From Interactive Applications to Distributed Laboratories

Beth Plale Greg Eisenhauer Karsten Schwan
Jeremy Heiner Vernard Martin Jeffrey Vetter*

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Distributed laboratories are environments where scientists and engineers working in geographically separated locations share access to interactive visualization tools and large-scale simulation computations, share information generated by such instruments, and collaborate across time and space to evaluate and discuss their results. The intent is to permit scientists, engineers, and managers at geographically distinct locations (including individuals telecommuting from home) to combine their expertise in solving shared problems by allowing them to simultaneously view, interact with, and steer sophisticated computation instruments executing on high performance distributed platforms. This paper reports on research efforts being undertaken at Georgia Tech that address the topic of distributed laboratories:

- Steering and monitoring tools and infrastructure used in the online observation and manipulation of two scientific computations developed jointly with end users.
- Middleware to transport the on-line monitoring and steering events. Can be changed dynamically to adjust event streams to current system loads and monitoring/steering needs.
- Visualization support to permit the definition of appropriate visual abstractions and the efficient representation on 2D and 3D graphical displays.
- Collaboration infrastructure and abstractions.

Keywords: distributed laboratories, interactive program steering, on-line monitoring, collaborative agents, complex scientific applications

1 Introduction

Motivation. Difficult research and engineering computations can be made more effective and efficient if users can easily contribute to the problems they address. One purpose of this paper is to demonstrate the potential increases in functionality and performance gained by the online interaction of end users with high performance *computational instruments* on single and networked parallel machines. Namely, we consider systems in which users interact with computations as if they were physically accessible laboratory instruments. Moreover, we consider systems in which entire *distributed laboratories* are constructed from sets of such computational instruments. Distributed laboratories are environments where scientists and engineers working in geographically separated locations share access to interactive visualization tools and large-scale simulation computations, share information generated by such instruments, and collaborate across time and space to evaluate and discuss their results. Within this context, our intent is to facilitate both (1) online interactions with single computational instruments and (2) interactions among multiple scientists and multiple instruments located at physically distributed sites where scientists may have dissimilar areas of expertise. For example, an atmospheric scientist working with a global chemical transport model, modeling the dispersion of chemical species through the atmosphere may enlist the help of a chemist at a remote research university to explain an observed phenomenon. The atmospheric scientist working locally starts the model,

*The Email addresses of the authors are: {beth,eisen,schwan,heiner,vernard,vetter}@cc.gatech.edu

brings up a visualization/steering interface, then contacts the chemist who in turn brings up a visualization suited to understanding model output from a chemical perspective. The atmospheric scientist and chemist may take turns steering the model, changing parameter values, viewing the effects of such changes, rolling back the model to earlier points in the simulation and trying a different set of parameters until a resolution is found. Obviously, a complex physical model such as this is likely to be implemented via a parallel or distributed system to achieve the necessary performance. However, extensive application interactivity has its own computational demands which must be met. When the possibility of multiple spatially separate users is considered, the display and interactivity system itself becomes complex, dynamic and computationally demanding distributed system. The principal goal of our research is to identify the unique demands of the interactivity system and to develop a software infrastructure that meets those demands.

This paper first explores the demands of a single user interacting with a single computational instrument during its execution, typically referred to as *interactive steering* [13][‡]. Interactive steering ranges from (1) rapid program changes performed by online algorithms using built-in, custom monitoring and facilities [16], to (2) changes performed between iterations of a program or when certain well-defined system states are reached[2], to (3) gradual changes to long-running programs performed by human users via user interfaces[23, 9, 12, 14].

The second part of this paper explores the use of multiple computational instruments by sets of end users, thereby moving from issues addressed in previous work toward interesting topics in future research. Specifically, interactivity with single instruments has been studied previously often using ad hoc, customized, and application-specific mechanisms and steering interfaces. Similarly, algorithmic program steering has been previously applied to improving the operation of single programs[2] or specific subsystems of larger applications.

Contributions. With our research and with the larger-scale *Distributed Laboratories* project at the Georgia Institute of Technology [11], we aim to improve the state of the art of interactive high performance computing for parallel and distributed applications on the variety of heterogeneous platforms now in common use by HPC users and researchers. Particularly, our goal is to develop a general framework for enhancing the interactivity of high performance applications. To realize the goal, we are pursuing research on several key components of the framework:

- *Online monitoring and steering agents* – mechanisms by which high performance applications may be inspected and/or steered at runtime by algorithms, human users, or both.
- *DataExchange and PBIO* – two components working together to provide a flexible infrastructure for transport across heterogeneous parallel and sequential target machines.
- *Collaborative agents* – mechanisms to facilitate interactions between multiple users needing to share one or more computational instruments. Support is provided through high level collaboration abstractions and through lower-level mechanisms that prevent deleterious interactions.

Research context. The local impetus for this research and the context in which it is being evaluated are provided by several high-performance applications developed jointly with end users, including a global atmospheric model, an interactive molecular dynamics simulation, and distributed discrete event simulations. Targeting important problems in science and engineering has required a collaborative, interdisciplinary process involving application scientists, experts in parallel high performance and distributed computing, computer networks, visualization specialists, and experts in user interfaces.

The computing platform utilized in our work, and shown in Figure 1, is but a small-scale example of the high performance computing infrastructure now being envisioned in the U.S. and elsewhere. Specifically, in Georgia Tech's College of Computing, an equipment grant provided by the National Science Foundation [11] has resulted in a heterogeneous infrastructure consisting of: (1) two computing servers (an SGI PowerChallenge multiprocessor and a cluster of SUN UltraSPARC workstations), (2) a storage server (a SUN 1000

[‡]Interactive steering may be defined as the 'online configuration of a program by algorithms or by human users, where the purpose of such a configuration is to affect the program's execution behavior'.

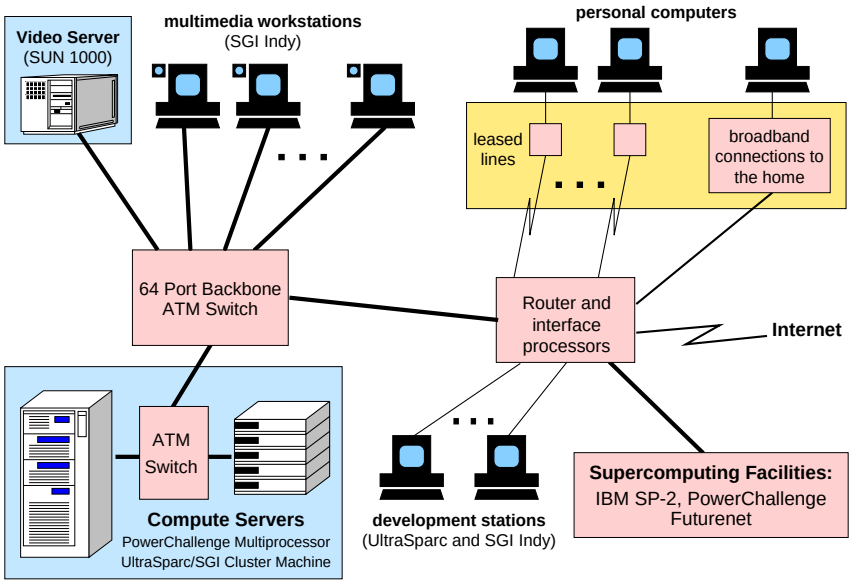

Figure 1: The Distributed Laboratories computing environment.

video server), (3) a number of development stations, and (4) a number of high end visualization engines, all of which are interconnected via a heterogeneous network infrastructure, including ATM links, and switched 100Mb and 10Mb Ethernet. In addition, at the campus level the Futurenet network effort will link these machines via ATM to several remote high performance and user interface engines, including an IBM SP-2 machine, a larger-scale SGI PowerChallenge, and visualization engines located in the College of Computing's visualization laboratories, in several engineering departments, and in Georgia Tech's School of Earth and Atmospheric Sciences.

The computational instruments described in this paper are run on any one or several of the compute engines shown in Figure 1; their input data may reside on machines specialized to store the large-scale datasets, and the visualization and user interface engines may utilize high performance graphics hardware required for real-time 2D or 3D data visualization.

Overview of paper. The following section describes in more detail two of the parallel and distributed scientific applications used in our research. Section 3 discusses interactivity in computational instruments and introduces the basic software architecture of the interactivity framework, including the components necessary for a single user to interact with a parallel computational instrument. Section 4 considers the support necessary to move to a Distributed Laboratory, an environment in which multiple users can work collaboratively to examine and control highly complex distributed applications. Related research is discussed in Section 5, and conclusions and future research are described in Section 6.

2 Examples of Interactive Computational Instruments

The requirements and design of a distributed laboratory cannot be meaningfully discussed in the absence of application requirements. This section presents two parallel applications used in our research and describes the manner in which these applications can benefit from the types of interactivity that the distributed laboratory can provide. This discussion will motivate the later description of distributed laboratories facilities.

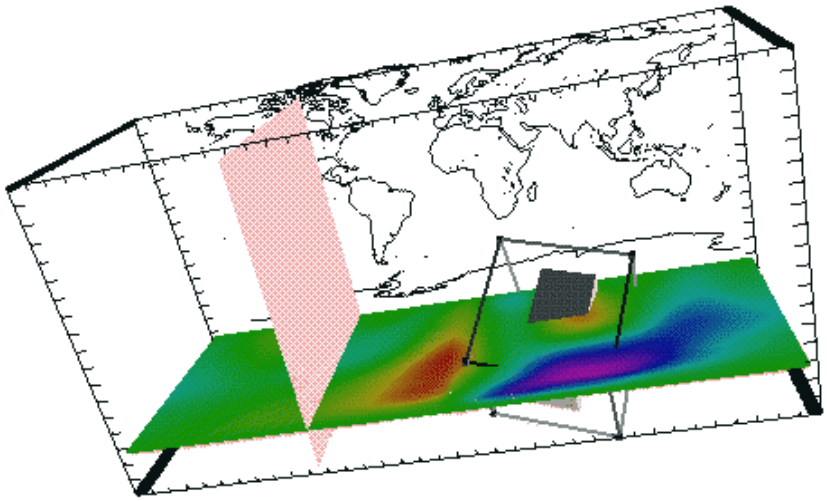

Figure 2: Three-dimensional visualization of atmospheric modeling data.

2.1 Atmospheric Modeling

In collaboration with atmospheric scientists at Georgia Tech, we have developed a parallel and distributed global chemical transport model capable of running on any of the HPC engines described in Figure 1. This model uses assimilated windfields [31] (derived from satellite observational data) for its transport calculations, and known chemical concentrations also derived from observational data for its chemistry calculations. Models like these are important tools for answering scientific questions concerning the stratospheric-tropospheric exchange mechanism or the distribution of species such as chlorofluorocarbons (CFCs), hydrochlorofluorocarbon (HCFCs) and ozone. Our model contains 37 layers, which represent segments of the earth's atmosphere from the surface to approximately 50 km, with a horizontal resolution of 42 waves or 946 spectral values. In a grid system, this corresponds to a resolution of about 2.8 degrees by 2.8 degrees. Thus in each layer 8192 gridpoints have to be updated every time step where a typical time step increment is 15 simulated minutes. Details of the model's solution approach, parallelization, and performance results are described in [19].

In order to assist end users in understanding model results and in steering the model's computations toward more useful data domains, visualization researchers at Georgia Tech have developed tools and visualizations for collaborative model steering [18, 26]. One such interface, depicted in Figure 2, displays a latitudinal and longitudinal slice of N^2O concentrations in the atmosphere where the concentration levels are represented as varying shades of grey. The N^2O concentrations are extracted from the model at each timestep using on-line monitoring mechanisms. Steering is accomplished by positioning the latitudinal and longitudinal planes, sizing and moving the cube to intersect a plane, then entering a desired concentration value to be applied to all gridpoints in the cube. The resulting set of values is forwarded to the computational instrument via the steering infrastructure and is available to the model at the next timestep.

This type of steering change is useful in playing "what-if" scenarios. Combined with checkpoint and rollback features built into the model code, the user can examine model execution, checkpoint at desired points in time, and at a future time, roll back to a checkpointed state, inject new concentration values, and restart the model. The process can be repeated until a desired outcome is achieved. The visualization techniques employed in this work are discussed more fully in Section 3.4.

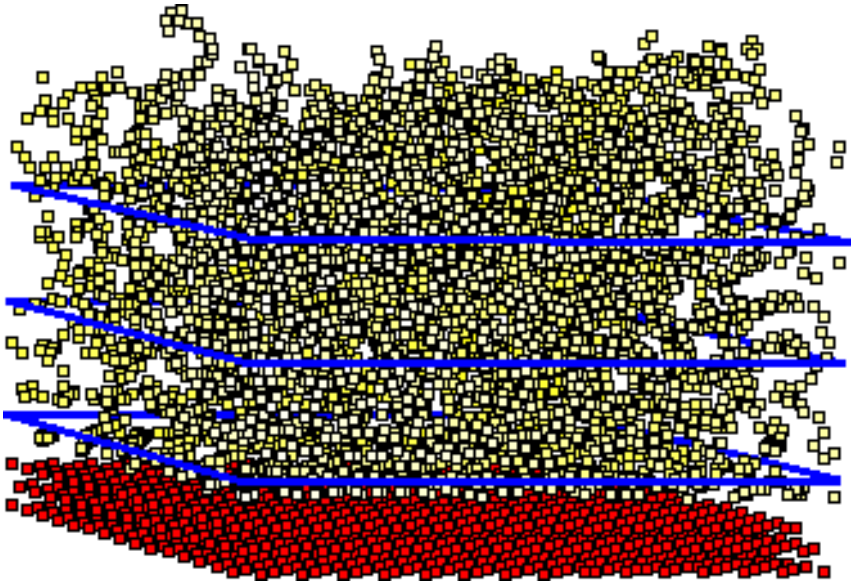

Figure 3: A visual representation of a sample system for the molecular dynamics simulation. The white-yellow particles are the pseudoatoms of the alkane chains. The red particles represent the gold substrate. The blue bands depict the boundaries between the spatial domains assigned to each processor.

2.2 MD – A Molecular Dynamics Application

The second example is of an interactive molecular dynamics simulation, called MD, developed in cooperation with a group of physicists exploring the statistical mechanics of complex liquids [9]. The specific molecular dynamics systems being simulated are n-hexadecane (C_{16}-H_{34}) films on a crystalline substrate $Au(001)$. In the simulation, the alkane system is described via intramolecular and intermolecular interactions between pseudoatoms (CH_2 and terminal CH_3 segments) and the substrate atoms. A typical small simulation contains 4800 particles in the alkane film and 2700 particles in the crystalline base. A visual representation of this physical system appears in Figure 3.

For each particle in the MD system, the basic simulation process takes the following steps: (1) obtain location information from its neighboring particles, (2) calculate forces asserted by particles in the same molecule (*intra-molecular forces*), (3) compute forces due to particles in other molecules (*inter-molecular forces*), (4) apply the calculated forces to yield new particle position, and (5) publish the particle's new position. Among these steps, the dominant computational requirement is calculating the long-range forces between particles, but other required computations with different characteristics also affect the application's structure and behavior. These computations include finding the bond forces within the hydrocarbon chains, determining system-wide characteristics such as atomic temperature, and performing analysis and online visualization.

The implementation of the MD application attains parallelism by domain decomposition. The simulation system is divided into regions, and the responsibility for computing forces on the particles in each region is assigned to a specific processor. In the case of MD, we can assume that the decomposition changes only slowly over time and that computations in different sub-domains are independent outside some cutoff radius. Inside this radius information must be exchanged between neighboring particles, so that different processors must communicate and synchronize between simulation steps. The resulting overheads are moderate for fairly coarse decompositions (e.g., 100-1000 particles per process) but unacceptable for finer grain decompositions (e.g., 10 particles per process).

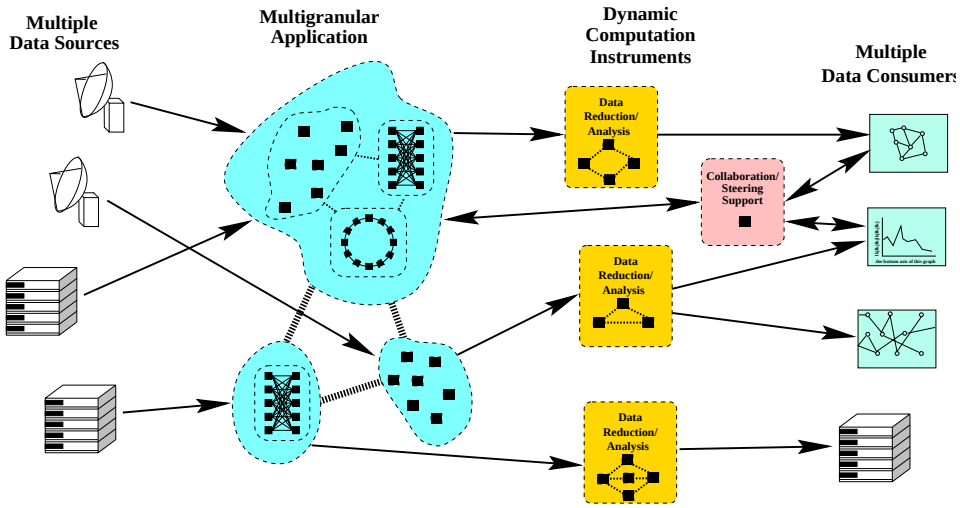

Figure 4: General Interactivity Framework

The MD simulation provides numerous opportunities for performance improvement through interactive steering:

- Changing decomposition geometries in response to changes in physical systems. For example, a slab-based decomposition may be useful for an initial system, but a pyramidal decomposition might be a better choice if a probe is lowered into the simulated physical system.
- Modifying the cutoff radius to improve solution speed by computing uninteresting time steps with some loss of fidelity.
- Shifting the boundaries of spatial decompositions for dynamic load balancing among multiple processes operating on different sub-domains.

One such interaction implemented by our group is depicted in Figure 3, where the MD computational instrument is steered interactively and without need for rollback by end users to balance computational loads across different processors by moving the boundaries of spatial decompositions (by 'grabbing' and moving the lines shown in the figure).

3 Interactivity in a Single User Framework

The needs and benefits of interactivity discussed above carry with them a price in terms of both computational demands and system complexity. The traditional approach to building interactive applications requires the user interface, data reduction, display and interactivity processing to be embedded in the application itself. A fundamental design goal of the distributed laboratories project is to provide this interactivity support *outside* the application. This separation simplifies the coding and optimization of application, allows the reuse of complex user interfaces and eases the distribution of interactivity processing onto other computational resources where their interference with application performance can be minimized. Before considering the requirements of the general case, we first consider the basic needs of an interactivity system that supports a single display interacting with a single application though external monitoring and steering mechanisms.

3.1 General Interactivity Framework

Figure 4 depicts a general interactivity framework for a complex collaborative application. The Distributed Laboratories project includes support for all aspects of this application, including visualizations,

steering interfaces, data communication and analysis middleware, collaboration support, and application monitoring and steering support. However, for the moment we avoid discussion of the general issues in distributed collaborative applications and instead focus more narrowly on what is required to support the types of interactivity required in the context of a single user interacting effectively with a single computational instrument.

Specifically, this section considers:

- the support required to efficiently monitor system- and application-level behavior in parallel and distributed programs;
- support for steering such a program though external interfaces; and
- visualization interfaces for both monitoring and steering.

Several key design goals influence the nature of the framework presented in Figure 4. First, as it is mentioned above, it is our goal that the facilities or algorithms deciding upon and making changes to computational instruments are implemented outside the computational instrument (application) itself. This is a goal that cannot always be strictly adhered to, particularly when steering latency requirements mandate tight coupling between the instrument code and monitoring and steering. Moreover, many programs already offer user interfaces or contain specific algorithms that adjust certain application parameters in order to maintain reasonable convergence of iterative solution methods, to ensure the validity of experiment parameters, etc. The framework in Figure 4 considers instead additional functionality to enable the instrument to be used in an experimental setting or to provide experimentation support beyond the instrument's basic functionality. However, we posit that in general such functionality should remain separately changeable from the instrument itself, for the same reason it is desirable that multiple user interfaces are not integral parts of data sets they visualize.

The second design goal is motivated by the high performance nature of computational instruments. Namely, the runtime overheads of interacting with a computational instrument should correspond to the level of interactivity being employed. This implies that program instrumentation and monitoring/steering are efficient, 'get out of the way' when not used, and may themselves be configured to offer exactly the functionality and corresponding overheads needed for a desired interactivity level. This assumption rules out the use of commonly available monitoring mechanisms such as trace files, or the use of simple approaches to steering like including blocking input statements in the instrumented code. We posit that such simple approaches would not be adopted by physical scientists who are already seeking ever increasing amounts of computation time for solving their scientific and engineering problems.

Finally, we assume the programmer is capable of evaluating the costs versus derived benefits of online instrument interactions. This is reasonable in light of the fact that we rely on the same programmers to implement the actual configurable program components required for steering. Although some research in software engineering and programming languages has addressed the automation of program transformations automating such configuration, generally useful solutions are not available to date.

3.2 Monitoring

Interacting with computational instruments in terms meaningful to end users requires online monitoring – the dynamic gathering of application-specific information from an instrument as it executes [28]. Depending on the type of data obtained, the data can be useful for debugging, performance tuning, or as part of an observation-reaction feedback loop used in combination with a steering system.

Distributed laboratories uses Falcon [12] to provide application-specific, event-based monitoring of parallel and distributed applications. Falcon is operational for both distributed and shared memory applications, on the SUN UltraSPARC, SGI, RS6000, and Windows NT platforms. The tool set consists of a sensor specification language and compiler for generating application sensors, and one or more local agents for online information capture, collection, filtering and analysis of event data.

A local agent, usually residing on the monitored program's machine, is responsible for capturing event

data. On shared memory architectures, this local agent is implemented as an additional user level thread operating in the instrument's address space. Local agents, due to their proximity to the computational instrument, can gather monitoring data quickly while minimizing interference with the instrument's operation. For physically distributed instruments, multiple local agents are employed and each of the instrument components is treated as a separately monitored entity.

Falcon's use is straightforward. The user describes sensors in a Sensor Specification Language which defines the data to be extracted. Sensor descriptions are compiled into functions in the native language of the application. The user places calls to these functions in the application code (instruments the application) at points where monitoring may be necessary. During execution, the sensors are invoked and generate data that is passed to the monitoring local agent.

Shared memory applications can generally be satisfactorily monitored using a single user level thread. For distributed applications that potentially span wide area networks (such as those in the distributed laboratories scenario), the monitoring mechanism necessarily becomes more complex, and is constructed as a hierarchy of agents forwarding and analyzing data based on constraints like network latency, available processing and data bandwidths, and desired steering response times. Our current work in monitoring addresses these distributed environments, based in part on earlier ideas presented in [29]. In addition, we are currently addressing monitoring highly dynamic systems, where little knowledge can be assumed at initialization time about the location of and requirements for monitoring agents, analysis and reduction clients, and visualization clients.

3.3 Interactive Steering

Characterizing Steering and its System Support.

A steering system used by external agents must provide the following functionality: (1) receive the computational instrument's runtime information from the online monitoring system, (2) analyze and display the information to the end user or submit it to a steering agent, (3) accept steering commands from the user or agent, and (4) enact those commands to affect the application's execution. As shown in Figure 5, in our implementation at least one local steering agent runs on the target machine along with the application. This agent performs steering actions requested by external agents. External agents are driven by monitoring data and may request steering actions directly based solely upon this data or in response to user manipulations on the data.

While monitoring can be performed with little application involvement, steering is often more intrusive in that it can involve modifications to application state that must be synchronized with the application's execution. Such synchronization may or may not be required, as demonstrated by three examples drawn from MD described in Section 2.2. For example, steering without synchronization may be performed on the size of the cutoff radius, since this radius is read from a global location at the beginning of each timestep. This cutoff radius is a single floating point number which can be written atomically by the steering system and read atomically by the application. Similarly, a steering agent can continuously make small changes to domain boundary locations (as long as domain boundaries do not cross each other) without synchronizing with the application, since the arrays specifying these locations are refreshed once per timestep. On the other hand, a steering action that initiates a switch to a different decomposition geometry would require synchronization with the application, since it affects much of the application's internal state.

Experiences with Steering. One important insight from the steering examples for MD is that computational instruments differ in terms of the ease with which certain steering actions may be implemented. In MD's case, its implementation permits some of its internal variables to be inspected and steered continuously, with little instrumentation of the code. This also facilitates the attainment of performance improvements via steering (e.g., by online load balancing.) Moreover, steering can improve MD's functionality by permitting end users to cause it to proceed more quickly through uninteresting portions of its molecular simulation. In

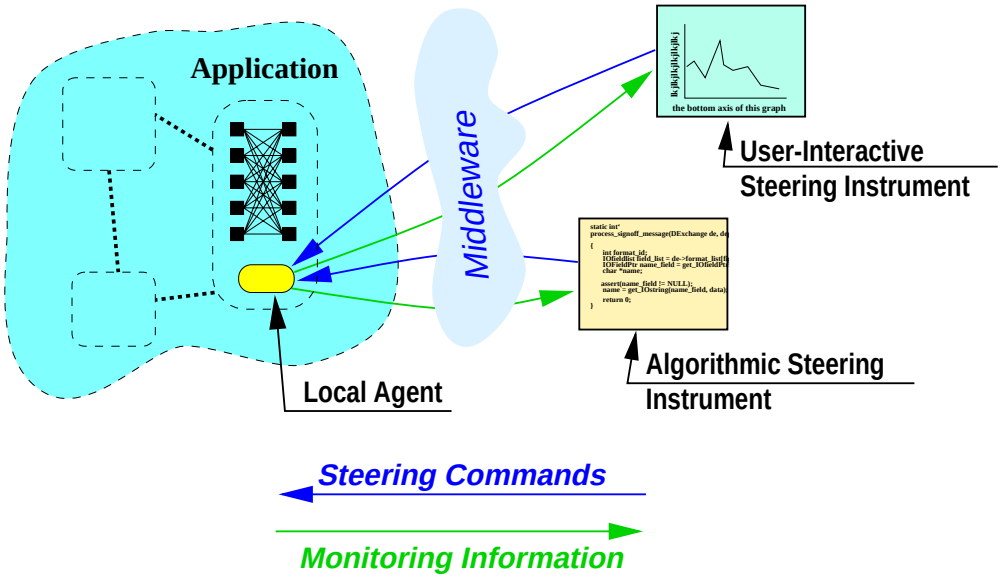

Figure 5: Our steering model.

general, however, potential improvements in performance or functionality by the addition of steering depend largely on an instrument's implementation and on the steering and monitoring actions required. We do not attempt to characterize instrument implementations that are well-suited for steering*. Instead, we address two topics: (1) the performance criteria to be applied to steering instruments and (2) the interfaces to be provided to steering developers and end users.

Another insight from the MD steering examples is that program steering must consider overheads not relevant to performance monitoring. These overheads are: (1) the perturbation to the application due to instrumentation for monitoring and steering [21], (2) the latency of the monitoring to enactment feedback loop, and (3) the costs of decision making part of this latency. Specifically, for steering, the end-to-end latency of the monitoring to enactment feedback loop is a critical performance constraint when steering actions cannot be based on 'stale' monitoring data, or when such actions become inappropriate after some future program state has been passed. Careful performance measurements on a multiprocessor platform show the performance effects of alternative implementations of the monitoring to enactment loop in which program perturbation is traded off against end-to-end latency. For improving performance when steering MD by moving domain boundaries, reductions in perturbation are critical, whereas reduced end-to-end latency is important in several real-time and multimedia applications we observed [16, 27].

Two Steering Systems.

We are experimenting with two additional approaches to implementing program steering for high performance computational instruments. These approaches differ with respect to two key aspects which are: (1) the manner in which actions to be performed by steering commands are specified, and (2) the manner in which those actions are synchronized with the application's control flow. In the remainder of this section we describe these approaches, called Progress and Eagle, and discuss their complementary goals.

*A broad community of researchers addressing the topic of configurable software presents its research results in the bi-annual 'Configurable Distributed Computing' conference[5]. We are not investigating program transformations, compiler actions, or generating self-modifying code to improve the performance or ease of implementing certain monitoring or steering actions.

The Progress Toolkit. The Progress toolkit [32] implements a steering interface based on the concept of *actuators*. Actuators are the mechanism through which steering actions are accomplished. Like sensors in the Falcon monitoring system, actuators are function calls inserted by the user into the instrument code at appropriate locations. Namely, they are placed where it would be 'safe' for the steering system to perform a steering action. The resulting steerable entities (application-level data structures) are registered with a local agent called the *steering server* by special function calls also inserted into the application code. The steering server maintains a database of steerable entities. An entity's entry in the server database may also specify particular steering actions to be taken in response to monitoring events associated with the entity. This provides a mechanism for accomplishing external algorithmic steering entirely on the target machine, using the local steering agent.

Steering actions supported by Progress include both changes to application state and the invocation of application functions which have been registered with the steering server. In either case, the steering server does not directly perform the steering action. Instead, it sets (or arms) one or more actuators in the application. When, in the course of normal execution, the application code executes an armed actuator, the steering action is performed synchronously *by the application's own thread of control.* In this fashion, Progress implicitly addresses the issue of synchronization with the application, by simply requiring the programmer to place actuators only where a steering action would not threaten the application's integrity or by requiring the programmer to implement additional, explicit safeguards against corrupting application state. For example, in the atmospheric model discussed in Section 2.1, actuators are used to manipulate model parameters and to cause checkpointing or rollback to occur. Because carefully placed actuators running in an application thread of control carry out these operations, steering actions only occur when the application is in a 'safe' state for carrying them out[†].

The research topics addressed by Progress focus on the scalability of program steering, where scalability addresses both the sizes of target programs and machines being monitoring and steered, as well as the amounts of information utilized for these tasks. Specifically, because Progress uses the same event transport infrastructure as the Falcon monitoring system, it is easy to address tradeoffs in the amounts of information required for certain steering actions vs. end-to-end steering latency. For example, it is straightforward to reduce the amounts of monitoring information extracted at runtime if steering actions can be migrated 'as close as possible' to the actual application program generating monitoring information. In one extreme case, a combined monitoring/steering entity in the program itself may steer and monitor itself semi-autonomously, perhaps only using some meta-level inputs (e.g., turn steering 'on' or 'off') from the outside. In another extreme case, all monitoring information may be forwarded to a remote client interface where a human user makes all steering decisions. The intent of Progress is to provide an infrastructure in which it is easy to realize tradeoffs in monitoring and steering latencies, throughputs, and 'accuracies' when addressing target programs running on any number of nodes of underlying machine platforms. Currently, Progress runs on shared memory machines, with its next version spanning most of the heterogeneous machines available at Georgia Tech.

Eagle toolkit: unifying monitoring and steering. The Eagle toolkit aims to create a framework encompassing program monitoring and steering into one uniform conceptual model. Eagle shares with Progress its methods for attaining scalability while also providing additional functionality to enable end users to perform program steering. Specifically, the Eagle approach views application-level entities as traditional *objects* with both state and a set of methods operating on that state. As a result, a steering action is simply an invocation of one of the methods of a steerable object. In contrast to Progress, where the local agent and actuators borrow program threads of control to accomplish steering actions, the Eagle local agent actively calls object methods in the application. Given this functionality, Eagle addresses the synchronization of steering actions with the application's execution by assuming that those method invocations perform any

[†]Progress also offers a 'probe write' action in which the steering server directly sets an application data value without any application synchronization.

application synchronization necessary to ensure application integrity.

Eagles object-based model of steering is natural if the application is programmed in an object-based parallel programming language. However, we have found it straightforward to apply it to non-object-based applications, by creating in the application the object-style abstraction of a datatype and procedures that operate on that type.

In this model of monitoring and steering, all program abstractions are potentially monitorable and steerable, once they have been registered with an information repository. Moreover, object-based compilers for describing monitoring and steering objects (using the IDL interface language) may be implemented to be compliant with current industry standards like CORBA [30] and OLE. An interesting outcome of this generality is an innovative application of CORBA's notion of Event Services to program monitoring. Namely, rather than controlling the monitoring of particular objects with explicit 'on-off' switches, Eagle can implicitly suppress monitoring events that are not of interest by interposing an *event channel* transport mechanism between monitored application objects and external agents. The interposed event channel requires parties interested in receiving events from a particular channel to register with the channel. This explicit registration of listeners permits the suppression of events on channels which have no listeners and therefore, provides a convenient basis for the online control of monitoring overheads.

Eagle and the CORBA event channel implementation it utilizes are being constructed as part of a larger project on high performance object representations on heterogeneous platforms, contributing to the general middleware infrastructure of the Distributed Laboratories project[1].

3.4 Application-specific Visualization and Steering Interfaces

While monitoring and steering support are essential for efficient, low-impact interactivity, it is often the final visual interface that determines the success and effectiveness of interactivity. The interactive 3D visualization of atmospheric modeling data shown in Figure 2 is constructed as a set of modules using Glyphmaker [26] and the SGI Explorer framework. These modules implement application specific functionality, in the case of the atmospheric model, modules that directly convert monitoring data extracted from the computational instrument from its spectral form within the instrument to a gridded form more suitable for visualization. Another module acts as a reader for converting the data being displayed to be printed on high resolution output devices used by atmospheric scientists (using the PV-Wave visualization system).

The notion of modules in Glyphmaker is not a new concept. Important is Glyphmaker's ability to have users select and focus on important regions in dense and complex data. For example, it offers a 'Conditional Box' for choosing a spatial region by direct manipulation; the data inside that region can then be bound to special glyphs, made to appear alone, and most importantly, manipulated and then re-injected into the application program using the steering infrastructure. GlyphMaker also offers graphical modules for depicting slices of data at various longitudes, latitudes, and altitudes and in various projections. With these modules, users can focus on the correlations between species concentrations and vertical wind fields (taken from satellite observations). These correlations are hard to see using traditional visualization methods, but can be critical in assessing the accuracy of the model, the accuracy of the vertical windfields themselves (which are not well understood), and in understanding the processes by which species spread through the atmosphere.

In contrast to the 3D visualizations providing excellent overviews of model behavior, a complementary Glyphmaker-based 2D steering interface operating with subsets of the atmospheric modeling application's data is shown in Figure 6. This interface's display presents the distribution of C^{14} at the single latitude of 2.8° N. It has two logical parts: one for showing both the computed and the observed concentration values of C^{14} atoms in air to the end user, and the other for accepting steering requests from the user. The computed results of the C^{14} distribution are represented by the circle plotted curve from atmospheric layer 0 to 37, which is updated for every model time step. The concentration of C^{14} actually observed at this point is represented by the triangle plotted curve. Although complete observational data does not

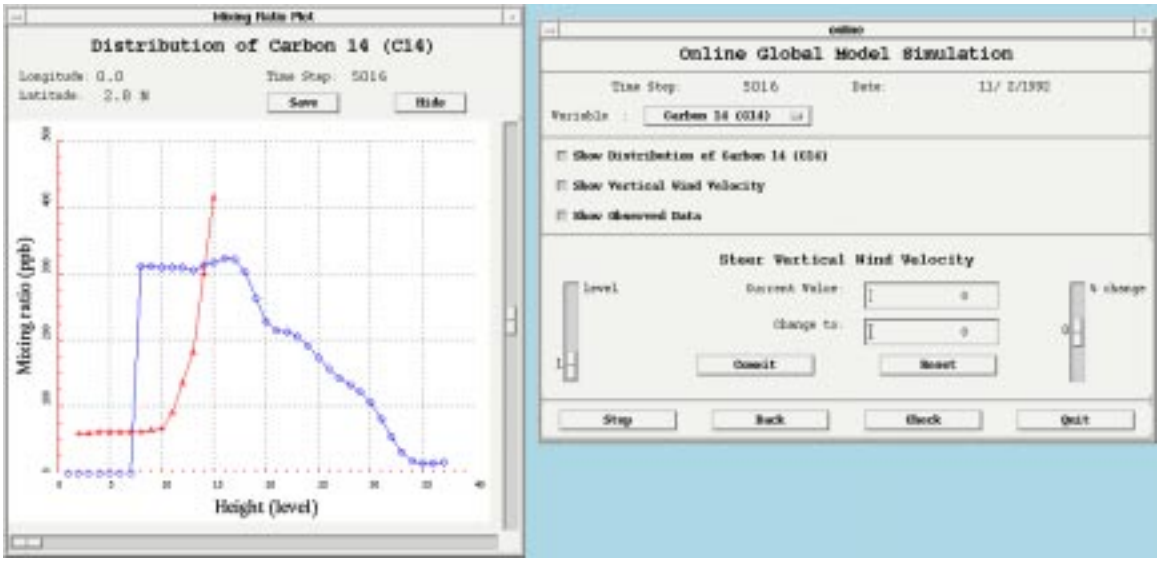

Figure 6: An application-specific display for online control of the atmospheric modeling code.

typically exist, the data available to us may be used to indicate when the current computation is going astray. When noticeable discrepancies between the calculated values and the observed values are detected, the user can dynamically modify the application execution to 'correct' the computations. For example, the triangle plotted observational data curve in Figure 6 shows that C^{14} concentrations are relatively constant until atmospheric level 10 where they rise sharply. In the computation curve (circles), the constant region is too low and the rise too sharp and too early. This may be indicative of a weakness or inaccuracy in the model that could be corrected by modifying some of its experimental values. In this example, the steering interface is being used to fine-tune vertical wind field values. After entering new vertical wind velocity values, the user clicks the Commit button to send the steering command to the application which uses the parameters for computations in the next time step.

The user can also stop the application's execution (by clicking the Stop button), change parameters, and restart the execution (by clicking the Go button which would replace the Stop button). Before restart, the user can rollback the computation to a previously checkpointed time step (by clicking the Back button). At any point the user can checkpoint the application execution (by pressing the Check button) or invoke a default checkpointing policy which automatically saves the application's execution history after a predefined number of time steps.

Discussion. The interfaces presented here have different strengths. The strength of the 3D interface is in its global view and ease of use. The 2D interface is better suited to offer the fine control needed when particular parameters must be examined and changed. The 3D interface has recently been ported to the SGI Open Inventor environment which facilitates its integration with collaboration support discussed in Section 4.2.

4 Toward Distributed Laboratories

The basic monitoring and steering support discussed in the previous section is the essential functionality required for simple interaction between a single user and a single computational instrument or application. When multiple users are involved, however, or the data reduction and display processing required by the interactivity grow to the level of being a significant distributed system in and of themselves, additional support is required. These requirements, and the tools to support them, are what separate the distributed

laboratories environment from simpler monitoring and steering systems.

The intent of distributed laboratories is to permit scientists, engineers, and managers at geographically distinct locations (including individuals telecommuting from home) to combine their expertise in solving shared problems by allowing them to simultaneously view, interact with, and steer sophisticated computation instruments executing on high performance distributed platforms. The goal of the distributed laboratories project is to develop the underlying, enabling technologies and software tools to support multiple users interacting with multiple instruments and each other and to develop working prototypes. The underlying, enabling technologies needed to support such a distributed laboratories vision includes:

- dynamic monitoring, adaption, and interactive steering of high performance computations.
- interconnectivity and data exchange infrastructure; and
- collaboration and shared visualization technologies.

Dynamic monitoring and steering were discussed in Section 3 in the context of a single user/single instrument environment. Novel communication and data analysis middleware and collaborative support are discussed in the following sections.

4.1 Communications and Data Analysis Middleware

The communication and data analysis needs of Distributed Laboratories differ from that of traditional parallel and distributed computing environments. As with traditional environments, communications must be highly efficient to support both high bandwidth and low latency data communications, potentially across heterogeneous architectures. But in contrast to current environments [10], we assume that tools and experimenters may come and go dynamically and may be interested in different types of data at different times. Additionally, the various programs that cooperate to make up a distributed laboratories may not all be under the control of a single group, compiled by the same compiler or written in the same language. While the need for efficiency makes binary data transmission imperative, the computational environment makes complete agreement on common and consistent format for the data difficult to begin with and virtually impossible to maintain in an evolving system. The dynamic nature of the environment and the amount of processing required to support scientific visualization also pose difficulties. Ideally, tools should be able to access any communication in the system without disruption or modification to the computational instrument being observed. One would also like easy mechanisms for distributing data gathering and analysis necessary for visualization to make full use of the available computational resources.

Two novel communication libraries, called DataExchange and PBIO, jointly address the dynamic and heterogeneous nature of Distributed Laboratories. Jointly, these libraries provide a communications infrastructure that allows instruments, data analysis tools and interactive client displays and visualization displays to be plugged into the system dynamically. First, the *PBIO* (Portable Binary Input Output)[7] library supports the transmission of binary records between heterogeneous machines. PBIO is essentially a data meta-representation. Users register the structure of the data they wish to transmit or receive, and PBIO transparently masks the representation differences across heterogeneous machine architectures. In particular, PBIO handles differences in the sizes, locations and even basic types of the fields in the records to be exchanged. Record meta-information is transmitted once, when record formats are registered. Thereafter, transmission occurs in the writer's native format, and the PBIO library on the receiver transparently handles discrepancies between the writer's format and the format required by the reader. In the case of transfers between homogeneous machines, the only additional overhead imposed by PBIO is the transmission of a 4-byte format ID. Between heterogeneous machines, overheads depend on the degree to which the record formats and atomic type representations differ across the machines.

While PBIO supports exchanging data between two clients, the *DataExchange* library[8] layered on top of PBIO provides support for establishing communication between agents, for resolving differences between data formats used by multiple agents, for forwarding data from agent to agent, and for processing data within an agent. The use of DataExchange as the basic communications package in the Distributed Laboratory

allows transparent multiplexing when necessary and supports dynamic attachment of instruments to different portions of the application. DataExchange also allows application handler functions to be bound to the arrival of new data. The DataExchange library augmented with a few application functions and a simple main program can thus serve as a configurable data filter. Coupled with the support for dynamic connections, this dramatically simplifies the creation of networks of cooperating agents that gather, analyze and distribute the data required for a particular display. Controlling the flow of data with this network of communicating entities is accomplished in part by allowing agents to specify which types of data they are interested in receiving and those they do not wish to receive.

PBIO and DataExchange constitute first steps in our efforts to develop a rich infrastructure for the program-program, program-human, and human-human communications in distributed laboratories. Given the need for laboratory instruments and user interfaces to interact with many diverse systems, including commercially available analysis and display packages, we are currently constructing object-based middleware layered on top of PBIO and DataExchange using the CORBA standard developed by OMG. The focus of this work is to attain high performance of distributed objects by using diverse object implementations. For coarse grain sharing, we have developed a distributed object system based on the CORBA-like Fresco toolkit, where performance is improved by employing caching techniques[20]. For finer grain sharing, object implementations may be fragmented[6] and/or configured online jointly with configuring the communication protocols used for inter-object communications[16].

4.2 Collaboration

Meaningful collaboration among multiple science or engineering end users should utilize the power of the computational instruments being deployed. This implies that end users will employ visualizations of the actual output data of computational instruments, depictions of program structures, and even jointly manipulate shared instruments. Since current visualization engines (e.g., the SGI Explorer system) do not offer significant collaboration support, in joint work with visualization [26] and CSCW[15] researchers, we are developing (1) mechanisms for coordinating what is rendered on separate machines and (2) abstractions for manipulating shared complex entities. Toward this end, many other systems provide fairly low-level 'sharing' abstractions, such as shared memory or remote procedure calls, and leave it to the application programmer to build the support for collaboration. In comparison, our goal is to create a framework and collaboration library with which application-specific interaction and collaboration abstractions are easily constructed, thereby removing some of this burden from the developer. The method we use to attain this goal is to add means for constructing relevant 'interactors' as part of the data visualizations being employed by end users (e.g., see Figure 2).

The approach used by our group relies on the directed acyclic graph (DAG) abstraction employed by many modern user-interface libraries. Such abstractions are used to describe the appearance and some of the functionality of the interface. This technique works well for both the traditional 2.5D 'desktop' interface as well as 3D interfaces. Our framework uses an example of the latter, OpenInventor, for interaction and rendering. Collaboration support then, is built into the OpenInventor graphics library by provision of abstractions and their inclusion with the DAG structure being rendered. In this type of library, a DAG is a collection of nodes where the nodes have attributes or *fields* which determine certain aspects of their appearance or behavior.

Thus coordinating the displays on separate machines becomes a problem of maintaining consistency in both the structure of the DAG and in the field values of the nodes. Toward this end, we have extended the OpenInventor library with a *sharer* node which handles these details. The sharer node has a field which specifies the name of a DataExchange port. To create a shared 3D scene, the application programmer places a sharer node somewhere in the scene DAG and, through common OpenInventor library calls, sets the port name field. The sharer node uses the port name to open a connection to a component that has been designated a DataExchange server. The DataExchange server is responsible for accepting connections

from clients and forwarding events it receives to all connections for which a client has registered and interest in the event. Once a connection is established, the sharer node watches for changes to the DAG below it. It conveys this information to any other sharer nodes that happen to be connected to the DataExchange, and also accepts notification of changes from the remote sharer nodes and mirrors the changes in the local DAG. Thus, for example, if the steering object is moved by one collaborator, the changes to its orientation field are sent out via the DataExchange and any other users that are connected will see the effects.

Shown in Figure 4 are two collaborative clients. Both clients, developed using the SGI Open Inventor library, have incorporated into their scene DAG a sharer node. Communication of scene information is handled through a DataExchange server shown in the figure as a separate component. It is also possible for one of the visualization/collaboration clients to assume the role of DataExchange server.

OpenInventor is quite extensible and allows the programmer to create new nodes, but since all OpenInventor nodes use fields to store attributes, the sharer node has a reasonable default method for handling these custom-built nodes — maintaining the consistency of their field values. But many collaboration scenarios call for a less strict form of consistency. Consider the camera node, which has fields that specify the location of the user within the 3D scene. Two scientists collaboratively examining a data set might reasonably expect to have control over their own viewpoint, rather than sharing a single camera. Our framework supports this with *collaboration-aware* nodes, which identify themselves to the sharer node and provide their own methods for maintaining consistency. In this case the collaboration-aware camera would allow the users to explore independently and also provide a mechanism to switch to another's viewpoint.

Two scenarios are now being studied using the mechanisms described above. In one scenario, a 'teaching' user offers 'share' nodes that simply enable other users to follow what the teacher is doing, perhaps at different levels of detail. In another scenario, two users can be 'aware' of each others' actions by seeing 'where' other users are actively exploring data or making changes but not being coupled to those explorations or changes.

5 Related Work

VASE [17] was one of the first systems to recognize the need for application control. It furnished application developers with tools to annotate their code for steering and provided an interface independent of a specific visualization and interaction method. VASE was developed on a SIMD platform and used the attributes of this architecture to help control the application. Most of the published literature for steering focuses on case studies or customized solutions rather than a general framework for steering. SCIRun [23] provides a general computational steering environment in which separate modules of an application are modeled as a dataflow graph. As data flows through this graph, changes are introduced to the computation through parameters to the modules. SCIRun runs in a single workstation or shared memory environment. Our work differs from VASE and SCIRun in its ability to support distributed computational tools. A more detailed review of related steering research can be found in [13] and [4].

The Falcon monitoring system differs from other research on performance monitoring in its emphasis on controlling monitoring latency and its attempt to remain generally applicable by not exploiting properties of specific target programming languages, as Malony does with C++[3], for example. In comparison, Miller's IPS [22] parallel program measurement system attempts to give users insight into program performance by assuming a hierarchical program model. Event capture and performance measurement are at several levels: program level, machine, process, procedure level, and at the primitive activity level. The system performs two types of offline (and recently, also online[5]) analysis on trace data: critical path analysis and phase behavior analysis. Reed's *Pablo* system [24, 25] is an event-based performance analysis environment providing performance data capture, analysis, and presentation. Pablo is recently also being used for online program steering, by application to certain operating system services, such as file servers supporting World Wide Web traffic. We differ from this group in our emphasis on low latency steering and online monitoring using threads-based monitoring techniques.

6 Conclusion and Future Work

High performance applications are becoming increasingly interactive. In addition, the complexity of the problems being addressed by those applications (e.g., global atmospheric modeling) often requires collaboration among multiple scientists in devising the applications, in evaluating their data outputs, and in running them with suitable adjustments to internal parameters and input data. For such complex applications, the framework required to support data collection, data reduction, display and collaboration between users itself constitutes a complex distributed system. Furthermore, the complexity of this "meta-application" is mirrored by the complex nature of the underlying computing infrastructure being employed. Specifically, underlying machines are likely to be heterogeneous in processing and connectivity, and be physically distributed across multiple sites. In part, such distribution is due to the relative scarcity of high performance computing resources at any one site. More importantly, such distributed applications reflect the physical distribution of the diverse scientific teams involved in investigating complex, interrelated problems.

The distributed laboratories project described in this paper aims to construct an infrastructure with which future scientists and engineers can easily construct complex parallel and distributed interactive applications and run them cooperatively without regard to physical location. This project leverages off our experience with single user/single computational instrument environments and the valuable feedback we have received from application scientists working with the interactive applications described herein. The foundation for the distributed laboratories project rests with the following closely integrated research efforts:

- Monitoring and steering tools and infrastructure used in the online observation and manipulation of scientific computations.
- Middleware to transport the events and their contents, where such transport can be changed at runtime to adjust event streams to current system loads and monitoring/steering needs.
- The visualization support permitting the definition of appropriate visual abstractions and their efficient representation on 2D and 3D graphical displays.
- Collaboration infrastructure and abstractions supporting several collaborators based on are provided using the OpenInventor graphical display framework.

The two scientific computations/instruments presented in this paper represent only a subset of the applications used in this research. Other applications being steered using our tools include a fluid flow code developed with engineering end users, an interactive 3D virtual environment being constructed by graphics researchers at Georgia Tech, and most recently the application of our tools to large-scale physical simulations at the Los Alamos National Laboratories.

Our future research is pursuing two distinct directions. First, the monitoring and steering tools are being extended to explore highly distributed and dynamic target platforms, as are commonly used when multiple, networked supercomputers solve single computational problems and are being accessed by multiple, distributed end user. The key issue to be addressed by these extensions is the manner in which high performance (i.e., appropriate latencies and/or bandwidths for monitoring and steering) may be maintained in underlying environments that are not under the sole control of a single large-scale application. Ongoing research in this area is developing configurable communication protocols [16] that can trade off communication amounts for accuracy by use of runtime compression, the use of ATM's quality of service parameters to maintain runtime quality guarantees for connections that vary in their execution time behavior, and the dynamic configuration of the monitoring system and application itself to change its analysis or computational loci in response to changing runtime needs. The second area of research addressed by our future work concerns flexible middleware for implementation of the information sharing in any future high performance systems. Toward this end, we are developing high performance implementations of CORBA objects able to be configured in their implementation such that the diverse and runtime-varying needs of high performance applications may be satisfied[1].

7 Acknowledgments

We thank Thomas Kindler and Dilma Silva for their work on the atmospheric modeling software and Weiming Gu for developing Falcon and contributing to the design of DataExchange and the steering system. We also acknowledge our collaborators in discrete event simulation (Richard Fujimoto), in Atmospheric Sciences (Fred Alyea, Mary Trauner), in data visualization (Bill Ribarsky and Yves Jean), in collaboration systems (Scott Hudson and Norberto Ezquerra), in the area of online monitoring and steering for distributed systems (Raja Das), and concerning high performance, object-based middleware (Mustaq Ahamad) and communication protocols (Ellen Zegura and Ken Calvert). We also wish to thank the reviewers for their helpful comments.

References

[1] Mustaque Ahamad and Karsten Schwan. The COBS Project. http://www.cc.gatech.edu/systems/projects/COBS.

[2] Devesh Bhatt, Rakesh Jha, Todd Steves, Rashmi Bhatt, and David Wills. SPI: an Instrumentation Development Environment for Parallel/Distributed Systems. In *Proceedings of The 9th International Parallel Processing Symposium*, pages 494–501. IEEE, April 1995.

[3] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel system. In *Proceedings of the 1993 Supercomputing Conference*, pages 588–597, November 1993.

[4] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering scientific computations. *IEEE Computational Science & Engineering*, 1(4):44–62, 1994.

[5] International conference on configurable distributed sytems (CDS3). http://www.cc.gatech.edu/systems/cds.

[6] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (DSA) on large-scale multiprocessors. In *Proc. of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 227–246. USENIX, September 1993. Also as TR# GIT-CC-93/25.

[7] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. http://www.cc.gatech.edu/tech_reports.

[8] Greg Eisenhauer and Beth (Plale) Schroeder. The DataExchange library. Technical Report GIT-CC-96-17, Georgia Institute of Technology, 1996. http://www.cc.gatech.edu/tech_reports.

[9] Greg Eisenhauer and Karsten Schwan. Design and analysis of a parallel molecular dynamics applicati on. *Journal of Parallel and Distributed Computing*, 35(1):76–90, May 25 1996.

[10] Message Passing Interface (MPI) Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[11] Richard Fujimoto, Karsten Schwan, Mustaq Ahamad, Scott Hudson, and John Limb. Distributed laboratories: A research proposal. Technical Report GIT-CC-96-13, Georgia Institute of Technology, 1996. http://www.cc.gatech.edu/tech_reports.

[12] 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*, pages 422–9, 1994. Also available as TR GIT-CC-94-21.

[13] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29(9):140–148, Sept. 1994.

[14] P. T. Highnam and A. Pieprzak. Implementation of a fast, accurate 3-d migration on a massively parallel computer. In *61st Annual International Meeting and Exposition of the Society of Exploration Geophysicists*, pages 338–340, 1991.

[15] Scott E. Hudson and Ian Smith. Techniques for addressing fundamental privacy and disruption tradeoffs in awareness support systems. In *To appear in Proceedings of ACM Conference on Computer Supported Cooperative Work*, 1996.

[16] Daniela Ivan-Rosu and Karsten Schwan. Improving protocol performance by dynamic control of communication resources. Technical Report GIT-CC-96-04, Georgia Institute of Technology, 1996. http://www.cc.gatech.edu/tech_reports.

[17] D.J. Jablonowski, J.D. Bruner, B. Bliss, and R.B. Haber. VASE: The visualization and application steering environment. In *Proc. Supercomputing '93*, pages 560–9, 1993.

[18] Yves Jean, Thomas Kindler, William Ribarsky, Weiming Gu, Gregory Eisenhauer, Karsten Schwan, and Fred Alyea. Case study: An integrated approach for steering, visualization, and analysis of atmospheric simulations. In *Visualization '95*. IEEE, Oct. 1995. Also published as GIT-GVU-95-15, http://www.cc.gatech.edu/gvu/reports.

[19] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[20] R. Kordale, M. Ahamad, and M. Devarakonda. Object caching in a CORBA compliant system. In *Proceedings of the Second Conference on Object-Oriented Technologies and Systems*, June 1996.

[21] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[22] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[23] S.G. Parker and C.R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proc. Supercomputing 95*, pages 1–1, 1995.

[24] D.A. Reed, R.D. Olson, R.A. Aydt, T.M. Madhyastha, T. Birkett, D.W. Jensen, B.A.A. Nazief, and B.K. Totty. Scalable performance environments for parallel systems. In *the Sixth Distributed Memory Computing Conf.*, pages 562–569. 1991.

[25] D.A. Reed, K.A. Shields, W.H. Scullin, L.F. Tavera, and C.L. Elford. Virtual reality and parallel systems performance analysis. *Computer*, 28(11):57–67, 1995.

[26] W. Ribarsky, E. Ayers, J. Eble, and S. Mukherjea. Glyphmaker: creating customized visualizations of complex data. *Computer*, 27(7):57–64, 1994.

[27] P. Schneck, E. Zegura, and K. Schwan. DRRM: Dynamic resource reservation manager. In *Proceedings of IC3N 96*. IEEE, October 1996. to appear.

[28] Beth (Plale) Schroeder. On-line monitoring: A tutorial. *Computer*, 29(6):72–78, June 1995.

[29] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A language and system for parallel programming. *IEEE Transactions on Software Engineering*, 14(4):455–471, April 1988.

[30] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[31] R. Swinbank and A. O'Neill. A stratosphere - troposphere data assimilation system. Climate Research Technical Note CRTN 35, Hadley Centre Meteorological Office, London Road Bracknell Berkshire RG12 2SY, March 1993.

[32] J. Vetter and K. Schwan. Progress: a toolkit for interactive program steering. In *Proc. 1995 Int'l Conf. on Parallel Processing*, pages II/139–42, 1995.