An Approach to Immersive Performance Visualization of Parallel and Wide-Area Distributed Applications

Luiz De Rose
laderose@us.ibm.com

Advanced Computing Technology Center
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Mario Pantano
mario.pantano@ac.com

Andersen Consulting
Largo Donegani 2
20121 Milan, Italy

Ruth A. Aydt Eric Shaffer Benjamin Schaeffer Shannon Whitmore Daniel A. Reed*
{aydt,shaffer1,schaeffr,swhitmor,reed}@cs.uiuc.edu

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

Complex, distributed applications pose new challenges for performance analysis and optimization. This paper outlines an online approach to performance analysis where developers are active participants, using integrated measurement and immersive performance visualization to tune parallel and distributed applications.

1 Introduction

Historically, performance analysis has been an iterative process — developers repeatedly instrument application and system software, analyze captured performance data, and modify the software to alleviate identified bottlenecks. This post-mortem optimization model requires developers and performance analysts to engage in a laborious cycle of code modification execution, particularly when the number of possible policy parameters or optimization points is large. Indeed, most performance analysis tools (e.g., SvPablo [4], Medea [3], Paragraph [6], and AIMS [10]) still render only post-mortem data. Consequently, performance analysis is passive and based on modification of quiescent code, rather than an active process of interaction with executing code.

However, emerging applications are increasingly distributed and heterogeneous, exploiting distributed resources (e.g., remote instruments, computational grids, and distributed data archives). In such environments, the set of resources for each application execution is drawn from a dynamic, heterogeneous pool. Hence, it is unlikely that subsequent executions will acquire the same resources or encounter the same operating conditions. Consequently, performance measurements from one execution may not be representative of future executions nor may they highlight critical bottlenecks. Finally, the resource needs of complex, multidisciplinary applications with adaptive meshing, discipline-specific algorithms, and real-time data analysis routines may vary dramatically even within a single execution.

Simply put, the confluence of complex applications, network-connected resources, and geographically distributed collaborative teams dictates a new model of performance measurement and analysis. Just as scientific data visualization has allowed scientists to understand evolving, complex phenomena, real-time performance measurement and immersive performance data visualization can enable collaborating groups to inter-

*This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049, F30602-96-C-0161,DABT63-96-C-0027, and N66001-97-C-8532, by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202, and by the Department of Energy under contracts DOE B-341492, W-7405-ENG-48, and 1-B-333164.

act with executing software, tuning its behavior to meet evolving resource needs and availability.

This paper outlines an online approach to performance analysis where application developers and performance analysts are active participants during code execution. The prototype system integrates measurement and immersive performance visualization, allowing users to steer and tune parallel and distributed applications. The remainder of this paper is organized as follows. We begin in §2 by outlining the features of an immersive performance environment for performance control. This is followed in §3 by a description of each components in our prototype environment. In §4, we outline our experiences using this environment to analyze the behavior of a large-scale solid rocket simulation code. Finally, §5 summarizes our conclusions and directions for future research.

2 Online Analysis Design Goals

As a motivating example, consider a distributed application that controls a remote radio telescope, transmitting raw data from the telescope site to a distributed data archive and concurrently convolving the data to create images for real-time visualization. Performance problems can arise as a result of unexpected interactions among diverse application, middleware, and network components (e.g., a processor scheduling decision on one system causing data transmission tasks to be starved, cascading to infrequent data transmission and, consequent lost bandwidth for transcontinental network pipelines).

The key points are that distributed applications require measurement and adaptation at many levels. The primary reason is not just the physical dispersion of the computational elements, but rather the greater complexity and frequency of component interactions and computation models, as well as the number of potential optimization gradients. However, raw data alone is insufficient — it must be correlated across levels, allowing semantic "drill down" to explore both the proximate and underlying causes of poor performance.

To realize the goal of online interaction and control of distributed computations, we minimally require integration of four critical features:

- *performance immersion* for visualization and interaction with complex data,
- *collaboration support* for distributed application teams,
- *flexible measurement tools* for application instrumentation, and
- *portable data capture and steering systems* for extracting performance data and controlling application behavior.

Below, we describe each of these features and the rationale for their use.

2.1 Immersion and Collaboration

From a sensory perspective, the computer need not be simply a low-bandwidth medium of interaction between users and a passive software structure; instead, via immersive virtual environments users could interact directly with the executing software (e.g., manipulating hierarchical graphs of software structure). In this model, the user is no longer an external observer of the system and its behavior, but an active participant with the system. This immersion in and direct manipulation of complex software systems would exploit our real-world abilities to react directly and reflexively based on sensory input and make possible interactive software configuration.

For complex, distributed applications with evolving behavior, such immersive visualization systems should illuminate application dynamics. Given application complexity, hierarchical representations would allow analysts to identify the relevant software components and their interactions, beginning with high-level abstractions and "drilling down" for additional detail where needed (e.g., from a geographic display of wide-area communication to procedure call dynamics).

Complementing immersion, collaboration tools are needed because metacomputing applications are often developed by geographically distributed teams. Members of these teams must collaborate across both time and space. Coupling the virtual environment to remote desktops and mobile devices enables distributed collaboration (e.g., shared discussion and collaborative manipulation of software and its representations), whereas annotation mechanisms allow researchers to identify and mark performance problems for later exploration by other collaborators.

2.2 Flexible, Portable Instrumentation

Given the heterogeneous nature of computational grids and the coupling of unique instruments, applications are often composed of modules written in multiple languages, exploit computer-controlled devices, and are executed on multiple architectures. The distributed radio telescope application described above is not unique; a wide range of unique scientific instruments and their data are now accessible via the Internet.

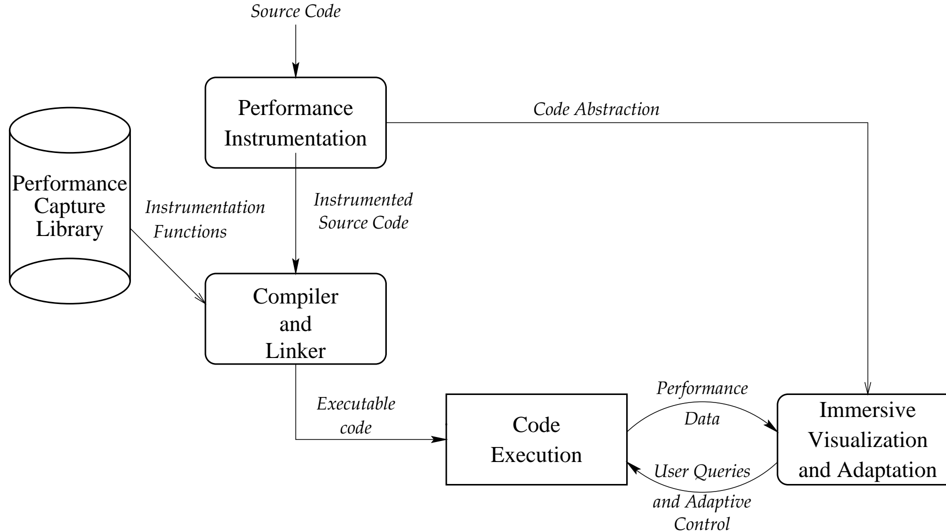


Figure 1. Instrumentation and Analysis Environment

Similarly, multilingual software toolkits are increasingly common. As an example, the Cactus toolkit for computational cosmology [2] consists of a general framework written in C with computational modules (thorns), written in Fortran 90. Consequently, any performance instrumentation and analysis tools must be cross-architecture and language independent.

Within this context, application and library code can be instrumented in a variety of ways. Interactive instrumentation provides detailed control, allowing users to specify precise points where performance data should be captured, albeit at the possible expense of excessive perturbation and inhibition of compiler optimizations. In contrast, automatic instrumentation relies on parsers, compilers or runtime systems to insert measurement probes. Although this decreases the probability of excessive perturbation, it sacrifices user control of instrumentation points. Because both approaches are appropriate in different circumstances, an instrumentation system should support both interactive and automatic instrumentation for a broad range of languages and architectures.

2.3 Portable Data Capture and Manipulation

Implicit in instrumentation of multilingual, distributed applications is the need for distributed data capture, extraction, and correlation. Moreover, given the complexity of component interactions, this instrumentation must capture data from multiple hardware and software levels, ranging from low-level processor and memory performance to wide-area network and storage access.

Complementing measurement, distributed control mechanisms should allow users or control software to modify run-time parameters or resource allocation during the application's execution. These tools can be operated by the user, while immersed in the virtual environment, or by an intelligent decision procedure.

3 Immersive Performance Analysis

Based on the ideas outlined above, we designed and developed a prototype environment for immersive performance visualization and analysis of applications that execute on a distributed collection of heterogeneous computing systems. Figure 1 shows the structure of this environment.

First, the application source code is instrumented with calls to software probes. These probes enable both capture of dynamic performance data and run-time access to the application for real-time adaptive control. During this phase, the application code is also analyzed to generate static call graphs and region graphs. The instrumented code is then compiled and linked with a performance data capture library.

During execution on the target distributed architecture, the instrumentation and control systems interact with the immersive visualization environment, providing a dynamic mechanism for performance evaluation and code adaptation. In turn, the visualization system generates three-dimensional views of application execution by combining static data (i.e., source code and call graphs) with dynamic performance data. By interacting with the hierarchical, three-dimensional representations of the application, users can interactively

```
main(int argc, char **argv)
{
  ...
  printf("start main\n");
  for(i=0; i < bounds; i++){
    ...
  }
  MPI_Bcast( ... );
  proc1();
  proc2();
  ...
}
proc1() {
  printf("start 1\n");
  for(i=0; i < bounds; i++) {
    ...
  }
  MPI_Sendrecv( ... );
}
proc2(){
  printf("start 2\n");
  MPI_Recv( ... );
  for(i=0; i < bounds; i++){
    ...
  }
  if ( ... ) {
    proc1();
    proc2();
  }
}
```

Figure 2. Call and Region Graph Example

pose performance queries or steer the application during execution.

Below, we elaborate on each of these phases, instrumentation, data capture, and visualization, emphasizing how the components interact and support one another.

3.1 Performance Instrumentation

To capture performance data and create control points for adaptive control, application source codes are instrumented via SvPablo [4, 7], a multi-language toolkit for instrumenting code and browsing dynamic performance data.[1] SvPablo parses the code and identifies all instrumentable regions, defined as the outer loops of loop nests and all procedure calls. This restriction balances instrumentation detail against perturbation, providing detailed insight while not overly distorting application behavior. Code can be instrumented interactively, via the SvPablo graphical user interface, or automatically, using parser command line options.

[1] Although SvPablo also supports desktop browsing of application source code and associated performance data, in this context, SvPablo is used only for instrumentation and data capture. Data analysis and visualization are provided by Virtue, described in §3.3.

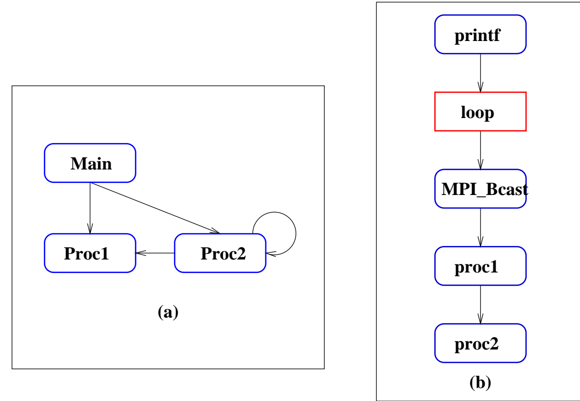


Figure 3. Associated Graphs for Figure 2

In addition to emitting instrumented source code, SvPablo generates static call graphs and code region graphs. Virtue, our immersive environment, uses these graphs as a basis for behavioral animation during or after application execution. As an example, the left of Figure 3 shows the call graph for the code in Figure 2 — each graph vertex represents a user-defined procedure with graph edges indicating the procedure call hierarchy. For more detailed analysis, code region graphs (e.g, like that shown on the right of Figure 3) define the control flow among instrumented constructs within each instrumented procedure.

3.2 Performance Data Capture

In a distributed execution environment, software components execute on a wide variety of physically distributed system components, requiring the performance measurement system to capture, extract, and correlate data from multiple sites. In our implementation, Autopilot [8] provides this real-time, distributed measurement.

More generally, Autopilot is designed to enable real-time, adaptive control of parallel and distributed applications. Autopilot is built atop Globus [5], which provides a shared address space across local and wide-area networks, as well as support for interprocess, intraprocess and intermachine data sharing. Globus also supports heterogeneity, allowing a single computation

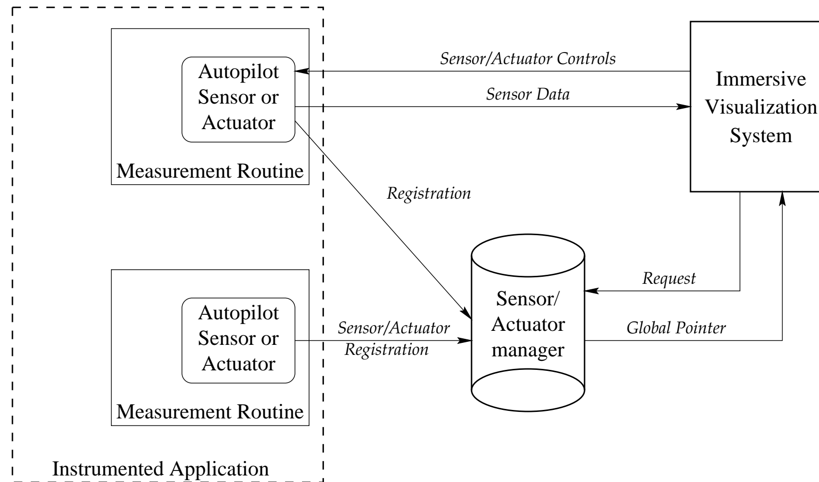


Figure 4. Software Component Interactions

to use multiple communication protocols, executables, and programming models.

Using Globus as a base, the Autopilot toolkit defines sensors, actuators, decision procedures, and sensor/actuator managers, all accessible via Globus "global pointers." Sensors are low overhead routines designed to capture real-time performance data from distributed software components and can be extended via user-defined functions to process raw performance data before transmission (e.g., computing a profile from event trace data). In turn, actuators define a mechanism for implementing control functions in software modules (e.g., changing policies or policy parameters in response to remote decision procedures).

Decision procedures implement fuzzy logic control of distributed software. They accept and evaluate real-time data from one or more sensors and generate actuator outputs in response. Finally, sensor/actuator managers can be viewed as access points for sensors and actuators. Decision procedures and visualization systems can query managers with sensor and actuator attributes and retrieve global points to all sensors and actuators that match the specified attributes.

Figure 4 illustrates the interactions among the application, Autopilot components, and the immersive visualization system. The application measurement routines inserted by SvPablo serve two roles. First, they gather performance data for each instrumented code region. Second, they define an interface between the application and Autopilot sensors and actuators.

By encapsulating Autopilot sensors and actuators within the SvPablo measurement routines, we isolate all sensor/actuator interactions from application code. This allows us to use the same instrumentation API for the SvPablo desktop user interface and Virtue immersive performance display system. It also makes the instrumented code portable — one can change the display mechanism without changing the instrumentation.

In addition to basic software performance metrics, such as elapsed times and execution counts, the extended SvPablo measurement routines can collect hardware performance metrics via processor performance counters [4]. Taken together, the hardware and software measures provide a complete set of metrics, enabling users to conduct a full behavioral analysis of the application.

As an example, Figure 5 shows a typical Autopilot sensor. Sensor properties identify the specific sensor and are registered with the Autopilot manager. Remote tasks, including the immersive visualization system, can obtain global pointers to sensors via property-based queries. The five data items are the performance data provided by the sensor in to all remote tasks that have attached to the sensor.

Although Autopilot sensors can operate in a wide variety of modes, SvPablo uses these sensors in four specific ways. Each supports a different data transmission policy and level of detail, enabling users and remote systems to balance detail and overhead.

Tracing sensor: Sensors of this type record each instance of measured data in an internal buffer. After activation, tracing sensors transmit buffered data to all registered recipients at the end of the execution of each code region. Within the immersive visualization system, data from tracing sensors allows users to explore detailed behavior in selected code regions. Because sensors can be dynamically enabled and disabled, Autopilot en-

```
PerfProp = new ApProperties(
          ApGlobal::contextAttributes,
          programName, managerName);
PerfProp->addProperty("Name",
          "PerfSensor");
PerfProp->addProperty("ProcName",
          "svPabloSensor");
PerfProp->addProperty("SensorProperty",
          "SensorValue");
// Record sensor data
PerfSet = new ApDataSet("Perf_buffer",
          tag++, 5);
          PerfSet->addItem(ProcessID);
PerfSet->addItem(CodeRegionID);
PerfSet->addItem(Count);
PerfSet->addItem(Inclusive_Time);
PerfSet->addItem(Exclusive_Time);
```

Figure 5. Autopilot Sensor Structure

ables visualization systems to enable data extraction only when users request the data.

Hardware sensor: These sensors provide access to hardware performance data via processor performance counters. Like the other sensors, hardware sensors are configurable, providing controls for data extraction frequency and properties for attribute-based queries.

Window sensor: These sensors differ from tracing sensors in their transmission policy. Window sensors transmit performance data after a specified time frame (window). Performance data associated with all code regions executed during this window are buffered and then forwarded (i.e., creating profiles of application behavior during the specified window). By adjusting window and buffer sizes, one can vary data extraction frequency, enabling fine or coarse-grained analysis.

Summary sensor: Finally, summary sensors provide aggregate performance data at the end of application execution. Intuitively, one can view summary sensors as the limiting case of window sensors, with the window size equal to the application's execution time.

Using these sensors, the immersive performance visualization system acts as a performance analysis client. It attaches to performance sensors and actuators, obtaining performance data and controlling application behavior, respectively.

3.3 Immersive Performance Data Visualization

As we noted in §2, advanced visualization techniques offer the same potential advantages for performance analysis as they do for scientific and data visualization. As applications continue to become larger and more complex, understanding and optimization their behavior requires commensurately more powerful data visualization techniques.

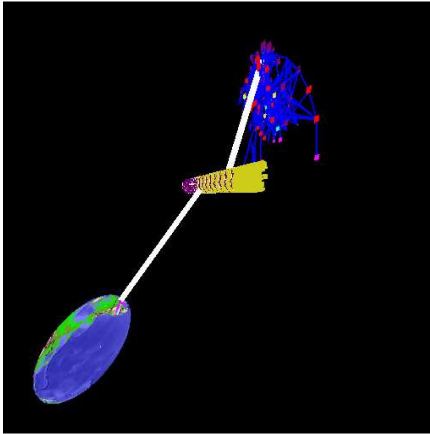
Equally importantly, geographically dispersed application teams need collaborative tools for code development and optimization. Different developers, connected only via networks, may need to visualize, analyze, and steer the performance of those code regions they created. To address these needs, we have drawn on insights from computer-supported cooperative work (CSCW) and virtual environments to create the Virtue environment for collaborative, hierarchical performance visualization and analysis [9].

Virtue is a general-purpose toolkit for visualizing hierarchical, three-dimensional graphs, with a rich language for mapping data to graph attributes (e.g., size, shape, color, and position), manipulating graphs and their representations, and annotating graph components with audio/video notes. A complementary collaboration system allows desktop and mobile collaborators to control graph displays and examine the associated data.

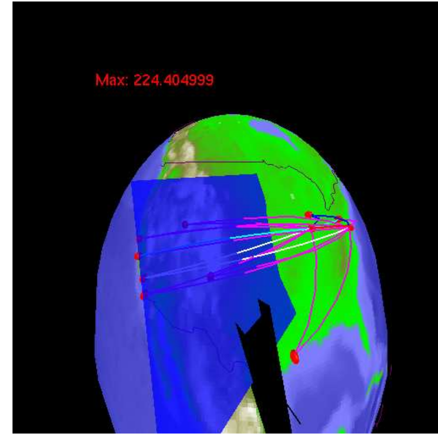
Virtue's graph description language is based on our extensible Self-Defining Data Format (SDDF) [1]. SDDF separates data structure from semantics, allowing one to quickly associate a new graph layout or data mapping with changes to only a few lines of the SDDF description. Because SvPablo also uses SDDF as its data representation, integrating graph descriptions (e.g., call graphs from SvPablo) with performance data mappings is straightforward. For example, the call graphs created by SvPablo instrumentation can be rendered as three-dimensional digraphs, with procedure invocation counts and execution times mapped to vertex size and color.

Virtue supports both post-mortem and real-time visualization — the SvPablo measurement system can write either SDDF files for post-mortem analysis or use Autopilot sensors (see Figure 4) to transmit performance data in real-time to Virtue. Moreover, Virtue provides a set of multimedia tools that enable distributed collaboration for behavioral analysis.

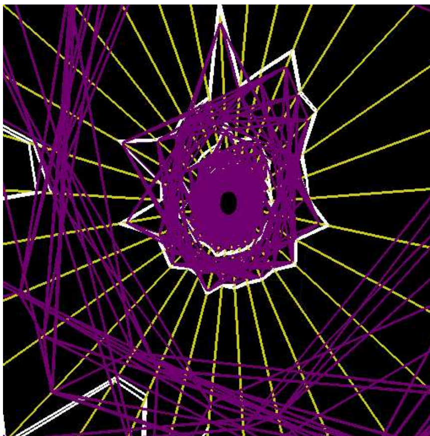
Within Virtue, the display hierarchy begins at a geographic scale, showing wide-area communication costs. A cutting plane, one of Virtue's suite of direct manipulation tools, allows users to compute metrics on



(a) Multilevel Display Hierarchy



(b) Geographic Display with Cutting Plane



(c) Time Tunnel Display with Communication

(d) Call Graph Display

Figure 6. Virtue Performance Displays

bisected graph edges.[2] From this geographic perspective, one can select a site and "drill down" to a time tunnel display of task interactions. The time tunnel consists of a cylinder of task lines, color coded based on activity type, with task interactions (e.g., message passing) shown as chords that cut through the cylinder interior.

From the time tunnel, one can select a task and drill down further to view the call or code region graph for that task. In Figure 6(a) for example, the task graph is being interrogated with a magic lens, one of Virtue's tools for viewing additional data associated with a graph objects. Finally, from the task graph, one can select a task and drill down to the source code associated with that task.

Virtue also provides three-dimensional interfaces to control actuators, not shown in Figure 6, that allow one to adjust the rate window sensors transmit data. Moreover, if actuators have been embedded in the application code, one can also manipulate application behavior.

4 Experimental Results

To evaluate the behavior and performance of our integrated measurement and immersive performance visualization system, we used SvPablo to instrument a large application from the Center for Simulation of

[2] In Figure 6(a), for example, the value shown is the maximum network latency for the bisected edges.

Advanced Rockets (CSAR). CSAR, one of five Department of Energy Academic Strategic Alliance Program (ASAP) sites. Given recent and sometimes spectacular failures of solid rocket boosters, the goal of this work is to create high resolution, fully three-dimensional, integrated modeling and simulations of complex component interactions and to understand potential instabilities and failure modes.

At present, the CSAR application consists of roughly 40,000 lines of F90 MPI code that models the fluid flow, combustion, and structures components of the space shuttle solid rocket booster (SRB) during a brief period. It is estimated that 200 hours on a 128 processor SGI Origin 2000 will be required to model only the first 0.5 seconds after SRB ignition. However, the ultimate goal is to model the entire two minutes of SRB burn.

Figure 6(c)-(d) shows a series of snapshots from execution of the CSAR code on 32 processors of an SGI Origin 2000, which required roughly 45 minutes and simulated 50 microseconds of the SRB burn.[3] In the figure, the yellow (gray) time tunnel lines correspond to execution of the structures solver, with MPI reductions and send/receives visible at the rear of the tunnel. Similarly, the call graph display shows several of the time intensive procedures of the code.

In general, our visualizations, both real-time and post-mortem, revealed that there were major opportunities for optimizing the CSAR code's initialization, which consumed a non-trivial fraction of its total execution time. Moreover, we have identified opportunities for dynamically adjusting the convergence criteria and execution balance across the fluid and structure components of the code. Working with CSAR application developers, we believe this offers major opportunities for tuning, based on propellant and other code parameters.

5 Conclusions and Future Work

The complexity of emerging distributed applications mandates real-time analysis and tuning of system and application performance. To meet this need, we have designed a prototype system that integrates collaborative, immersive performance visualization with real-time performance measurement and adaptive control of applications on computational grids. This system combines the SvPablo instrumentation system, the Autopilot real-time adaptive control toolkit, and the Virtue virtual environment.

The combination of these three tools enables physically distributed collaborators to explore and steer, in real-time, the behavior of complex software. Users can pose interactive queries and modify application parameters and behavior during execution.

Although we have validated our approach with a large-scale application, much work remains. In particular, we are developing new behavioral controls, enabling users to control a broader range of application behavior. Finally, we are exploring techniques that would fully couple the virtual environment to mobile devices for "anywhere, any time" access and control.

[3] In this test case, we executed the code at only one site.

References

[1] R. Aydt. The Pablo Self-Defining Data Format. Technical report, Department of Computer Science at the University of Illinois at Urbana-Champaign, April 1994.

[2] C. Bona, J. Masso, E. Seidel, and P. Walker. Three Dimensional Numerical Relativity with a Hyperbolic Formulation. *Physics Review*, to appear.

[3] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, November 1995.

[4] L. DeRose, Y. Zhang, and D. Reed. Svpablo: A Multi-Language Performance Analysis System. In R. Puigjaner, N. Savino, and B. Serra, editors, *Computer Performance Evaluation Modelling Techniques and Tools*, pages 352–355. Lecture Notes in Computer Science, vol. 1469, Springer-Verlag, September 1998. 10th International Conference, Tools'98.

[5] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997.

[6] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, Sept. 1991.

[7] D. A. Reed, R. A. Aydt, L. DeRose, C. L. Mendes, R. L. Ribler, E. Shaffer, H. Simitci, J. S. Vetter, D. R. Wells, S. Whitmore, and Y. Zhang. Performance Analysis of Parallel Systems: Approaches and Open Problems. In *Proceedings of the Joint Symposium on Parallel Processing (JSPP)*, pages 239–256, June 1998.

[8] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceecings of the Seventh IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 172–179, 1998.

[9] E. Shaffer, R. Aydt, and S. Whitmore. Virtue: A Collaborative Virtual Environment for Data Visualization. Technical report, Department of Computer Science at the University of Illinois at Urbana-Champaign, February 1998.

[10] J. C. Yan, S. R. Sarukkai, and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience*, 25(4):429–461, April 1995.