separate socket with the master), only one thread reads from the sockets when data is available. The general result is a limiting of performance in slave to master communication and, subsequently, client buffer filling. The speedup limit will be determined by the relative amount of time spent in synchronization and memory copying, and by the additional thread scheduling overhead that arises. Unfortunately, we did not have the low-level instrumentation support to fully investigate Nexus and TCP/IP communication performance, nor was that our intent necessarily. But here we conclude that performance improvements in the full probe path will come primarily from improvements in communications performance; see Section §VI for more discussion. In the next section we investigate further the performance of client buffer filling.

D.3 Client Buffer Filling

When the master receives data from the slaves, those data represent slave-local sections of the array request. These must be assembled into a global buffer to be sent back to the client tool. On the slave side, a "gather" operation is performed, reading (possibly) non-contiguous pieces of slave-local data and packing them into the dense, contiguous buffer that is sent to the master. Once the master receives a buffer from a slave, a "scatter" operation occurs to unpack the slave data into the appropriate (possibly) non-contiguous subsection(s) of the client buffer. The algorithm to fill the client buffer is nearly identical to the slave data access algorithm. As before, we observed the best and worst case behavior for client buffer filling; only the worst case is reported below. In the best case, a single, contiguous memory copy is performed. In the worst case, many memory copies are required. In our experiments, we gathered timestamps beginning from the time the first master thread begins to fill the client buffer to the time the last master thread finished. The array sizes selected were $4*10^4$, $4*10^5$, and $4*10^6$ bytes.

Table V shows the worst case performance results for four master threads performing client buffer filling. The buffer numbers represent the data received from individual slaves. The time required to "scatter" buffer data is given

[7]A pointer to the data and the byte length is given the procedure, and, we assume, Nexus performs a single memory copy operation to move the data.

in $\mu secs$ along with the percentage of the actual time for client buffer filling. The sum of the buffer scatter times is shown and an overlap value computed as $Sum/Actual$ indicating a degree of simultaneous client filling (a value of 1.00 means no overlap occured; a value of 4.00 means there was full overlap). We first notice that client buffer filling is slower than slave data access. There are two reasons for this. The master begins filling as soon as the first buffer is received. However, due to the pipelining effect discussed above, buffer receipt is staggered. Also, scatter operations reek havoc with the memory system, particularly in the cache where write conflicts on the lines cached from the client buffer can occur.

As the array size increases, more overlap (i.e., parallelism) is seen in client buffer filling. The slave data is still being received staggered, but the time between the first and last slave buffers being received is small compared to how long it takes to write them to the client buffer. For an array size of $4 * 10^6$, a nearly total overlap of 3.67 is achieved.

E. Client Interaction

Client interaction with the parallel application using the DAQV interaction system can take many different forms, depending on the number of clients involved, where the clients are located, the size of the data interactions, and the interaction frequency, to name a few factors. The scenario that we selected to study involves a 4-way parallel application that performs 100 iterations of a loop that first "computes" then executes a yielding probe. The "compute" part actually sleeps a specified number of seconds (0.5, 1.0, 1.5, 2.0, 2.5, and 3.0 seconds) allowing us to control the ratio of computation to DAQV overhead. Experiments were performed with 1 and 2 clients each requesting a 5 million element array during each probe. All processes (application, master, and client) were executing on the same 8-processor SGI machine that was used in the other experiments.

We believe a DAQV user should have performance data that relates to their interaction needs. The experiment reported is representative of a range of experiments that could be done to determine DAQV overhead in an application context where choices of the frequency and size of interaction are of concern. Here probe requests are for a relatively large amount of data. The concern might be with the computation granularity such that per iteration probes do not degrade execution time significantly. For instance, a user might use a slowdown of two times as an acceptable criteria.

Figure 12 compares three experiments where the computation granularity varies from 0.5 to 2.5 seconds. In each case, a snapshot of one iteration out of the 100 iterations performed is shown with intervals presented for application, slave/master, and client operations. One can see the overlap of master to client communication with application computation. But because yielding probes are used, the application will wait at the `DAQV_PROBE` call until the client process issues a continue; the client then makes another probe request. In the 0.5 second case, the computation granularity is too small relative to the size of the probe request and the application suffers a delay while waiting for the next client requests, in addition to the probe access overhead. The slowdown for this case is approximately 4.5 times. In contrast, the client receives the probe data well before the end of the computation loop in the 2.5 second case; the slowdown is approximately 1.6 times. When the compute time is 1.5 seconds, client communication is balanced with application computation and the slowdown is approximately 1.8 times. Because communication from the master to a client is sequential, this communication is the dominant factor in application slowdown.

With two clients, DAQV's ability to perform concurrent client communication is observed; see Figure 13 for 2.5 seconds of compute time. Since each client is requesting all 5 million array elements, twice the amount of data transferred and the probe access time doubles as expected. However, the efficiency of client communication improves due to parallel data transfer, shifting the dominant factor in application slowdown to the data access; the slowdown is also approximately 1.8. Simply put, doubling the number of clients in this case does not result in a doubling of the computation time needed to maintain a desired slowdown. This demonstrates good scaling characteristics of the DAQV system.

VI. Conclusions and Future Work

The DAQV program interaction system meets the functional objectives described in Section §III. DAQV has been successfully applied in building a domain-specific environment for seismic tomography [22], a tool interaction system for coupling parallel simulations [23], and a online performance monitor [24]. In each of these cases, the functional capabilities of DAQV were significant in reducing the complexity of code development by providing an interaction abstraction that was easy to understand and apply.

Certainly, performance was also a concern in each case and will be so in any future DAQV application. We believe that an extensive performance study of the DAQV system (or any program interaction system, for that matter) is important so that users can weigh functional and non-functional requirements when deciding to adopt the DAQV technology. The performance analysis in this paper is a step in that direction. Not only have our experiments demonstrated performance behavior that would undoubtedly be of interest to users, it has also prompted us to investigate performance bottlenecks in more detail to find possible design and/or development alternatives. The tradeoff for us, DAQV's designers, is in the functional robustness of the system vs. the choice of technologies to meet non-functional objectives, mainly performance, and the functional limitations they impose.

Other interaction systems that have close similarity with DAQV face similar design challenges and tradeoffs. The PAWS software infrastructure is used for the dynamic coupling of parallel applications in a component-style model [19]. Like DAQV, it uses Nexus for its underlying communication, but implements a data interoperability model that is more dataflow in operation. Parallel data transfer is important in PAWS to avoid serialization bottlenecks and the PAWS API explicitly represents accessible parallel data structures so that communication schedules can be computed. Dataflow operation and explicit data representation can lead to high-performance interaction, but may be limited for other interaction needs. For instance, PAWS is not focused on issues of computational steering.

The Magellan computational steering system [25] is a program interaction system with a server-master-client architecture very similar to DAQV. Interestingly, Magellan's designers sought to improve the performance of the system by optimizing how steering commands are performed and reducing the amount of data that must be transferred to steering tools. The approach emphasizes interaction with program operation and execution to improve steering frequency and therefore concentrates on improving performance of interpreting steering commands and processing monitored data close to the application. A different balance between functional and non-functional requirements is reached in Magellan than DAQV or PAWS because of different application objectives and system environment targets.

Improving DAQV's capabilities and performance must continue to balance objectives. For instance, we are considering extending program state interactions by allowing different types of distributions and relaxing the restriction of accessing only regular, dense subsections of registered arrays. Doing so will undoubtedly complicate array processing, but this portion of the DAQV overhead achieves very good performance and the extra work will not seriously affect performance overall. Although not discussed in detail, CYCLIC distribution handling was difficult to optimize and can easily degenerate into poor-performing access behaviors. Consideration of other distributions should include the susceptibility of poor access patterns. Also not discussed was the mutate access type which has performance considerations similar to probe. Both probe and mutate access may be extended to work with distributions that regard data as being replicated on each application process.

When functional design affords flexibility in system implementation, performance optimizations can be found. We continue to improve DAQV's performance in protocol implementation and communications management. We hope to see a significant performance when using Nexus shared memory communications between slave and master processes on the same machine. It may also be possible to replace the Nexus runtime system layer with a higher-performing system such as Tulip [26]. However, doing so may restrict the system environments on which DAQV runs. Smarter request handling can also offer improvements. For instance, detecting identical client probe requests (i.e., probe requests for the same array subsection) could eliminate redundant data accesses – the master needs to request the data only once and then copy the data to the requesting clients. We could also extend request types to include a "probe with continue" request that effectively eliminates the extra communication between master and slaves to issue a continue command.

Program interaction systems will become more commonplace as distributed computing use increases. We suspect that different forms of interaction systems will develop depending on functional and non-functional objectives in their application. Providing users with comprehensive information about interaction system performance will continue to be a challenging endeavor.

Acknowledgments

This work was supported in part by the Department of Energy ASCI program (Contract No. C70660017-3) and DOE 2000 program (Agreement No. DEFC 0398 ER 259 986), and a National Young Investigator grant (NSF ASC-9457530) from the National Science Foundation. We would like to thank the Los Alamos National Laboratory for their sponsorship of the DAQV project. In particular, Jeff Brown, Pete Beckman, and William Humphreys of LANL were helpful in providing technical feedback. Lastly, the authors sincerely acknowledge the invaluable contribution of Chad Busche of the Department of Computer and Information Science, University of Oregon, in the implementation of the DAQV system.

References

[1] R. Stallman and R. Pesch, "Debugging with gdb," Tech. Rep., Free Software Foundation, 1995.

[2] D. Cheng and R. Hood, "A portable debugger for parallel and distributed programs," in *Proc. Supercomputing '94*. ACM, Nov. 1994, pp. 723–732.

[3] M. Meier, K. Miller, D. Pazel, J. Rao, and J. Russell, "Experiences with building distributed debuggers," in *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, May 1996, pp. 70–79.

[4] C. Pancake and R. Netzer, "A bibliography of parallel debuggers," *SIGPLAN Notices*, vol. 28, pp. 169–186, Dec. 1993.

[5] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simmons, and R. Title, "A scalable debugger for massively parallel message-passing architectures," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 2, no. 2, pp. 50–56, Summer 1994.

[6] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, and O. Wolf, "Event and state-based debugging in tau: A prototype," in *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, May 1996, pp. 21–30.

[7] L. Hicks and F. Berman, "Debugging heterogeneous applications with pangaea," in *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, May 1996, pp. 41–50.

[8] R. Hood, "The p2d2 project: Building a portable distributed debugger," in *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*. ACM, May 1996, pp. 127–136.

[9] T. Ludwig, R. Wismuller, M. Oberhuber, and A. Bode, "An open interface for the online monitoring of parallel and distributed programs," *Int'l. Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, 1997.

[10] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T Newhall, "The paradyn parallel performance measurement tools," *IEEE Computer*, vol. 28, no. 11, Nov. 1995.

[11] R. Ribler, J. Vetter, H. Simitci, and D. Reed, "Autopilot: Adaptive control of distributed applications," in *Proc. of the Seventh Int'l. Symp. on High Performance Distributed Computing (HPDC7)*. IEEE, July 1998.

[12] J. Hollingsworth and P Keleher, "Prediction and adaptation in active harmony," in *Proc. of the Seventh Int'l. Symp. on High Performance Distributed Computing (HPDC7)*. IEEE, July 1998.

[13] R. Haimes, "pv3: A distributed system for large-scale unsteady cfd visualization," in *Proc. American Institute of Aeronautics and Astronautics*, 1994.
[14] A. Tuchman, D. Jablonowski, and G. Cybenko, "Runtime visualization of program data," in *Proc. Visualization '91*. IEEE, 1991, pp. 255–261.
[15] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter, "Falcon: On-line monitoring and steering of large-scale parallel programs," in *Proc. Frontiers '95*, 1995, pp. 422–429.
[16] G. Geist II, J. Kohl, and P. Papadopoulos, "Cumulvs: Providing fault tolerance, visualization, and steering of parallel applications," *Int'l. Jour. of Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, pp. 224–235, 1997.
[17] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber, "Vase: The visualization and application steering environment," in *Proc. Supercomputing '93*. ACM, Nov. 1993, pp. 560–569.
[18] W. Gu, J. Vetter, and K. Schwan, "An annotated bibliography of interactive program steering," *SIGPLAN Notices*, vol. 29, no. 9, pp. 140–148, 1994.
[19] P. Beckman, P. Fasel, W. Humphrey, and S. Mniszewski, "Efficient Coupling of Parallel Applications Using PAWS," in *Proc. 11th Int'l. Symp. on High-Performance Distributed Computing,*. IEEE, July 1998.
[20] High Performance Fortran Forum, "High performance fortran language specification, ver. 1.0," Tech. Rep. CRPC-TR92225, Center for Research on Parallel Computation, Rice Univ., 1993.
[21] I. Foster, C. Kesselman, and S. Tuecke, "The nexus approach to integrating multithreading and communication," *Jour. of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 70–82, Aug. 1996.
[22] J. Cuny, R. Dunn, S. Hackstadt, C. Harrop, H. Hersey, A. Malony, and D. Toomey, "Building domain-specific environments for computational science: A case study in seismic tomography," *Int'l. Jour. of Supercomputing*, vol. 11, no. 3, pp. 179–196, Fall 1997.
[23] C. Harrop, S. Hackstadt, J. Cuny, A. Malony, and L. Magde, "Supporting runtime tool interaction for parallel simulations," in *Proc. Supercomputing '98*. ACM, Nov. 1998.
[24] S. Shende, S. Hackstadt, and A. Malony, "Dynamic performance callstack sampling: Merging tau and daqv-ii," in *Proc. of the Fourth Int'l. Workshop on Applied Parallel Computing (PARA98)*, June 1998.
[25] J. Vetter and K. Schwan, "High performance computational steering of physical simulation," in *Proc. IPPS '97*, 1997, pp. 128–132.
[26] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine, "Hpc++ and the hpc++lib toolkit," Tech. Rep., Dept. of Computer Science, Indiana Univ., 1998.
[27] S. Hackstadt, C. Harrop, and A. Malony, "A framework for interacting with distributed programs and data," in *Proc. of the Seventh Int'l. Symp. on High Performance Distributed Computing (HPDC7)*. IEEE, July 1998.
[28] S. Hackstadt and A. Malony, "Daqv: Distributed array query and visualization framework," *Jour. of Theoretical Computer Science*, vol. 196, no. 1-2, pp. 289–317, Apr. 1998.
[29] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "Integrated pvm framework supports heterogeneous network computing," *Computer in Physics*, vol. 7, no. 2, pp. 166–175, Apr. 1993.
[30] Message Passing Interface Forum, "Mpi-2: Extensions to the message-passing interface," Tech. Rep., Univ. of Tennessee, 1997.

`DAQV_INIT(int *procId, int *numProcs)` Initializes process `procId` of an application with `numProcs` processes.
`DAQV_REGISTER(void *array, int *rank, int dim[], int *type, char *name, int distType[], int distParam[], int *procRank, int proc[])` Registers distributed data contained in `array`, whose rank, dimensions, type, and textual name are also provided; `distType[]` and `distParam[]` describe the distribution in each dimension onto an abstract processor grid of rank `procRank` with dimension contained in `proc[]`.
`DAQV_PROBE(int *type, void *array1, void *array2, ...)` Fills pending probe requests for any of the arrays listed; if `type` is yielding,then the call waits for additional requests from clients and does not return until explicitly told to do so.
`DAQV_MUTATE(int *type, void *array1, void *array2, ...)` Same as probe, except the arrays listed can be mutated.
`DAQV_EXIT()` Cleans up and terminates DAQV access operation.

TABLE I
Application Data Access Procedural Interface

Category	**Parameter**	**Values**
Global	location of execution	local, remote
Slaves	array rank	1D, 2D, 3D
	array size	10^3 - 10^7 elements
	# processors	1-8
	data distribution	various BLOCK
	duration of loop	0.25 - 3.00 secs
	threading	multiple, single
Clients	# of clients	1, 2
	# of requests	1, 4

TABLE II
Experimental Parameters

Routine	**Overhead** (*msecs*)		
	Disabled	**Enabled**	
		Local	*Remote*
`DAQV_INIT()`	.008	121.022	198.267
`DAQV_REGISTER()`	.003	2.428	4.780
`DAQV_PROBE()`	.003	*	*
`DAQV_EXIT()`	.002	1,004.250	1,004.941

* see Table IV

TABLE III
Application Access API Overhead with DAQV Enabled/Disabled

Type	**Overhead** (*msecs*)	
	Master local	Master remote
Non-yielding	1.465	2.872
Yielding		
w/ tool local	3.728	5.811
w/ tool remote	4.626	8.402

TABLE IV

Probe Access API Overhead

Size	**Buffer**	**Time** ($\mu secs$) **(% Actual)**
$4 * 10^4$	1	36,500 (73%)
	2	24,633 (49%)
	3	13,087 (26%)
	4	12,597 (25%)
	Sum	86,816
	Actual	50,287
	Overlap	1.73
$4 * 10^5$	1	421,700 (74%)
	2	512,760 (90%)
	3	521,141 (91%)
	4	484,232 (85%)
	Sum	1,939,832
	Actual	572,631
	Overlap	3.39
$4 * 10^6$	1	6,648,635 (94%)
	2	6,509,077 (92%)
	3	6,593,553 (94%)
	4	6,101,742 (87%)
	Sum	25,853,007
	Actual	7,044,857
	Overlap	3.67

TABLE V

Worst Case Client Buffer Filling

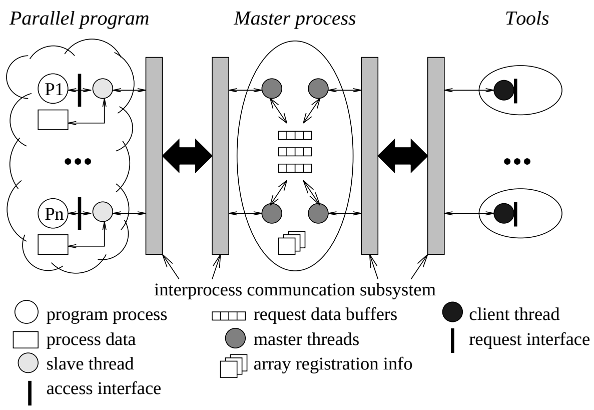


Fig. 1. DAQV Architecture

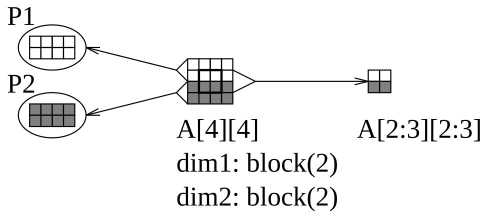


Fig. 2. Distributed Array Access

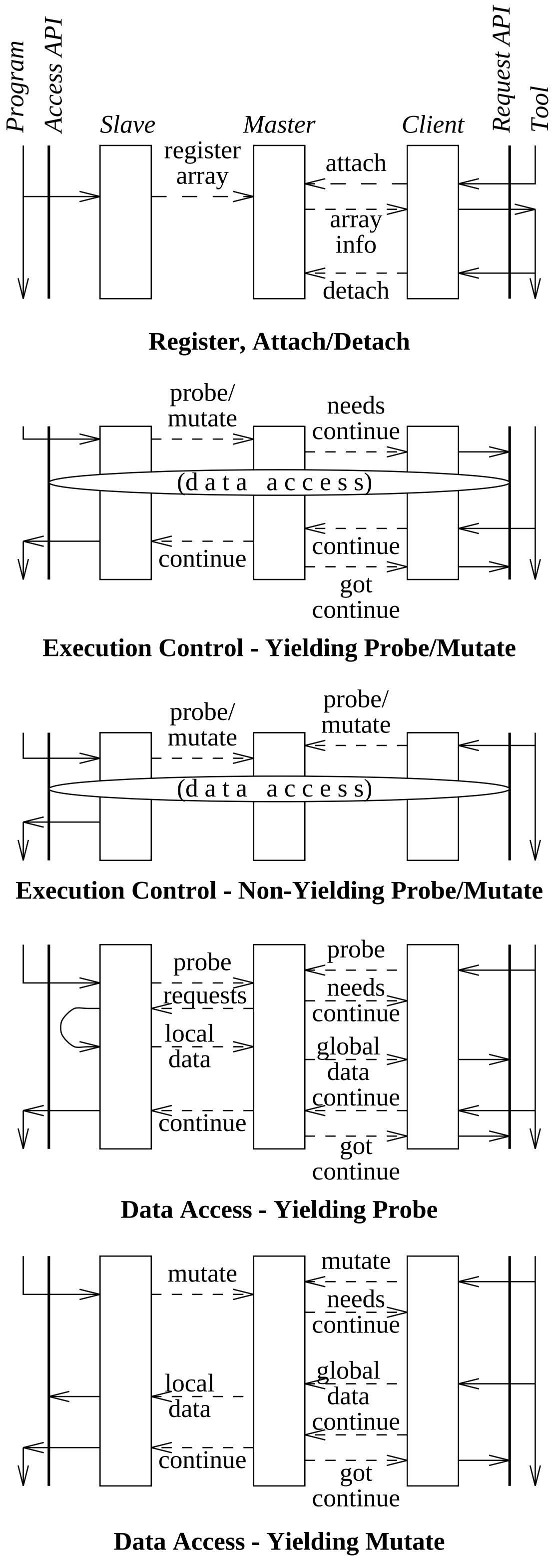
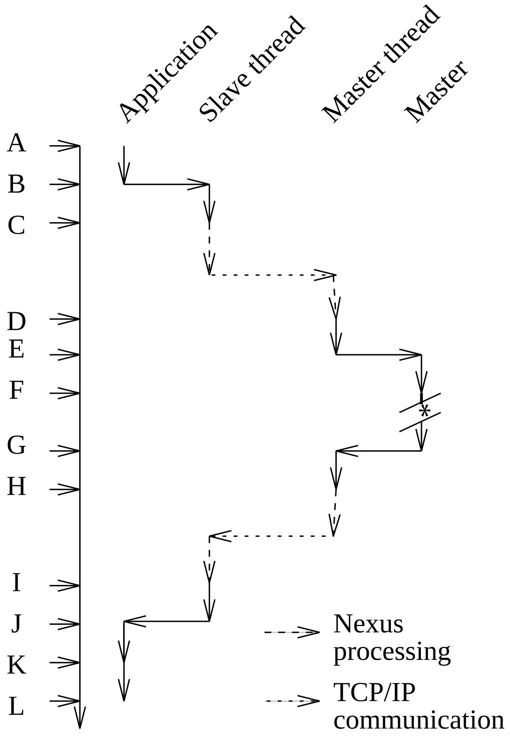


Fig. 3. DAQV Protocols



A	beginning of DAQV_PROBE call
B	prior to sending probe access to master
C	begin Nexus communication slave-to-master
D	a master thread receives probe access
E	prior to signalling master process
F	master process notified of probe
G	prior to sending continue to slave thread
H	begin Nexus communication master-to-slave
I	slave thread receives continue
J	prior to signalling application to continue
K	application notified of continue
L	end of DAQV_PROBE call
*	processing of client requests

Fig. 4. Non-yielding Probe Event Timeline

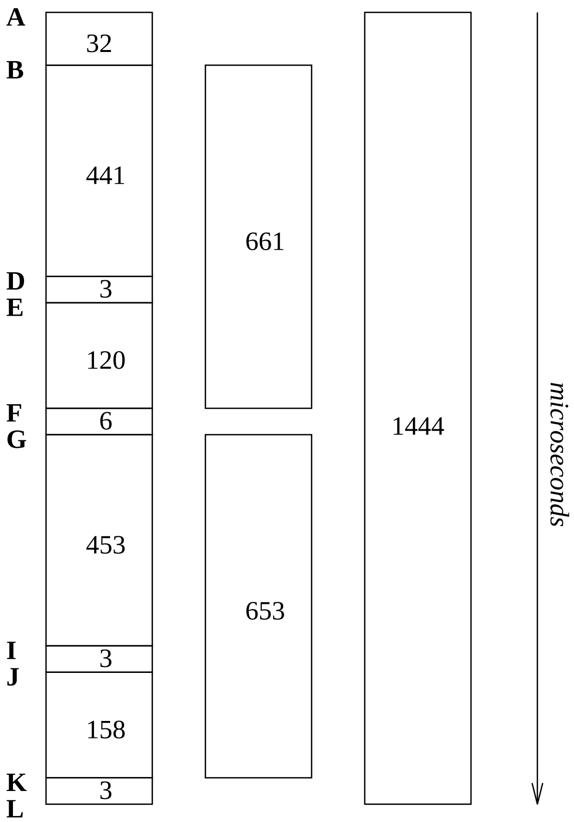


Fig. 5. Non-yielding Probe Performance Profile (letters denote events shown in Figure 4

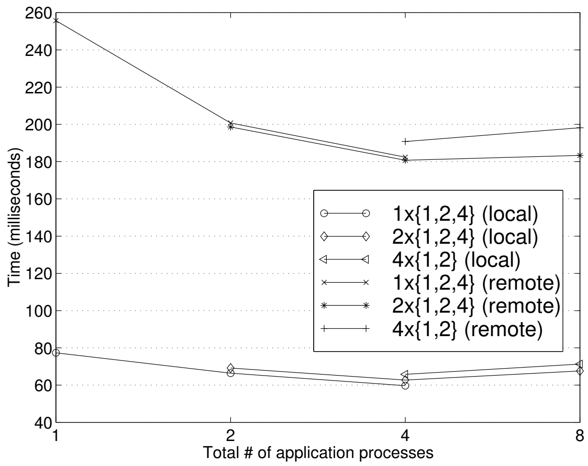


Fig. 7. Multiple Process Data Access for 500 x 500 Array

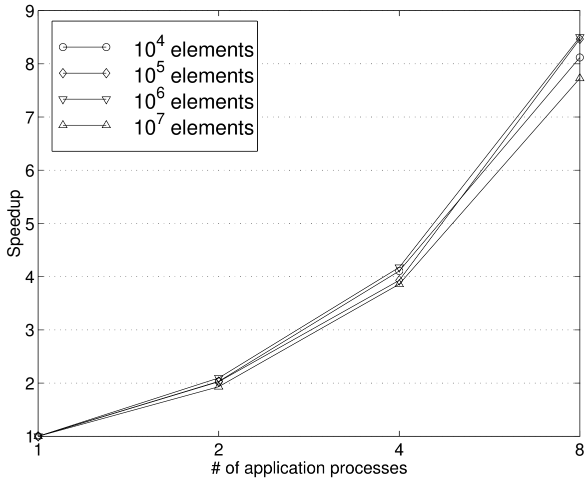


Fig. 8. Worst Case Data Access

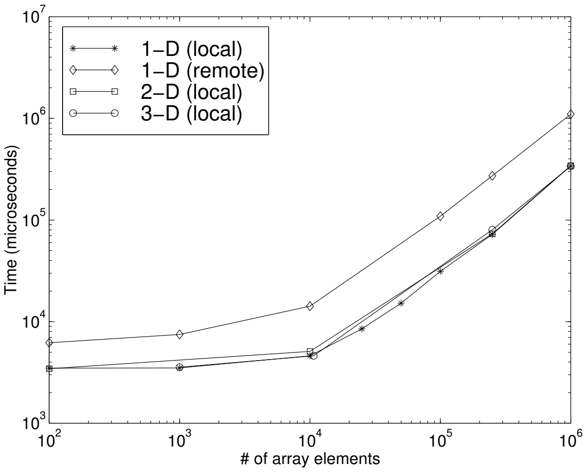


Fig. 6. Single Process Data Access

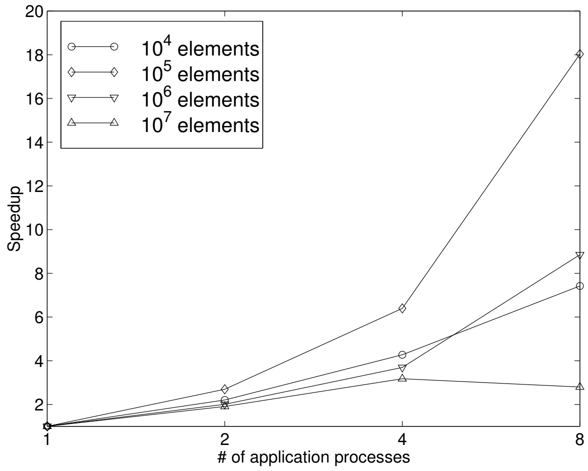


Fig. 9. Best Case Data Access

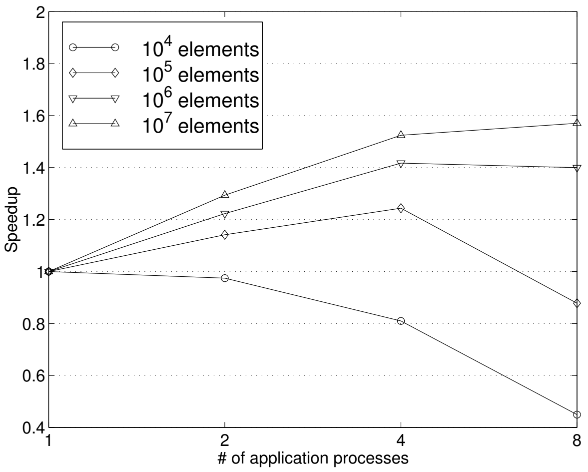


Fig. 10. Probe Performance – Best Case Data Access

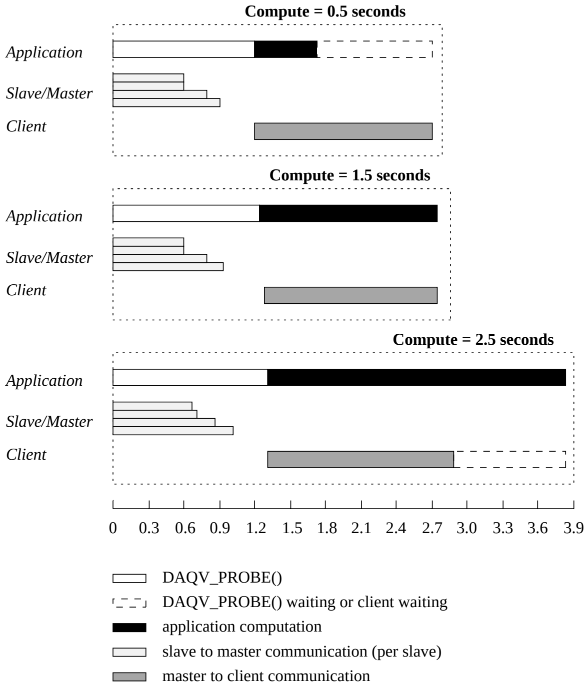


Fig. 12. One Client with Varying Computation Granularity

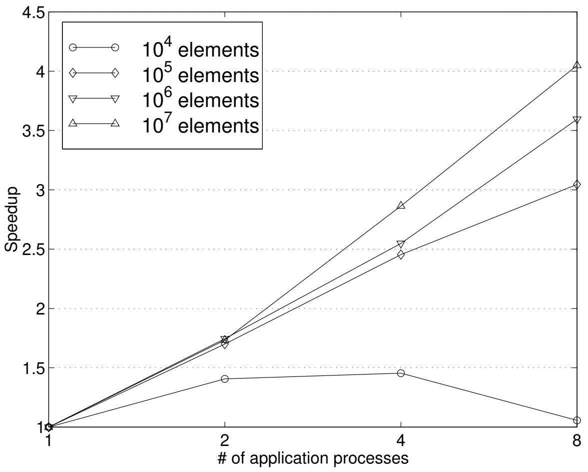


Fig. 11. Probe Performance – Worst Case Data Access

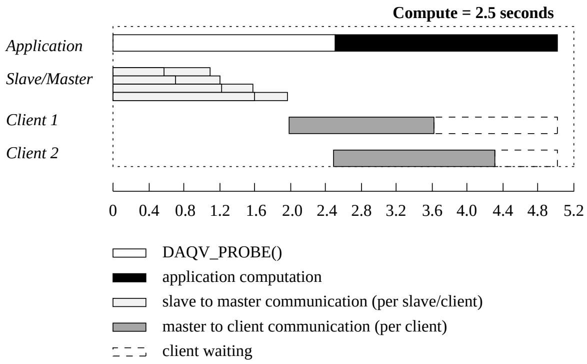


Fig. 13. Two Clients with Varying Computation Granularity