Development of an Intelligent Monitoring and Control System for a Heterogeneous Numerical Propulsion System Simulation[1]

John A. Reed
Abdollah A. Afjeh
University of Toledo

Henry Lewandowski
Cleveland State University

Patrick T. Homer
Richard D. Schlichting
University of Arizona

Abstract

The NASA Numerical Propulsion System Simulation (NPSS) project is exploring the use of computer simulation to facilitate the design of new jet engines. Several key issues raised in this research are being examined in an NPSS-related research project: zooming, monitoring and control, and support for heterogeneity. The design of a simulation executive that addresses each of these issues is described. In this work, the strategy of *zooming*, which allows codes that model at different levels of fidelity to be integrated within a single simulation, is applied to the fan component of a turbofan propulsion system. A prototype monitoring and control system has been designed for this simulation to support experimentation with expert system techniques for active control of the simulation. An interconnection system provides a transparent means of connecting the heterogeneous systems that comprise the prototype.

1. Introduction

Designing and implementing new propulsion technologies can be an expensive and time-consuming process. The Numerical Propulsion System Simulation project, sponsored by NASA Lewis Research Center, is bringing new computer simulation techniques and parallel hardware to bear on this problem [Claus92, Claus91]. Specifically, it is fostering the development of parallel simulations to improve both the execution time and accuracy of the simulations. A simulation executive will be developed that will support complete engine simulations made up from improved component simulations. Research on the simulation executive includes developing the monitoring and control techniques needed to manage the simulation, and exploring the use of expert systems techniques to assist the user in controlling the simulation.

Several key issues must be addressed in the design of the propulsion system simulation. One is the integration of simulation codes at different levels of fidelity. Low fidelity modelling requires empirical data that are not available at the preliminary design stage. On the other hand, high fidel-

[1] This work has been supported in part by the following grants: NSF grant ASC-9204021, NASA grants NGT-50966, NAG3-1560, and NCC-3-207.

ity modelling overcomes this limitation, but at a substantial computational cost. *Zooming* allows selected components to be modelled in detail and integrated into a low-level engine simulation. Additionally, during a low fidelity simulation, zooming provides a means of selectively examining in detail the physical processes within components of the engine.

A second issue is the use of a monitoring and control system. A monitoring tool will allow the user to observe the progress of the simulation through displays of its key parameters. An expert system can further improve the simulation by continuously monitoring and actively steering the simulation. This requires support in two areas: The first area is the collection of knowledge and the formulation of rules that govern the design and operation of jet engines. The second area is the integration of expert system software into the simulation executive to assist the user in executing the simulation.

A third issue is heterogeneity. The engine component codes and the expert system take advantage of a variety of vector and parallel platforms, and employ a variety of programming models and languages. An interconnection system allows components to execute on the most appropriate platform with minimum effort on the part of the user and the scientific programmer. The user should not see individual simulations that execute in isolation, but rather a single integrated simulation.

This paper describes a prototype simulation executive designed to address all three issues: zooming, a monitoring and control system, and heterogeneity support. The prototype employs a one-dimensional model of a complete engine. In this model, the operational characteristics of the individual system components are supplied in the form of performance maps that are constructed from experimental data. To provide descriptions of the physical processes occurring in an engine component beyond that supplied by a performance map, a higher fidelity component simulation is used. The simulation executive uses a monitoring tool that provides information about the engine simulation to the user and the expert system. Based on this information, the expert system can provide warnings and errors to the user and will be able to actively steer the engine simulation. Heterogeneity is addressed in the simulation executive through an interconnection system that provides the software framework to connect the various tasks.

Section two describes an engine model that demonstrates zooming on the fan component of NASA's Energy Efficient Engine [Davis85]. Section three describes the design of a monitoring tool and expert system that assists the user in executing the simulation and will be used to explore

techniques for intelligent control. The last section gives the current status of the project and outlines some of its future directions.

2. Simulation Strategy

The simulation strategy utilizes a high-fidelity flow solver in a low-fidelity simulation, and has been implemented in a prototype zooming framework consisting of the following systems:

- TESS - A propulsion system simulator [Reed93] running with the Application Visualization System (AVS) [AVS92].
- ADPAC - a fully three-dimensional Navier-Stokes/Euler flow analysis package capable of providing detailed flow analysis of the fan component in a turbofan engine [Hall93].

This system is depicted in Figure 1. Here the fan component of the one-dimensional "baseline" engine model has been "zoomed" to a three-dimensional analysis.

2.1 Zooming

Implementation of the zooming concept is difficult, due mainly to the inability to accurately resolve high-fidelity data fields from a single value as supplied from the low-fidelity system simulator. In order for the zooming to be accurate, the upstream and downstream boundary values (which are single valued), must be extrapolated to define a suitable three-dimensional distribution of field variables such that when integrated over, the original single-valued boundary conditions are recovered.

This process begins with the single inlet boundary values for stagnation pressure, stagnation temperature, and Mach number, and the exit boundary value of static pressure from the fan component of the one-dimensional engine model. These are then extrapolated to appropriate three-

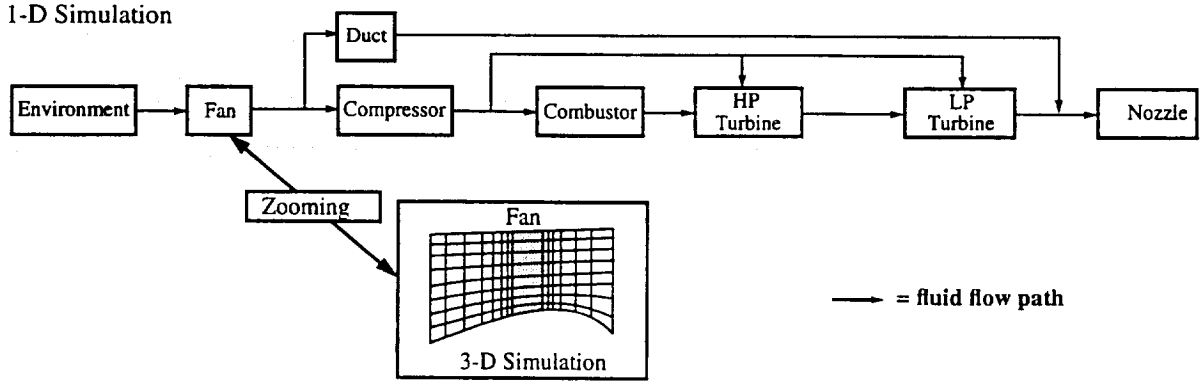


Figure 1: System schematic representing *zooming* on the fan component

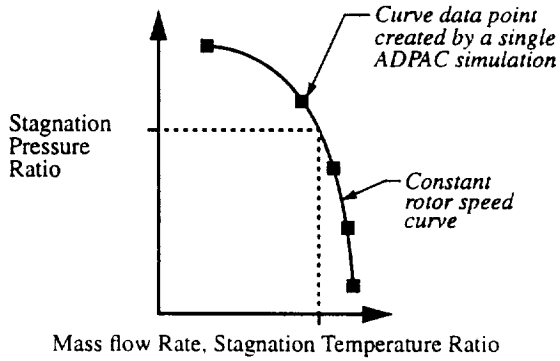


Figure 2: Single curve fan map created by zooming

dimensional field distributions and applied as boundary conditions to the fan simulation. The results of the fan 3-D simulation are then integrated to determine the mass-flow rate, and the mass-averaged values of outlet/inlet ratios for the stagnation pressure and stagnation temperature. The averaged stagnation pressure ratio is then compared with the stagnation pressure ratio computed across the engine model. If the values are identical, then the extrapolated field distributions are proved to be suitable representations and the averaged values of mass-flow rate and stagnation temperature ratio may be used in the one-dimensional simulation.

Typically, however, the averaged stagnation pressure ratio will not initially match the low-fidelity simulator value, and the three-dimensional boundary condition representations must be redefined and the above process repeated until the necessary match is found. An iterative approach to boundary value matching was found to be computationally unstable, requiring many iterations to achieve a balance. Worse, in many instances, the iterative approach led to an oscillatory mode where convergence could not be achieved.

A solution is the construction of a performance map from multiple runs of the three-dimensional component. A single-curve performance map, such as that shown in Figure 2, is constructed and the appropriate value can then be chosen from the map, interpolating as needed. To shorten the overall time for the simulation, the multiple runs can be performed in parallel when the necessary computational resources are available.

2.2 Simulation Tools

This section presents an overview of TESS and ADPAC. Also presented is PVM [Sunderam90], a message-passing package, that transfers data and control between TESS and the multiple 3-D fan

simulations needed to implement the zooming strategy.

2.2.1 Turbofan Engine System Simulator (TESS)

The low-fidelity system simulator used in the current research is the Turbofan Engine System Simulator (TESS). TESS is an object-based, one-dimensional, transient, thermodynamic aircraft engine simulator which runs under AVS. This integrated system provides the graphical user interface and operating environment for construction of arbitrary engine configurations, selecting and controlling steady-state and transient engine operation, and graphical display of simulation results.

The Network Editor of AVS provides a visual interface for creating dataflow programs. For TESS, the dataflow is used to model the flow of air through the engine. Engine components (e.g., compressor, turbine, duct, etc.) are represented graphically as AVS module icons, or simply *modules*. Each module has a control panel where the operational characteristics of the engine component are defined by the user (e.g., the mass flow rate, design point performance data). An engine is created by selecting the modules needed and placing them in the work space of the Network Editor. The dataflow network is then created by connecting the modules to establish the physical connections of the engine. Figure 3 shows a typical TESS engine network that models a two spool, two stream turbofan engine.

Once all of the components have been connected and their operational parameters have been entered, the user selects the length of time for the transient, and defines how the governing equations are to be solved numerically for both the steady-state and transient portions of the simulation. Currently, for steady-state solutions, the user may choose either Newton-Raphson or Fourth-order Runge-Kutta methods. For transient solutions, the user may choose either Modified Euler, Fourth-order Runge-Kutta, Adams, or Gear methods. When simulation execution is begun, TESS first attempts to balance the engine at the initial operating point using the steady-state balancing method. Once the engine is balanced, the transient is begun and proceeds up to the number of simulation seconds defined by the user.

2.2.2 Advanced Ducted Propfan Analysis Code (ADPAC)

The high-fidelity flow solver program used to model the operation of the fan component is the Advanced Ducted Propfan Analysis Code (ADPAC) [Hall93]. ADPAC is a three-dimensional Euler/Navier-Stokes numerical analysis tool developed to study high-speed ducted propfan aircraft propulsion systems. The program utilizes a three-dimensional, time-marching numerical

procedure along with a flexible, coupled 2-D/3-D multiple block geometric grid representation to predict the flow field in and around the fan. Multiple runs of ADPAC are needed to create the single-curve performance map used in the zooming strategy.

2.2.3 Parallel Virtual Machine (PVM)

PVM is a message-passing system that permits a network of heterogeneous Unix computers to be used as a single large parallel computer. Using PVM, a user-defined collection of different computers, known as the *virtual machine*, is used to provide aggregate power for solving large computational problems

The PVM system is composed of a daemon which resides on all of the computers making up the virtual machine, and a library of PVM interface routines which supply user-callable routines. These functions, along with the PVM daemon, allow a PVM application on one computer to auto-

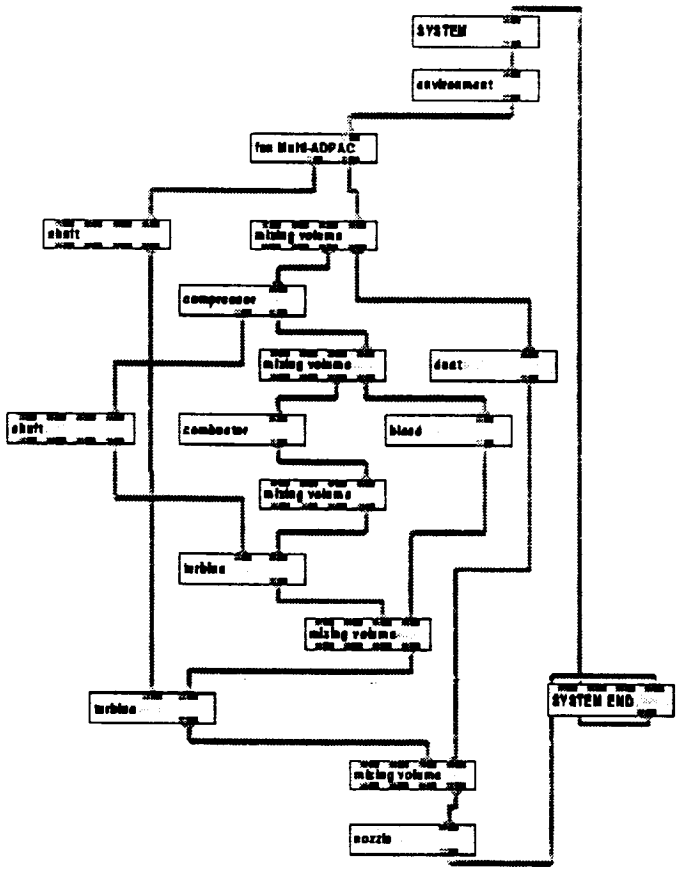


Figure 3: TESS engine model dataflow network

matically start up *tasks* (computational processes) on other computers in the virtual machine and communicate data among the tasks by sending and receiving messages.

2.3 Prototype Zooming System

The prototype zooming system is defined by two suites of codes: The first suite, residing on the user's workstation, runs AVS and TESS. The second suite consists of ADPAC and associated codes [Reed94]. One instance of this second suite exists for each of the multiple fan simulations used in the zooming strategy.

A new TESS engine component module, *fan Multi-ADPAC*, was created to provide the user interface and functionality for the zooming system. The module

- Handles the basic AVS data transfer for the fan component within TESS,
- Establishes the PVM virtual machine,
- Spawns the remote ADPAC tasks, and
- Controls the data transfer between TESS and the ADPAC simulations.

To utilize the *fan Multi-ADPAC* module in a TESS engine simulation, the user defines the ADPAC control parameters and the remote machines on which to spawn the ADPAC simulations. Figure 4 shows the AVS pop-up windows used to accept this input from the user. PVM daemons are started on each remote machine specified by the user to create the virtual parallel machine.

Each time TESS needs fan performance data during a simulation, *fan Multi-ADPAC* creates the needed remote instances of ADPAC on the virtual machine and sends each its boundary condition parameters. *fan Multi-ADPAC* then waits for the simulation results. Each result is matched with its boundary conditions, then used to create data points on the performance curve (see Figure 2). Once all the values have been received, the performance curve is interpolated to match the stagnation pressure ratio across the fan, impressed by the TESS simulation, to determine the stagnation temperature ratio and mass flow rate. These values are then used by TESS to continue the complete propulsion system simulation.

The source code for ADPAC is maintained by a separate group at NASA Lewis Research Center, and has been unavailable for this project. As a result, the operation of each remote instance of ADPAC requires several codes, illustrated in the flow chart in Figure 5. The first program, `makeinput`, creates the ADPAC input data file from the boundary parameters. Then, ADPAC executes, reading its grid file and the input data file. The output file produced by ADPAC

is then read by the third program, `mbave`. This is a multi-block averaging program and integrates the three-dimensional flow solution to give the single (space-averaged) flow values which are

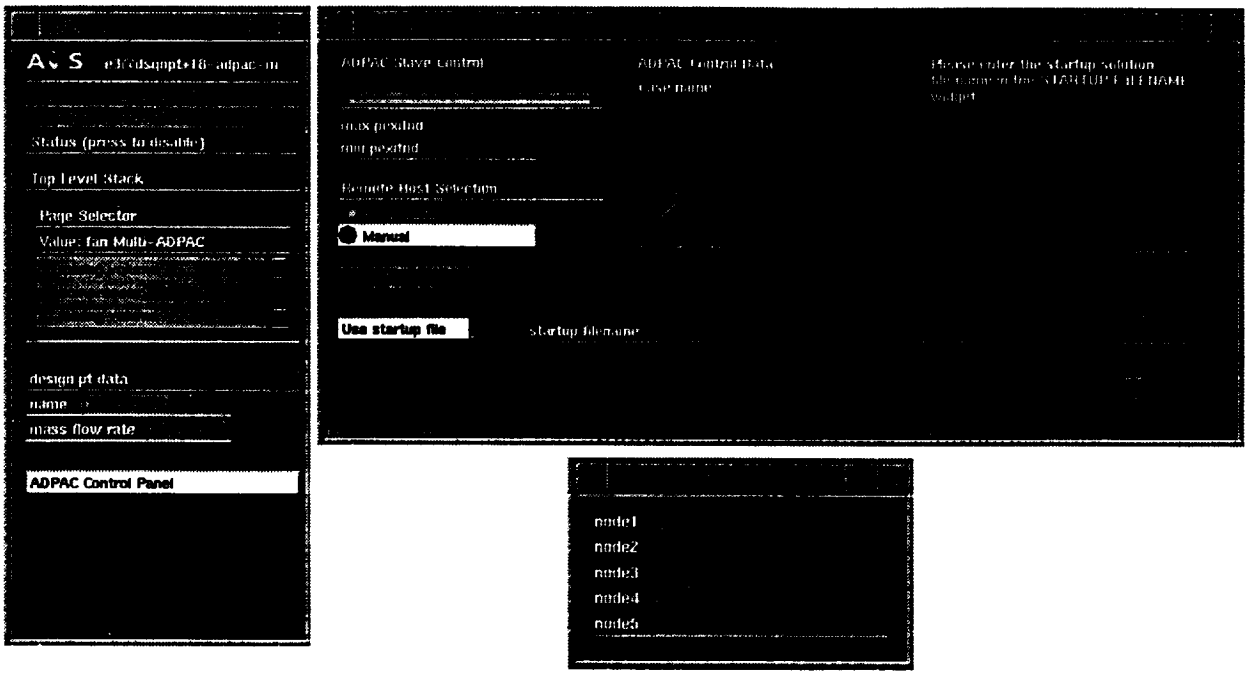


Figure 4: *fan Multi-ADPAC* module control panels

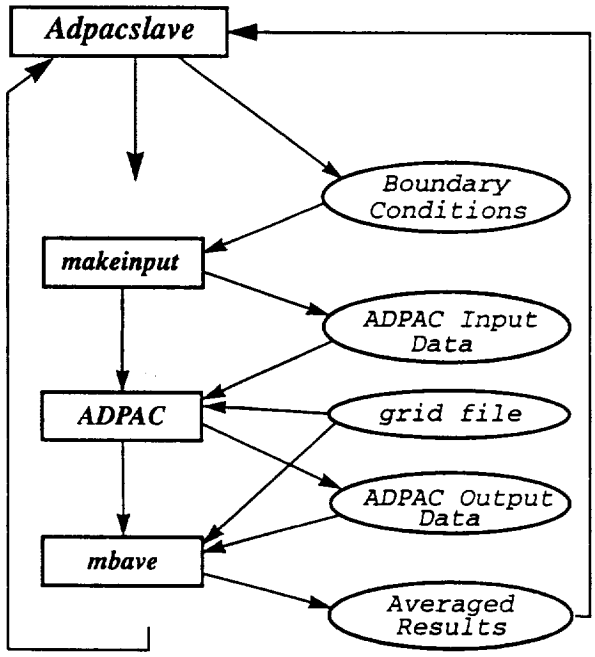


Figure 5: ADPAC code suite flow chart

needed by TESS. The `adpacslave` program coordinates the execution of the other three programs, and handles the PVM communications with *fan Multi-ADPAC* receiving the input data for `makeinput` and returning the results from `mbave`.

A new performance curve is created by *fan Multi-ADPAC* each time fan performance data is needed by TESS. To reduce the overall simulation time, the space-averaged values are retained and used to create an overall fan performance map. Before running flow solutions, this data is checked to see if the current operating conditions are within the data range. If so, the data is interpolated and used in the system simulation. In this manner, the simulation time may be significantly reduced. This also has the added benefit of creating an overall fan performance map which can be used in subsequent, non-zooming TESS simulations.

3. Intelligent Monitoring and Control

A monitoring tool and expert system have been designed for use with the TESS/ADPAC simulation. It will provide the user with information about the progress of the simulation and allow the inclusion of rules to steer ADPAC runs and determine when new performance curves are needed. The design of the system is complete and implementation is underway. The following systems are being used:

- Monitoring tool constructed using the TAE+ (Transportable Application Environment) package [TAE]
- Expert system constructed using CLIPS (C-Language Integrated Production System) package [CLIPS], and
- The Schooner interconnection system [Homer94a, Homer94b]

This section first describes the overall approach to the problem, then gives a description of each system used, and finally presents some details of the implementation.

3.1 Control Strategy

Intelligent monitoring and control is necessary due to the complexity of engine simulations. A large number of variables can affect the outcome of a simulation and monitoring them can place a severe burden on the user. Two types of problems that arise are physically unrealistic boundary conditions imposed on a component and numerical instabilities that arise within a component. One simple example is the addition of fuel to a combustor component. This should result in a rise in temperature through the combustor. As another example, the fan component should produce a

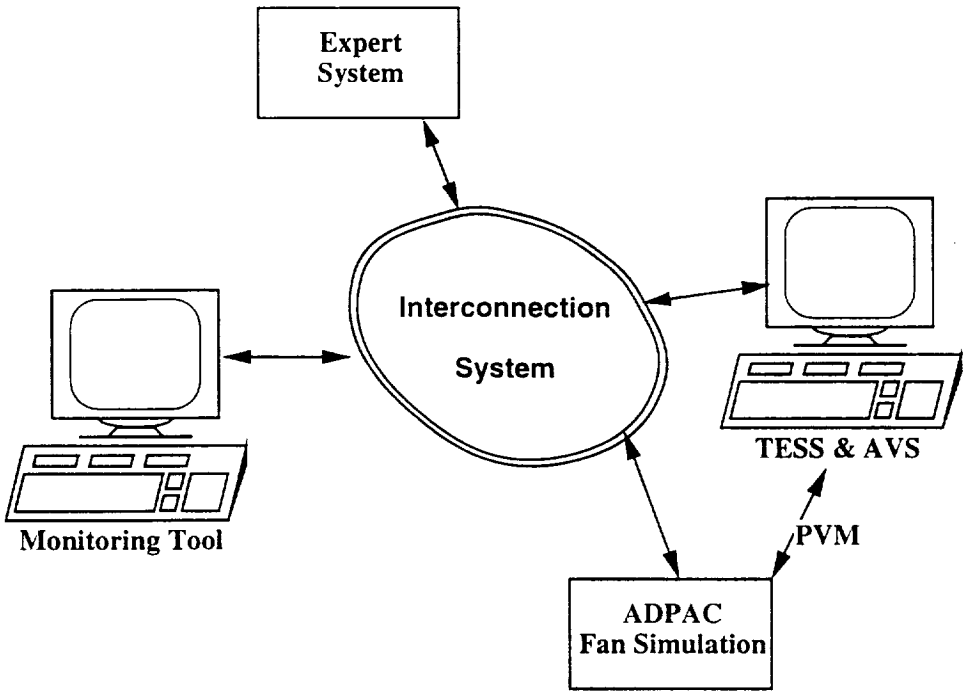


Figure 6: Monitoring and control system

rise in air pressure at the outlet. Both of these cases are relatively easy for an expert system to check. A more complicated example arises from numerical instabilities in the fan component. These can produce artificial, numerically-induced vortices in the air flow which reduce the effective area for flow through the fan and cause pre-mature choked flow. An expert system would need more complicated rules to detect such a problem and implement the series of corrective steps needed.

The immediate goal of this research is to build a monitoring and control system that can detect some of these types of problems and warn the user when they arise. The longer-range goal is the development of more complex rules and the corresponding controls to allow the expert system to actively steer the simulation. To realize the immediate goal, the monitoring tool displays information about the progress of each of the ADPAC simulations. The expert system receives data from the monitoring tool and passes back appropriate warnings to be displayed for the user. The interconnection system provides a transparent means of connecting the different parts into a single application. Figure 6 illustrates this system. For clarity, only one ADPAC instance is shown in the figure. As experience with the system is gained, the expert system will be extended to provide active control, through the interconnection system, of ADPAC and TESS.

3.2 Control System Components

This section presents an overview of the TAE+, CLIPS, and Schooner systems that are used in the monitoring tool and expert system.

3.2.1 TAE+

TAE+ is a package that supports the rapid prototyping and construction of X-windows graphical user interfaces. It provides a workbench that facilitates the design and layout of the application's windows, allowing easy placement of the various objects within each window. A programming tools package allows the user to add code to the interface to provide program control over the various objects that make up the interface. Finally, a code generator automatically generates code in a number of languages for creating the interface and building the main event loop for the application.

There are three basic building blocks available for use in designing windows for a TAE+ application. The first is a set of user-entry objects that allow the user to interact with the application through buttons, pull-down menus, and text fields. Second, there are data-driven objects that graphically display information from the application in real time through dials, strip-charts, thermometers, etc. The third category is information objects, such as text displays and help screens that provide the user with information or instructions about the application. The data-driven objects are particularly useful in a monitoring tool as they easily support receiving and reporting of continuous data during execution. A set of pre-defined objects are available that can be used to create vertical and horizontal scales, rotating dials, strip charts, etc. The user can also build objects specific to the application by creating a custom-object using the supplied drawing tools in TAE+ and defining the type of rotational, sliding, stretching, etc. data that will be supplied to the object. The data-driven object was a major reason for selecting TAE+ for this project, as it easily supports the type of monitoring needed for ADPAC.

3.2.2 CLIPS

An expert system built with CLIPS begins with a user-defined set of rules. The rules are written in a functional language and describe actions to take when specified conditions occur in the application. Typically, CLIPS supplies a window that displays information to the user about the application and shows the progress of the knowledge engine as the package works through the rules. A C language interface is also available that by-passes the CLIPS window and allows an application to

be tied directly to the knowledge engine. This latter feature is being used in this current project. The C interface consists of function calls that pass data to CLIPS and return results from the rules. Callbacks through the C interface allow CLIPS to pass control commands to other components in the system.

3.2.3 Schooner interconnection system

An interconnection system provides a model of computing that connects applications and implements a configuration management system, thus creating a *meta-computation* [Khokhar93]. Each application contains one or several computations that accomplish a specific set of tasks, for example, the TESS or monitoring tool applications in Figure 6. An application can be developed using the combination of programming language, model, or architecture that is most suitable. Thus, the meta-computation is a heterogeneous, distributed program. At runtime, each application exports operations that can be invoked from other applications. For example, the monitoring tool exports operations that can be invoked by instances of ADPAC to report their progress. The interconnection system transparently handles the transfer of data and control among the applications making up the meta-computation.

A meta-computation requires configuration tools to assist the user in starting and controlling the component applications. The configuration management features of the system give the user both static and dynamic configuration control. Static control allows the user to select the applications that will be needed, such as the expert system and monitoring tools, and to begin execution. Dynamic control then allows applications to be added or removed as needed by the user or through commands issued by the applications themselves. Dynamic control is used by the monitoring tool to establish and break connections with instances of ADPAC.

The Schooner interconnection system realizes this model of scientific computing by supplying a software configuration and control mechanism for executing heterogeneous distributed computations. There are four, mostly orthogonal, parts to Schooner: a specification language, and intermediate data representation and accompanying data exchange library, a set of stub compilers, and a runtime support system. The Universal Type System (UTS) provides both the specification language and the intermediate data representation [Hayes89]. The specification language is machine- and language-independent and is used to describe the interface for each component application. The UTS intermediate data representation provides a medium for exchanging data across machine architectures and handling data structure differences among languages. The stub compilers, one

for each supported language, read the UTS specifications and create the interface. The runtime system implements application-level remote procedure call (RPC) control transfer between components, as well as configuration and control features. It provides the user with a means of configuring the various applications in the computation, and provides the underlying communication and management support.

3.3 Prototype Monitoring and Control System

To accomplish the immediate research goals, a monitoring tool has been constructed that allows the user to observe the progress of ADPAC runs and provides information to an expert system that can raise several warning panels. The monitoring tool, designed with TAE+, consists of windows for each instance of ADPAC. Figure 7 shows a snapshot of one such window taken at the end of an ADPAC run. The name of the machine executing this instance of ADPAC is shown at the top center of the window. The chart on the lower-right portion is a strip-chart and plots the residual on a log scale over the most recent 100 iterations. The residual provides a measure of how well ADPAC is approaching convergence. Convergence is generally achieved when the residual has

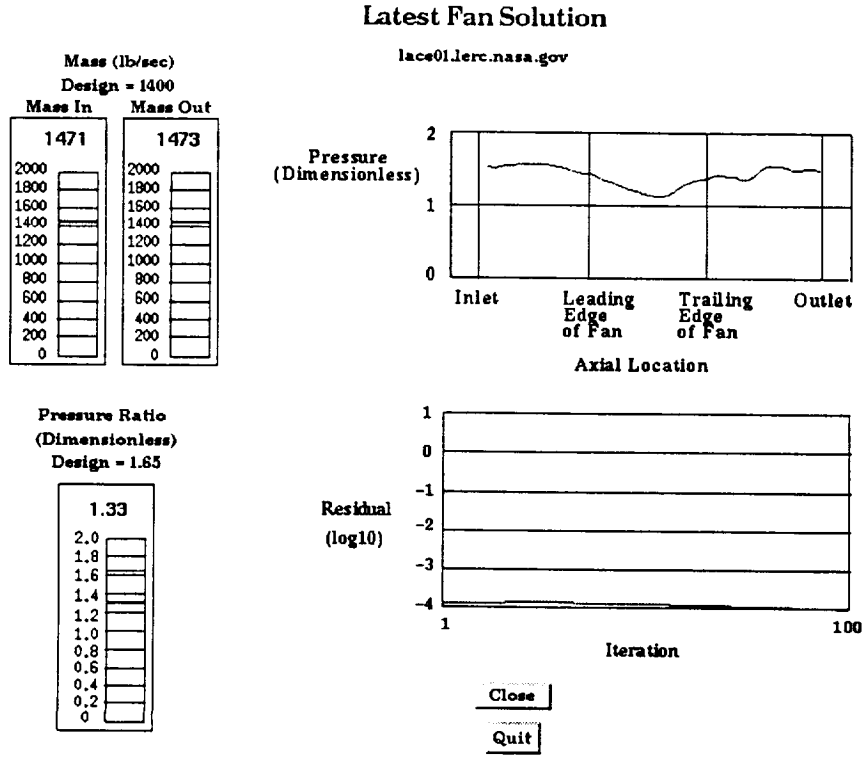


Figure 7: ADPAC Monitoring Tool

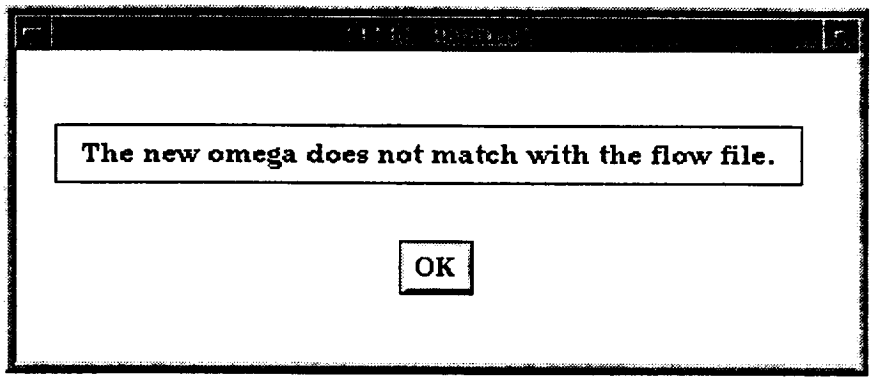


Figure 8: Pop-up warning window

dropped four orders of magnitude, while an oscillating residual is a symptom of a problem within ADPAC. Experience with TESS-ADPAC indicates that convergence is reached in most cases in 500 to 800 iterations. When a computation is finished, the strip-chart on the upper-right shows the pressure plot at 52 points along a slice through the fan. The vertical scales on the upper left report the flow rate of air into and out of the fan. The vertical scale on the lower left shows the final pressure ratio (outlet/inlet) computed by ADPAC. Each of the vertical scales also show the design point provided by the one-dimensional TESS model.

A prototype expert system receives data from the monitoring tool and provides several warning messages to the user in the form of pop-up windows. One example is shown in Figure 8. The monitoring tool uses the C language interface to CLIPS to pass data to the expert system. The pop-up windows, when appropriate, are then triggered by procedure callbacks from CLIPS to the monitoring tool.

Since the source code for ADPAC was not available for this project, the output data files are monitored instead. One of the data files is updated by ADPAC on each iteration during a run to report a number of quantities, including the desired residual, and several types of warnings. One limitation of this approach is an inability to affect ADPAC once execution has started. Thus, the expert system is currently limited to displaying warnings and errors in the monitoring tool, rather than being able to actively steer ADPAC.

To simulate the type of monitoring desired given the constraints, a watch-dog process is created on each machine executing ADPAC. This process uses an infinite loop to continuously check ADPAC's output file for new data. Whenever the file changes, the watch-dog examines the file for values of interest, specifically the residual values from each iteration and warnings of interest to

```
# starting watch-dog
export monitor_files dispatch( "filename" val string[40])

# residual report
import residual_report prog(
        "iteration" val integer,
        "residual"  val float)

# warning reports
import warning_report1 prog( "message"  val string[-])
import warning_report2 prog( "message"  val string[-])

# final reports
import mass_in_report  prog( "mass_in"  val float)
import mass_out_report prog( "mass_out" val float)
import pressure_report prog( "pressure_ratio" val float)
import pressure_plot_report prog( "pressure" val array[52] of float)
import last_report prog()
```

Figure 9: ADPAC Watch-dog UTS Specification

the expert system. It also reads the average results from the `mbave` program at the completion of the ADPAC run.

Schooner connects the watch-dog processes to the monitoring tool through the use of UTS specification files. The specification file for the watch-dog process is shown in Figure 9. An `import` specification indicates a service the watch-dog process will call from another component, in this case from the monitoring tool. An `export` is a service the watch-dog process provides to the monitoring tool. A analogous specification file is used with the monitoring tool. Each procedure specification lists the arguments using a Pascal-like syntax.

Execution of the watch-dog process is started after the monitoring tool receives from TESS the list of machines on which ADPAC is executing, and the corresponding list of output file names. Schooner's dynamic configuration library allows watch-dog processes to be started whenever needed by the simulation. Once the watch-dog has been started, the `monitor_files` dispatch is called. This tells the watch-dog the name of the output file to monitor, and starts the infinite monitoring loop. During execution, the watch-dog makes simple procedure calls, for example

```
residual_report(iteration, max_err);
```

is called to report the residual from the current iteration. The Schooner system transparently han-

dles communications and data conversions among the machines.

4. Current Status and Future Directions

This is an on-going research project. This section describes the current state of the implementation and then outlines some of our plans for expansion of the system.

4.1 Current State

The TESS-ADPAC system has been fully implemented. It has been tested with a subsonic engine model and compared with experimental data from the Energy Efficient Engine. The machine suite used for the tests consisted of a Silicon Graphics Iris 4D/440VGX at the University of Toledo for the TESS system. The ADPAC instances were executed on a variable number of nodes of the Lace cluster, a network of 32 IBM RS6000 workstations located at the NASA Lewis Research Center.

The monitoring tool was originally designed to monitor a single ADPAC instance. This was a result of the original zooming strategy which envisioned a single high-fidelity component simulation used in an iterative approach (see Section 2.1). The monitoring tool has been tested on a Sun Sparc 10 workstation located at the Lewis Research Center and monitoring an ADPAC run on a node of the Lace cluster. In this prototype, the expert system executes on the same platform as the monitoring tool, since the initial number of rules is small. The system is currently being extended to monitor all instances of ADPAC and allow the user to select all, or a subset, to observe. The expert system can be moved to a separate platform as soon as the complexity of the rules increases to the point where this will be necessary.

4.2 Future Directions

One obvious direction is to modify the source code of ADPAC to allow it to communicate directly with TESS and the monitoring and control system, rather than through its output files. A principal reason for not modifying the source initially is the desire to prove the feasibility of this approach and identify the specific changes desired. There is another positive feature to the approach of using the output files: It would be relatively straight-forward to substitute a different high-fidelity fan simulation and provide a similar level of monitoring through watching its output file. This technique allows for easy testing of different fan simulations without the initial need to involve the authors of the simulation.

The ADPAC code is currently being re-written to take advantage of parallel machines and workstation clusters. Once this work is completed, the parallel-ADPAC will be tested with the TESS system.

Another zooming approach being studied is to use an intermediate fan simulation, specifically a two-dimensional, axi-symmetric simulation. This has the advantage of not requiring as much execution time as the three-dimensional ADPAC simulation when less accuracy is needed. In addition, it will be possible in some cases to use the solution from the medium-fidelity simulation to jump-start the three-dimensional solution, thus shortening the execution time of the high-fidelity simulation.

Fault detection and fault tolerance techniques are being studied for use with the multiple ADPAC runs. Currently, the system does not gracefully handle the failure of an ADPAC instance. In general, the desired single curve performance map can be created even when one or two ADPAC instances fail, allowing the simulation to proceed. This is an area where rules are needed for the expert system so the user will not have to constantly monitor a long simulation in case a fault occurs.

Acknowledgments

The NPSS project is managed by the Interdisciplinary Technology Office (ITO) at NASA Lewis Research Center (LeRC). This work was performed in part on computing resources at the Advanced Computational Concepts Laboratory (ACCL) and the Computer Services Division at LeRC. Thanks are due to G. Follen, C. Putt and C. Miller of LeRC.

References

[AVS92] Advanced Visual Systems Inc. *AVS Developer's Guide* (Release 4.0), Part number: 320-0013-02, Rev B, Advanced Visual Systems Inc., Waltham, Mass., May 1992.

[Claus92] R. W. Claus, A. L. Evans, G. J. Follen. Multidisciplinary propulsion simulation using NPSS. *4th AIAA/USAF/NASA/OAI Symposium on Multi-disciplinary Analysis and Optimization*, Cleveland, OH (September 1992).

[Claus91] R. W. Claus, A. L. Evans, J. K. Lylte, and L. D. Nichols. Numerical propulsion system simulation. *Computing Systems in Engineering 2*, 4 (April 1991), 357-364.

[CLIPS] CLIPS Reference Manual, Basic Programming Guide. Software Technology Branch, Lyndon B. Johnson Space Center. CLIPS Version 5.1, September 10, 1991.

[Davis85] D. Y. Davis and E. M. Stearns. Energy Efficient Engine—Flight propulsion system

Final Design and Analysis. NASA CR-168219, contract report prepared by General Electric Company, August 1985.

[Hall93] E. J. Hall, R. A. Delaney, and J. L. Bettner. Investigation of Advanced Counterrotation Blade Configuration Concepts for High Speed Turboprop Systems, Task 5 — Unsteady Counterrotation Ducted Propfan Analysis Computer Program User's Manual, NASA CR-187125, Jan. 1993.

[Hayes89] R. Hayes. UTS: A Type System for Facilitating Data Communication, Ph.D. Dissertation, Department of Computer Science, University of Arizona, August 1989.

[Homer94a] P. T. Homer and R. D. Schlichting. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing 21*, (June 1994), 301-315.

[Homer94b] P. T. Homer and R. D. Schlichting. Using Schooner to support distribution and heterogeneity in the Numerical Propulsion System Simulation project. *Concurrency—Practice and Experience 6*, 4 (June 1994) 271-287.

[Khokhar93] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban and C. Wang. Heterogeneous computing: Challenges and opportunities. *IEEE Computer 26*, 6 (June 1993), 18-27.

[Reed94] J. A. Reed and A. A. Afjeh. Distributed and parallel programming in support of zooming in numerical propulsion system simulation, OAI/OSC/NASA Symposium on Application of Parallel and Distributed Computing, Columbus, Ohio. April 1994.

[Reed93] J. A. Reed. Development of an Interactive Graphical Aircraft Propulsion System Simulator. Master of Science Thesis, University of Toledo, August 1993.

[Sunderam90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience 2*, 4 (December 1990) 315-339.

[TAE] Transportable Applications Environment Plus. Programmer's Manual, Version 5.2. Goddard Space Flight Center, National Aeronautics and Space Administration. December 1992.

Opportunities and Tools for Highly Interactive Distributed and Parallel Computing

Greg Eisenhauer
Weiming Gu
Thomas Kindler
Karsten Schwan
Dilma Silva
Jeffrey Vetter

Georgia Institute of Technology
Atlanta, Georgia 30332

Abstract

Advances in networking, visualization and parallel computing signal the end of the days of batch-mode processing for computationally intensive applications. The ability to control and interact with these applications in real-time offers both opportunities and challenges. This paper examines two computationally intensive scientific applications and discusses the ways in which more interactivity in their computations presents opportunities for gain. It briefly examines the requirements for systems trying to exploit these opportunities and discusses Falcon, a system that attempts to fulfill these requirements.

1 Introduction

The world of computationally intensive computing is moving away from the batch-oriented style of processing. Users accustomed to spreadsheets and WYSIWYG word processing are not satisfied with the traditional hands-off, you'll-get-your-data-when-the-batch-queue-empties mode of running parallel programs. At the same time, high-speed network interfaces and the proliferation of high-end graphics workstations offer an opportunity to open new windows into application behavior. Falcon, a system developed at Georgia Tech, provides tools and techniques for exploiting these developments, and it uses them to create new opportunities for application understanding, debugging and tuning.

Traditional debuggers rely on halting the system in order to examine and modify the program state. While such debuggers are useful, they are often inadequate to detect the race conditions, synchronization errors or other problems endemic to parallel and distributed programs. Similarly, traditional uniprocessor code profilers and analyzers have a role in tuning parallel programs, but they are ineffectual for analyzing synchronization overheads, bursty computational demands, or other problems more unique to non-sequential applications. Neither type of tool provides the insight into dynamic program behavior that is often necessary to debug and tune parallel and distributed programs. Perhaps more importantly in the long term, neither type of tool encompasses mechanisms for dynamically manipulating running programs.

To address these deficiencies one needs mechanisms for "observing" a running application and "adjusting" its state or behavior. Collectively, these mechanisms are a *monitoring and steering* system[GVS94]. The on-line manipulation or steering of parallel and distributed programs has been shown to result in performance improvement in many domains. Examples of such improvement include the automatic configuration of small program fragments for maintaining real-time response in uniprocessor systems[MP89], the on-line adaptation of functional program components for realizing reliability versus performance tradeoffs in parallel and real-time applications [BS91, GS89, GS93], and the load balancing or program configuration for enhanced reliability in distributed systems[SGB87, MW91, Bec94].

Further benefits are gained if monitoring and steering mechanisms are not limited to system-level constructs but are instead made available in a reasonable way at the application level. In addition to supporting standard program tuning practices, application-level monitoring and steering have the potential to produce real gains in application productivity by allowing users to accomplish more useful work with the same number of compute cycles. How is this possible? Truly interactive parallel programs, created with application-level monitoring and steering, will give users significant insight into the *progress* of the computation. If users have access to the computation and the ability to guide or direct the computations at runtime, they have an unparalleled power to evaluate and experiment with the program.

Consider the case of scientific computing, where many applications are trying to model or simulate the real world. A truly interactive program would let users interact with that world as it evolves. Rather than planning a dozen batch-style simulation runs with a dozen different parameter values, the user can adjust the application dynamically and examine the response. Rather than discovering at the end of a twenty hour simulation run that the system wandered into an unreasonable state early on, it can be monitored for reasonableness as it progresses. These *process tuning* situations are often neglected because they do not fall into the traditional realm of program tuning or debugging. However, the advantages of additional high-level insight into the application in these cases are an important benefit of interactive parallel computing.

This paper first examines two computationally intensive scientific applications and discusses the ways in which more interactivity in their computations presents opportunities for gain. It then briefly discusses the requirements and techniques for exploiting these opportunities and examines the aspects of Falcon which fulfill these requirements. The paper also presents our conclusions and plans for future work.

2 Interaction Opportunities in Selected Applications

Parallel and distributed programming models and applications vary widely, as do the situations in which one might wish to employ a monitoring and steering system. The utility of a general, low-perturbation monitoring system in both debugging and performance tuning is widely accepted. Unfortunately, most such systems do not make their facilities easily accessible at the application level. If we are to realize the application-level process gains discussed in the introduction we must consider the demands of application-level monitoring and determine how steering might be used. This section examines two large parallel applications and discusses ways in which a monitoring and steering system might benefit each.

2.1 MD

MD is an interactive molecular dynamics simulation developed at Georgia Tech in cooperation with a group of physicists exploring the statistical mechanics of complex liquids [XORL92, EGSM94]. 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. The calculational cell is a square cylinder which is periodically repeated in the x-y directions. Temperature is controlled via infrequent scaling of the particles' velocities. The alkanes remain associated in a chain with very predictable bond lengths throughout the simulation. 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 as Figure 1.

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. 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 on-line 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

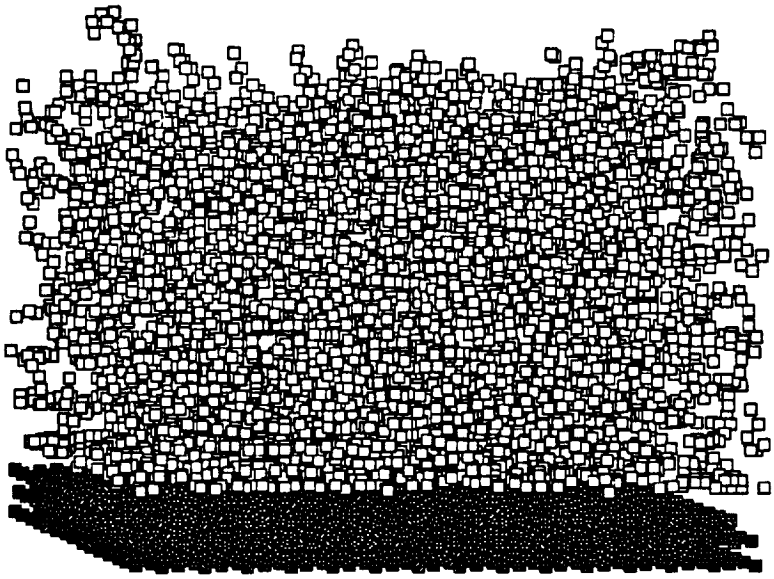


Figure 1: A visual representation of a sample system for the molecular dynamics simulation. The white-yellow particles are the pseudoatoms of the alkane chains. The red particles represent the gold substrate.

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).

The MD simulation offers many opportunities to improve the performance through both on-line interactions with the end user and program steering by algorithms, including:

- Decomposition geometries could be changed to respond 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.
- The interactive modification of the cutoff radius could improve solution speed by computing uninteresting time steps with some loss of fidelity, if this is desired by the end user.
- The boundaries of spatial decompositions could be shifted for dynamic load balancing among multiple processes operating on different sub-domains. This can be performed by an algorithm or by end users.
- Global temperature calculations, which are expensive operations requiring a globally consistent state, could be replaced by less accurate local temperature control. On-line analysis could determine how often global computations must be performed based on the temperature stability of the system.

From our experience with MD, we believe that these are important opportunities to exploit in order to increase the usability and efficiency of the application. For example, we have seen that the performance of the application is extremely sensitive to load balance shifts which can dramatically limit efficiency with even moderate numbers of processors. The ability to dynamically rebalance and perhaps even reconfigure the decomposition to match the evolving physical system is essential to performance for a long-running system.

2.2 Atmospheric Modeling

The simulation of complex global natural phenomena is one of the biggest challenges facing computational science because of its extreme computational and data handling requirements. The ultimate goal in climate

modeling, the simultaneous simulation on a global scale of physical and chemical interactions in ocean and atmosphere, is still far from reach. It is difficult to run and test a model with typical runtimes of hours for each simulation day. One simple reason for this is that changes to a model often do not have the desired effects upon the model results. This occurrence is particularly common when parameters must be chosen to simulate processes that are not well understood or whose influence can only be approximated at the scale of the current model. The result in these cases is a set of sometimes arbitrarily chosen parameters that must be adjusted individually. On-line visualization, interaction and program steering have potential to simplify and significantly shorten model development time and improve model results as well as to help to improve traditional measures of simulation performance.

Earth and atmospheric scientists at Georgia Tech have developed a global chemical transport model (GCTM)[KSS+94] which uses assimilated windfields [SO93] for the transport calculations. These types of models are important tools to answer scientific questions concerning the stratospheric-tropospheric exchange mechanism or the distribution of species such as chlorofluorocarbons (CFC's), hydrochlorofluorocarbon (HCFC's) and ozone. This model uses a spectral approach to solve the transport equation for each species. In a spectral model, all variables are expanded into a set of orthogonal spherical basis functions, called spherical harmonics. Derivatives with respect to the latitude or the longitude are more easily and accurately calculated in this spectral domain, though the variables must be transformed back into a grid domain for the chemistry calculations. Details of this solution approach, which is quite common in global models, can be found in [Hau40], [Sil54], [KHYK61], [WP86] or [FW94]. 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. A typical time step increment is 15 simulated minutes. Figure 2 represents a visual sample from this application.

There are many ways in which more interactivity in this parallel application could significantly benefit end users. For example, a typical problem in model development is that there are dramatic differences in scale between some global phenomenon and the many physical processes that comprise it. Gross measures such as vertical windfields have small values on a global scale, yet on a smaller scale phenomenon such as thunderstorms cause large vertical air displacements and play important roles in vertical mixing in the atmosphere. Computing the entire globe on a scale where all such phenomena could be accurately represented is far too computationally expensive to consider. One way of approaching this problem is to use parameterizations inside models which bear an indirect relationship to the small-scale phenomenon and that attempt to match observed phenomenon on the global scale. Unfortunately, the construction of these parameterized models is an exploratory and error-prone process. Section 3.1.1 describes ways in which basic interactive monitoring and steering can aid in this model construction process.

Other more ambitious approaches to the same problem might involve allowing the user to interactively identify interesting subareas for simulation at a higher resolution in time and/or space. This differential focus approach would allow those regions to be modeled with better fidelity without invoking the huge computational cost of using a higher resolution uniformly over the entire model. In some cases these areas might be selected algorithmically, but in other cases what constitutes an interesting situation or area could depend upon a subjective judgment by a human observer.

One can certainly imagine writing a program with a user interface that allows this level of interaction, but our goal is to achieve this without turning scientists into graphical user interface (GUI) programmers. The next section presents some tools and techniques that we have developed to further this goal.

3 Requirements and Tools for Interactive Computing

This section examines the general techniques and tools required to support program tuning in general and specifically to support the user/program interactions presented above. We first examine examples of some displays that support the interaction goals discussed previously. Then we discuss the monitoring and steering systems required to create and support these displays.

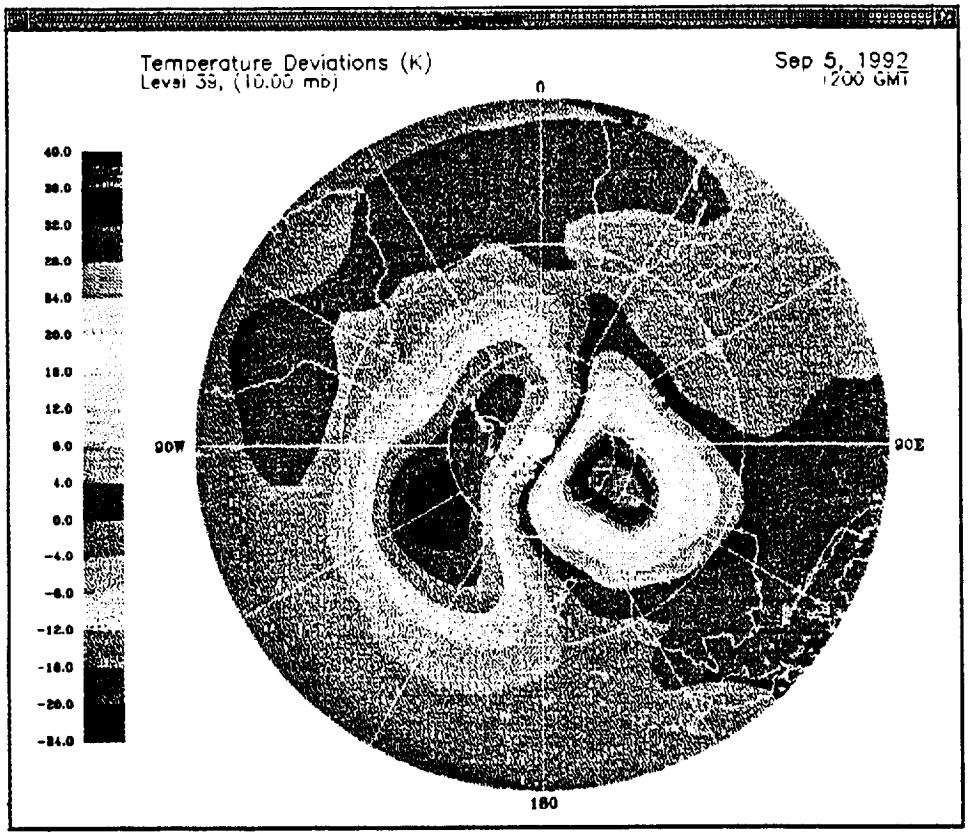


Figure 2: A sample plot of southern hemisphere temperature distribution as used by the global climate transport model.

3.1 Displays

Displays which are useful for understanding application behavior vary as widely as do applications and programming models. It is not possible, within the bounds of this paper, to survey all possible displays or even all useful approaches to display construction. Instead we present sample displays and interactions so that we can explain the monitoring and steering infrastructure required to support them. This section presents two types of displays targeted to different levels of abstraction in a arallel program. The first example is an application-specific display of the type required to achieve some of the process-oriented gains in application development discussed above. The second example is a programming-model specific display useful for program debugging and tuning.

3.1.1 Application Specific Displays

The previous section has indicated that user interactive steering has the potential to improve an application's performance and functionality. However, many uses of steering are application specific and so are graphical displays that are used to present the run-time program and performance information to the end user and to accept the user's steering commands. By examining a sample display used for steering the atmospheric modeling code, we can explore how these displays are used to understand and control the application and how they are interfaced with other parts of the monitoring and steering system.

The graphical display discussed in this section is specifically built for interactive steering of an atmospheric modeling code that simulates the distribution of atmospheric species such as Carbon 14 (C^{14}) and CFC. Figure 3 shows a screen display of the distribution of C^{14} at a latitude of 70° N. The display has two logical

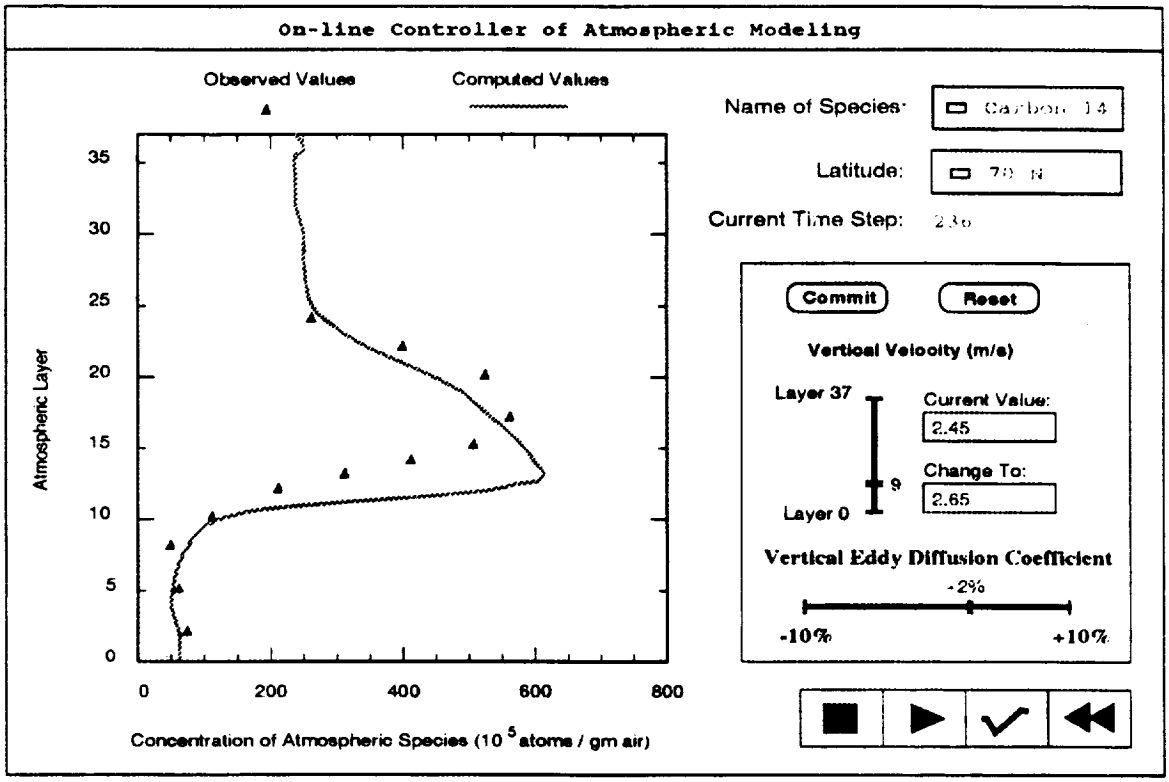


Figure 3: An application specific display for on-line control of the atmospheric modeling code.

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 is represented by a plotted curve from atmospheric layer 0 to 37, and it is updated for every model time step. The observed C^{14} concentration at a number of atmospheric layers is represented by discrete triangular points, and is used to judge whether the current computation is "correct" or "wrong." 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 curve shown in Figure 3 demonstrates that the computed concentration of C^{14} is consistently higher than the observed values from layer 10 to 15, but it is lower from layer 16 to about 23. The simulation may be adjusted to remove this discrepancy; the end user can alter the vertical wind velocity at these atmospheric layers. After typing in new vertical wind velocity values, the user needs to click the commit button on the display to send the steering command to the application. The program will use these new parameters for computations from the next model time step. The user can also stop the application's execution (by clicking the □ or stop button), change parameters, and restart the execution (by clicking the ▷ or play button). Before restart, the user can rollback the computation to a previously checkpointed time step (by clicking the ◁◁ or rewind button). At any point the user can checkpoint the application execution (by pressing the checkpoint button). The user can also use the application's default checkpointing policy which automatically saves execution history after a predefined number of time steps.

The above application specific display has a two-way communication link with the application code. In one direction, it receives computed and observed concentration values of atmospheric species from the application, and displays these values to the user. In the other direction, the display accepts steering commands and sends them to the application. A clean interface between the display and the application code is needed. Our Falcon system provides a flexible mechanism of dynamically connecting and disconnecting displays to the application. This mechanism will be addressed in Sections 3.2 and 3.3.

3.1.2 Programming Model Specific Displays

Both of the application programs described in Section 2 have been implemented on shared-memory multiprocessors using a threads-based programming model. In this model, independent threads of computation are created on the various processors, and they control access to shared data by using mutex locks and conditions[CD88]. The amount of time a thread spends waiting to be granted a lock or for a particular condition to occur directly impacts the amount of useful work it can perform in a given time. Therefore, understanding the interactions of threads over time is one of the most important aspects in understanding the behavior of these programs.

This section examines one display that we have found useful in diagnosing performance problems in threads-based programs. The threads lifetime view depicted in Figure 4 shows thread behavior over time. In particular, it represents each thread as a horizontal bar which assumes different colors and patterns when the thread is in different states, such as running, waiting for a mutex, or waiting for a condition. A vertical line is drawn from the parent thread to the child thread at the time of thread fork event. When a thread joins another thread after it exits or when a detached thread calls thread_exit, the narrow bar representing the thread terminates. In the case of thread_join, another vertical line is drawn from the caller thread to the thread to which it joins. The space after a joined thread can be reused by threads forked later. The display contains buttons with which one can move around and zoom in on different threads and regions of time. In addition to this lifetime display, there is another simultaneous display that relates thread colors to their names. We have excluded this name mapping view for space considerations.

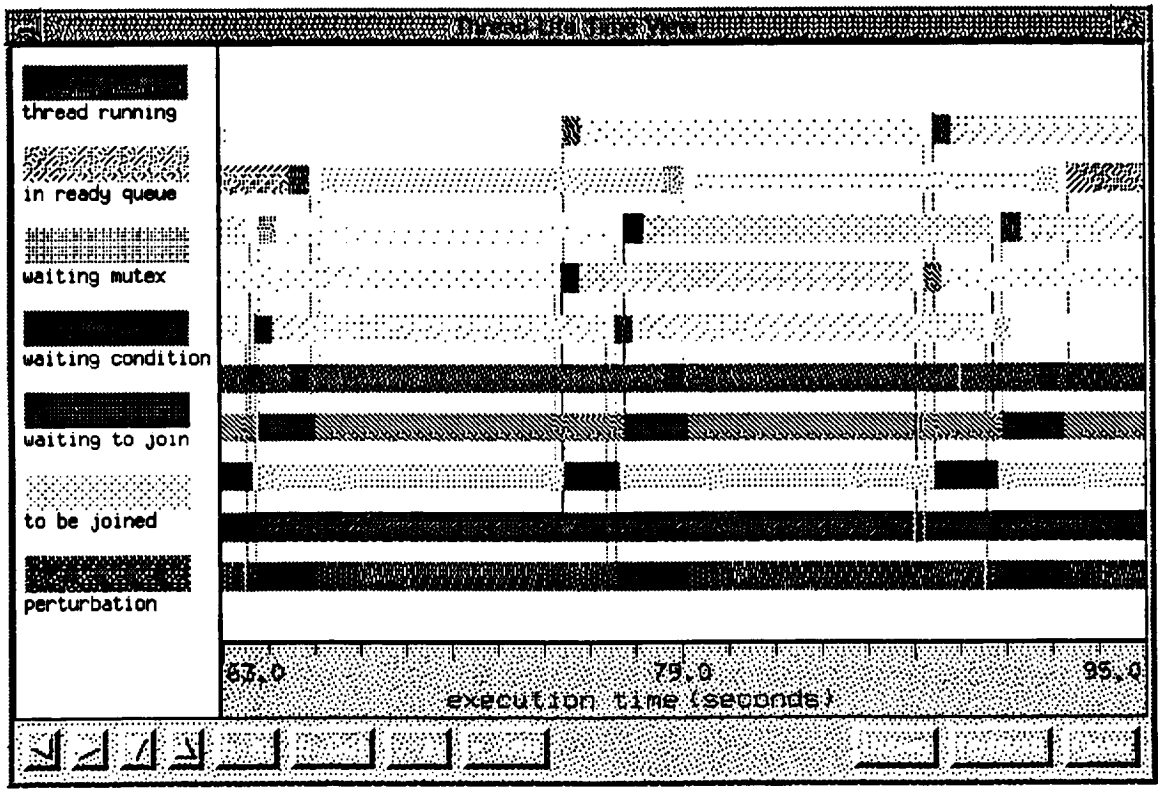


Figure 4: A thread life-time display derived from traces of MD program behavior.

The particular set of threads shown in Figure 4 represents a snapshot of the MD application's execution with the molecules partitioned into five domains. The bottom five threads are threads responsible for calculating the inter-molecular forces for each domain. These threads live for the duration of the calculation. The top five positions are held by threads that calculate intra-molecular forces. These threads are forked

by the inter-molecular thread only for a single timestep in the simulation. At the end of their calculations for that timestep, they perform a join operation and exit when both they and the thread that forked them complete the calculations for that timestep. When the next iteration begins, a different thread (with a different color) is forked and the process repeats.

Figure 4 is interesting for MD because it makes clear some aspects of thread synchronization that are difficult to determine without such a display. Each domain must acquire updated particle location information from its neighboring domains before it can proceed with the next iteration. This waiting time appears as the grey "waiting for condition" state in the display. Rather than requiring *all* threads to finish before *any* thread begins the next iteration, MD domains only synchronize with their immediate neighboring domains. This allows individual domains to begin the next iteration even before other threads have completed the current iteration. This flexibility helps MD compensate for the effects of minor variances in load balance between domains. In the figure one can see that blocks of solid compute time, which occur when a domain starts a new iteration, occur first in the second thread from the bottom and later in threads of more distant domains.

To produce displays of this type, the important constructs in the programming model must be instrumented. In this case, we have instrumented the Cthreads parallel programming library[Muk91] so that every operation that can affect the state of a thread produces a record in an event stream. In order to produce a reasonable display, these events must contain accurate timestamps and they should not excessively disturb normal execution of the program. The next section will discuss the system requirements and tools in Falcon that support the sample displays presented above.

3.2 On-line Monitoring

The first step to interactivity is gaining easy access to the applications' run-time information. This information ranges from records of the utilization of processors to detailed execution and waiting times spent by each processor and from values of certain variables (e.g., "temperature" of a simulated molecular system, "concentration" of an atmospheric species) to complete current program states of the application. Therefore, the capture, collection, and analysis of on-line program and performance information should be an integral component of any system which supports interactivity. Instead of focusing on supporting on-line interactivity, however, past work in program monitoring has focused on helping programmers understand the performance of their parallel codes, minimizing or correcting program perturbation due to monitoring, reducing the amounts of monitoring or trace information captured for parallel or distributed program debugging [OSS93, HMC94], and the effective replay [LMC87] or long-term storage of monitoring information. In comparison, interactivity, in the form of on-line program steering, specifically requires its *on-line* monitoring system to be able to: (1) capture *application-specific* information, (2) impose *controlled overheads* on the execution of monitored applications, (3) deliver monitoring information with *low latency*, and (4) provide incremental analysis of monitoring information vital for on-line steering.

The monitoring system is required to handle application-specific data because much of program steering is inherently application-specific. With MD, for example, steering can be used to improve load balance based on the molecule partitions and boundaries of these partitions. The boundaries can be adjusted by the user during program execution to obtain a better load balance. In steering the atmospheric modeling code, parameters concerning certain atmospheric species can be dynamically changed to effect different results on these atmospheric species. In addition, application-specific monitoring permits non-computer science end users to view, analyze, and steer their programs in terms of their specific attributes (e.g., "time step size" or "current energy").

Controlled monitoring overheads are useful for several reasons. First, since one purpose of application steering is to improve program performance, excessive monitoring overheads can easily offset the performance gains obtained by steering. Second, steering decisions based on inaccurate information may produce unexpected results. In the case of MD steering based on the work load information of each processor, perturbed information can cause inaccurate, sometimes unnecessary, adjustments of partition boundaries. In the worst case, thrashing of boundaries can occur and application execution will actually be slowed.

Steering latency is the period of time between the occurrence of a program activity or state and the time when it is acted upon by a steering agent; monitoring latency is the period of time between the capture of an activity by the on-line monitor and the passage of that activity to the steering mechanism. Excessive

monitoring and steering latencies can cause steering decisions to be made based on obsolete program and performance information, which can result in unpredictable and often negative effects on an application's execution. In the atmospheric modeling code, if the visualized windfield and values of atmospheric species are presented to end users several time steps behind the actual application execution, users may adjust parameters based on "old" information.

Falcon – an integrated system for on-line monitoring and steering of large-scale parallel and distributed applications – is designed to incorporate the attributes necessary for effective on-line monitoring and steering. An overview of the Falcon monitoring system is presented next, followed by discussions of its mechanisms for code instrumentation, event collection, and on-line trace data analysis[1]. Falcons program steering system will be described in Section 3.3.

3.2.1 System Overview of Falcon

Falcon is a set of tools that collectively support on-line program monitoring and steering of parallel and distributed applications. There are three major conceptual components of the on-line monitoring component of Falcon: (1) a monitoring specification and instrumentation mechanism, which consists of a low-level *sensor specification language*, a high-level *view specification language*, and an instrumentation tool, (2) mechanisms for on-line information capture, collection, filtering, and analysis, and (3) a graphical user interface and some graphical displays for interfacing with the end user. These components are shown in Figure 5.

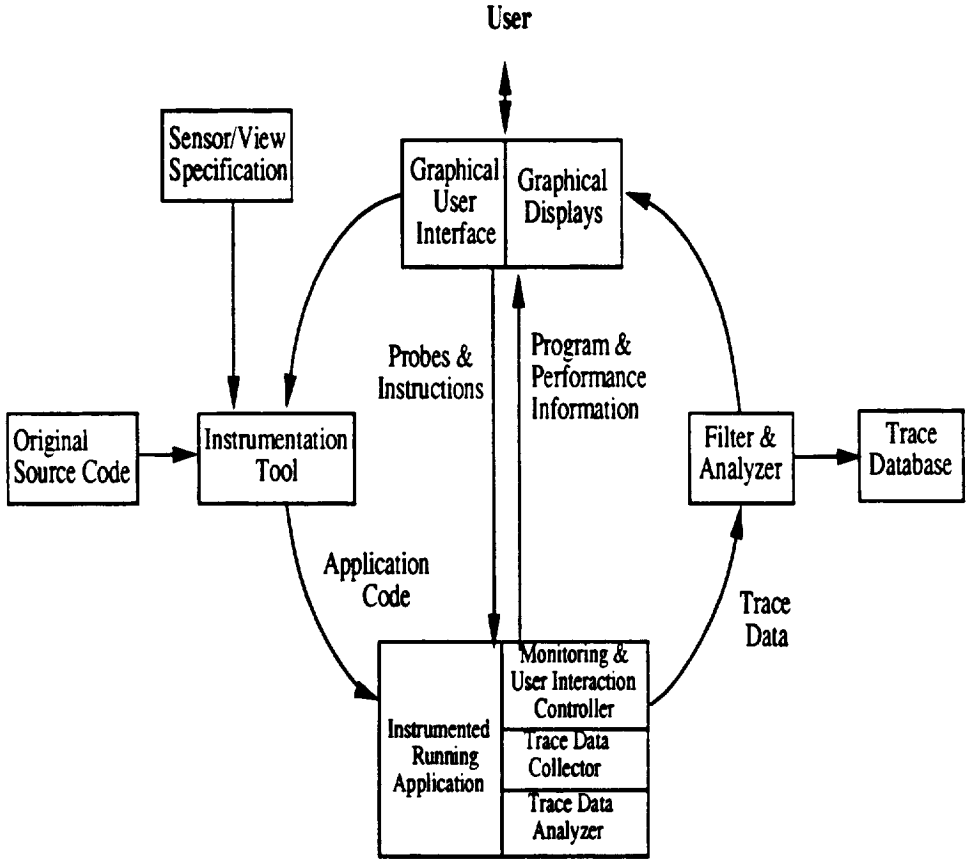


Figure 5: Conceptual components of Falcon.

The following steps are taken when using Falcon. First, application code is instrumented with sensors

[1] A more detailed description of the complete Falcon system and its performance can be found in [GEK+94]

and probes generated from sensor and view specifications. Such monitoring specifications allow users to express specific program attributes to be monitored and based on which steering may be performed. During program execution, program and performance information of interest to the user and to steering algorithms is captured by the inserted sensors and probes, and the information collected is partially analyzed. Falcon's runtime facilities consist of monitoring data output queues attaching the monitored user program to a variable number of additional components performing low-level processing of monitoring output. Partially processed monitoring information is then fed to the central monitor and graphical displays for further analysis and for display to end users. Trace information can also be stored in a trace data base for postmortem analysis.

The monitoring and user interaction 'controllers' in the Falcon runtime system activate and deactivate sensors, execute probes or collect information generated by sampling sensors, and also react to commands received from the monitor's user interface. For performance, these controllers are divided into several *local monitors* residing on the monitored program's machine so that they are able to rapidly interact with the running program. In contrast, the *central monitoring controller* is typically located on a front end workstation or on a processor providing user interface functionality.

3.2.2 Instrumentation and Monitoring Specification

Instrumentation of a target application and its run-time system is the first step toward application steering. Hardware monitoring and data collection require instrumentation of the hardware platform on which the target application is running. Software monitoring and data collection require instrumentation of the program's source code, the system libraries, the compiler, or any combination of the above. We do not rely on hardware monitoring due to its cost, inherent inflexibility and inability to provide high-level application-specific monitoring information.

Software instrumentation points are called *sensors* in Falcon. Falcon offers three types of sensors: sampling sensors, tracing sensors, and extended sensors. A *sampling sensor* is associated with a counter or an accumulator. When a sampling sensor is activated, the associated counter value is updated. A *tracing sensor* generates timestamped event records that may be used immediately for program steering or stored for postmortem analysis. In either case, trace records are stored in *trace queues* from which they are removed by local monitors. An *extended sensor* is similar to a tracing sensor except that it also performs simple data filtering or processing required for steering before producing output data. Sampling sensors inflict less overhead on the target application's execution than tracing and extended sensors. However, the more detailed information collected by tracing sensors may be required for diagnosis of certain performance problems in parallel codes. Furthermore, the combined use of all three sensor types enables users to balance low monitoring latency against accuracy requirements concerning the program information required for program steering.

In order to control monitoring overheads, sensors can be controlled dynamically *selectively* to monitor only the information currently being used by the end user or the steering algorithms. First, sensors may be turned off if events captured by those sensors are not currently used by the end user or the steering algorithm.[2] Second, sampling and tracing rates can be dynamically reduced or increased depending on monitoring load and tolerance of inaccuracies in monitored information. For example, a tracing sensor that monitors a frequently accessed mutex lock can reduce its tracing rate to every five mutex lock accesses, thereby improving monitoring perturbation at the cost of reducing trace accuracy. A selective monitoring example can be found in the MD code, where a large amount of execution time is spent in a three-level nested loop computing forces between particles. At each loop level, distances between closest points of particles and bounding boxes of molecules are calculated and compared with the cutoff radius to eliminate unnecessary computations at the next loop level where specific particles are considered. To evaluate the efficiency of this scheme, at each loop level we use a "cheap" sampling sensor to monitor the hit ratio of distance checks and a more "expensive" tracing sensor to monitor the correlations between the calculated distance and hit ratio at the next loop level. To reduce the perturbation, the "expensive" tracing sensor is not turned on until ineffective distance checks are detected.

Using Falcon's monitoring specification language [Sno87], programmers may define application-specific sensors for capturing both the program and performance behavior to be monitored and the program attributes

[2] Related work by Hollingsworth and Miller [HMC94] removes instrumentation points completely to reduce the overheads of these turned-off instrumentation points to zero.

based on which steering may be performed. The specification of a sample tracing sensor is shown below:

```
sensor work_load {
        attributes {
                int     domain_num;
                double  work_load;
        }
};
```

The sensor work_load is used to monitor the work load of each molecule domain partition in MD. It simply describes the structure of the application data to be contained in the trace record generated by this sensor. This declaration generates the following sensor subroutine.

```
int
user_sensor_work_load(int process_num, double work_load)
{
    if (sensor_switch_flag(SENSOR_NUMBER_WORK_LOAD) == ON) {
        sensor_type_work_load  data;
        data.type = SENSOR_NUMBER_WORK_LOAD;
        data.perturbation = 0;
        data.timestamp = cthread_timestamp();
        data.thread = cthread_self();
        data.process_num = process_num;
        data.work_load = work_load;

        while (write_buffer(get_buffer(cthread_self()), &data,
                    sizeof(sensor_type_work_load)) == FAILED) {
            data.perturbation = cthread_timestamp() - data.timestamp;
        }
    }
}
```

Note that there are four *implicit fields* for any event record that describe the event's sensor type, timestamp, thread id, and perturbation. The body of this subroutine generates entries for an event data structure, then writes that structure into a trace buffer. A local monitor later retrieves this structure from the buffer. Each sensor's code body is also surrounded by an `if` statement, so that it can be turned on or off during program execution.

3.2.3 Event Collection and On-line Trace Analysis

In many monitoring systems, all monitoring activities, including trace data capture, collection, and analysis, are performed by code inline with the thread of user computation. One problem with this approach is that the target application's execution is interrupted whenever a monitoring event is generated and processed. The lengths of such interruptions are arbitrary and unpredictable if complicated on-line trace analysis is used. This may be acceptable with off-line monitoring mechanisms in which monitoring events are written into files for postmortem consumption. For on-line monitors, however, this approach can produce unacceptable perturbation. Instead of performing monitoring activities in the user's code, Falcon uses concurrent monitoring, where most monitoring activities are on processors not running application code.

As depicted in Figure 5, local monitors perform trace data collection and processing, concurrently and asynchronously with the target application's execution. Local monitors and steering controllers typically execute on the target program's machine, but they may run concurrently on different processors, using a buffer-based mechanism for communication between the application and the monitoring mechanism. Therefore, the only direct program perturbation caused by Falcon is the execution of embedded sensors and the insertion of trace records into monitoring buffers. Such perturbation is generally predictable, and its effects on the correctness of timing information can be eliminated using straightforward techniques for perturbation analysis [MRW92].

In order to control monitoring overheads and latency, Falcon's runtime system may itself be configured or steered in several ways, including changing the number of local monitors and communication buffers to configure the system for parallel programs and machines of different sizes. Such changes permit the selection of suitable monitoring performance for specific monitoring and steering tasks, and they may be used to adapt the monitoring system to dynamic changes in workload imposed by the target application. For example, when heavy monitoring is detected by a simple monitor-monitor mechanism, new local monitoring threads may be forked. Similarly, when bursty monitoring traffic is expected with moderate requirements on monitoring latency, then buffer sizes may be increased to accommodate the expected heavy monitoring load. Such parallelization and configuration of monitoring activities is achieved by partitioning user threads into groups, each of which is assigned to a specific local monitor. When a new application thread is forked, it can be added to the local monitor with the least amount of work.

The amount of trace data generated by inserted sensors and collected by the run-time monitoring mechanism is usually too large and the information too low-level to be directly useful to any human user. Trace data filtering and analysis must be performed to generate information that is interesting to end users. Related research concerning on-line trace analysis includes Snodgrass' work on *update networks* [Sno82] and our own past work on real-time monitoring [OSS93]. In [Sno87], information to be monitored is modeled by temporal relations in a hierarchical structure with primitive relations at the bottom of the structure and composed relations at the top. The resulting hierarchy of relations is transformed into an update network - a directed acyclic graph, in which the tuples of the primitive relations enter the nodes at the bottom and the tuples of the composed relations flow out of the nodes at the top.

Falcon offers a flexible on-line trace analysis mechanism similar to update networks. However in Falcon's approach, trace data is processed in different physical components of the monitoring system. At the lowest level, simple trace data filtering and analysis can be performed by the extended sensors. For example, in the atmospheric modeling application, values of windfields may be filtered or eliminated since their complete visualization is expensive. At the local monitor level, trace data is further analyzed to produce higher level information. As in the steering of the atmospheric modeling code in Section 3.1.1, discrepancies between the computed values of an atmospheric species and the observed values can be detected by simple algorithms. Finally, trace data analysis can be performed by separate processes linked with the central monitor. An example of such an analysis process is presented next, and problems to be dealt with when performing on-line trace analysis will be discussed in more detail.

3.2.4 On-line Event Ordering

Displays like the thread life-time view of Figure 4 can provide users with insights into program progress and correctness. However, such displays generally have strict requirements in terms of the accuracy of the timestamps that they expect and the order in which events are presented to them. Misorderings can both confuse users and cause failures of the animation itself. For example, natural causal ordering would require that a thread_fork event precede any event executed by the newly created thread. A display that shows a child running before it has been forked by its parent does not make any sense. Furthermore, suppose that the first event for this child thread is a condition_wait event. In the thread life-time view of Figure 4, this event is represented by a change in the color and fill pattern of that thread's horizontal bar. However, if the thread_fork event has not been received by the display system, the horizontal bar does not yet exist. When the display system attempts to perform a color-change action on this non-existent object, it may crash.

The out-of-order events that cause problems for the display system cannot have occurred in the program's execution. Instead, misorderings existing in the event stream are due to the buffering and processing methods employed in the monitoring system. The diagnosis and correction of out-of-order events is a common problem in parallel and distributed monitoring systems. Existing systems (e.g., ParaGraph[HE91] and SIEVE[SG92]) rely on a sort by timestamp value to impose a total order on all events stored in event files. The on-line nature of the Falcon monitoring system precludes any use of such a solution, and sorting by timestamp order does not entirely eliminate the problem of out-of-order events[BS93]. In addition, coarse clock granularities and poor clock synchronization among different processors may lead to event timestamps that do not accurately reflect the actual order of program execution.

Falcon offers a general mechanism for approaching this problem. In particular, all events are processed by an *ordering filter* before they are sent to the display system. This filtering algorithm follows a "minimum-

intervention policy." Specifically, it examines each event in the stream arriving from the monitoring system, checks the applicable ordering rules for this event type, and if no rules are violated, forwards the event to the display system. If a rule violation is indicated, the event is held back until the rules are satisfied. As an example, consider the ordering rule that the lifetime view of Figure 4 uses to enforce orderings for a mutex lock event. Actually, a mutex lock is recorded as two separate events: a mutex_begin_lock event indicating that a thread has attempted to obtain the lock and a mutex_end_lock event indicating that a thread has succeeded in obtaining the lock. The following ordering rule is observed by the filter for a mutex_end_lock:

```
mutex_end_lock t m n <- ((thread_init t || thread_fork pt t) &&
                         (mutex_init m || mutex_alloc m) &&
                         (mutex_unlock m n-1) )
```

The parameters associated with the event mutex_end_lock are t, the id of the thread attempting to obtain the lock, m, the id of the mutex variable, and n, the sequence number indicating the number of successful lock attempts on this particular mutex variable. This rule may then be translated as: "a mutex_end_lock event with parameters t, m, and n, may be passed on to the display system if thread t has been initialized or forked by a parent thread, mutex variable m has been initialized or allocated, and the mutex_unlock event for variable m, sequence number n - 1 has already been passed on to the display system." Armed with similar rules for other events, the ordering filter can enforce sufficient ordering to ensure proper functioning of the thread lifetime display. Note that at present, this system only addresses the issue of event ordering. This is adequate to compensate for minor clock variations, but perhaps insufficient when the clocks on different processors vary widely. However this system may provide the basis for more general approach to the timestamp problem as we extend Falcon to more distributed systems.

This section has examined the basic components of the on-line monitoring system of Falcon. The next section presents our approach to the other component of interactivity, on-line steering.

3.3 Interactive Steering

As high performance computing applications move away from the batch-oriented style of processing, making these applications interactive is a daunting task. The challenge exists not only in building new applications with interactivity, but also in reengineering existing applications to become interactive ones. A few programmers turn directly to integrated graphical user interfaces to build interactivity into their applications, but this approach is fraught with difficulties. First, most developers of the high performance computing applications are non-computer scientists, who may not have the background or the inclination to become GUI programmers. Second, most high performance computing systems are not known for high performance graphics support. Increasingly high performance front-end workstations tend to offer better graphics and visualization support, both in hardware and software, and are therefore a better place for running graphics-intensive code. However, the construction of such distributed computation and visualization systems is far from easy.

The interactive steering discussed in this paper offers an alternative way of providing interactivity to the high-performance applications. This approach separates the interactive activities from the computation-intensive part of the application and provides a dynamic link between these components. The responsibilities of such a steering component are to receive the application's run-time information from its coupled on-line monitoring system, display the information to the end user or submit it to a steering agent, accept steering commands, and enact changes that affect the application's execution. The application code is not directly exposed to the interaction with the user or other steering agents, but it needs to be instrumented with sensors which capture run-time information and provide entries for steering commands which may change the program's execution behavior. The basic requirement for steering is that the application code should behave correctly under any valid steering command. Other requirements can be derived by examining its use.

3.3.1 What is interactive steering?

Interactive steering can be defined as the interactive control and tuning of an application and its resources to improve application functionality and performance. This control and tuning is interactive in that an

external entity interacts with the application to accomplish it. That outside entity may be a user sitting at a workstation, or it may be another program responding to application events and driven by a previously-encoded steering algorithm.

We call steering *human-interactive* if a human watching a display is the primary initiator of a steering action. If instead the initiator is an outside program we call the steering *algorithmic*. Algorithmic steering may not be commonly associated with interactive programs, but it is a natural extension of the facilities and requirements presented earlier in this paper. For example, to expect a human watch an application and adjust it to compensate for load imbalances may be reasonable on an occasional basis, but no one is likely to babysit a 36-hour simulation that requires adjustments every five minutes. In this situation the solution is to feed the load information to an algorithm which is capable of balancing the load without human involvement. Using steering for this instead of embedding the load balancing algorithm into the application is still beneficial because it allows the algorithm to be expressed separately, where it is more easily understood, replaced and reused in other applications.

The different goals and types of steering exert different requirements on the steering system. For steering for performance, low overhead costs in monitoring and steering support are critical, simply because excessive overheads can easily offset performance gain obtained by on-line steering. Low steering latency may also be a critical requirement, particularly for algorithmic steering. Program events related to steering must be captured and processed, and the corresponding steering decision must be made while the decision is still relevant to the situation. Consider the on-line configuration of mutex locks presented in [MS93], where on-line algorithms change lock behavior from spin to blocking locks. Lock type is determined at runtime based on the time a lock call must wait before it obtains the lock. When the waiting time is above a certain threshold, the lock is a blocking lock. When the waiting time is relatively short, the lock is a spin lock. Since the reaction times required are on the order of a few tens of processor cycles, this application presents a formidable challenge for a steering system.

In the case of human-interactive steering, the demands on the steering system are not so extreme, as human response times will typically dwarf the latency times imposed by the system. However, if human interaction is to develop basic insight or to experiment with alternative solution methods and experimental parameters, more cooperation from the application may be necessarily. To accomplish the parameter tuning described in Section 3.1.1 for example, it is necessary to synchronize the parameter modifications with the phases of the application to ensure that steering does not invalidate the computations. In some cases the design of the application makes this easy. The load balancing of the MD application described in [EGSM94] was facilitated because mechanisms were in place to handle molecules moving from domain to domain. These worked without modification when the domain boundaries themselves moved. In other cases it is clear that desired manipulations cannot be carried out without the direct cooperation of the application. A good example of this is the checkpointing and rollback facility discussed in Section 3.1.1. It is unlikely that such functionality could be provided without the knowledge of the application. A continuing challenge in steering is to define the application interface to the steering system.

3.3.2 Falcon's Steering System

Falcon's on-line steering component is a natural extension of its monitoring facilities. Figure 6 depicts some internal features of steering as well as its relationship with other components of Falcon. Similar to local and central monitors, a *steering server* on the target machine performs steering, and a *steering client* provides the user interface and control facilities remotely. The steering server is typically created as a separate execution thread to which local monitors forward only those monitoring events that are of interest to steering activities. Such events tend to represent a small proportion of the total number of monitoring events, in part because simple event analysis and filtering is done by local monitors rather than by the steering server. Steering decisions are then made based on specific attributes of those events by human users or steering algorithms. Therefore, the primary task of each steering server is to read incoming monitoring events and to take the appropriate action in response. These responses are based on previously encoded decision routines and actions, which are encoded in an steering event/action repository in the server. This repository contains entries for each type of steering event, specifying the appropriate action to take in response. The responses represented here may perform some actual steering action on the application, note the occurrence of some monitoring event for future reference, or simply forward the event to the client for

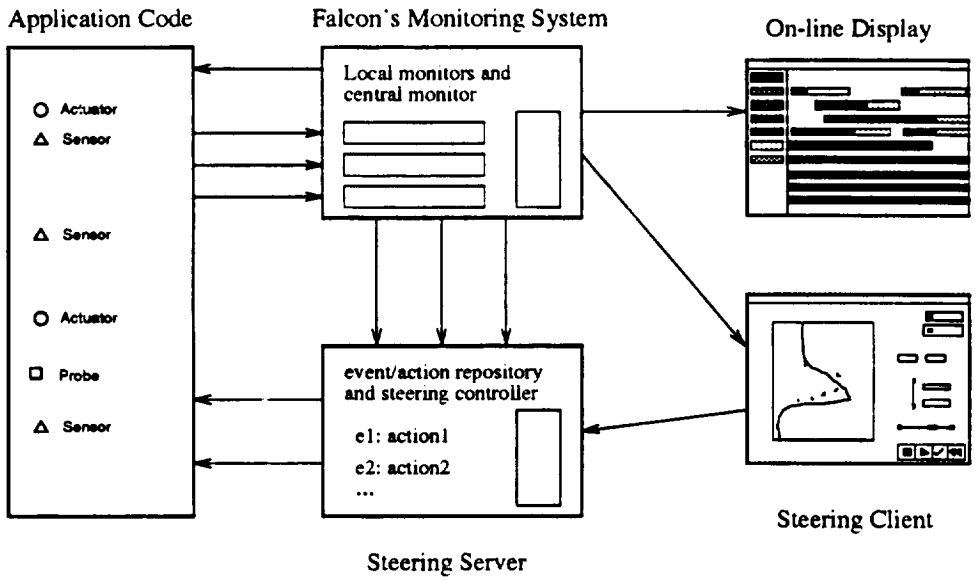


Figure 6: Overall structure of the steering system.

display or further processing. The secondary task of each steering server is to interact with the remote steering client. The steering client is used to enable/disable particular steering actions, display and update the contents of the steering event repository, and input steering commands directly from end users to the server.

Falcon's steering library introduces several abstractions. The first of these is *program attributes*. Program attributes are defined by application developers and they represent values or characteristics in the application that can be modified by the steering system. They are defined in an object-based fashion, where developers may associate with each specific program abstraction one or multiple attributes and then export methods for operating on these attributes. This type of association is called a *steerable object* and it must be "registered" with the steering system. The steering repository in the steering server maintains a list of all registered steerable objects and their associated program attributes. In our initial implementation, we assume that all attributes correspond to specific variables in the application program.

Steering actions are composite operations to be performed by the steering system in response to monitoring events in the program. Steering actions may examine and modify program attributes, perform computation, and even initiate other actions. Falcon defines two mechanisms for modifying program attributes, *steering probes* and *actuators*. A steering probe is the simplest form of steering action. It is used in actions to query or update a specific program attribute asynchronously to the program's execution. However, if a program attribute can only be updated synchronously, it must be associated with an actuator.

An actuator is a portion of code that the developer inserts into his code at locations in which it is "safe" for the steering system to take some action concerning the program attribute. Most of the time when the application executes the actuator code there are no pending actions and the actuator immediately returns control to the application. However, if the program attribute is to be synchronously modified by the steering system, the actuator becomes the instrument of that action. In particular, to update the program attribute synchronously, the steering system *asynchronously* sets the actuator so that the next time it is executed by the application code, it invokes a particular action *in the context of the application's thread of execution*. In this way, the responsibility for managing the synchronization of the steering system with the application rests with the application programmer and depends solely upon the placement of the actuators in the code. In simple situations, the action programmed into the actuator may just write a new value into the program attribute. For example, in the implementation of the steering of the atmospheric model as described in Section 3.1.1, the program variables corresponding to "Vertical Velocity" and "Vertical Eddy

Diffusion Coefficient" would be identified by the application programmer as program attributes and be registered with the steering system. The programmer would place actuators at points in his code, perhaps between iterations, where those values could be changed without invalidating the calculations. When a human triggered a change at the user interface, an actuator action would be "programmed" or "armed" with an action which would write the changed value into the target program variable at the next opportunity. However, actuator actions are capable of encoding much more complex operations than this. For example, they should be capable of the operations necessary to ensure that modifications of program state do not violate program correctness criteria as in [BS91].

The discussion above presents a brief overview of the abstractions in the Falcon steering library and the manner in which they interact with the remainder of the Falcon system. The implementation and integration of the steering library and other steering facilities is not yet complete, though proof-of-concept demonstrations as in [EGSM94] have been quite successful. However, we believe that the steering system, together with Falcon's monitoring facilities represent a powerful and flexible basis upon which to build interactive computing and through which users can exploit the opportunities presented in this paper.

4 Conclusions and Future Research

We have discussed the utility and potential of interactive parallel programming in the context of two large-scale parallel application programs. We have also explained how an on-line program steering and monitoring system can assist in realizing this potential. At present, ambitious and determined applications programmers can create their own interactivity by building user interfaces for their applications. These are valid interactive programs, but they are point solutions. Scientists are interested in computing to the extent that it helps them do science. Accordingly, the goal of our work on Falcon and monitoring and steering in general is to make this functionality more easily available to non-expert users.

The MD and atmospheric modeling codes as well as the Falcon system are implemented on a 64-node KSR shared memory supercomputer. Falcon is also available on several other shared memory platforms, including SGI and SUN Sparc parallel workstations. A version of Falcon currently being completed also works with PVM across networked execution platforms. Similar portability is attained for the graphical displays used with Falcon. Notably, the Polka animation library can be executed on any Unix platform on which Motif is available [SK93]. Falcon's low-level monitoring mechanisms have been available via the Internet since early Summer 1994. A version of Falcon offering on-line user interfaces for monitoring and monitor control will be released in 1995.

Current extensions of Falcon not only address additional platforms (e.g., an IBM SP machine now available at Georgia Tech and the monitoring of PVM programs running Cthreads, C, or Fortran programs), but also address several essential additions to its functionality. Currently users can insert into their code simple tracing or sampling sensors, where sensor outputs are forwarded to and then analyzed by the local and central monitors. We are now generalizing the notion of sensors to permit programmers to specify higher level 'views' of monitoring data like those described in [KS91, OSS93, Sno88]. Such views will be implemented with library support resident in both local and central monitors. We are also developing notions of composite and extended sensors that can perform moderate amounts of data filtering and combining before tracing or sampling information is actually forwarded to local and central monitors. Such filtering is particularly important in networked environments, where strong constraints exist on the available bandwidths and latencies connecting application programs to local and central monitors.

Our future work will address how such customized mechanisms may be used in conjunction with the remainder of the Falcon system. In addition, work in progress is addressing the monitoring of object-oriented, parallel programs, including the provision of default monitoring views and performance displays[MSSG95].

An important component of our future research is the use of Falcon with very large-scale parallel programs, either using thousands of execution threads or exhibiting high rates of monitoring traffic. For these applications it will be imperative that monitoring mechanisms are dynamically controllable and configurable. It must be possible for users to focus their monitoring on specific program components, to alter such monitoring dynamically, and to process monitoring data with dynamically enabled filtering or analysis algorithms. Moreover, such changes must be performed so that monitoring overheads are experienced primarily by the program components being inspected. Dynamic control of monitoring is also important for the efficient

on-line steering of parallel programs of even moderate size. Specifically, program steering requires that monitoring overheads are controlled continuously so that end users or algorithms can perform steering actions in a timely fashion.

Longer term research with Falcon will address the integration of higher level support for program steering, including graphical steering interfaces, and the embedding of Falcon's functionality into a programming environment supporting the process of developing, tuning, and steering threads-based parallel programs, called LOOM. In addition, Falcon will be a basis for the development of distributed laboratories in which scientists can inspect, control, and interact on-line with virtual or physical instruments (typically represented by programs) spread across physically distributed machines. The specific example being constructed by our group is a laboratory for atmospheric modeling research, where multiple models use input data received from satellites, share and correlate their outputs, and generate inputs to on-line visualizations. Moreover, model outputs (e.g., data visualizations), on-line performance information, and model execution control may be performed by multiple scientists collaborating across physically distributed machines.

References

[Bec94] Thomas Becker. Application-transparent fault tolerance in distributed systems. In *Proc. of the Second International Workshop in Configurable Distributed Systems*. IEEE Computer Society Press, May 1994.

[BS91] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[BS93] Adam Beguelin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, Pittsburgh, PA, June 1993.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical report, Computer Science, Carnegie-Mellon University, CMU-CS-88-154, June 1988.

[EGSM94] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3)*, pages 26–34, San Francisco, CA, August 1994.

[FW94] I.T. Foster and P.H. Worley. Parallel algorithms for the spectral transform method. Technical Report ORNL/TM-12507, Oak Ridge National Laboratory, April 1994.

[GEK+94] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. Technical Report GIT-CC-94-21, Georgia Institute of Technology, Atlanta, GA 30332-0280, April 1994.

[GS89] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106–125, July 1989.

[GS93] Ahmed Gheith and Karsten Schwan. Chaos-arc – kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[GVS94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29(9):140–148, September 1994.

[Hau40] B. Haurwitz. The motion of atmospheric disturbances on the spherical earth. *Journal of Mar. Res.*, 3:254–267, 1940.

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, pages 841–850, Knoxville, TN, May 1994.

[KHYK61] S. Kubota, M. Hirose, Y.Kichuchi, and Y. Kurihara. Barotropic forecasting with the use of surface spherical harmonic representation. *Pap. Meteorol. Geophys.*, 12:199–215, 1961.

[KS91] Carol E. Kilpatrick and Karsten Schwan. ChaosMON – application-specific monitoring and display of performance information for parallel and distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 57–67, Santa Cruz, California, May 20-21 1991.

[KSS+94] T. Kindler, K. Schwan, D. Silva, M. Trauner, and F. Alyea. A parallel spectral model for atmospheric transport processes. Technical report, Georgia Institute of Technology, Atlanta, 30332 GA, 1994.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[MP89] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–201. SIGOPS, Assoc. Comput. Mach., December 1989.

[MRW92] 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.

[MS93] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable microkernel. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 45–60, September 1993.

[MSSG95] Bodhisattwa Mukherjee, Dilma Silva, Karsten Schwan, and Ahmed Gheith. Ktk: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*, 1995. To Appear.

[Muk91] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[MW91] Keith Marzullo and Mark Wood. Making real-time reactive systems reliable. *ACM Operating Systems Review*, 25(1):45–48, January 1991.

[OSS93] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762–778, July 1993.

[SG92] Sekhar R. Sarukkai and Dennis Gannon. Parallel program visualization using SIEVE.1. In *International Conference on Supercomputing*. ACM, July 1992.

[SGB87] Karsten Schwan, Prabha Gopinath, and Win Bo. CHAOS – kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904–916, July 1987.

[Sil54] I.S. Silberman. Planetary waves in the atmosphere. *J. Meteorol.*, 11:27–34, 1954.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[Sno82] Richard Snodgrass. *Monitoring Distributed Systems: A Relational Approach*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, December 1982.

[Sno87] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[SO93] 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.

[WP86] W.M. Washington and C.L. Parkinson. *An introduction to three-dimensional climate modeling.* Oxford University Press, 1986.

[XORL92] T. K. Xia, Jian Ouyang, M. W. Ribarsky, and Uzi Landman. Interfacial alkane films. *Physical Review Letters*, 69(13):1967–1970, 28 September 1992.

Technical Report GIT-CC-94-21

Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs[1]

Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan
John Stasko, and Jeffrey Vetter

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract – Falcon is a system for on-line monitoring and steering of large-scale parallel programs. The purpose of such interactive steering is to improve its performance or to affect its execution behavior. The Falcon system is composed of an application-specific on-line monitoring system, an interactive steering mechanism, and a graphical display system. In this paper, we present a framework of the Falcon system, its implementation, and evaluation of the system performance. A complex sample application – a molecular dynamics simulation program (MD) – is used to motivate the research as well as to evaluate the performance of the Falcon system.

1 Introduction

The high performance of current parallel supercomputers is permitting users to *interact* with their applications during program execution. Such interactive executions of large-scale parallel codes typically make use of multiple networked machines working in concert on behalf of a single user, as computational engines, display engines, input/output engines, etc. Our research explores the potential increases in performance and functionality gained by the *on-line* interaction of end users with their supercomputer applications. Specifically, we are investigating the *interactive steering* of parallel programs, which is defined as 'the on-line configuration of a program by algorithms or by human users, with the purpose of affecting the program's performance or execution behavior'. Interactive program steering does not involve simply the on-line or postmortem exploration of program trace or output data, as being investigated by researchers in program debugging[28, 18] or in computer graphics[6]. Instead, program steering targets the parallel code itself, and it can range from rapid changes made by on-line algorithms to the implementation of single program abstractions (e.g., a mutex lock [35]) to the user-directed improvement of or experimentation with high-level attributes of parallel codes (e.g., load balancing in a large-scale scientific code – see Section 2.2). In either case, program steering is based on the on-line capture of information about current program and configuration state [7, 31, 46, 40], and it assumes that human users and/or algorithms inspect, analyze, and manipulate such information when making and enacting steering decisions.

[1]This research was supported in part by NASA grant No. NAGW-3886 and with funding from Kendall Square Research Corporation.

Falcon is a system for the *on-line* monitoring and steering of threads-based parallel programs. This paper focusses on Falcon's contributions to program monitoring:

- *Application-specific monitoring* – in addition to providing default program information Falcon permits users to capture and analyze application-specific program information, ranging from information about single program variables to program states defined by complex expressions involving several program components distributed across different processors of a single underlying parallel machine. These capabilities are especially useful for non-Computer Science end users, who wish to view, analyze, and steer their programs in terms of program attributes with which they are familiar (e.g., 'time step size', 'current energy', etc.).

- *Scalable, dynamically controlled monitoring performance* – by using concurrency and multiple mechanisms for capturing and analyzing monitoring information, the performance of the monitoring system itself can be scaled to different application needs, ranging from high-bandwidth and low-latency event-based monitoring to lower bandwidth sampling of accumulated values. Moreover, the resulting tradeoffs between monitoring latency, throughput, overhead, and accuracy may be varied dynamically, so that monitoring performance may be controlled and adjusted to suit the needs of individual applications and target machines. In addition, simple mechanisms are provided so that users can evaluate program perturbation due to monitoring.

- *On-line analysis, steering, and graphical display* – monitoring information captured with Falcon may be attached to arbitrary user-provided analysis code and subsequent (if desired) steering algorithms and/or graphical views. Analyses may employ statistical methods, boolean operators like those described in [40], or simply reorder the events being received, as described in Section 5.4. Graphical views may be displayed with multiple media or systems, currently including X windows, Motif, and the SGI Explorer environment. In addition, Falcon offers default on-line graphical animations of the performance of threads-based parallel programs. For such Motif-based displays, the Polka system for program animation provides users with easy-to-use tools for creating application-specific 2D animations of arbitrary program attributes[49].

- *Extension to multiple heterogeneous computing platforms* – an extension of Falcon addresses both single parallel computing platforms running threads programs as well as distributed computational engines using PVM as a software basis.

Falcon runs on several hardware platforms, including the Kendall Square Research KSR-1 and KSR-2 supercomputers, the GP1000 BBN Butterfly multiprocessor, the Sequent multiprocessor, SGI workstations, and SUN SPARCstations. Falcon is now in routine use at Georgia Tech by non-Computer Science end users, and it is available for public release for the KSR-1, KSR-2, SGI, and SUN SPARCstation platforms.

In the remainder of this paper, Section 2 presents the motivation for this research by examining the monitoring and steering needs of a sample parallel application, a molecular dynamics simulation (MD) used by physicists for exploring the statistical mechanics of complex liquids. Section 3 presents details of the implementation and performance of the Falcon system itself. The overall performance of the Falcon system as well as its performance with the MD code is evaluated in Section 4. Section 5 examines the nature and requirements of Falcon's graphical displays. Related research is described in Section 6. The final section presents conclusions and future research.

2 Monitoring and Steering a Parallel Code

Program monitoring and steering derive their value from their utilization in understanding and improving program behavior, and in permitting users to experiment with program characteristics that are not easily understood. Clearly, it will be hard to prove that promises of enhanced utility or performance of parallel applications can be fulfilled more easily by steered programs than by non-steered ones. However, it is

inevitable that program steering will be performed in the future, in part because scientists now have available to them the computational and network power for interactive execution of interesting physical simulations and the means for interactive data visualization or even for virtual reality interfaces to their programs. To further motivate our work, this section briefly describes a particular parallel code, its potential for utilizing program steering, and the required support for on-line monitoring.

2.1 The MD Application

MD is an interactive molecular dynamics simulation developed at Georgia Tech in cooperation with a group of physicists exploring the statistical mechanics of complex liquids [51, 8]. In this paper, the physical MD system being simulated contains 4800 particles representing an alkane film and 2700 particles in a crystalline base on which the film is layered. 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. The dominant computational requirement is calculating the inter-molecular forces between particles, and other important computations include finding the bond forces within the hydrocarbon chains, determining system-wide characteristics such as atomic temperature, and performing on-line data analysis and visualization.

The implementation of the MD application attains parallelism by domain decomposition. That is, 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 processes 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).

2.2 Steering MD – Experimentation and Results

The on-line manipulation of parallel and distributed programs has been shown to result in performance improvement in many domains. Examples include the automatic configuration of small program fragments for maintaining real-time response in uniprocessor systems[32], the on-line adaptation of functional program components for realizing reliability versus performance tradeoffs in parallel and real-time applications [5, 14, 12], and the load balancing or program configuration for enhanced reliability in distributed systems[26, 43, 31].

The MD simulation offers opportunities for performance improvement through on-line interactions with end users and with algorithms, including:

- Decomposition geometries can be changed to respond to changes in physical systems. For example, a slab-based decomposition is useful for an initial system, but a pyramidal decomposition may be a better choice if a probe is lowered into the simulated physical system.
- The interactive modification of cutoff radius can improve solution speed by computing uninteresting time steps with some loss of fidelity, which typically requires the involvement of end users.
- The boundaries of spatial decompositions can be shifted for dynamic load balancing among multiple processes operating on different sub-domains, performed by end users or by a configuration algorithm.
- Global temperature calculations, which are expensive operations requiring a globally consistent state, can be replaced by less accurate local temperature control. On-line analysis can determine how often global computations must be performed based on the temperature stability of the system.

To demonstrate the utility of program steering, we next review some results of interactive MD steering applied to the problem of improving system load balance. In particular, we examine the behavior of the MD simulation when spatial domain of the physical system is decomposed vertically. In this situation, it is quite difficult to arrive at a suitable load balance when decomposing based on static information (such as counting the number of particles assigned to each process, etc.). This is because the complexity of MD computation depends not only on the number of particles assigned to each process, but also on particle distances (due to cutoff radius). Furthermore, the portions of the alkane film close to the substrate are denser than those on the top and therefore require more computation. In fact, fairly detailed modeling of the code's computation is required to determine a good vertical domain decomposition without experimentation, and there is no guarantee that an initial 'good' decomposition will not degrade over time due to particle movement or other changes in the physical system. As a result, it appears easier to simply monitor load balance over time and then steer the application code to adjust load balance (by adjusting domain boundaries) throughout the application's execution. In this paper, such steering is performed interactively by end users. Necessary algorithmic support will be developed in the future; it will enable users to interact with the application only when automated steering is not successful.

For interactive steering of MD, the Falcon system is used to monitor process loads on-line, the resulting trace information is analyzed, and workloads are displayed in bar graph form (see Figure 1). In addition, the MD code performs on-line visualization of particles and of current domain boundaries. The load balance view of Falcon and the MD system's data displays are depicted in Figures 1 and 2, respectively, for a sample simulation run with four domains on four processors. Associated with these displays is a textual user interface (also part of Falcon) that permits the user to change selected program attributes (in this case, shift individual domain boundaries) while the application is running.

The effects of dynamic steering when used to correct load imbalances can be quite dramatic, as shown in Figure 3. In this figure, several steering actions significantly improve program performance by successive adjustment of domain boundaries. These results are important for several reasons. First, they demonstrate that it is possible to improve program performance by use of on-line steering, rather than degrade performance due to steering and monitoring costs. Second, it should be apparent that user interactions with the code can be replaced or assisted by on-line steering algorithms, in effect giving users the ability to migrate their experiences and experimental knowledge into their application codes, without requiring extensive program changes. Third, and more broadly, these results indicate the potential of on-line steering for helping end users experiment with and understand the behavior of complex scientific codes.

2.3 The Requirements of Steering

While the steering of MD code by adjustment of domain boundaries as presented in Section 2.2 is straightforward, important to our work are the future opportunities presented by on-line steering and monitoring. Toward this end, our group is now experimenting with interactive parallel programs in several domains, including (1) the interactive simulation of complex systems used in conjunction with some physical system, for on-line diagnosis of problems or for trying out certain fault containment strategies[13] (e.g., telecommunication systems), and (2) the on-line experimentation with scientific or engineering applications. For example, we are developing an interactive global atmospheric modeling code, where scientists can easily experiment with alternative values for atmospheric quantities to adjust model runs in accordance with actual measured atmospheric data obtained from satellite observations (e.g., concentrations of certain pollutants or strengths and directions of wind fields). Similarly, we are using on-line steering to give users the ability to interact with their large-scale optimization codes, to direct program searches out of local minima, to detect and correct searches possibly leading to infeasible solutions, etc.

To realize on-line program steering, several assumptions must be made, some of which may be removed or ameliorated by our future work. First, program steering requires that application builders must write their code such that steering is possible. Second, users must provide the program and performance information necessary for making steering decisions. Third, it is imperative that such information can be obtained with the latency required by the desired rate of steering. Concerning the first requirement, in the MD code,

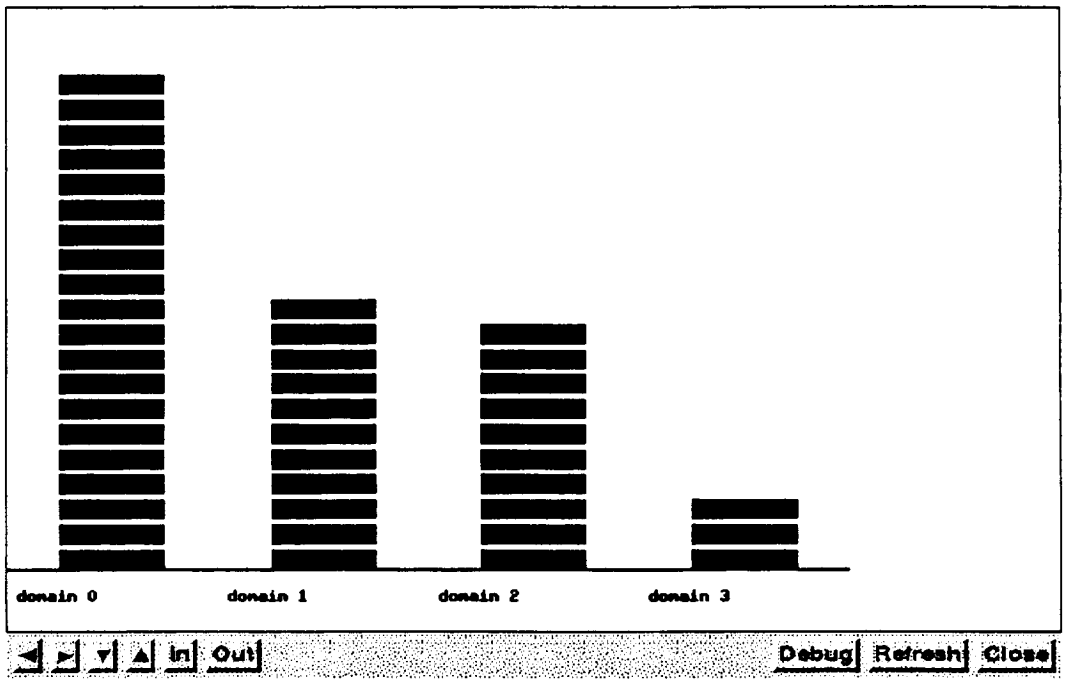
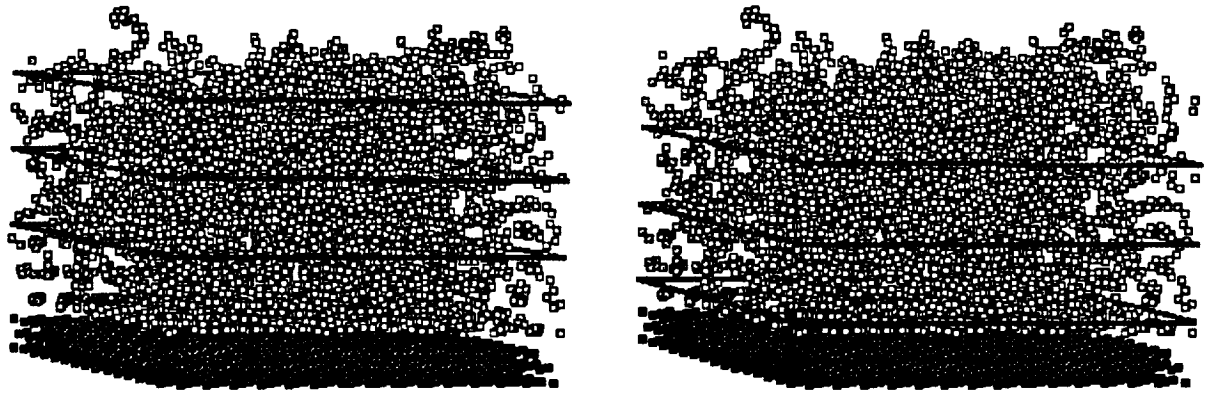


Figure 1: The load balance view of MD.



(a) Initial Decomposition

(b) Balanced Decomposition

Figure 2: Initial and balanced decompositions of the steered system. The horizontal frames mark the boundaries between processor domains. The dark particles are the fixed substrate while the lighter particles are the alkane chains.

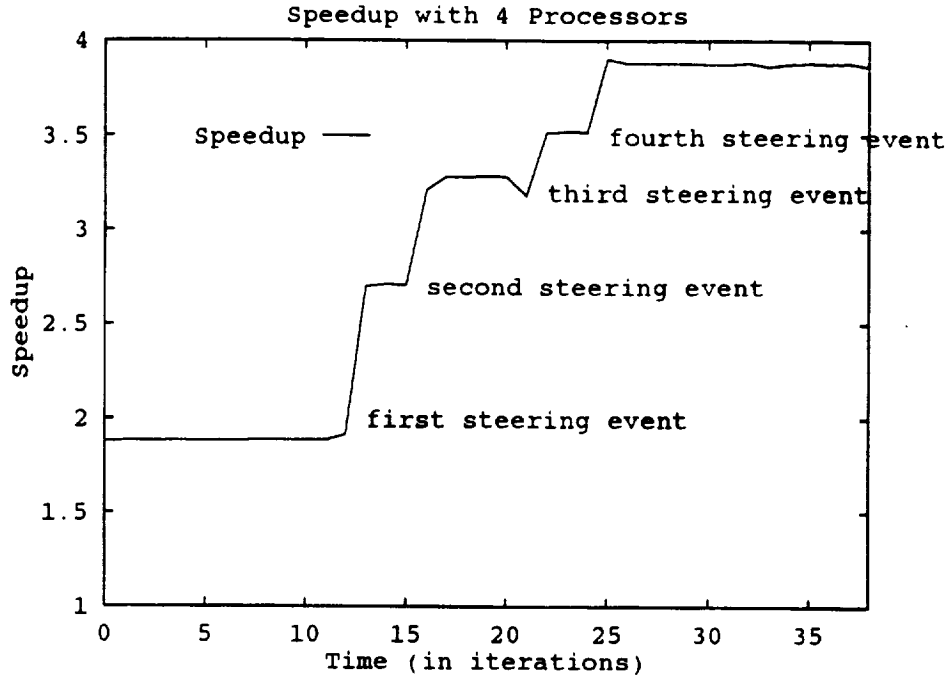


Figure 3: The effect of steering on performance over time with 4 processors.

domains are represented such that their boundaries are easily shifted to make steering for improved workload balance possible. In general, however, programs can be made steerable only by requiring end users to write them accordingly, by requiring substantial compiler support[46], or by requiring that the programming language offer stronger mechanisms of abstraction than those existing in parallel Fortran or in the Cthreads library used in our work (e.g., the object model [5, 11, 26, 14]). We are currently designing higher level language primitives for definition of steering actions and for inclusion of such actions with application code. At this time, however, Falcon relies on user-directed inclusion of *actuators* with the application code. These actuators are then stored into a runtime library which serves as a catalogue of names as well as an interface to the Falcon's on-line monitoring mechanism (see Section 3.4 for a description and brief evaluation of the steering library).

One of the primary concerns of this paper is the second requirement for on-line steering: the on-line provision, analysis, and display of information to users about current program behavior and performance, at rates suitable for program steering. Examples of such information used in graphical displays include the on-line data visualizations depicting molecular distributions in MD, the associated current values of domain boundaries (see Figure 2), and performance information about threads depicted in graphical views like the thread life-time view shown in Figure 13. Examples of such information used by on-line steering algorithms include lock contention values, which are used by on-line configuration algorithms to adjust individual mutex locks (see [35]) based on changes in a program's locking pattern.

A third requirement of on-line steering is that steering is effective only if it can be performed at a rate higher than the rate of program change. In the case of load balancing by dynamic domain shifting in MD, human users can detect load imbalances and shift domain boundaries faster than the rate of occurrence of significant particle movements (which require several minutes for moderate size physical simulations on our KSR-2 machine). However, when steering is used to dynamically adjust lock waiting strategies, changes in locking patterns must be detected and reacted upon in every few milliseconds[35]. As a result, any on-line monitoring support for program steering must permit users to realize suitable tradeoffs in the bandwidth versus latency of monitoring.

In response to the requirements listed above, Falcon gives users the ability to control instrumentation by permitting them to explicitly include program-specific *sensors* of different types into their application codes. A sensor definition language generates sensor implementations for target C and Fortran programs, and runtime-configurable monitoring libraries capture, analyze, and store/forward or display sensor outputs as desired by users. In addition, Falcon offers efficient system I/O (for data visualizations) and underlying communications across computer networks (for all remote mechanisms).

The description and evaluation of on-line monitoring in Falcon is the primary focus of this paper. However, to demonstrate the usability of Falcon, we also briefly describe and evaluate Falcon's interfaces to program animation and graphical data rendering tools.

3 The Design and Implementation of Falcon

3.1 Design Goals

Past work in program monitoring has focussed on helping programmers understand the correctness or performance of their parallel codes[33, 41], on minimizing or correcting for program perturbation due to monitoring[30], on reducing the amounts of monitoring or trace information captured for parallel or distributed program debugging[40], and on the effective replay[28] or long-term storage[47] of monitoring information.

Falcon has three important attributes. First, Falcon supports the *application-specific monitoring/steering, analysis, and display* of program information, so that users can capture, process, and understand and steer exactly the program attributes relevant to steering or to the specific performance problems being diagnosed or investigated. That steering requires application-specific program information is clearly demonstrated by the MD application steered in Section 2.2, where program variables capturing domain boundaries are adjusted based on monitoring output describing workload in terms of durations of molecular computations across different domains. Section 4 will also demonstrate that such specialization of monitoring to capture only specific program attributes can also significantly improve monitoring system performance and scalability compared to standard tools like GProf or compared to the default monitoring performed by Falcon.

Second, the primary focus of Falcon is to *reduce or at least control monitoring latency* throughout the execution of a parallel program, while maintaining acceptable monitoring workload imposed on the underlying parallel machine. Dynamic control of monitoring overhead is important because the effectiveness of program steering can depend on the delay between the time at which a program event happens and the time at which the event is noted and acted upon. In addition, excessive monitoring overheads not only offset performance gains achieved by steering, but also alter the order of occurances of program events. Finally, for scalability to large-scale parallel machines and programs, the Falcon system is configurable in its offered total performance and associated resource usage.

A third attribute of Falcon is its support for *scalable monitoring*, by varying the resources consumed by its runtime system in accordance with machine size and program needs. In Section 4, we show that Falcon can be used to monitor programs of any size running on our 64-node KSR multiprocessor, such that monitoring overheads and latencies can be adjusted in conjunction with program and machine size.

3.2 System Design

Falcon is constructed as a toolkit that collectively supports the on-line program monitoring and steering of parallel and distributed programs. There are four major conceptual components, as shown in Figure 4: (1) monitoring specification and instrumentation, which consists of a low-level *sensor specification language*, higher level *view specification constructs*, and an instrumentation tool, (2) runtime libraries for information capture, collection, filtering, and analysis, (3) mechanisms for program steering, and (4) a graphical user interface and several graphical displays of program behavior and performance information.

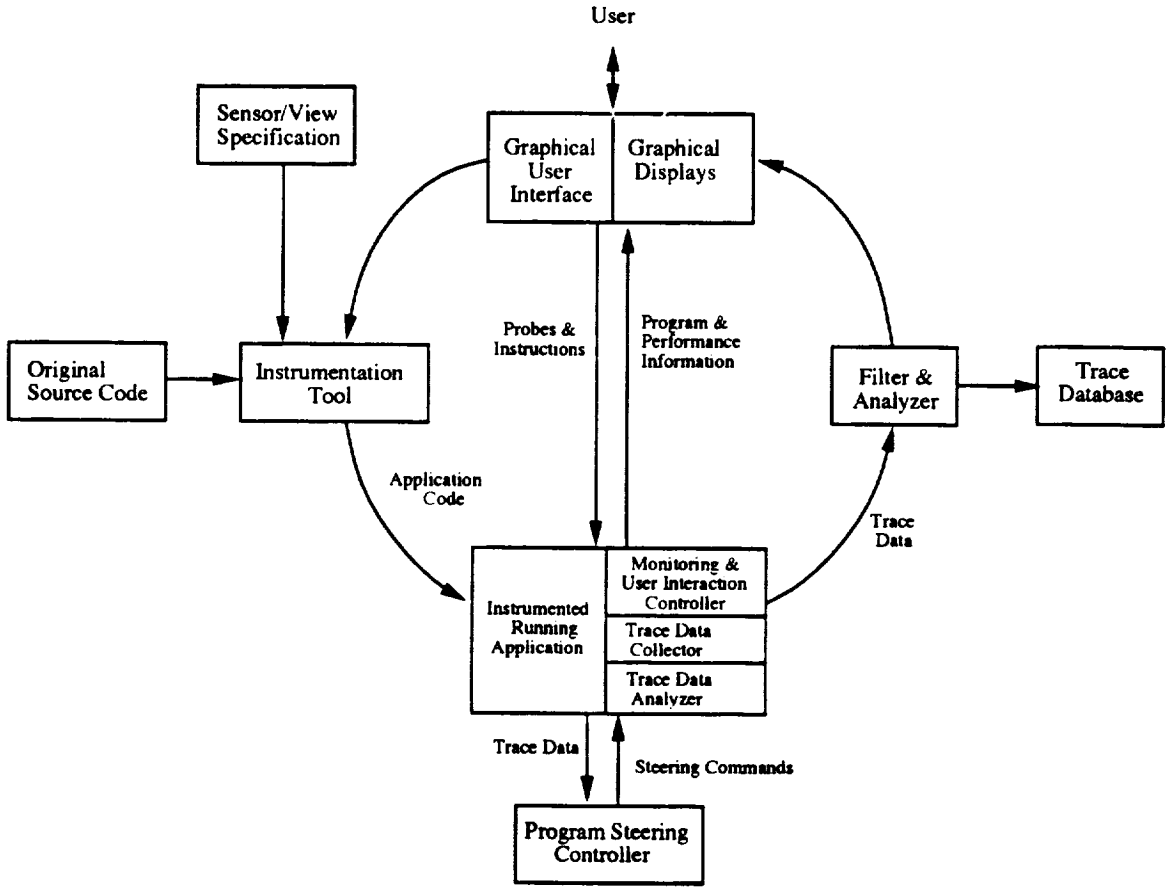


Figure 4: Overall architecture of Falcon.

The following steps are taken when using Falcon. First, the application code is instrumented with the sensors and probes generated from sensor and view specifications. Toward this end, monitoring specifications allow users to expose specific program attributes to be monitored and based on which steering may be performed. User programs and/or Falcon's user interface or analysis/steering algorithms directly interact with the runtime system in order to gain access to information about runtime-created sensor and actuator instances. When the application is running, program and performance information of interest to the user and to steering algorithms is captured by the inserted sensors and probes, and is collected and partially analyzed by Falcon's runtime monitoring facilities. These facilities essentially consist of monitoring data output queues attaching the user program being monitored to a variable number of additional components performing steering and low-level processing of monitoring output (discussed in detail in Section 3.3 below). Partially processed monitoring information is then fed to steering mechanisms for effecting on-line changes to the program or to its execution environment; or it is fed to the central monitor and graphical displays for further analysis and for display to end users. Trace information can also be stored in a trace data base for postmortem analyses.

The monitoring, steering, and user interaction 'controllers', as part of the Falcon runtime system, activate and deactivate sensors, execute probes or collect information generated by sampling sensors, maintain a directory of program steering attributes, and also react to commands received from the monitor's user interface. For performance, these controllers are physically divided into several *local monitoring controllers* and a *steering controller* residing on the monitored program's machine so that they are able to rapidly interact with the program. In contrast, the *central monitoring and steering controller* is typically located on

a front end workstation or on a processor providing user interface functionality.

Falcon uses the Polka system for the construction and use of graphical displays of program information[49]. Several performance or functional views (e.g., the aforementioned bargraphs and thread visualizations) have been built with this tool. However, in order to attain the speeds required for on-line data visualization and to take advantage of other performance display tools, Falcon also interfaces to custom displays and to systems for the creation of high-quality 3D visualizations of program output data, like the SGI Explorer tools.

3.3 System Implementation

Falcon's implementation relies on a Mach-compatible Cthreads library[36] available on several hardware platforms, including the Kendall Square Research KSR-1 and KSR-2 supercomputer, the GP1000 BBN Butterfly multiprocessor, the Sequent multiprocessor, and uni- and multi-processor SGI and SUN SPARC workstations. Figure 5 depicts the system's implementation. It is discussed next in the context of the

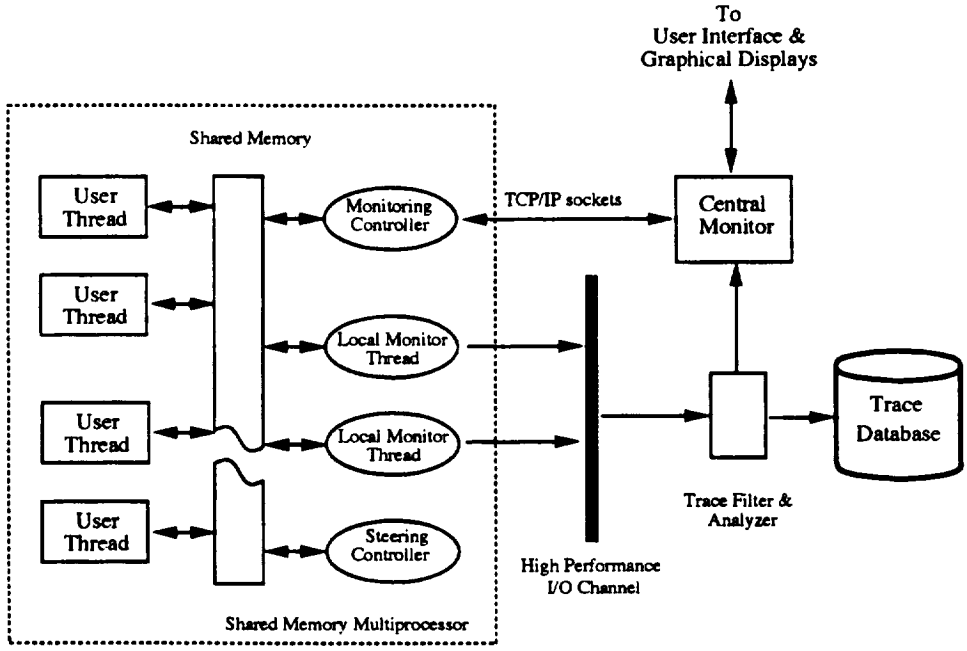


Figure 5: Implementation of the monitoring mechanism with Cthreads.

basic contributions of Falcon to the monitoring literature: (1) low monitoring latency and varied monitoring performance, also resulting in system scalability, (2) the ability to control monitoring overheads, and (3) the ability to perform application-specific monitoring and on-line analyses useful for steering algorithms and graphical displays.

Application-specific monitoring – sensors and sensor types. Using a simple specification language, programmers may define application-specific *sensors* for capturing (a) the program and performance behaviors to be monitored and (b) the program attributes based on which steering may be performed. The specification of a tracing sensor is shown in Figure 6. It simply describes the structure of the application data to be contained in the trace record generated by this sensor. From this declaration is generated the sensor subroutine shown in Figure 7. The body of this subroutine generates entries for an event data structure, then writes that structure into a buffer. A local monitoring thread later retrieves this structure from the buffer. Each sensor's code body is also surrounded by an `if` statement, so that the sensor can be turned on or off during program execution (ie., the monitoring system itself may be dynamically steered).

```
sensor work_load {
        attributes {
                int     domain_num;
                double  work_load;
        }
};
```

Figure 6: Specification of sensor work_load.

```
int
user_sensor_work_load(int process_num, double work_load)
{
    if (sensor_switch_flag(SENSOR_NUMBER_WORK_LOAD) == ON) {
        sensor_type_work_load  data;
        data.type = SENSOR_NUMBER_WORK_LOAD;
        data.perturbation = 0;
        data.timestamp = cthread_timestamp();
        data.thread = cthread_self();
        data.process_num = process_num;
        data.work_load = work_load;

        while (write_buffer(get_buffer(cthread_self()), &data,
                    sizeof(sensor_type_work_load)) == FAILED) {
            data.perturbation = cthread_timestamp() - data.timestamp;
        }
    }
}
```

Figure 7: Generated code of sensor work_load.

Figure 6 shows the specification of the tracing sensor that monitors the workload of each domain partition in MD, and Figure 7 depicts the generated sensor code. There are four *implicit fields* for any event record that describe the event's sensor type, timestamp, thread id, and perturbation. The purpose of the *perturbation field* is to record the additional time spent by the sensor waiting on a full monitoring buffer, if any. This 'buffer full' information is important for generating comprehensible execution time displays. A more detailed explanation of this problem appears with the discussion of Figure 13 in Section 5.3.

It is important to realize that each single sensor specification generates an event type; but its corresponding sensor code may be inerted to many different places within a single parallel program. Moreover, since new threads can be forked during an application's execution time, sensor instances are dynamic. The monitoring system identifies such dynamically created sensors using a combination of thread identifier and sensor type. In addition, users may explicitly register individual *instrumentation objects*, which correspond to specific calls to sensor code made by the target program. Such registration gives the monitoring system the ability to control (e.g., turn on or off) single invocations of sensor code instead of controlling all instances of a certain type of sensor as a whole.

Controlling monitoring overheads – sensor types and sensor control. The monitoring overheads experienced with sensor invocations may be controlled by use of different sensor types: sampling sensors, tracing sensors, or extended sensors. A *sampling sensor* simply writes its output into a structure located in shared memory periodically accessed by the monitor's runtime components also resident on the parallel machine, called *local monitoring threads*. A *tracing sensor* generates timestamped event records that may be used immediately for program steering or stored for postmortem analysis. In either case, trace records are stored in *trace queues* from which they are removed by local monitoring threads. Last, an *extended sensor* performs simple analyses before producing output data, so that some data filtering or processing required

for steering may be performed prior to output data generation. It is evident that sampling sensors inflict less overhead on the target application's execution than tracing and extended sensors. However, as shown in Section 4, the more detailed information collected by tracing sensors may be required for diagnosis of certain performance problems in parallel codes. Furthermore, the combined use of all three sensor types may enable users to balance low monitoring latency against accuracy requirements concerning the program information required for program steering.

Monitoring overheads may be controlled during each program run by direct interaction of user programs and/or Falcon's user interface and/or analysis/steering algorithms with the monitor's runtime system. First, sensors can be turned on or off during the application's execution[47]. Second, sensors can dynamically adjust their own behavior to continously control overall monitoring overhead and latency. For example, a tracing sensor that monitors a constantly accessed mutex lock can reduce its tracing rate to every five mutex lock accesses, thereby improving monitoring perturbation at the cost of reducing trace accuracy. In this paper, we use such dynamic sensor configuration for *selective monitoring* of a parallel program, where during a single program run, different monitoring methods are employed at different points in time. This is attained by enabling or disabling specific sensors, by switching from sampling to tracing sensors, and by changing the behavior of individual sensors (e.g., sensor sampling rates). Experimentation described in Section 4 will demonstrate the utility of selective monitoring with the MD code.

Controlling monitoring overheads – concurrent monitoring and steering. As depicted in Figure 5, local monitoring and steering threads perform trace data collection, processing, and steering concurrently and asynchronoulsy with the target application's execution. Local monitors and steering controllers typically execute on the target program's machine; but they may run concurrently on different processors, using a buffer-based mechanism for communication between application and monitoring threads.

An alternative approach performs all monitoring activities, including trace data capture, collection, and analyses, in the user's code. One problem with this approach is that the target application's execution is interrupted whenever a monitoring event is generated and processed, and the lengths of such interruptions are arbitrary and unpredictable if complicated on-line trace analyses are used. In contrast, the only direct program perturbation caused by Falcon is the execution of embedded sensors and the insertion of trace records into monitoring buffers. Such perturbation is generally predictable (results on the KSR-2 are presented in Section 4), and its effects on the correctness of timing information can be eliminated using straightforward techniques for perturbation analysis [30].

Falcon's runtime system itself may be configured (steered) in several ways, including disabling or enabling sets of sensors, varying activation rates, etc. One such on-line variation explored in detail in this paper is changing the number of local monitoring threads and communication buffers to configure the system for parallel programs and machines of different sizes. Such changes permit selection of suitable monitoring performance for specific monitoring and steering tasks, and they may be used to adapt the monitoring system to dynamic changes in workload imposed by the target application. For example, when heavy monitoring is detected by a simple monitor-monitor mechanism, new local monitors may be forked. Similarly, when bursty monitoring traffic is expected with moderate requirements on monitoring latency, then buffer sizes may be increased to accommodate the expected heavy monitoring load. Such parallelization and configuration of monitoring activities is achieved by partitioning user threads into groups, each of which is assigned to one local monitor. When a new application thread is forked, it is added to the local monitor with the least amount of work.

On-line analysis and display. Monitoring information partially processed by local monitors can be fed to Falcon's steering mechanism to effect on-line changes to the program and its execution environment. It can be sent to Falcon's central monitor for further analysis and for display of program behavior and application performance to end users. It can be stored in a trace data base for postmortem analysis. The central monitor, user interface, graphical displays, and trace database may reside on a different machine to reduce interference from monitoring activities to the target application's execution, and to capitalize on efficient graphics hardware and libraries existing on modern workstations. Section 5 describes some on-line analysis typically required for the on-line display of monitoring information: the need to reorder information produced by Falcon prior to its presentation to users. Falcon's interfaces to systems for the creation of high-quality

3D visualizations of program output data are out of scope of this paper. For the MD application, custom visualizations were constructed in order to gain the speeds required for on-line data viewing and steering.

3.4 On-line Steering Mechanisms

As described in Sections 1 and 2, program steering requires functionality in addition to that being offered by Falcon's monitoring components. Falcon's on-line steering component is a natural extension of its monitoring facilities. Similar to local and central monitors, steering is performed by a *steering server* on the target machine and a *steering client* providing user interface and control facilities. The steering server is typically created as a separate execution thread to which local monitors forward only those monitoring events that are of interest to steering activities. Such events tend to be a small proportion of the total number of monitoring events, in part because simple event analysis and filtering is done by local monitors rather than by the steering server. Steering decisions, then, are made based on specific attributes of those events, by human users (interactively) or by steering algorithms.

Falcon's steering system permits users to implement on-line control systems that operate on and in conjunction with the programs being steered. As a result, the primary task of each steering server is to read incoming monitoring events and then 'decide' what actions to take, based on previously encoded decision routines and actions, both of which are stored on a steering event database which is part of the server. This database contains entries for each type of steering event, where each event may either perform some actual steering action on the parallel program or simply note the occurrence of some monitoring event for future use in steering or for inspection by users from the client's user interface. Accordingly, the secondary task of each steering server is to interact with the remote steering client. The steering client is used to enable/disable particular steering actions, display and update the contents of the steering event database, and input direct steering commands from end users to the server. The steering client is not addressed by the target (on the parallel machine) performance measurements shown below. Its functionality and performance are discussed in more detail elsewhere.

At the lowest level of abstraction, a steering action that modifies an application is either a *probe* or an *actuator*. A probe updates a specific program *attribute* asynchronously to the program's execution. These attributes are defined by application programmers in an object-oriented fashion, where each specific program abstraction can define one or multiple attributes and then export methods for operating on these attributes. The steering event database lists all steerable objects and their program attributes. Also, actions are stored in the database with each event type. Actions are defined methods able to operate on the attributes of these objects. A complete object-oriented framework for defining and operating on program attributes is defined in [38]. The definition and dynamic adjustment of operating system level attributes is described in [35]. For purposes of this paper, the reader should assume that such attributes correspond to specific program variables (ie., to specific locations in the program's data). The steering server uses probes to update such variables at any time it chooses, and it uses actuators to have the program's execution threads enact certain steering actions on its behalf. Such actuators may also execute additional functions to ensure that modifications of program state do not violate program correctness criteria[5].

The performance of steering is assessed in Section 4.5 below.

4 System Evaluation

To understand the performance of the Falcon monitoring system, we evaluate its implementation on a Kendall Square Research KSR-2 parallel machine[2]. This machine has 64 processors interconnected by two rings. The KSR-2 supercomputer is a NUMA (non-uniform memory access) shared memory, cache-only

[2]The 64 node KSR-2 machine at Georgia Institute of Technology was upgraded from a 64 node KSR-1 during our experimentations. Therefore, some of the results presented in this paper are obtained on the KSR-1 machine, while others are obtained on the KSR-2. Programs running on the KSR-2 are roughly twice as fast as those running on a KSR-1 due to differences in machine clock speeds.

architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes. Each node consists of a 64-bit processor, 32 MBytes of main memory used as a local cache, a higher performance 0.5 Mbyte sub-cache, and a ring interface. CPU clock speed is 20 MHz on the KSR-1 and 40 MHz on the KSR-2, with a peak performance of 20 and 40 Mflops per node for KSR-1 and KSR-2, respectively. Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheaper. The parallel programming model implemented by KSR's OSF Unix operating system is one of kernel-level threads which offer constructs for thread fork, thread synchronization and shared memory between threads. This kernel-level thread facility is called Pthreads. Falcon itself employs Cthreads, a user-level threads facility that is built on top of Pthreads.

In the remainder of this section, we first evaluate the basic performance of Falcon's monitoring mechanisms, including measurements of the average costs of tracing sensors and of minimal and expected monitoring latencies. Next, using the MD code, we evaluate Falcon's ability to control monitoring overheads and to scale to different performance requirements. The overheads incurred by individual elements of the runtime steering library are evaluated last.

4.1 Sensor Performance

The perturbation, latency, and throughput of sensors depend on three factors: (1) the size of the event data structure, (2) the cost of event transmission and buffering from sensors to local monitors, and (3) sensor type. A tracing sensor generating a 'large' event containing many user-defined and implicit attributes will execute longer than one generating a 'small' event. Event transmission and buffering costs are affected by a variety of factors, including the number of event queues and local monitor threads, and the actual event processing demands placed on local monitors. Factor (1) is evaluated in Table 1, which depicts the basic costs of executing a sensor modulo its size, where basic costs include: (a) accessing the sensor switch flag, (b) computing the values of sensor attributes, and (c) writing the generated sensor record into an event queue. The table displays measured execution times on a KSR-2 machine.

Event record length	32 bytes	64 bytes	128 bytes
Cost (microseconds)	6.8	7.9	9.6

Table 1: Average cost of generating a sensor record on the KSR-2.

The results in Table 1 indicate that the direct program perturbation caused by inserted sensors should be acceptable for many applications for moderate amounts and rates of monitoring. Specifically, if an application can tolerate from 5% to 10% perturbation, then Falcon's monitoring mechanism can produce monitoring events at a rate from 7,500 to 15,000 events per second on the application's critical execution path. Given these costs, total perturbation of a parallel program can be derived as the cumulative cost of generating all of the sensor records in the program's critical path. A more complex perturbation model is required when considering side effects of such direct program perturbation[30].

The dominant factor in sensor execution is the cost of accessing the buffer shared between application and monitoring threads. The use of multiple monitoring buffers (one per user thread) in Falcon reduces the contention of buffer access by user and monitoring threads, so that the effective cost of buffer access is the cost of copying a sensor record to the buffer. This latter cost depends on the size of the sensor record, as clearly evident from the measurements in Table 1. It should be noted that these costs do not include perturbation that might be caused by bottlenecks in the processing and transmission of the events (which would result in delays in obtaining buffer space). However, such worst case perturbation may be avoided by making dynamic monitoring adjustments provided by Falcon's runtime monitoring mechanisms, such as turning off non-critical sensors, reducing a sensor's tracing rate, forking new local monitoring threads, etc.

4.2 Monitoring Latency and Perturbation

Monitoring latency is defined as the elapsed time between the time of sensor record generation and the time of sensor record receipt and (minimal) processing by a local monitoring thread. Low latency implies that steering algorithms can rapidly react to changes in a user program's current state[37]. Monitoring latency includes the cost of writing a sensor record to a monitoring buffer, the waiting time in the buffer, and the cost of reading the sensor record from the monitoring buffer. While the reading and writing times can be predicted based only on sensor size, the event waiting time in the monitoring buffer depends on the rate at which monitoring events can be processed by local monitors.

Buffer size (bytes)	Record length		
	32 bytes	64 bytes	128 bytes
256	69	73	87
1,024	68	71	84
4,096	68	70	83
16,384	69	73	85

Table 2: Minimum monitoring latency (in microseconds) on the KSR-2.

Buffer size (bytes)	Record length		
	32 bytes	64 bytes	128 bytes
256	164	181	242
1,024	201	264	294
4,096	211	277	498
16,384	256	347	556

Table 3: Latency at moderate monitoring rates (in microseconds) on the KSR-2.

Tables 2 and 3 depict the results of two experiments with a synthetic workload generator instrumented to generate sensor records of size 32 bytes at varying rates, using a single local monitoring thread. In Table 2, monitoring latency is evaluated under low loads, resulting in an approximate lower bound on latency. Results vary with event record sizes, but demonstrate the independence of monitoring latency on the size of the monitoring buffer at low loads. Table 3 uses higher monitoring loads[3] and experimentally demonstrates the expected result that larger monitoring buffers reduce program perturbation, but also increase monitoring latency for buffered events. Specifically, latency is not affected by buffer size at low rates, but increases with increasing buffer sizes even at moderate monitoring rates. This would indicate the use of smaller buffers. However, program perturbation can be larger with small buffers since programs must wait until buffer space is available when attempting to produce an event. Figure 8 demonstrates that the maximum event processing rate of a single local monitoring thread is about 40,000 to 45,000 events per second on the KSR-2 (assuming no significant processing of events in the local monitoring thread). However, monitoring latency remains acceptable when the monitoring rate is less than this saturation point.

The bottleneck due to limitations on the processing ability of single local monitors can be remedied by use of parallelism. Figure 9 shows that monitoring delay is reduced when multiple local monitors are used

[3] The measurements in Table 3 use a monitoring rate of approximately 40,000 events per second, which almost saturates the

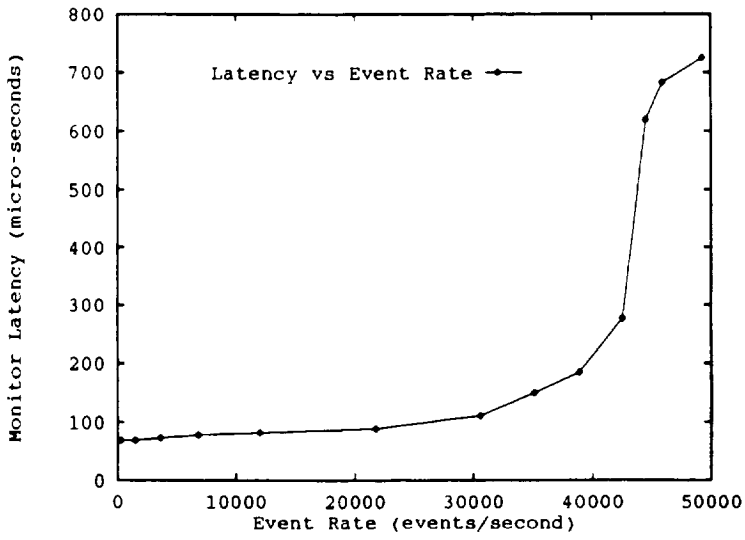


Figure 8: Monitoring latency versus event rate on the KSR-2.

to monitor the MD application. In this experiment, all procedure calls to the Cthreads library are traced. As MD runs on more processors, the frequency of calls to the Cthreads library increases, resulting in higher event rates. It is evident from the results shown in Figure 9 that additional local monitors are effective in reducing monitoring delay when this delay exceeds some threshold (around 200 microseconds for the MD code on the KSR-2). Below this threshold, the additional overheads associated with multiple vs. single local monitoring threads prevents their effectiveness.

In general, the measurements shown in Tables 2 and 3 and in Figures 8 and 9 demonstrate that there exists no general means of attaining both low monitoring latency and perturbation at arbitrary rates of monitoring (other than using additional hardware support). The approach taken by Falcon toward addressing this problem is simply to permit the configuration of the monitoring system itself (buffer sizes, number of trace buffers, number of local monitoring threads, and attachment of monitoring threads to buffers – monitoring load distribution) to offer the performance characteristics desired by the application program. Such configuration can be performed dynamically in a fashion similar to on-line program steering, where the saturation points for local monitors may be used as triggers for configuring the monitoring system itself.

4.3 Monitoring the MD Code

This section demonstrates the overall performance and utility of Falcon's monitoring mechanisms, again using the MD application. Measurements in this section are taken on a 64-node Kendall Square Research KSR-1 machine. The specific MD simulation used in these measurements uses a cylindrical domain decomposition; MD performance and speedups with different decompositions are evaluated in detail elsewhere[9].

Table 4 depicts the results of four different sets of MD runs, normed against a run of MD without monitoring. These experiments compare the performance and perturbation when using Falcon for five different cases: (1) when no monitoring performed (Original MD), (2) when tracing only MD calls to the underlying Cthreads package (Dft Mon Only), (3) when tracing Cthreads events as well as sampling (using sampling sensors) the 10 most frequently called procedures in MD (Dft Mon & Sampling), (4) when using the Unix GProf profiler existing on the KSR-1 machine (MD with Gprof), and (5) when tracing Cthreads events as well as the 10 most frequently called procedures in MD (Tracing All Mon Events). The table and figures list computation times and speedups with different numbers of processors. These measurements do not consider

single local monitoring thread used in the experiment.

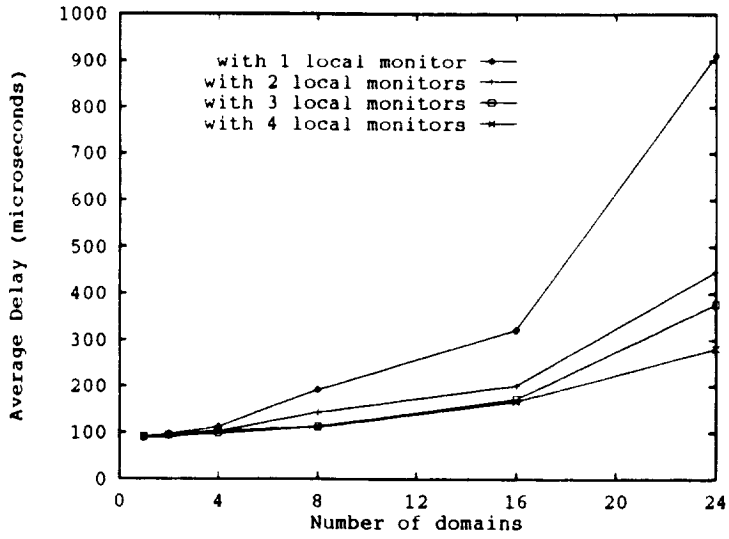


Figure 9: Monitoring latency with multiple local monitors on the KSR-2. (Each domain of particles is assigned to one processor.)

Number of Processors	Execution Time of Each Iteration (seconds) & Monitoring Overhead				
	Original MD	Dft Mon Only	Dft Mon & Sampling	Tracing All Mon Events	MD with Gprof
1	8.19	8.19(< 1%)	9.61(17%)	114.60(1299%)	22.53(175%)
4	2.65	2.65(< 1%)	3.21(21%)	59.30(2140%)	7.29(175%)
9	1.45	1.45(< 0%)	1.72(19%)	65.33(4406%)	4.28(195%)
16	0.62	0.63(1%)	0.73(17%)	54.29(8628%)	1.71(175%)
25	0.30	0.31(2%)	0.35(16%)	41.56(13776%)	0.82(173%)
36	0.19	0.20(4%)	0.23(16%)	33.65(17245%)	0.54(195%)

Table 4: Average execution time and perturbation of each iteration of MD with different amounts of monitoring or profiling on KSR-1.

the costs of either forwarding trace events to a some front end workstation or storing them in a trace data base, since those costs are not dependent on Falcon's design decisions but rather on the performance of the networking code and/or file system implementation of the KSR-1 machine. Specifically, measurements with trace events essentially 'throw away' events at the level of local monitors, whereas the measurements with sampling sensors actually use local monitors to retrieve and evaluate sampling sensor values stored in shared memory on the KSR-1 machine.

The MD application's performance with different amounts of monitoring or profiling is depicted in Figure 10, and the resulting program perturbation due to monitoring is shown in terms of speedup degradation in Figure 11. Evaluations of each experiment are presented next. The first experiment (Dft Mon Only – default monitoring) measures the overhead of monitoring when Falcon traces all calls to the underlying Cthreads package. Specifically, this is the amount of monitoring required for the thread life-time view

[4]Super-linear speedups are due to the KSR-1's ALLCACHE memory architecture. When MD runs on a large number of processors, it can load all of its code and data into the fast sub-caches or local caches associated with these processors, while

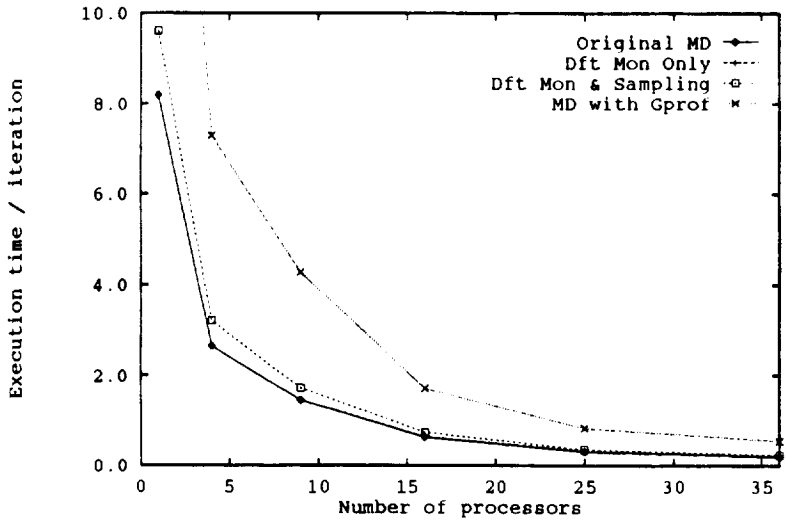


Figure 10: Comparing average execution time of each iteration of MD on the KSR-1.

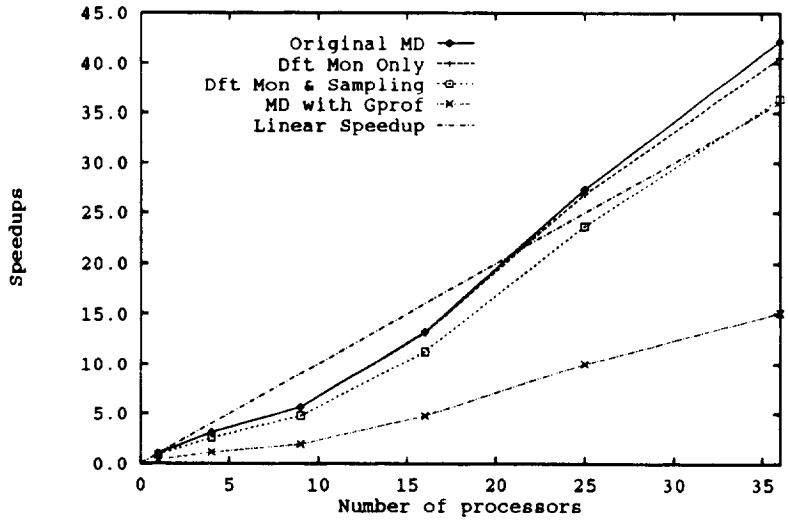


Figure 11: Comparing speedups of MD on the KSR-1[4].

shown in Figure 13 in Section 5.2. The monitoring information being collected includes the runtime activities associated with each thread (such as thread_fork, thread_join and thread_detach events), synchronization calls, and all other information displayed in the thread life-time view. It is apparent from Figures 10 and 11 that default monitoring does not noticably perturb the execution of MD. However, monitoring overheads increase slightly with an increasing number of processors, which is caused by an increasing number of events (more user threads imply more cthreads calls, and hence more monitoring events) generated during a shorter execution time and beginning to saturate the available local monitoring threads. The creation of additional local monitors can remedy this problem.

The second experiment compares the overhead of Falcon monitoring with that of commonly used program profiling tools, namely, with Gprof. The KSR implementation of Gprof used in these measurements has been optimized to take advantage of the machine's memory architecture in several ways, including replicating counters on each processor to avoid remote accesses. To compare fairly, we exclude the time spent on writing the results to file from the presented Gprof execution times. Using Falcon, we monitor the 10 most frequently called procedures in MD. These calls constitute about 90% of all procedure calls made in the program. Each procedure is monitored by a sampling sensor, which increments a counter for each procedure call being monitored. Counter values are sampled each millisecond by local monitoring threads. The result of this experiment is the addition of 20% to MD's total execution time. In comparison, with Gprof, the execution time of MD is increased by approximately 180%. Similar advantages of Falcon to other profiling tools are demonstrated when using Prof. Experimental results not reported in detail here show that Prof's overhead is approximately 130% [15]. The results described above are not surprising, since profiling tools typically maintain large amounts of compiler-derived information about a parallel program's attributes. In comparison, Falcon only maintains the specific information required for taking certain program measurements.

While the first two experiments clearly demonstrate the importance of monitoring only the program attributes of interest to the user, the third experiment shows that it is also important to adjust or select the techniques being used for information capture. In this experiment, tracing sensors are used in place of sampling sensors for monitoring the 10 most frequently called procedures in MD, which results in a very significant increment of monitoring overheads. The excessive performance penalties arising from this 'misuse' of tracing sensors are primarily due to the direct perturbation caused by monitoring tens of millions of procedures calls and are exacerbated by the saturation of the single local monitoring thread being used in the experiment. The resulting (lack of) performance clearly demonstrates two points. First, since tracing sensors are too expensive for procedure profiling, any monitoring system must offer a variety of mechanisms for information capture and analysis, including both sampling and tracing sensors. Second, since tracing can help users gain an in-depth understanding of code functionality and performance (see Sections 4.4 and 5), users should be able to both control the rates at which tracing is performed and the specific attributes of the application that are captured via tracing. We call the user's ability to focus monitoring on specific system attributes *selective monitoring*. It is explained in more detail in the next section.

In general, the experiments with MD presented in this section demonstrate that the multiple monitoring mechanisms (e.g., tracing vs. sampling sensors) supported by Falcon can be employed such that monitoring overheads remain moderate for realistic parallel application programs.

4.4 An Example of Selective Monitoring Using Falcon

In this experiment, the MD code's most computationally intensive component is monitored using Falcon's sampling and tracing sensors. Both types of sensors are needed since programmers require both summary (e.g., total number of invocations) and sequencing or dependency information (e.g., 'b' was done after 'a' occurred) to understand and evaluate code performance. Such dynamically selective monitoring is useful since programmers can focus on different phenomena at different times during the performance evaluation process. The specific purpose of the selective monitoring demonstrated in this section is to understand the

it cannot do so when running on a single processor.

effectiveness of certain, commonly used 'short cuts' which are intended to eliminate or reduce unnecessary computations in codes like MD.

The dominant computation of each domain thread in the MD code is the calculation of the pair forces between particles, subject to distance constraints expressed with a cut-off radius. This calculation is implemented with a four-level, nested loop organized as follows (pseudocode is shown below):

```
for (each molecule mol_1 in my domain) do
    for (each molecule mol_2 in domains within cut_off_radius) do
        if (within_cutoff_radius(mol_1, mol_2)) then continue;
        for (each particle part_1 in molecule mol_1) do
            if (within_cutoff_radius(part_1, mol_2)) then continue;
            for (each particle part_2 in molecule mol_2) do
                if (within_cutoff_radius(part_1, part_2)) then continue;
                calculate_pair_forces(part_1, part_2);
            end for
        end for
    end for
end for
```

The inner three levels of this loop check the distances between molecules and particles to eliminate all particles outside the cut-off-radius. When the distance between two molecules is checked, three dimensional bounding boxes are used for each molecule. Each molecule's bounding box includes all of its particles. The minimum distance between two molecules is defined as the distance between their bounding boxes' closest points, whereas the minimum distance between a particle and a molecule is the distance from the particle to the molecule's bounding box' closest point.

The question to be answered with selective monitoring is whether the additional costs arising from the use of bounding boxes is justified by the saved costs in terms of the resulting reduction in the total number of pair force calculations. More specifically, does the reduction is total number of pair force calculations justify the additional computation time consumed by bounding box calculations? A simple selective monitoring mechanism is used to answer this question, by dynamically monitoring the performance of this four-level loop. Specifically, a sampling sensor is first used to monitor the hit ratios of the distance checks at all levels. When a hit ratio at some loop level falls below some threshold, say 10%, a tracing sensor monitoring this loop level is activated to obtain more detailed information. The intent is to correlate the low hit ratio with specific properties of domains or even of particular molecules. Specifically, for each 'hit' distance check at the 2nd level loop, we trace the distances between particles and molecules at the 3rd level loop. The motivation is to understand the relationships of distances between molecules' bounding boxes and with distances between specific particles of a molecule with the bounding boxes of other molecules. In other words, what is the effectiveness of the second level distance check?

The performance of such dynamically selective monitoring is presented in Table 5. In these measurements, we use a MD data set that contains 300 molecules with 16 particles each. This relatively small system is then monitored by insertion of sampling and tracing sensors at one, two, three, or all levels of the nested loop (the outermost level is numbered zero, while the innermost three). Tracing at all levels results in overheads that are somewhat unacceptable, especially when the same tracing is performed for larger systems. This is apparent from the increases in monitoring overheads experienced when tracing at all levels for increasing system sizes (e.g., 9 vs. 16 domains). On the other hand, when tracing only at lower levels (e.g., levels 1 or 2), overheads are less than 1% for smaller systems and no more than 5% for larger systems, and sampling overheads remain small for all system sizes.

These results indicate that selective monitoring is quite effective, even when applied to this highest frequency set of loops in the MD program's execution. Furthermore, the strategy of sampling execution and only initiating tracing when some problem (e.g., a low hit ratio) is experienced should result in composite monitoring overheads that approximate the sampling overheads experienced with Falcon for long system

No. of domains	Execution Time of each MD time step (seconds) & Monitoring Overhead					
	No Monitoring	Sampling Hit-Ratio	Tracing at Level 1	Tracing at Level 2	Tracing at Level 3	Tracing at All levels
4	1.28	1.28(< 1%)	1.28(< 1%)	1.34(5%)	1.38(8%)	1.46(14%)
9	0.703	0.706(< 1%)	0.708(< 1%)	0.734(4%)	0.742(5%)	0.794(13%)
16	0.301	0.301(< 1%)	0.304(1%)	0.316(5%)	0.323(7%)	0.356(18%)
25	0.147	0.147(< 1%)	0.149(1%)	0.155(5%)	0.158(7%)	0.188(28%)

Table 5: Performance of selective monitoring of the MD's main computation component on the KSR-2.

runs. In conclusion, the on-line 'steering' of Falcon's monitoring mechanisms themselves can be used to control runtime monitoring overheads.

4.5 Performance of On-line Steering

As outlined in Section 3.4, the steering component of Falcon operates in conjunction with its monitoring components, by receiving and processing selected monitoring events, then controlling the application's execution based on such runtime state information. This section presents low-level measurements that highlight the basic performance and operation of program steering when viewed as a low-level control system. Therefore, tor these measurements, networking is disabled and, hence, no remote operations are performed with the steering client. Three processors are used, one running an application thread, a second running a single local monitoring thread, and a third running the steering server. Algorithmic steering is used in order the evaluate the basic costs of observing some interesting program state via the (1) local monitor and (2) steering server, (3) making a simple decision based on that observation, and (4) enacting that decision by taking a steering action. These costs are evaluated in the first experiment, which measures the latency of actions (1)-(4) for a lightly loaded system:

Measurement	microseconds
Avg. Latency	610
Min. Latency	224
Max. Latency	4483

Table 6: Latency for closed-loop steering.

Table 6 describes the closed-loop latency for steering under the following conditions: a total of 100,000 sensor events are generated by the application program, they are received by local monitors, and they are then forwarded to the steering server without any additional filtering or processing. (4) The steering server performs a simple action in response to each event's receipt. This action consists of a write to a variable in the application program. (1)-(4) are performed for an application program that repeatedly performs the following tasks. First, it generates a monitoring event using a Falcon sensor. Second, the program waits on some pre-specified memory location that will be asynchronously updated by the steering server. Third, the steering server receives the event from the local monitor, reads the event type, accesses its database of steering events to determine the actions required for this event type, and then uses a probe to enact this action. The probe essentially changes the value of the memory location on which the program is waiting.

For these measurements, the database only contains a moderate number of different steering event types and their respective actions.

The results depicted in Table 6 demonstrate an average latency of 610 microseconds for algorithmic program steering using Falcon. This implies that program steering can be performed using Falcon at rates approximating the execution times of the set of inner loops in programs like MD. However, it is not possible to use Falcon's current mechanisms to perform steering of program abstractions accessed with high frequencies, like the adaptable locks described in [35]. Such high-rate and low-latency steering must be performed by local monitors themselves, possibly using custom implementations of sampling sensors. Two surprising results depicted in the table are (1) the high maximum latency for servicing a steering event (4,483 microseconds), which is due to mismatches in the scheduling of application, monitoring, and steering threads, and (2) the low minimum latency of 224 microseconds for steering, which is comprised mainly of the costs of event transmission from the application, to local monitor, to steering thread, respectively (recall that monitoring latency is approximately 70 microseconds).

The second experiment evaluates more complex steering actions, by forcing the steering thread to take multiple actions for each received steering event. Specifically, Table 7 depicts the latencies of steering when for each received steering event, the steering server takes some variable number (1, 10, 100) of actions involving both probes and actuators. The purpose of this experiment is to determine the incremental costs of steering.

Complex action	Number	Microseconds	Gain
Probe write	1	643	-
Probe write	10	2930	4.6
Probe write	100	15418	23.9
Actuator	1	627	-
Actuator	10	1207	1.9
Actuator	100	7870	12.6

Table 7: Average closed-loop latency with complex actions.

First, consider the costs of probe-based steering. For each probe, different memory locations have to be accessed. In this experiment, worst case costs are evaluated by forcing the steering server to access its database once for each received event. As the complexity of the action increases, the execution time required by the steering server to execute this action increases. For a complex action, the server must execute this action before accepting any other events from the monitor. As seen from Table 7, the basic probe write costs 643 microseconds. These costs increase by a factor of 4.6 for a complex action that requires 10 probe writes (2,930 microseconds), and they increase by a factor of 23.9 for very complex actions (to 15,418 microseconds for 100 probe writes). These measurements indicate that the steering server's construction is sensible in that it permits the basic costs of steering to be amortized over the costs of increasingly complex actions.

The second portion of Table 7 addresses actuator costs. These measurements are interesting in their demonstration of scalability for actuators versus probes in terms of the resulting costs arising for steering servers. Specifically, 100 actuator activations do not correspond to 100 executions of actuator code by the steering server. Instead, the server simply enables the actuator once (for 100 executions), and then relies on the user program to execute steering actions using the enabled actuator. As expected, actuator-based steering costs do not depend on the number of steering actions taken; they depend only on the number of times actuators are enabled or disabled! However, in order for the steering server to program the actuators, the server must write to a buffer shared by the server and the application. A lock must prevent simultaneous access to this region. Unfortunately, this lock is a point of contention between the server and the application

and, as such, the performance of the server is somewhat degraded.

From these measurements and in accordance with earlier results presented in this paper and in [2] addressing the time required for analyzing monitoring output, it should be apparent that the steering component of Falcon is sufficiently fast to (1) keep up with fairly high rates of monitoring and (2) steer programs at rates and with overheads enabling medium grain on-line program configuration[5] and application steering.

5 The On-line Presentation of Monitoring Information

The process of on-line user interaction with a target application includes (1) obtaining application-specific information through monitoring mechanisms, (2) displaying this information to the user, and (3) controlling program execution based on (2). Steps (1) and (3) have been discussed in the previous sections. This section presents Falcon's methods for presenting monitoring information to end users.

5.1 Falcon's On-line Display System: An Overview

Graphical displays have been shown useful in presenting data structures [39], algorithms [48], runtime program behaviors [29], and performance information[17, 41] to human users. However, most current work deals primarily with off-line graphical and animated presentations of program and performance information. Falcon's specific goals concerning the presentation of information to end users are to evaluate: (1) how on-line displays of program information can help users understand a target program's performance and runtime behavior, and (2) how users can use such an understanding to steer their parallel codes. The resulting necessary attributes of graphical displays used for program steering include:

- *Application-specific displays* - program information should be presented to end users in familiar terms, that is, by reference to abstractions in their programs rather than by reference to machine or operating system details with which they may not be familiar.
- *Behavior-preserving displays* - program monitoring and information display cause program perturbation and they exhibit a lag between the time of information generation and its display to end users. Programmers and end users must be made aware of both monitoring perturbation and information delay and should, potentially, be able to control them when performing program steering.

The Falcon information display system offers functionality addressing both attributes. First, the central monitor in Falcon is able to 'attach' any number of event streams from local monitors to its input ports, and 'route' events to any number of analysis packages and subsequent displays through its output ports. Such attachments to either input or output ports may be changed during program execution, if needed. As a result, event streams may be subjected to multiple analysis packages and then displayed by any method of display chosen by end users. Attachments are created and dissolved by commands to the central monitor, and alternative displays may be associated with event streams by use of class hierarchies within the display process. Figure 12 demonstrates the use of three alternative display methods for a sample event stream. The first method, built on the X window system and the Athena and Pablo [41] widget sets, uses the **work_load** events for the display of the application's load balance information. The second method applies statistical methods to analyze events from the application and then presents the resulting summary information to end users in a textual format. The third method, built on the Polka animation system and the Motif widget set, uses specific events from the on-line event stream to animate the program's behavior at the threads level (also see Figure 13).

In this paper, we focus on 2D graphical displays of program behavior or performance, which have been shown useful for on-line steering in previous sections, where graphically displayed load balance information is used to direct the execution of the MD code (e.g., see Figure 1). Our future work is combining event streams from the monitoring system with program output typically generated via file system calls, so that users can

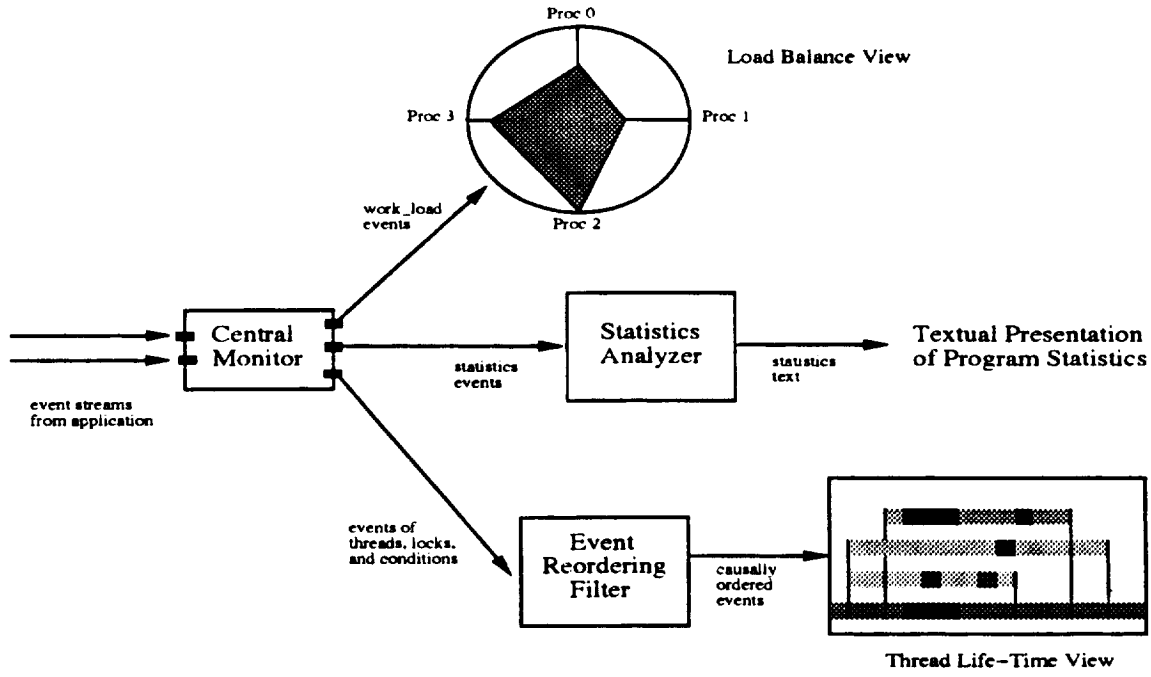


Figure 12: A sample on-line display system for an application.

understand and direct program execution in terms of individual program variables (e.g., 'energy levels' or 'molecular positions' in the MD code). Toward this end, we are now developing and integrating into Falcon interactive 3D data visualization tools. These tools are being applied to a large-scale atmospheric modeling application.

Falcon attempts to preserve the original behavior of the parallel program when displaying program performance. Two issues arise: (1) monitoring can perturb program execution, and (2) the monitoring system's method of event collection via buffers does not preserve the actual time ordering of events being produced and displayed. Specifically, since monitoring events are first buffered on the parallel machine and local monitoring threads are not perfectly synchronized, events received by the central monitor and ultimately, by analysis and display packages are not guaranteed to be in order. For off-line monitoring, event files can be sorted. For on-line monitoring, event reordering must be performed on-line and with suitable efficiency. Furthermore, in order to preserve the behavior of the original program when presenting such information to users, reordering must be performed so that the causal order of events exhibited by the executing program is preserved and enforced.

The remainder of this section describes how Falcon displays address both program perturbation and monitoring delay, by providing on-line perturbation information displayed as *perturbation events*, and by reordering and displaying monitoring events according to known information about a program's causal execution order. In both cases, the system-level default information about threads available in Falcon is utilized.

5.2 The Thread Life-Time View: Performance of Threaded Programs

The graphical *thread life-time* view described next is one contribution of the Falcon project toward understanding the dynamic behavior of threads-based parallel programs. Available with Cthreads programs running on SGI and SPARC workstations and on KSR machines, this view uses the default sensors embedded in Cthreads. The view is implemented with the Polka animation library [49]. Since Polka provides a variety of graphical objects, animation primitives, and user interface facilities, the program defining the thread

life-time view only consists of roughly 200 lines of application-level Polka code. Polka runtime libraries provide a flexible animation scheduling policy, permit different temporal mappings of program events to their animations, and therefore facilitates the construction of on-line displays. Polka is described and evaluated in detail in [49].

The thread life-time view shows the different states of threads over time. The state information depicted in the view includes thread execution time, blocking time, waiting time in ready queues, the identifiers of conditions or mutex locks on which threads are blocked, and thread identifiers. From this information, users can easily discern the time a thread spends doing useful computation versus waiting for other threads, the degree to which different threads' executions are synchronized, processor utilization, and other useful program and performance information. Figure 13 shows a snapshot of the MD program's execution on four processors (ie., molecules are partitioned into four domains) on the KSR-1. When a new thread is forked,

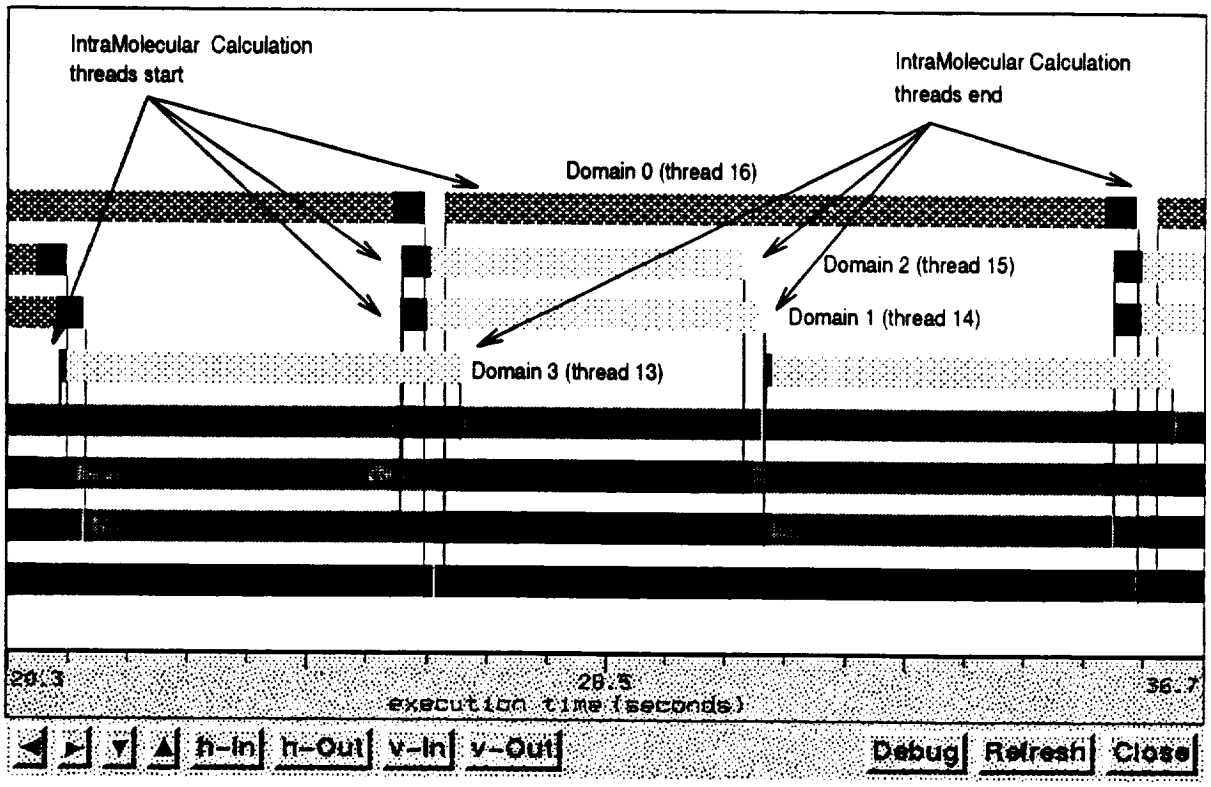


Figure 13: An annotated thread life-time view of MD. The actual display uses colors to represent different threads and different thread states.

a narrow horizontal bar is created to represent the new thread's life-time, and a vertical line is drawn from the parent thread to the child thread at the time of the fork event. A narrow bar terminates when the thread, which the narrow bar represents, joins another thread after it exits or when a detached thread calls thread_exit. In the case of thread_join, another vertical line is drawn from the caller thread to the thread it is joining. The resulting empty space in the display can be reused for depiction of a new thread, if there is any. Since the color display has to be rendered into monochrome for this presentation, some annotations are added to compensate for lost information. Specifically, the thread life-time view uses different colors and patterns to represent thread states. In Figure 13, the solid black pattern represents a thread in a running state, while the dark gray pattern represents a thread waiting for a condition. The lightly dotted pattern indicates that a thread has called thread_exit and is waiting to join to another thread. The heavily dotted pattern indicates that a thread is in a processor's ready queue; it is waiting for another thread using the processor to complete its execution.

The bottom four bars in Figure 13 represent four threads each computing properties of the molecules in their respective domains (numbered 0, 1, 2, and 3 from bottom to top). Each such 'domain' thread forks a second 'helper' thread in every iteration. These 'helper' threads are shown as short bars above the 'domain' threads (they may not be in the same order as the 'domain' threads). They calculate intramolecular forces while domain threads wait for information from neighboring domains. At the end of each iteration, domain threads perform neighbor-to-neighbor synchronization. As apparent from the figure, the intramolecular calculations of domain 3 proceed and perform useful computations while the domain thread is waiting for completion of the computations of neighboring domains' threads. However, domain thread 3 experiences significant wait time since its domain computation is finished quickly, whereupon it must wait for completion of neighboring threads' computation (it needs their data from the current iteration before starting the next iteration). Therefore, it is also clear from this view that work load is imbalanced: domain 3 has little work to do, while domain 0 is almost always busy. This illustrates a problem with the slab-based domain decomposition strategy used in this run of the MD program: Domain 0 is responsible for additional molecules in the substrate on which the liquid being modeled is layered; this substrate is much denser than the liquid and therefore, contains many more molecules.

5.3 Perturbation Events

The thread life-time view shown in Figure 13 can help users understand program performance problems only if the thread running and waiting time shown is due to application's execution and synchronization rather than monitoring perturbation caused by executing extra code and additional synchronization between application threads and monitoring threads. Section 4.1 shows that the basic perturbation due to sensor code execution is quite small, provided that monitoring buffers are sufficiently large and local monitoring threads can process events fast enough to keep monitoring buffers from being completely full. Furthermore, the direct perturbation due to the execution of a sensor is easily predicted from the sensor's type. As a result, such perturbation can be removed from the event trace by application of simple perturbation analysis. However, if local monitoring threads cannot keep up with the rate of event generation, then monitoring buffers will eventually become full, and application threads will have to wait for some time until events have been removed from such full buffers. Since such waiting time caused by filled monitoring buffers is not due to the program's code or data, the resulting thread life-time view can be incomprehensible or misleading to end users.

Figure 14 depicts a potentially misleading thread life-time view constructed with an event trace captured from a large-scale atmospheric modeling code. The problems in this view are due to an unsuitable configuration of the monitoring system (a single local monitoring thread), which is quickly overwhelmed by events from a large number of computational threads. Without perturbation events, it would appear to programmers that their computational threads execute for different amounts of time. This is misleading since in this program, each of the computational threads have the same amount of work. In fact, when first using Falcon, one of the atmospheric code's implementors spent several hours chasing a non-existing load imbalance indicated by the life-time view (without perturbation events). A more precise inspection of the view in Figure 14 shows pure black bars that represent 'worker' threads, each responsible for a partition of the computation space. The three bars below each 'worker' thread are 'helper' threads, which are employed by the 'worker' thread to help calculate separate terms in its computation. The iterative algorithm performs barrier synchronization after threads finish their work and before the next iteration starts. The figure indicates that the third 'helper' thread of the top 'worker' thread and the first 'helper' thread of the bottom 'worker' thread waited a very long time for mutex locks. In addition, the second and third 'helper' threads of the bottom 'worker' thread have significantly longer computation times than other 'helper' threads. These superficial observations imply that the program has unbalanced work loads and improper synchronizations. However, the improved threads life-time view with perturbation events presents a different picture.

Figure 15 shows the same execution of the atmospheric modeling code, with special perturbation events depicting the total blocking times experienced by its threads on the monitor's event buffers. It is clear from this 'correct' view that the 'helper' threads have balanced work loads, but that their execution times are extended due to monitoring perturbation experienced by the 'helper' threads of the bottom 'worker'.

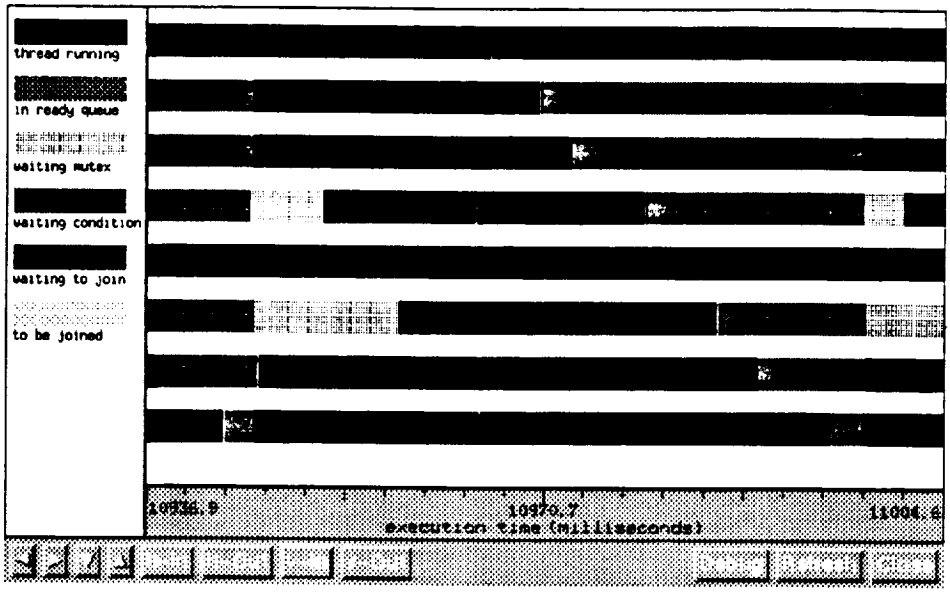


Figure 14: A thread life-time view that shows perturbation events.

Specifically, the extremely long blocking times for mutex locks apparently experienced by the third 'helper' thread for the top 'worker' and the first 'helper' thread for the bottom 'worker' (shown in Figure 14) are not due to mutex locking. They are due to additional thread waiting times experienced when writing mutex_end_lock events to the monitoring buffers. From this example, it should be evident that perturbation events help users understand the monitoring system's contribution to total thread execution and wait times. Our current work is generalizing this straightforward notion of perturbation events to apply more sophisticated sequential perturbation analyses (e.g., see [30]). Another type of monitoring perturbation, causing misordered event streams, is discussed next.

5.4 On-line Event Reordering

Event orderings and program animation. Displays like the thread life-time view of Figure 13 can provide users with insights into program progress and correctness. However, the perturbation example described above already demonstrates that graphical views can be quite misleading and confusing if the information being displayed does not correspond to the program's actual execution. This section focusses on another issue with on-line graphical views, namely, on the fact that the graphical animation order determined by the receipt and display of monitoring events does not correspond to the actual or causal order in which program events occur! Such misorderings can both confuse users and more critically, cause failures of the animation itself. For example, causal ordering would require that the thread_fork event creating a thread precede any event executed by the new thread. A display that shows a child running before it has been forked by its parent does not make any sense. Furthermore, suppose that the first event for this child thread is a condition_wait event. In the thread life-time view of Figure 13, this event is represented by a change in the color and fill pattern of that thread's horizontal bar. However, if the thread_fork event has not been received by the display system, the horizontal bar does not yet exist. When the display system attempts to perform a color-change action on this non-existent object, it crashes. Some of these crashes could be avoided by adding a layer of error-checking code to the display system, but this adds execution overhead, makes displays more difficult to design, and still leaves the viewer with displays that may not be useful.

The out-of-order events that cause problems for the display system *cannot* have occurred in the program's execution, since they would violate causal event orderings determined by program and language semantics.

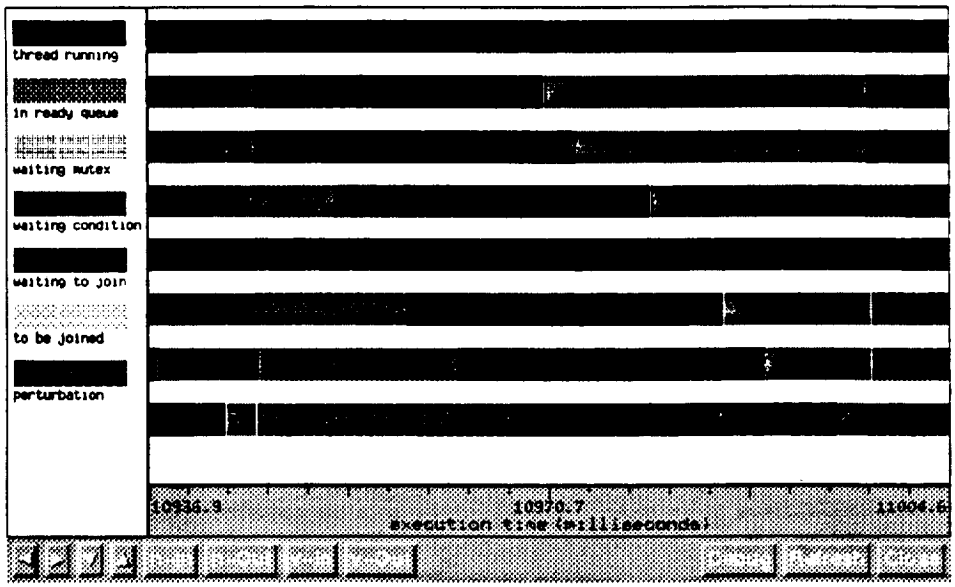


Figure 15: A thread life-time view that shows perturbation events.

Instead, misorderings existing in the event stream are due to the buffering and processing methods employed in the monitoring system. Specifically, high monitoring performance (i.e., low perturbation) requires that events for each thread be buffered until a local monitor is ready to process them. Furthermore, different local monitors send events to the central monitor at their own speeds, in part because the number of events to be processed and the processing requirements of individual events may differ among local monitors. As a result, while the event stream reaching the display system is in-order with respect to each individual thread (recall that each thread uses only a single event buffer), it may be out of order with respect to thread events from different threads.

On-line event reordering. The diagnosis and correction of out-of-order events is a common problem in parallel and distributed monitoring systems. Existing systems (e.g., ParaGraph[17] and SIEVE[42]) rely on a sort by timestamp value to impose a total order on all events stored in event files. The on-line nature of the Falcon monitoring system precludes using such a solution, and sorting by timestamp order does not entirely eliminate the problem of out-of-order events[4]. In addition, coarse clock granularities and poor clock synchronization among different processors may lead to event timestamps that do not accurately reflect the actual order of program execution. For example, if the system clock changes only every 10 milliseconds, and if two events occur within this time frame, then the ordering of these two events cannot be determined within this period. A more realistic concern on the KSR supercomputer used in our work is poor clock synchronization, where one processor's clock can be sufficiently ahead of another processor's clock so that the elapsed time between a thread fork and the first event executed by the child appears to be negative. This problem is exacerbated when threads are allowed to migrate across processors, something we avoid in Cthreads but is permitted in the Pthreads parallel programming library on the KSR machine.

Ordering rules. The previous discussion of out-of-order events makes apparent that the use of timestamps is not sufficient for determining and enforcing suitable, global event orderings. Falcon addresses this issue by employing an *ordering filter* between the central monitor and the display system (see Figure 12). This filter ensures that the event stream reaching the display system adheres to a pre-specified, known causal ordering among thread events. This ordering filter has knowledge of all execution threads, mutex locks, and conditions identified occurring in the event stream. The algorithm employed by the filter follows a "minimum-intervention policy". Namely, it examines each event in the stream arriving from the monitoring system, checks the applicable ordering rules for this event type, and if no rules are violated, forwards the

event to the display system. If a rule violation is indicated, the event is held back until the rules are satisfied.

As an example, consider the ordering rule for a mutex lock event. Actually, a mutex lock is recorded as two separate events - the mutex_begin_lock event indicating that a thread has attempted to obtain the lock, and the mutex_end_lock event indicating that a thread has succeeded in obtaining the lock. The following ordering rule is observed by the filter for a mutex_end_lock:

```
mutex_end_lock t m n <- ((thread_init t || thread_fork pt t) &&
                         (mutex_init m || mutex_alloc m) &&
                         (mutex_unlock m n-1) )
```

This rule may be translated as: "The mutex_end_lock event with parameters t, m, and n, may be passed on to the display system if thread t has been initialized or forked by a parent thread, mutex variable m has been initialized or allocated, and the mutex_unlock event for variable m, sequence number n - 1 has already been passed on to the display system." Accordingly, the parameters associated with the event mutex_end_lock are t, the id of the thread attempting to obtain the lock, m, the id of the mutex variable, and n, the sequence number indicating the number of successful lock attempts on this particular mutex variable. Among these parameters, the most interesting parameter is n, since it required an unforeseen augmentation of the Falcon system and since it enables the efficient implementation of on-line event ordering discussed below.

The rule applied to a mutex lock is one of many rules implemented by the reordering filter (see Appendix A for a complete listing of these rules). Moreover, even for this single rule, with each of its expressions is associated another set of ordering rules that must be met. The rules appearing in Appendix A are written to reflect the logic of the current filtering code. Our future work is addressing the automatic generation of filtering code from formal rule specifications like the one shown above.

Implementation of on-line reordering. Figure 16 outlines the implementation of the event reordering filter. The event stream at the left arriving from the central monitor is only partly ordered with respect to

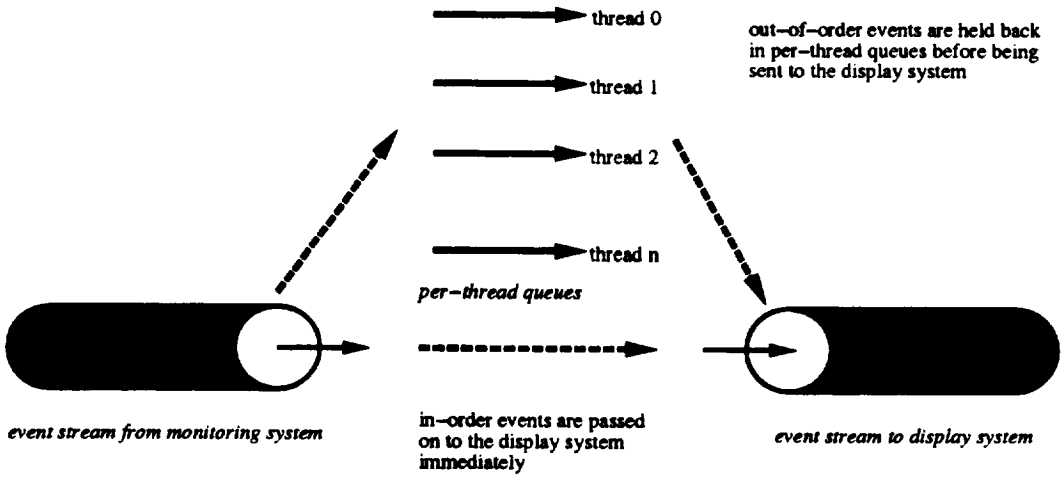


Figure 16: Architecture of the on-line trace reordering filter.

each thread_id. The event stream forwarded to the display system shown at the right hand side of the figure is ordered according to the specified ordering rules. To attain this ordering, the filter maintains an ordered queue for all events with the same thread_id encountered in the event stream, shown at the center of the figure. This queue only contains events that are not ready to be processed (that do not yet satisfy the rules), whereas other events are immediately forwarded to the display system. The ordering filter then continues to examine new events, checking the head of each active queue in every round to see if it is now possible to

place the event in the stream going to the display system. Note that these queues are not activated until a thread_init event (in the case of the program's initial thread), or a thread_fork event (all other threads) is processed for that thread_id. Processing of the queue is turned "off" again when a thread_exit is encountered. Straightforward generalizations of this code would entail dynamic queue creation and deletion at some cost in runtime performance.

Additional data structures in the ordering filter are assigned to mutex_ids and condition_ids, each of which is represented by a data structure that keeps track of the sequence numbers associated with this abstraction that have been processed thus far. An example of these data structures is shown in Figure 17, where threads are waiting on both condition variables. These data structures are dynamically allocated as the events are

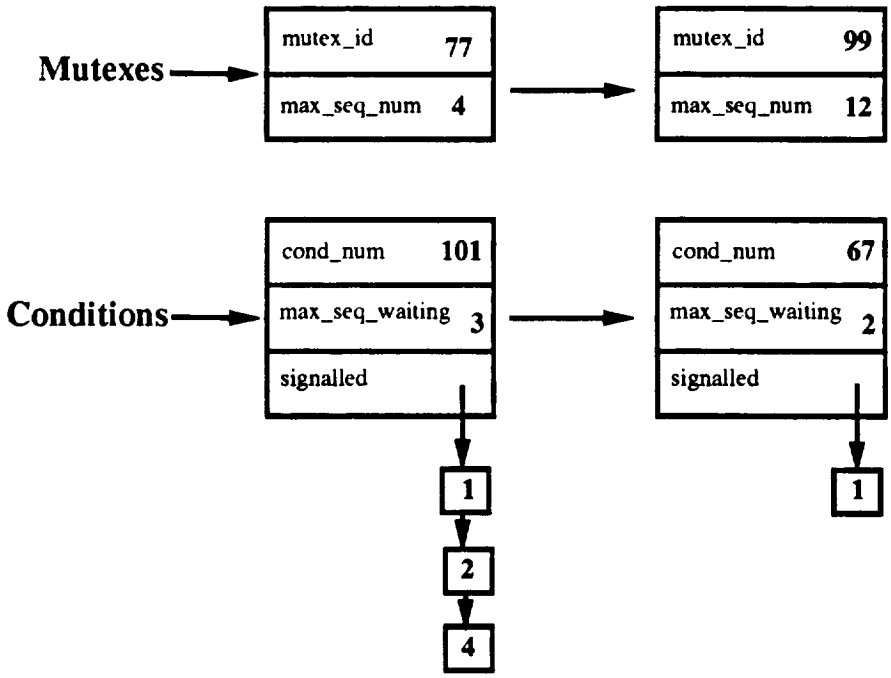


Figure 17: Detail.

observed in the stream. A mutex_init or mutex_alloc event causes data structure allocation for this mutex_id, and the sequence number for the mutex is initialized to 0. No event associated with this mutex_id may be processed until after the mutex_init or mutex_alloc events have occurred. In addition, mutex_end_lock and mutex_unlock events have a sequence number, and are required to be processed in sequence number order. Similar data structures exist for events concerning condition variables, again requiring that condition_init or condition_alloc events precede each condition's use and using sequence numbers initialized to 0. Specifically, a condition_end_wait event for sequence number *n* must be preceded by a condition_signal on sequence number *n* or a condition_broadcast on a range of sequence numbers containing *n*. In turn, a condition_signal on *n* must be preceded by a condition_begin_wait on *n*. A condition_broadcast on *n1* ... *n2* must be preceded by a condition_begin_wait on *n2*. The condition_begin_wait on sequence number *n* must be preceded by condition_begin_wait on sequence number *n-1*.

Evaluation. Meaningful performance numbers for the efficiency of the ordering filter are difficult to obtain. Because online monitoring *requires* the ordering filter to prevent display crashes, it is not possible to compare the appearance and execution of the display with the ordering filter versus without the ordering filter. Instead, we have attempted to evaluate the effects of the ordering filter on the appearance and speed of the displays under three offline conditions. The degree to which the events are misordered may also have an effect on the delay or "drag" that the reordering filter may impose on the display. Accordingly, we have produced trace

files with varying degrees of misordering and have developed a metric to describe the degree of misordering in an event stream or file. Traces were collected from four executions of the MD application. For each run the buffer size of the local monitor was varied in order to produce trace files with varying ratios of out-of-order events. The use of large buffers should produce more out-of-order events in the trace file (but less perturbation in the program), and smaller buffers should cause fewer out-of-order events (but more perturbation in the program).

As a measure of the misordering of the events, we calculated a *hold-back ratio*. Recall that the reordering code will temporarily hold back any event that violates causal ordering. If a misordered event is held back for multiple times, it will be counted for as many times. In our experiments, the hold-back ratios for the four trace files range from 0.60 (9,020 events held back in a file of 14,970 records) for the smallest size local buffer to 2.81 (40,903 events held back in a file of 14,552 records) for the larger local monitor buffer. The results clearly confirms the hypothesis that smaller event buffers cause more out-of-order events and larger buffers causes less out-of-order events.

For each trace file, we run a sorting program to produce another version of the trace file with all event records totally ordered by their timestamps. The thread life-time display code is then executed, observed, and timed for each of four trace files under the following three conditions: (1) the thread life-time view reading directly from the sorted file, (2) the reordering filter reading from the *sorted* trace file, passing event records to the thread life-time display through a socket, (3) and the reordering filter reading from the *original* trace file, passing event records to the thread life-time display through a socket.

Not surprisingly, the total running time of the display under the second condition exceeds the first in every case, ranging from a 3% increase to a 6% increase in display time. We attribute this delay primarily to CPU contention between the display and reordering code (they run on the same machine). However, the running times of the thread life-time display under the second and the third conditions are not significantly different. The degree of misordering does not significantly affect display execution time simply because the reordering code is much faster, from 10 to 30 times, than the display code itself, which relies on relatively more expensive X-windows call to show events to end users. In other words, the reordering filter is sufficiently fast to supply the display code with a steady stream of events.

6 Related Research

Interactive program steering. The concept of steering can be found in many interactive scientific visualization and animation applications which allow users to directly manipulate the objects to be visualized or animated [22, 21]. For example, in a wind tunnel simulation, users can interactively change shapes and boundaries of objects in the wind tunnel in order to see the effects on the air flow. Research has also addressed the provision of programming models and environments to support the interactive steering of scientific visualization. In [22], DYNA3D and AVS (Application Visualization System from AVS Inc.) are combined with customized interactive steering code to produce a time-accurate, unsteady finite-element simulation. The VASE system [21] offers tools that create and manage collections of steerable Fortran codes.

The idea of steering has also been used in parallel and distributed programming to dynamically change program states or execution environment for improving program performance or reliability [5, 35, 8]. Early work in this research area focusses on the dynamic tuning of parallel applications in order to adapt them to different execution environments [44, 45]. Recent experiments demonstrate that changes to specific program states or program components, such as locks [35] and problem partition boundaries [8], can significantly improve overall performance. Our research interests are to provide a mechanism for programmers easily take advantage of this dynamic tuning capability as well as supporting the on-line capture of program and performance information necessary for efficient program steering. While we can base some of our work on past research on the monitoring of parallel and distributed programs for correctness and/or performance debugging, on-line and dynamic monitoring are relatively new topics[40]. We refer the reader to [16] for a brief survey of current research on interactive steering and on-line monitoring.

Program monitoring. Past work in monitoring of parallel and distributed programs focuses on performance understanding and debugging. These performance monitoring systems (e.g. Miller's IPS[34] and IPS-2[33], Reed's Pablo[41]) provides programmers with execution information about their parallel codes, and leads their attention to those program components on which most execution time is spent. A variety of performance metrics, such as normalized processor time[1], execution time on the critical execution path [33], etc., are employed to describe the program's runtime performance. One limitation of these performance metrics is the difficulty to relate measured performance numbers to specific program details. Instead, most such research measures program execution times at the procedure level. However, program steering can depend on program information derived from specific program variables or statements, such as the analyses of the workloads of each domain when steering the MD application.

Some recent work has addressed application-specific program monitoring[47, 40]. In these systems, users can explicitly specify what variables or program states to monitor using specification languages [40, 23], some of which are based on the Entity-Relational model[47]. The W^3 search model described in [20] addresses this problem in a different fashion: performance data is collected using hooks either inserted by the compiler or by programmers; based on this data, potential performance bottlenecks are identified and resources causing these bottlenecks are found and then, corrected by application programmers.

Data and perturbation analysis. Monitoring information may be refined with trace data analysis techniques, such as the Critical Path Analysis and Phase Behavior Analysis described in [33], often in an off-line manner. More sophisticated analysis techniques may be used to reduce and correct perturbation to the measured program performance due to monitoring [30]. In addition, performance data may be subjected to various statistical filtering techniques prior to its display to users. All such techniques may be applied to Falcon's monitoring data, as well.

A number of systems have addressed the problem of "out-of-order" events, events that violate causality. These events violate the "happened-before" relationship described in [27] and [10]. Post-mortem display systems such as ParaGraph[17] and SIEVE[42] may sort the trace files by timestamp. Instant Replay[28], Makbilan[52], TraceViewer[19], the Animation Choreographer[25], and Xab[3] have all used a causality graph as on ordering tool for the post-mortem display of the execution of parallel programs. These methods are not effective for run-time performance display because they rely on fully available trace files that may be sorted prior to their display. In contrast, Xab[3], a tool for monitoring PVM programs, uses a timestamp adjustment approach. Each processor calculates time as the sum of its local clock and an "offset" value. This offset value is adjusted whenever a process a message with a a later timestamp than the receiving process's current time. However, it was found that lower-level changes to PVM were required to eliminate some "out-of-order" events. These changes are in part analogous to the Cthread-based support provided for on-line event reordering in the Falcon system.

On-line program steering utilizes current and past efforts concerning the efficient linkage of multiple supercomputer engines, which is being addressed by several Gigabit testbeds efforts in the United States. Systems like PVM[50] and Express offer software support for constructing large-scale distributed and parallel codes.

7 Conclusions and Future Work

The Falcon monitoring system enables programmers to capture and view precisely the program attributes of interest to them. Such monitoring may be performed on-line (during the program's execution) with low latency and more importantly, with dynamically controlled monitoring overheads. To attain such controls, Falcon's monitoring mechanisms themselves may be configured on-line to realize suitable tradeoffs in monitoring latency, overhead, and perturbation.

Falcon performs program monitoring on-line, namely, monitoring information is captured, analyzed, and stored or displayed during the target program's execution. This permits programmers to view their long-running parallel codes interactively, and then steer their execution into more appropriate data domains or

simply, to play 'what if' games with alternative parameter settings. Toward this end, Falcon also offers an integrated library for interactive program steering, as well as support for the on-line provision of monitoring information both to algorithms controlling program configuration and to graphical displays based on which users can perform program steering.

This paper demonstrates the utility and potentials of on-line program steering and monitoring with a large-scale parallel application program, a molecular dynamics simulation used by physicists to study the interfacial properties of lubricants. Additional measurements are based on an atmospheric modeling code used by scientists to study global atmospheric phenomena. When Falcon is used with these programs, it becomes apparent that programmers should be permitted to perform monitoring and steering at multiple levels of abstraction within a single parallel program, ranging from inspecting and steering individual program variables to steering at the threads or process level. The evaluation of Falcon's performance with these applications also demonstrates the importance of supporting multiple degrees of granularity (and accompanying overheads) with which monitoring may be performed. Detailed performance studies on a 64-node KSR shared memory multiprocessor show how changes in the methods of capturing program information can result in distinct differences in monitoring performance. In other publications, we also also demonstrate some limitations on applying Falcon's functionality, notably when using it for the steering of individual operating system abstractions used by parallel programs (e.g., mutex locks[35]). To support the monitoring and steering rates required for such fine grain program control, monitoring mechanisms must be customized. Our future work will address how such customized mechanisms may be used in conjunction with the remainder of the Falcon system. In addition, future work is addressing the monitoring of object-oriented, parallel programs, including the provision of default monitoring views and performance displays[38].

The MD and atmospheric modeling codes as well as the Falcon system are implemented and evaluated on a 64-node KSR shared memory supercomputer. However, the Falcon system is available on several shared memory platforms, including SGI and SUN Sparc parallel workstations. A version of Falcon currently being completed also works with PVM across networked execution platforms. Similar portability is attained for the graphical displays used with Falcon. Notably, the Polka animation library can be executed on any Unix platform on which Motif is available [49]. The Falcon system has been in routine use at the Georgia Institute of Technology by non-Computer Science end users. Its low-level mechanisms are available via the Internet since early Summer 1994. A version of Falcon offering on-line user interfaces for monitoring and monitor control will be released in 1995.

Current extensions of Falcon not only address additional platforms (e.g., an IBM SP machine now available at Georgia Tech and the monitoring of PVM programs running Cthreads, C, or Fortran programs), but also concern several essential additions to its functionality. First, currently, users can insert into their code simple tracing or sampling sensors, where sensor outputs are forwarded to and then analyzed by the local and central monitors. We are now generalizing the notion of sensors to permit programmers to specify higher level 'views' of monitoring data like those described in [24, 40, 47]. Such views will be implemented with library support resident in both local and central monitors. Second, we are developing notions of composite and extended sensors that can perform moderate amounts of data filtering and combining before tracing or sampling information is actually forwarded to local and central monitors. Such filtering is particularly important in networked environments, where strong constraints exist on the available bandwidths and latencies connecting application programs to local and central monitors.

An important component of our future research is the use of Falcon with very large-scale parallel programs, either using thousands of execution threads or exhibiting high rates of monitoring traffic. For these applications, it will be imperative that monitoring mechanisms are dynamically controllable and configurable. Namely, it must be possible for users to focus their monitoring on specific program components, to alter such monitoring dynamically, and to process monitoring data with dynamically enabled filtering or analysis algorithms. Moreover, such changes must be performed so that monitoring overheads are experienced primarily by the program components being inspected. Dynamic control of monitoring is also important for the efficient on-line steering of parallel programs of moderate size. Specifically, program steering requires that monitoring overheads are controlled continuously, so that end users or algorithms can perform steering actions in a timely fashion.

On-line control of monitoring performance will be performed in Falcon by affecting the rates of data collection by individual or sets of sensors, the degrees of parallelism used by local monitors, and the amounts of filtering done by local monitors prior to information transfers to central monitors. In addition, we are developing on-line control algorithms that permit Falcon's use with real-time applications.

Longer term research with Falcon addresses the integration of higher level support for program steering, including graphical steering interfaces, and the embedding of Falcon's functionality into a programming environment supporting the process of developing, tuning, and steering threads-based parallel programs, called LOOM. In addition, Falcon will be a basis for the development of distributed laboratories in which scientists can inspect, control, and interact on-line with virtual or physical instruments (typically represented by programs) spread across physically distributed machines. The specific example being constructed by our group is a laboratory for atmospheric modeling research, where multiple models use input data received from satellites, share and correlate their outputs, and generate inputs to on-line visualizations. Moreover, model outputs (e.g., data visualizations), on-line performance information, and model execution control may be performed by multiple scientists collaborating across physically distributed machines.

Acknowledgements. We thank Niru Mallavarupu for contributing to early implementations of Falcon components. Thomas Kindler is responsible for the parallel implementation of the atmospheric modeling code.

References

[1] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, Boston, May 1990.

[2] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 11–22, Madison, Wisconsin, May 1988.

[3] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6):88–95, June 1993.

[4] Adam Beguelin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, Pittsburgh, PA, June 1993.

[5] Thomas E. Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[6] Gretchen P. Brown, Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza. Program visualization: Graphical support for software development. *IEEE Computer*, 18(8):27–35, August 1985.

[7] Bernd Bruegge. A portable platform for distributed event environments. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 184–193, Santa Cruz, California, May 20-21 1991. ACM Press. ACM SIGPLAN NOTICES 26(12), December 1991.

[8] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3)*, pages 26–34, San Francisco, CA, August 1994. IEEE, IEEE Computer Society.

[9] Greg Eisenhauer and Karsten Schwan. Md - a flexible framework for high-speed parallel molecular dynamics. In Adrian Tentner, editor, *High Performance Computing - 1994*, pages 70–75, P.O. Box 17900, San Diego, CA 92177, April 1994. Society for Computer Simulation, Society for Computer Simulation. Proceedings of the 1994 SCS Simulation Multiconference.

[10] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, August 1991.

[11] Ahmed Gheith, Bodhi Mukherjee, Dilma Silva, and Karsten Schwan. Ktk: Kernel support for configurable objects and invocations. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 92–103, Pittsburgh, Pennsylvania, March 1994. The IEEE Computer Society Press.

[12] Ahmed Gheith and Karsten Schwan. Chaos-arc - kernel support for multi-weight objects, invocations, and atomicity in real-time applications. *ACM Transactions on Computer Systems*, 11(1):33–72, April 1993.

[13] Kaushik Ghosh, Kiran Panesar, Richard M. Fujimoto, and Karsten Schwan. PORTS: A parallel, optimistic, real-time simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, July 1994. College of Computing, Georgia Institute of Technology. to appear.

[14] Prabha Gopinath and Karsten Schwan. Chaos: Why one cannot have only an operating system for real-time applications. *SIGOPS Notices*, pages 106–125, July 1989. Also available as Philips Technical Note TN-89-006.

[15] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of FRONTIERS'95*, February 1995. To appear. Also available as Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology.

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

[17] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[18] David P. Helmbold, Charles E. McDowell, and Jian-Zhong Wang. Determining possible event orders by analyzing sequential traces. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):827–840, July 1993.

[19] David P. Helmbold, Charlie E. McDowell, and Jian-ZhongWang. Traceviewer: A graphical browser for trace analysis. Technical Report UCSC-CRL-90-59, Univ. of California at Santa Cruz, Santa Cruz, CA, October 1990.

[20] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 185–194, Tokyo, Japan, July 1993.

[21] David Jablonowski, John Bruner, Brian Bliss, and Robert Haber. VASE: The visualization and application steering environment. In *Proceedings of Supercomputing'93*, pages 560–569, November 1993.

[22] David Kerlick and Elisabeth Kirby. Towards interactive steering, visualization and animation of unsteady finite element simulations. In *Proceedings of Visualization'93*, 1993.

[23] Carol Kilpatrick, Karsten Schwan, and David Ogle. Using languages for describing capture, analysis, and display of performance information for parallel and distributed applications. In *International Conference on Computer Languages '90, New Orleans*, pages 180–189. IEEE, March 1990.

[24] Carol E. Kilpatrick and Karsten Schwan. ChaosMON - application-specific monitoring and display of performance information for parallel and distributed systems. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 57–67, Santa Cruz, California, May 20-21 1991. ACM Press. ACM SIGPLAN NOTICES 26(12), December 1991.

[25] Eileen Kraemer and John T. Stasko. Toward flexible control of the temporal mapping from concurrent program events to animations. In *Proceedings Eighth International Parallel Processing Symposium*, pages 902–908, 1994.

[26] Jeff Kramer and Jeff Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424-436, April 1985.

[27] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the Association for Computing Machinery*, 21(7):558-565, July 1978.

[28] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471-481, April 1987.

[29] Allen D. Malony, David H. Hammerslag, and David J. Jablonowski. Traceview: A trace visualization. *IEEE Software*, pages 19-28, September 1991.

[30] 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.

[31] Keith Marzullo and Mark Wood. Making real-time reactive systems reliable. *ACM Operating Systems Review*, 25(1):45-48, January 1991.

[32] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191-201. SIGOPS, Assoc. Comput. Mach., December 1989.

[33] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206-217, April 1990.

[34] Barton P. Miller and Cui-Qing Yang. IPS: An interactive and automatic performance measurement tool for parallel and distributed programs. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 482-489, Berlin, West Germany, September 1987. IEEE.

[35] Bodhi Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *Proceedings of the International Conference on Parallel Processing, Michigan*, pages 205-208. IEEE, Aug. 1993.

[36] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101-112, June 1991.

[37] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of Second International Symposium on High Performance Distributed Computing (HPDC-2)*, pages 59-66, July 1993. Also TR# GIT-CC-93/17.

[38] Bodhisattwa Mukherjee, Dilma Silva, Karsten Schwan, and Ahmed Gheith. Ktk: kernel support for configurable objects and invocations. *Distributed Systems Engineering Journal*. Expected to be out early 95.

[39] Brad A. Myers. INCENSE: A system for displaying data structures. *Computer Graphics*, 17(3):113, July 1983.

[40] D.M. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762-778, July 1993.

[41] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801, November 1992.

[42] Sekhar R. Sarukkai and Dennis Gannon. Parallel program visualization using SIEVE.1. In *International Conference on Supercomputing*. ACM, July 1992.

[43] Karsten Schwan, Prabha Gopinath, and Win Bo. CHAOS - kernel support for objects in the real-time domain. *IEEE Transactions on Computers*, C-36(8):904-916, July 1987.

[44] Karsten Schwan and Anita K. Jones. Flexible software development for multiple computer systems. *IEEE Transactions on Software Engineering*, SE-12(3):385-401, March 1986.

[45] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and Dave Ogle. A system for parallel programming. In *9th International Conference on Software Engineering, Monterey, CA*, pages 270-282. IEEE, ACM, March 1987. Awarded best paper.

[46] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, and David Ogle. A language and system for the construction and timing of parallel programs. *IEEE Transactions on Software Engineering*, 14(4):455-471, April 1988.

[47] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157-196, May 1988.

[48] John T. Stasko. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27-39, September 1990.

[49] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258-264, June 1993.

[50] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315-339, 1990.

[51] T. K. Xia, Jian Ouyang, M. W. Ribarsky, and Uzi Landman. Interfacial alkane films. *Physical Review Letters*, 69(13):1967-1970, 28 September 1992.

[52] Dror Zernik and Larry Rudolph. Animating work and time for debugging parallel programs - foundation and experience. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 46-56, Santa Cruz, California, May 20-21 1991. ACM Press. ACM SIGPLAN NOTICES 26(12), December 1991.

Appendix A: A Complete List of Cthreads Events Reordering Rules

Here is the terminology used in describing the ordering rules:

```
<- = "is allowable if preceded by"
t  = thread number
c  = condition number
m  = mutex number
n  = sequences number
pt = parent thread
ct = child thread
jt = join_to thread number
bn = beginning sequence number
en = ending sequence number
p  = processor number
x  = don't care value
```

The ordering rules for all events from default monitoring of Cthreads programs are listed below. A brief explanation for each rule is provided. For each mutex number m and condition number c, it is initially set to 0.

```
thread_init t          <- ();
```
This is the initial event for thread t. All prior events pertaining to this thread are ignored. An internal buffer is created for this thread number, and it is turned "on".

```
thread_fork pt ct      <- ( (thread_init pt) && ((pt == 0) && (thread_fork t pt)) );
```
The parent thread must be "on" for this event to be processed. An internal buffer is created for the child thread and it is turned "on". It is required that the parent thread is initialized before this event.

```
thread_exit t          <- ( (thread_init t) && (thread_fork pt t) );
```
The internal buffer is de-allocated and the thread is turned "off". Any succeeding events recorded by this thread are ignored.

```
thread_begin_join t jt <- (thread_init t);
```

```
thread_end_join t jt   <- ( (thread_init t) && (thread_exit jt) );
```
The `thread_exit jt` event for the join_to thread jt must have occurred before this event.

```
thread_detach t        <- (thread_init t);

thread_yield t         <- (thread_init t);

thread_set_name t      <- (thread_init t);

mutex_init t m         <- ( (thread_init t)
                            && !( (mutex_init x m) || (mutex_alloc x m) ) );
```
No prior `mutex_init x m` or `mutex_alloc x m` event may have occurred.

```
mutex_alloc t m        <- ( (thread_init t)
                            && !( (mutex_init m) || (mutex_alloc m) ) );
```
No prior `mutex_init x m` or `mutex_alloc x m` event may have occurred.

```
mutex_begin_lock t m n <- ( (thread_init t)
                            && ( (mutex_init x m) || (mutex_alloc x m) ) );
```
A `mutex_init x m` or `mutex_alloc x m` must precede this event.

```
mutex_end_lock t m n   <- ( (thread_init t)
                           && ( (mutex_init x m) || (mutex_alloc x m) )
                           && (mutex_end_lock x m n-1) )
```

A **mutex_init x m** or **mutex_alloc m** must precede this event. The **mutex_end_lock m, n-1** must have occurred. The initial value of this term is **mutex_end_lock x m 0**, which is always true.

```
mutex_unlock t m n     <- ( (thread_init t)
                           && ( (mutex_init x m) || (mutex_alloc x m) )
                           && !(mutex_end_lock x m n+1) );
```

A **mutex_init x m** or **mutex_alloc x m** must precede this event. The **mutex_end_lock m, n+1** may not have occurred.

```
mutex_free  t m        <-  (thread_init t);

mutex_clear t m        <-  (thread_init t);

mutex_set_name t m     <-  (thread_init t);

condition_alloc t c    <-  ( (thread_init t)
                           && !(condition_init x c || condition_alloc x c) );
```

No prior **condition_init x c** or **condition_alloc x c** event may have occurred before this one.

```
condition_init t c     <- ( (thread_init t)
                           && !( (condition_init c) || (condition_alloc c) ) );
```

No prior **condition_init x c** or **condition_alloc x c** event may have occurred before this one.

```
condition_free t c    <- (thread_init t);

condition_clear t c   <- (thread_init t);

condition_begin_wait t c n m <- ( (thread_init t)
                                  && ((condition_alloc x c) || (condition_init x c))
                                  && (condition_begin_wait t c n-1 m) );
```

A **condition_init x c** or **condition_alloc x c** event must have occurred, and so do the preceding **condition_begin_wait t c n-1 m** event.

Progress: a Toolkit for Interactive Program Steering[1]

Jeffrey Vetter[2]

Karsten Schwan

Technical Report GIT-CC-95-16, August 1995

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
vetter@cc.gatech.edu, 404/853-9389, Fax: 404/853-9378
schwan@cc.gatech.edu, 404/894-2589

Abstract

Interactive program steering permits researchers to monitor and guide their applications during runtime. Interactive steering can help make end users more effective in addressing the scientific or engineering questions being solved with these programs, and it may be used to improve the performance of complex parallel and distributed codes. **Progress** is a toolkit for developing steerable applications. Users instrument their applications with library calls and then steer parallel applications with Progress' runtime system. Progress provides *steerable objects* which encapsulate program abstractions for monitoring and steering during program execution. Once created, steering objects are known to and manipulated by Progress' two components: (1) a server executing in the same memory space as the target program and capable of inspecting and manipulating program state, and (2) a potentially remote client providing command and graphical interfaces. Developers instrument their applications with the Progress toolkit library to create and maintain these steering objects. The server maintains information about the steering objects and performs steering actions on the application. This toolkit provides sensors, probes, actuators, function hooks, complex actions, and synchronization points. Progress' server is built on a Mach-compatible Cthreads library; it is a general toolkit for use with a variety of multithreaded, C programs executing on multiprocessors. Progress has been applied to several large-scale parallel application programs, including a molecular dynamics code and an N bodies simulation. It is currently being used with a complex global atmospheric modeling code.

Keywords: steering, dynamic, visualization, monitoring, environments.

1. Introduction

If high performance computing continues to remain 'non-interactive' [McCormick88], end-users and program developers alike will not capitalize on new techniques for interactive data visualization and program animation [Jablonowski93,Stasko90], remote and collaborative work, interactive debugging and monitoring [Gu95,Ogle93], and on-line program adaptation [Mukherjee93]. For example, when run in 'batch mode', erroneous or uninteresting results produced by large-scale scientific or engineering simulations are not apparent until after the computations complete, sometimes days or weeks after program initiation. *Program*

---

[1] This report was originally submitted to the 24th International Conference on Parallel Processing 1995 (ICPP 95). It was accepted and published as a shorter version.

[2] Vetter is financially supported by a NASA Graduate Student Researchers Program (GSRP) grant.

steering provides end-users with the capability to monitor and guide their applications during runtime. The goal of our research in program steering is the exploration of the opportunities, limitations, and challenges inherent in the development and use of interactive high performance programs, on parallel and distributed execution platforms. More specifically, the **Progress** toolkit presented and evaluated in this paper provides facilities through which existing high performance multiprocessor programs are extended for increased interactivity [McCormick88, Jablanowski93, Eisenhauer94]. Once these extensions have been performed on a target application, a user can 'connect' to her high performance program, extract application-specific data regarding the program's execution state, perform steering actions, if desired, and then 'disconnect' from the program. The runtime overheads of these tasks depend only on the desired degrees of interactivity, ranging from minimally perturbing program execution in the absence of steering to being highly intrusive when Progress calls explicitly stop and restart programs.

High performance of Progress-instrumented programs is attained by use of *selective* and *application-dependent* runtime monitoring and program steering. Namely, programmers encapsulate program abstractions in *steering objects* that explicitly identify those program components as steerable. Steering objects are created, catalogued, and manipulated at execution time by Progress' runtime system. The runtime system consists of two distinct components: a server and a client. The server is an additional thread associated with the multithreaded multiprocessor application program. This thread responds to runtime application events, performs steering actions in response application events, executes commands from the client, and communicates with the client to provide consistent object information. The client provides command, recording, scripting, and user interface facilities, and it interfaces to existing program animation or data visualization facilities offered by the Falcon monitoring system [Eisenhauer94FAL] and by data visualizers [Ribarsky94] employed by Progress users.

This paper explores the requirements and opportunities of on-line program steering. Its contributions focus on the usage of steering during the execution of high performance parallel programs, similar to ongoing research on dynamic program monitoring [Eisenhauer94FAL,Hollingsworth93]. Stated briefly, the hypothesis we explore in this research is "program steering should be dynamically initiated, enabled and disabled, used selectively, and changed in scope and functionality." The Progress toolkit provides facilities with which we test this hypothesis.

1.1 Interactive Program Steering

Program steering is defined as the runtime manipulation of an application program and its execution environment. The goal of this manipulation is either performance improvement or increased functionality, such as focusing program execution toward more interesting data domains or improving resource usage

through manual load balancing. An essential characteristic distinguishing program steering from work on interactive data visualization and navigation is the latter's lack of feedback from data manipulation to the program producing the data. *Program steering* affects the programs producing data, whereas data navigation is usually performed after program execution has completed. Similarly, a distinction of program steering from research in on-line program adaptation [Mukherjee93, Bihari91] is that steering enables both algorithms and users "in the loop" when application programs take actions in response to changes in program state or output. Our hope is to utilize this interactivity to improve productivity in the specific scientific and engineering processes being undertaken. For example, in the interactive atmospheric modeling code being developed by our group, multiple researchers will be able to inspect and manipulate data objects in shared 3D data visualizations. One such manipulation concerns changes in the concentration of atmospheric constituents, so that researchers can play "what if" games concerning the model's global effects. Another manipulation allows alterations to the vertical constituent movement in the simulation's transport model. This movement remains a poorly understood phenomenon, and model outcomes are significantly affected or even invalidated by the settings of simulation parameters controlling movement and the computational methods used for describing it. Steering improves the application by providing constant, selective feedback to the end user. The user can terminate the program, stall the program for inspection, schedule steering actions for execution, and effect a host of other modifications.

A concise breakdown of interactive program steering has three components: data collection, data interpretation, and steering. Data collection has a rich history of research in monitoring and data analysis. Data interpretation also has an extensive past with visualization and other techniques such as filtering, clustering, and queries [Bates86,Snodgrass88]. Steering, however, is rather immature. We review several systems in the Section 4 (Related Work); a complete review of systems related to this topic is available in [Gu94].

To summarize, *program steering* permits users to control program execution in terms of program abstractions familiar to them. Such abstractions may be encapsulations of computations producing program output like 'constituent concentration', or they may address program resource usage like 'average thread wait times' (typically of interest to program developers). In either case and in contrast to traditional research on program monitoring and debugging, steering must be based on abstractions specific to each application program, using runtime support that minimally perturbs program performance. This implies that steering cannot rely on automated methods for code inspection that may require disabling compiler optimizations, and it cannot require default instrumentation at any level of program abstraction. Furthermore, since steering is on-line, Progress cannot utilize existing post mortem methods for trace analysis as used in program debugging

and in off-line monitoring[LeBlanc87, Mal91]. This prompts us to adopt the Falcon system's [weiming] event- and view-based models of program monitoring first advocated by Bates [Bates86] and Snodgrass for the collection of program state in Progress. Similarly, the Progress toolkit assumes that users employ visualization and animation toolkits [Ribarsky94,Kra93] for constructing the application-specific displays of program behavior required for steering. The Progress prototype targets the actual tools that software developers need to create generally 'steerable' applications and the additional functionality required from the operating system, and the monitoring system.

Computational fluid dynamics is a interesting target application area for steering. Biomedical researchers at GT are developing a parallel version of a spectral element fluid dynamics simulation program. CFD is an attractive area for program steering because most computations meet the time requirements with average simulations lasting days or weeks. Intermediate analysis of results could both pinpoint errors in initial conditions and allow modifications to error convergence limits. CFD data is usually visualized for final analysis but with on-line, interactive systems, various data might be visualized and modified throughout the simulation.

MD [Eisenhauer94] is a molecular dynamics simulation that explores the statistical mechanics of complex liquids. The dominant computational requirement is the calculation of long-range forces between particles. MD is an interesting steering target because load balancing is difficult and standard heuristics are not effective in managing the load. With the aid of a steering system, however, a human user may manage resources manually resulting in a low cost, effective simulation.

Weather and climate modeling presents a fascinating area for experimenting with Progress due to the enormous data sets garnered from satellites and other remote sensing stations. These data sets sometimes require repeated processing. End users can easily experiment with alternative model parameters, conveniently evaluate and re-evaluate the behavior of specific processes being modeled, and affect or change model execution to improve performance. Also, selecting certain data sets for evaluation or 'focusing' the calculation is useful with large data sets.

1.2 Paper Outline

This paper describes the design objectives for Progress, its architecture and implementation, and the evaluation of Progress with a real-world application. Section 2 describes the construction of the Progress steering system. Section 3 evaluates Progress with an application. Section 4 reviews related work. Section 5 concludes this paper with a review of research goals and future research.

2. The Progress Steering Toolkit

Progress is an acronym for *Pro*gram and *Re*source *S*teering *S*ystem. The goal of ProgReSS is "to provide applications developers with a complete set of tools and facilities for creating steerable parallel applications." The Progress steering toolkit has several concepts and stages. First, the application developer must understand the steering object model to properly instrument the target application. The Progress object model encapsulates components of the application that the user might wish to steer or monitor. Second, the application developer actually instruments the application with calls to the Progress library to create and maintain these steering objects throughout the application's lifetime. The developer understands the application's operation sufficiently to allow external changes to its state while the application is executing. And finally, the end-user controls the application with the steering runtime system. The steering system is composed of a steering server and steering client. The steering server executes as a separate thread in the same memory space with the application. The client presents a graphical user interface the end-user for control of the remote steering server.

Progress is not intended as an advanced remote visualization system, or a parallel debugger. Remote visualization systems focus on the visualization of application output or performance visualizations. While interactive program steering will combine visualization of data and control of the application, its goal is not primarily visualization of data. Progress admittedly does not attempt to duplicate the functionality of advanced parallel debuggers. Parallel debuggers are far more general and flexible in their exploration of program information.

2.1 Design Requirements

Our design requirements for Progress include three fundamental notions: basic steering should be possible with many dynamic applications, the steering library should provide functionality for steering beyond updates to simple variables, and the user interface should allow end-users the capability to explore their executing programs. While designing Progress, it became obvious that certain types of programs are more 'steerable' than others. There is a distinct range of options facilitated by steering and not all these options are feasible with every program.

The library should support complex steering operations. Steering should provide tools for changing an executing application that protect application integrity and allow changes to application state that do not degrade application performance. As opposed to changing one integer variable in a SPMD application, steering should allow a variety of changes to all types of data within the application. These tools should also provide a level of abstraction that is effective for an end-user and useful for steering. Steering, as opposed to

debugging, should not allow access to every program variable and the ability to arbitrarily interrupt program flow. Steering concentrates on observing and changing parameters of a correct application which is contrary to debugging in that debugging focuses attempts to locate faults in an application [McDowell89]. Types should not be limited to standard data types provided by the language. User-defined types including structures and arrays must be accessible to allow complex steering operations on arrays and structures of data.

Because the interface to the steering system might be predominately used by non-computer scientists, the interface to the steering system must follow typical guidelines for user interface design. The end user, not the application developer, controls the steering system through this steering user interface. This interface must provide various types of steering actions consistently on a continuous range of steerable programs. One steering interface for all steering. The steering interface to a CFD application and an atmospheric modeling application are the same; however, the steering objects accessible from this interface are application specific. Operations on steering objects are consistent between different applications. From the steering interface, the user should be able to selectively monitor and modify steering objects within the application. Also, the interface must provide ways of interpreting and analyzing large amounts of data produced by the steering system [Kra93].

2.2 Steering Object Model

An object model allows the steering system (and the user) to concentrate on only those components of the application that the developer explicitly declared prior to execution. This abstraction is necessary to limit amount of information that the steering system must contend with as well as the end-user. This object model, distinct from other notions of object-orientation, provides a convenient mechanism for naming and manipulating program components. Progress does not support inheritance and other characteristics of object oriented languages; Progress' object model encapsulates and identifies components of the program for monitoring and steering. This concept is identical to LeBlanc's instant replay mechanism [LeBlanc87], where the user identifies objects within the code. Replays guarantee to simulate interactions between these objects, but not between *every* program component.

An object becomes known to the steering server (and the user) when the object is registered. An object is a convenient way of describing data within the program that is of interest. A registered object can be accessed in three ways throughout the Progress system. Information about each object is stored in the application as a handle, in the server's object registry, and in the client's object registry. This object registration delimits the data of interest within the application, classifies the data type, and assigns a unique ID number to that object. Registration also enters a record into the steering server's registry of steering objects. The steering server uses this registry record to control the steering object within the application. The application receives a

handle to the object so that it can perform operations on the object. The application may want to unregister an object, synchronize itself with the steering server and the end-user, or check an object to determine if any synchronous modifications are scheduled. After registration, the steering server can control steering objects. The steering server controls the object by monitoring the object or steering (modifying) the object. These operations are described fully in Section 2.3. At application termination or unregistration for a steering object, the server's registry record is removed and future references to this object are either ignored or produce an error.

2.3 Steering Object Operations

Once a steering object is defined within the application, the steering server manipulates the object with several different operations. These operations allow a variety of synchronous and asynchronous accesses to the application. Synchronous and asynchronous access are important to differentiate because, in some cases, asynchronous access to steering objects may produce inconsistent views of the object or actually invalidate the application results! To perform synchronous operations, the developer must instrument the application. The operations available on all steering objects are probe read, probe write, sense, and actuate. Operations available on specialized steering objects are synch points, function execution, and scripts.

Probe reads and writes to the steering object. A probe is the simplest of the steering operations because once an object is registered the steering server can just read the object's memory. After registration the developer does not have to introduce any additional code into the application. The steering server performs probes without respect to the applications control flow; therefore, probes are considered asynchronous. Probes are particularly useful for inspecting stalled programs or updating non-critical variables in the application.

Sense captures an object's state within the application and forwards it to the monitoring system for analysis. When the application encounters a 'Sense' call in its thread of execution, it copies the object state to an event record, and places the record into a buffer destined for the monitoring system. Because sense is executed within the control flow of the application, sense is synchronous. The steering server can enable or disable 'Sense' for each particular steering object. Both probes and sensors are investigated in [Ogle93] as part of an application specific monitoring system. In Progress, these mechanisms monitor steering objects instead of language specific application components.

Actuate performs a modification on the steering object. Actuate is a new steering mechanism that is analogous to the sense mechanism because it is synchronous with the application's control flow and it performs an opposite action on the application. When the application thread executes this actuate call, actuate checks its buffer to determine if any changes are intended for its steering object. These changes are 'programmed' for a steering object by the steering server. when the application executes an actuate call,

actuate checks to decide if any changes to it's steering object are necessary. Actuate is synchronous with the control of the application because the modification is not performed on the application until the application encounters an actuate instrumentation point. If no modifications await the object, the actuate call exits. If a change is waiting in the buffer, then the actuate call modifies the object as specified.

	Synchronous	**Asynchronous**
Monitor	Sense	Probe Read
Steer	Actuate	Probe Write

Table 1 - Summary of Primary Operations for Steering Objects

For enhanced functionality, Progress provides a set of specialized objects. These objects have particular functionality that helps a user to steer an application. These operations are synch points, function execution, and scripts. **Synch Points** stall an application so that the end user can explore the state of the application and the steering client views a consistent snapshot of the application state. Synch Points are activated by the user so that the application stalls. The application threads spin waiting for a release command from the server. Because the threads are essentially halted, the user can interactively inspect and modify program state without fear of corrupting an executing application. The user can also be certain that the information stored in the object registries, on both the client and server, are consistent with the actual application state.

Functions allow the developer to register application functions with the steering server so that the server can execute application functions. Usually, these functions are complex operations that either gather information from that application or update some application state. Functions are regular C functions registered as steering objects with the server. Either the application or the server thread might execute these registered functions. The application could just call the function as it would any other function as could the server thread; however, for this function to be known to the registry and the client, the application must register the function. Registration publishes the name of the function to the steering system including the end user. When the calling thread is the server thread, functions can have one parameter and their return value is ignored. Functions are useful for accomplishing tasks other than just reading or writing a data value. These functions can alter application specific data structures within the executing application. For example, a user might stall an application and execute a registered function to remove an element from a queue. The function knows how to update the data structure, while updating all the variables for the queue data structure per se is tedious and error prone. In certain cases, these functions must synchronize with the application to prevent

corruption of the application. Alternatively, a function might calculate a global value and store it in a steering object where it can be accessed asynchronously with a probe.

Scripts provide users with the functionality of combining other steering operations in a language form for repeated execution. Scripts are different from functions because they are executed either on the client or in the steering server; the values they use and actions they execute are derived only from the registry - scripts cannot access program data other than registered objects.

2.4 Runtime System

The runtime system is composed of a server and a client. This separation is convenient because the server runs on the same machine as the application, and the client, which presents a graphical user interface to the user, usually executes remotely on a separate machine. With the server on the same machine as the application, the server can access and control the application with low latency [Gu95]. Because the client is remote, latencies introduce network delays. These network delays are closer to human interactive response times; however, they are burdensome for the server.

This architecture is also advantageous because the server must exist through the application's lifetime while the client does not. The client is transitory and can connect to the server many times throughout the server's (application's) existence. Because the client may use visualizations for data interpretation, the end-user may choose to run the client on a high-performance graphics system.

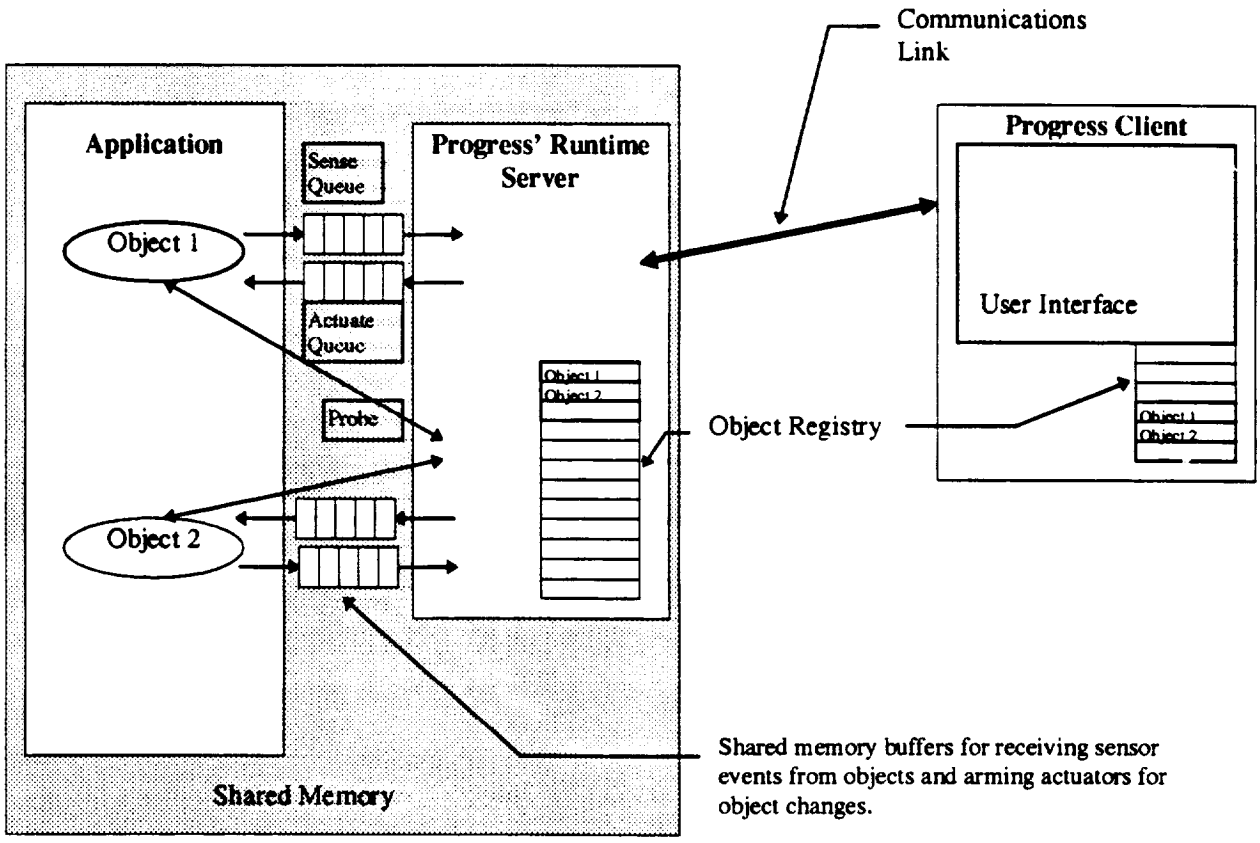


Figure 1 - Progress Architecture

2.4.1 Steering Server

The steering server is a separate thread that executes in the same memory space as the application. The server architecture (Figure 1) allows the application to execute normally. In fact, the application can execute entirely without interference from the server. The server thread has three basic tasks: interact with the steering client, gather monitoring output from the application, and steer the application via the steering objects. This server continuously maintains a registry of steering objects within the executing application. Each object that the application registers is stored by name in the registry. Registry records contain enough information to access the application's object at anytime and interpret information generated from the object. The registry is also used to route information from the application to the client and visa-versa.

The server communicates with the client through UNIX sockets. When the server starts, it allocates a socket, publishes the socket number, and listens to that socket for pending client connections. Once the client connection contacts this server socket, the server creates a new stream connection for that client-server pair. Thereafter, the message protocol allows bi-directional messages between the client and server. The server polls the client's socket for incoming messages. When a message is received, the server decodes the message

and executes any action required. The server assumes that incoming messages from the client will be relatively infrequent compared to the other server tasks. The server is required to execute actions requested by the client. Occasionally, those actions require extensive time.

The server monitors the application by receiving sensor events and probing objects. The server does simple analysis to filter irrelevant events out of the stream. The server controls monitoring so that the end-user can selectively observe different steering objects throughout the application's lifetime. Based on the commands from the end user, the server enables and disables sensors and probes objects to gather information for the end user. This monitoring has two levels of application specific filtering. First, the application developer registers only those objects that may possibly be of interest to an end user. This level of filtering discards all temporary or uninteresting data values in the application. The developer selects data that the user may want to observe or control. Second, the end user elects which steering objects to monitor through the client user interface. The end user only receives information about objects that are registered with the server *and* that he has selected through the client interactively. The user cannot arbitrarily access components of the application that are not registered with the server.

The server steers the application through steering objects using the steering operations detailed in Section 2.3. Several actions can trigger a steering operation. The user can manually request a steering operation on a steering object, or the server can execute a steering operation in response to an event received from the application.

2.4.2 Steering Client

The steering client is a remote application to control the resident steering server. The client is a single threaded Motif application that communicates with the steering server through a customized message protocol. The graphical user interface for the steering client is presented in Figure 2. The client has three main tasks: interact with the user, communicate with the steering server, keep relatively consistent state information about all the steering objects. The client receives all of its information from the steering server; it receives periodic updates to its registry from the server, based on the frequency of activity at the server. The client has a mirror registry of the server's registry. The interface allows the user to selectively monitor and modify steering objects. Each of the steering actions available through the steering server can be initiated from the steering client albeit the latency is higher. A user can enable monitoring of a steering object through sensors, probe an object for its current state, or modify the object through a probe write or an actuator.

An additional task of the steering client is easy manipulation of possibly large datasets resulting from monitoring. If a user selects monitoring for an object at a high frequency, simple textual display of the results

fails [Kra93]. However, because the structure of the steering object is known to the client, the client can map the objects updates onto a simple graph or other visualization.

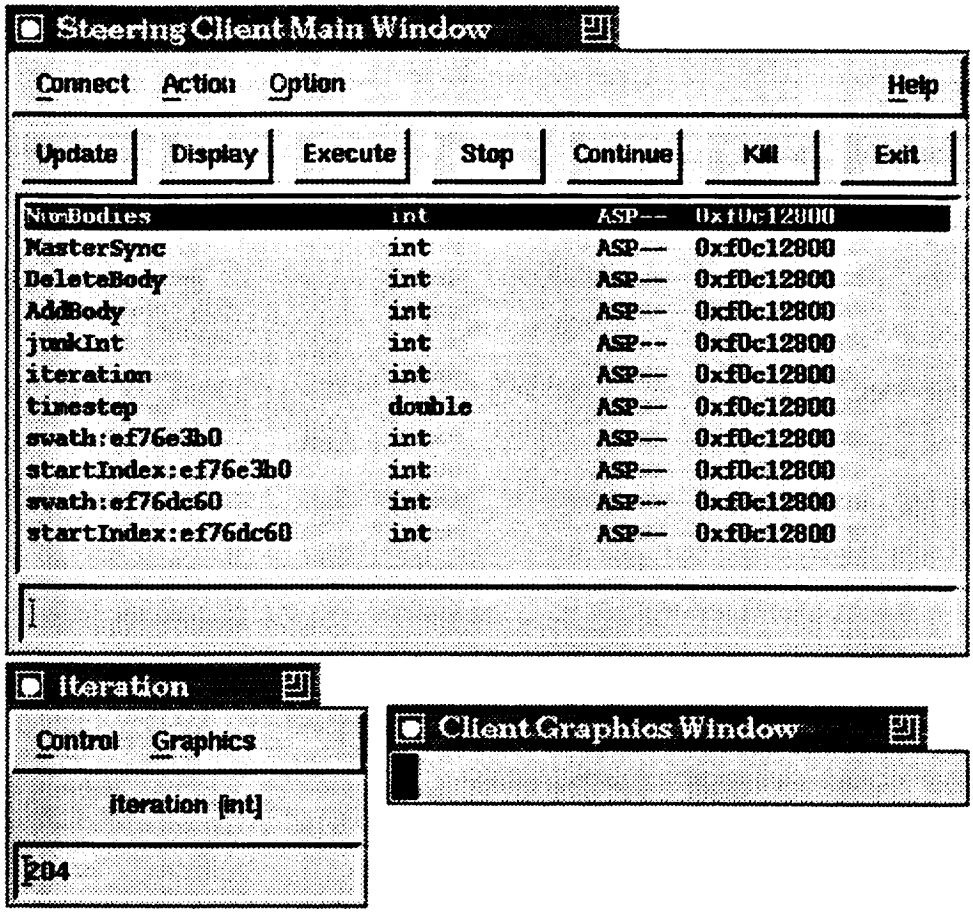


Figure 2 - Graphical User Interface for Steering Client

The client's user interface is build with Motif to present steering objects in a consistent and general manner. This same interface is used with all steering applications whether the application is a atmospheric modeling simulation or a CFD application. Steering objects are presented to the user in a consistent fashion for investigation and modification.

Figure 2 shows the main screen of the Progress client. The main menubar is at top. Next are two boxes: one list box and one text box. The list box displays all registered steering objects in the application. All objects are listed here whether they are regular steering objects or more specialized steering objects. The text box is an event log. As objects produce events, they are logged here in the event log. This log information can be captured in an ASCII file for post-mortum review. The command line allows the user to enter complex expressions and other statements not easily entered with the graphical user interface. Below the command line, the message log displays messages specific to the operation of the client, but not the server. A popup window (not shown), the object information box, provides object specific information about the selected object.

This box displays the object ID, object type, object name, etc. and any attributes particular to the object type. Menu sensitivities change as different objects are selected in the object list box. For example, the 'enable' and 'disable' menu options are disabled when a function is highlighted in the object list box because these menu options do not work with functions.

2.5 Instrumenting an application

Instrumenting the application builds entirely on the steering object model (Section 2.2). Instrumentation creates the steering server, and registers and manages the steering objects. The application registers these steering objects with the steering server.

```
1   main()
2   {
3       int i;
4       pSteeringServer steeringServer;
5       pSObject sobj;
6               .
7               .
8       CreateSteeringServer( &steeringServer );
9               .
10              .
11      RegisterSteeringObject( steeringServer, &sobj, &i, A_INTEGER );
12              .
13      i = 1;
14              .
15      while( ++i < 10 )
16      {
17              Sense( sobj );
18      }
19              .
20      while( i != 0 )
21      {
22              Actuate( sobj );
23      }
24              .
25      printf( "%d", i );
26              .
27      DestroySteeringServer( steeringServer );
28  }
```

Figure 3 - Steering Server Calls

Figure 3 outlines the necessary calls to the Progress toolkit for a simple application. CreateSteeringServer creates data structures, initializes the state, and spawns that steering server for this application. The steering server's internal structure is described in Section 2.4.1. Once the steering server is initialized, the application can register steering objects. RegisterSteeringObject creates an record in the application containing information about the object, and it also enters a record in the steering server's object registry. The registration requires a pointer to the application state of interest (e.g., integer i on line 11) and the data type of that application state (e.g., A_INTEGER on line 11). After the registration, the server can access the object with steering operations and the steering client is notified of the registration, so that the user can interrogate the new object and possibly steer it.

The 'Sense' call uses the information stored in the steering object record to determine its operation. First, the call checks whether sensing is enabled for this object. If sensing is disable, the call returns without further action. Otherwise, it continues. The call, using the information in the steering object record, copies the object state to an event record and places it in a shared memory buffer. The buffer was previously created by the steering server and the steering object record contains the name of this buffer. In this example (Figure 3), the event record would contain an object identifier, a timestamp, a thread id, and the value of the steering object (integer i=1). Once the event record is inserted into the buffer, the sense call returns to the application. Notice that the sense call only requires a steering object handle. The Sense call gathers all the necessary information about the object from the steering object handle. This design is convenient for several reasons. First, if the frequency of state updates for a steering object is too low, then the developer can insert more 'Sense' calls using the same steering object handle. Second, if multiple objects are to be monitored, then a sense call for each object must be inserted with a handle to each object. Third, if a user wants to disable monitoring for a particular object, then they disable monitoring for that object, not each sense call. Because the event records generated by each sense call update the state of an object at the server and eventually the client, it is advantageous to enable and disable monitoring on a per object basis rather than a per sensor basis. The object model allows the user to focus on objects of interest within the program and only those objects. The user does not have to enable and disable multiple sensor insertion points for each object they wish to observe; they merely enable or disable all sensors for each object.

The 'Actuate' call is a steering operation that modifies the steering object. Actuate uses information stored in the steering object record to determine if any modifications to the object state are required. If the actuator is not armed, then the call just returns to the application. If the object is to be updated, then the actuate call retrieves a record from a shared memory queue. This record contains all the necessary information to update the steering object. When the actuate call is executed, it checks this shared memory queue for entries. If the queue contains an entry, then the actuator retrieves the record and updates the steering object state appropriately. The actuator is armed by the steering server. The server arms an actuators on a per object basis. In Figure 3, the actuate call forces the value of i to equal 0, otherwise the program loops forever. The external steering server places an record into the actuate queue for 'sobj.' Then, when the application executes the actuate call on sobj, the application retrieves the record and updates the object appropriately. In our example, the actuate call is constantly checking the sobj actuate queue to determine when a change must occur. Because the steering server places only one record in this object's actuate queue, the majority of calls to actuate just return without modifying the sobj (or the integer i).

However, on the last call to actuate, the server has placed a record in the actuate queue to change the object to 0. On this last call, the object is updated to 0 and the loop terminates on its next loop test (i != 0).

Actuate operates on a per object basis. Many actuate calls can service one steering object. Placement of the actuate calls throughout the source code identifies time frames when the steering object can be updated without corrupting the application during execution. A higher frequency of these actuate calls allows steering objects to respond more rapidly to user steering requests. As with sense, each individual actuate call cannot be disabled.

Actuators also have pre- and post-conditions [Bihari91]. Preconditions allow a binary test to check a state within the application before firing the actuator. Post conditions execute a function after the actuator has modified the application state. Postconditions usually update application state based on the change performed by the actuator. An actuator might even return a steering object to its previous value because the postcondition failed with the new value.

3. Evaluation

To evaluate Progress, we use N-body because it is a well-understood and concise example with which we can describe the functionality and performance of our toolkit. Additional functionality of Progress is evaluated by demonstrating new techniques of interacting with the executing application. Performance is important because this toolkit cannot prohibitively degrade performance of the application. Users will not tolerate excessive performance penalties.

3.1 N-body

The numerical N-body of gravitation [Greengard90] simulates dynamical behavior of large stars with only gravitational forces acting between them. This simulator uses a straightforward N^2 algorithm for calculating pairwise gravitational forces and updating the velocities and positions of the bodies. The N-body application is implemented as a collection of worker threads calculating gravitational forces and new positions for each of the bodies in synchronous timesteps. N-body is built on a user-level threads library. Multiple threads divide the work by allocating each thread a group of bodies to update. Each worker thread reads the positions and parameters of its neighbors, and then updates the position and velocity for each body assigned to it.

Integrating N-body with the Progress system required two distinct steps. First, the developer added appropriate calls to create and initialize the steering server. Second, registration calls for all steerable objects were inserted into the source code. Third, all synchronous objects had the instrumentation code inserted in the application to identify points where these objects could be accessed.

Creating and destroying the steering system only required adding two calls to the N-body source code. Straightforward steering objects provided information on the iteration number, timestep, body count, and various other parameters. These objects were registered with the server through the code, usually near their declaration. At various points in the master thread, the sense calls produce one event to update these objects, if monitoring is enabled. These activation points are not limited to one location. For example, the timestep variable controls the outermost loop in the simulation. In the master thread and the slave threads, these loops are identical with barriers synchronizing the loops. Timestep is the variable loop variable. The timestep object is registered once in the master thread's loop and it has only one sense call within the master thread's loop. This thread generates one event per loop for the entire application. Because the timestep only changes once per loop, then updating it more often with additional sense calls would be repetitive and inefficient. Also, if the slave threads had sense calls within their loop, the timestep object would generate one event per thread per sense call. In other words, at every timestep the number of events generated updating the timestep object would be equal to the number of threads in the application including the master and its slaves. However, all of these events would be identical updates to the timestep object because the timestep loop is synchronized across all threads. Thus, these multiple updates are redundant.

Prior to the steering integration, user interaction was limited to file I/O. The user created an input file with various parameters. N-body then read this file and began processing, occasionally, producing an output message for feedback to the user. At various intervals, the application dumped body position and velocity to a file. While this type of interaction could be customized to provide interactivity, there are no generic methods or toolkits to facilitate this type of selective, application-specific interaction.

Progress allows far more interaction with the simulation than file I/O. With Progress the user interactively explores the intermediate results of the simulation. For example, one particular body's velocity can be traced during the simulation using a sensor to determine if the parameters are realistic. Another sensor can report the timestep value. Yet another sensor can track a load average for that thread. If the user notices an error in the parameters, then he can stall the application and inspect all other objects. Furthermore, if the user decides to take corrective action, the user can update the parameter with a probe write or an actuator, and then, allow the application to continue.

Users may add new bodies and delete existing bodies to the executing simulation with the aid of Progress. As the application is advancing through its timesteps, the Progress server manipulates the application so that a new body with specific parameters is added to the simulation. In our N-body simulation, two application procedures were created that add one body to the current simulation and delete one body from executing simulation. The results of these procedures are cumulative. Executing AddBody twice will add two bodies to

the executing simulation. These two application procedures are necessarily application specific. They have access to the necessary variables and data structures so that the simulation can be updated properly. Additionally, this AddBody procedure can register these new bodies with the steering server so that the user can alter their parameters. Once the procedure is complete and the body parameters are adjusted, the user can continue the simulation.

3.2 Performance

The utmost concern of Progress' designers was the performance degradation due to instrumentation with the Progress library. Obviously, the user can chose to degrade performance by controlling the program; however, this option remains with the user. As illustrated in ***Table 2***, the application's performance degradation due to the addition of Progress is minimal when compared to other common debugging and profiling techniques. The steering system did not interact with the application during the test. These tests did varied proportionally with longer execution times and they did not vary considerably from architecture to architecture (KSR, SGI, Sparc). The measurements in ***Table 2*** were gathered on an SGI 8-node multiprocessor with the standard SGI compiler and multiprocessor library.

Measurement	Time (sec)
N-body optimized	55.55
N-body w/ Progress	57.23
N-body w/ gprof	69.73
N-body w/ -g compiler option	72.11

Table 2 - Application Performance

Latency is important because the information presented at the client, and used for decision making at the server must be close to real time. Unusual feedback might occur if the information is too far out of date, impeding any performance gains due to steering and limiting the usefulness of the steering system. Also, the server must be fast enough to execute steering actions. In ***Table 3***, the performance of several measurements are outlined.

Table 3 describes the latency for several type of steering operations under the following conditions: a total of 100,000 operations are executed by the server and they are measured from this beginning of the operation until they complete including the object registry query time. Additional filtering or processing on the operations are not used. For the synchronous operations, the server time is the time the server requires to place a record into the objects actuate buffer. Enabling and disabling sensors is essentially a probe write to a

memory location that the sensor checks during each call this is why the time is close to that of the probe read/write.

4. Related Work

Several steering systems exist [Gu94] as well as research in dynamic applications and adaptable systems. Because Progress focuses on interactive systems, we limit our review to systems that allow interactivity. Functionality and generality of these systems vary but they are consistent with Progress' goals of performing interactive program steering. As discussed earlier, interactive program steering implies that human users have the option of interpreting program data and providing feedback to the program during its execution. Other research on dynamic systems discusses feedback and adaptation; however, the feedback is usually the product of an algorithm.

Tuchman, et al. [Tuchman91] created the *Vista* system for simulation-time visualization of data. Vista

Action	Server Time (µs)
Probe Read/Write	643
Actuate	627
Sensor Event	181
Enable/Disable Sensor	651

Table 3 - Progress Performance

provides a window into the application by showing program data automatically during execution. The system architecture is designed for a distributed or remotely executing application. The Vista model allows a trace file to replace the executing application, providing a visualization 'data browser' for data from past simulation runs. Data from the executing application are interactively selected and displayed. Vista did not concentrate on steering the application; however, the interactive selection and display of application data is similar to Progress' interactive selection and control of steering objects. Progress goes further by allowing the user to propagate changes from the user interface back into the executing simulation.

Program directing is investigated in [Sosi92]. Program directing is synonymous with program steering. *Dynascope* monitors a program, presents the data to a user or program, and allows for possible feedback actions. Dynascope provides basic monitoring and controlling in distributed environments. The system is integrated with existing programming tools and uses a few generic operating system and networking primitives. Dynascope provides a complete set of tools for interacting with an application, including feedback; however, Dynascope did not concentrate on high performance, parallel applications. The tools available for

interacting with the executing program gave the operator access to the entire program and Dynascope did not allow the developer to clearly define how and where a user could steer the application. Progress focuses both the developer's and user's attention on the steering objects allowing efficient and meaningful access to program components.

The *VASE* system [Jablonowski93] presents an abstraction for a steerable program and offers tools that create and manage collections of steerable codes. VASE annotates existing Fortran code to create a high-level model of the application; therefore, users do not have to work at the source code level. Software developers must annotate the existing code, however. Once the source code is annotated, VASE coordinates the execution of these codes in the distributed environment. VASE supports only the SPMD model of parallel execution. A powerful 'C'-like scripting language provides flexibility for data selection and steering during execution. The SGI Iris Explorer renders output data visualizations. Progress resembles VASE in several ways; however, there are differences. Both VASE and Progress provide a user interface for interacting with a remotely executing application. They both also provide a technique for abstracting uninteresting details from the steering process. VASE, however, concentrates of abstracting blocks of code and control flow, whereas Progress focuses on abstracting important data (steering objects) and time windows for accessing those data items (sense,actuate). Both of these system recognize the difference between the application developer and the application user. The application developer is responsible for instrumenting the application code so that the end user can control the application with a general steering interface.

DYNA3D and AVS (Application Visualization System from AVS Inc.) are combined with customized interactive steering code to produce a time-accurate, unsteady finite-element simulation in [Kerlick93]. Rudimentary steering is demonstrated in a distributed environment consisting of a supercomputer and multiple graphics workstations. Although steering was demonstrated, [Kerlick93] did not present a general toolkit for steering any application program. Progress attempts to define a library and runtime system that will work with a variety of MIMD applications.

[Parris93] describes challenges for a real-time visualization of a complex physical simulation. The goal of this real-time visualization is a virtual world where human users interact with the visualization in a 3D environment. The implementation spans a network of several specialized computer systems. [Parris93] is interesting in the context of Progress because of the feedback techniques used. However, the results of this work were customized for a particular high performance graphics system and network of computers, and it did not address the creation of a general toolkit for steering common high performance parallel applications.

5. Conclusions

Progress is a prototype steering toolkit for the specific purpose of evaluating the necessary components of a general steering system and the essential functionality required by interactive steering. Progress has successfully provided a testbed for interactive steering, and we have outlined a set of general features for inspecting and modifying executing applications: the steering object model and their respective object operations. Progress' steering object model provides a useful technique for identifying components of the application to export to the end-user. Probes, sensors, actuators, synch points, and functions furnishes a developer with numerous mechanisms to allow an end-user to observe and modify his application at runtime.

Two improvements to the existing Progress system are essential. First, the object registry system must allow complex user defined types including arrays and structures. Existing technology forces our toolkit to define these user types are runtime, rather than compile time. The developer must add additional toolkit calls to the application to describe any user defined types that he may want to register as steering objects. Arrays are extremely valuable because the user may want to adjust an entire array of values and rather than registering each element of the array with the server, the developer could just register the array itself. Structures are also important because to allow a user access to the fields within a structure, the developer must now register each field individually instead of just registering the entire structure.

Second, visualizations of steering objects at runtime is necessary to interpret the massive amounts of information that the user might select [Appelbe91,Bem93,Cou93]. Eventually, the goal of Progress is direct manipulation of graphical models of the simulation with appropriate feedback into the executing simulation. High performance graphics systems could possibly provide complex graphics in real-time. For example, one sensor in the application captures three variables: timestep, convergence error, and data region. The Progress runtime periodically forwards this event to the graphics system over the network. At the graphics system, a user binds these three values to a 3D surface graph. Using this visualization, the user easily locates convergence problems with portions of the data regions. As the simulation executes, the user could view an animation of the convergence errors and their respective data regions in the simulation.

6. References

[Appelbe91] William F. Appelbe, John T. Stasko, Eileen Kraemer. "Applying Program Visualization Techniques to Aid Parallel and Distributed Program Development." Report GIT-CC 91/34, College of Computing, Georgia Institute of Technology. July 1991.

[Bates86] Peter Bates. "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior." *ACM/ONR Workshop on Parallel and Distributed Debugging* (1988), pp. 11-22.

[Bem93] Thomas Bemmerl, Peter Braun. "Visualization of Message Passing Programs with the TOPSYS Parallel Programming Environment." *Journal of Parallel and Distributed Computing*, 18(2):118-128, June 1993.

[Bihari91] Thomas E. Bihari, Karsten Schwan. "Dynamic Adaptation of Real-Time Software." *IEEE Transactions on Computer Systems*, 9(2):143-174, May 1991.

[Cou93] Alva L. Couch. "Categories and Context in Scalable Execution Visualization." *Journal of Parallel and Distributed Computing*, 18(2):195-204, June 1993.

[Eisenhauer94] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. "Falcon -- toward interactive parallel programs: The on-line steering of a molecular dynamics application." In *Proceedings of The Third International Symposium on High-Performance Distributed Computing*, San Francisco, CA, August 1994.

[Greengard90] Leslie Greengard. "The Numerical Solution of the N-Body Problem." *Computers in Physics*, March/April 1990, pp. 142-152.

[Gu94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. "An Annotated Bibliography of Interactive Program Steering." *ACM SIGPLAN Notices*.29(9):140-148, September 1994.

[Gu95] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs." *Proceedings of FRONTIERS'95*, February 1995.

[Hollingsworth93] Jeffrey K. Hollingsworth, Barton P. Miller. "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems." *Proceedings of International Conference on Supercomputing* (1993).

[Jablonowski93] David Jablonowski, John Bruner, Brian Bliss, and Robert Haber. "VASE: The Visualization and Application Steering Environment." In *Proceedings of Supercomputing 93*, pp. 560--569.

[Kerlick93] David Kerlick and Eliabeth Kirby. "Towards Interactive Steering, Visualization and Animation of Unsteady Finite Element Simulations." In *Proceedings of Visualization 93*.

[Kilpatrick91] Carol E. Kilpatrick, Karsten Schwan. "ChaosMON -- Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems." *ACM/ONR Workshop on Parallel and Distributed Debugging* (1991).

[Kra93] Eileen Kraemer, John T. Stasko. "The Visualization of Parallel Systems: An Overview." *Journal of Parallel and Distributed Computing*, ():, May 1993.

[LeBlanc87] Thomas J. LeBlanc, John M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay." *IEEE Transactions on Computers*, C-36(4):471-481, April 1987.

[Mal91] Allen D. Malony, David H. Hammerslag, David J. Jablownski. "Traceview: A Trace Visualization Tool." *IEEE Software*, 8(5):29-38, September 1991.

[Malony92] Allen D. Malony, Daniel A. Reed, Harry A. G. Wijshoff. "Performance Measurement Intrusion and Perturbation Analysis." *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433-450, July 1992.

[Marinescu90] Dan C. Marinescu, James E. Lumpp, Thomas L. Casavant, Howard Jay Siegel. "Models for Monitoring and Debugging Tools for Parallel and Distributed Software." *Journal of Parallel and Distributed Computing*, (9):171-183, 1990.

[McCormick88] B. H. McCormick, T. A. DeFanti, M. D. Brown. "Visualization in Scientific Computing." *ACM SIGGRAPH Computer Graphics*, 21(6):, November 1988.

[McDowell89] Charles E. McDowell, David P. Helmbold. "Debugging Concurrent Programs." *ACM Computing Surveys*, 21(4):593-622, December 1989.

[Mukherjee93] Bodhisattwa Mukherjee and Karsten Schwan. "Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package. *Proc. of Second International Symposium on High Performance Distributed Computing (HPDC-2)*, July 1993, pp. 59-66.

[Ogle93] David M. Ogle, Karsten Schwan, Richard Snodgrass. "Application Dependent Dynamic Monitoring of Distributed and Parallel Systems." *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762-778, July 1993.

[Parris93] Mark Parris, Carl Mueller, Jan Prins, Adam Duggan, Quan Zhou, Erik Erikson. "A Distributed Implementation of an N-body Virtual World Simulation." In *Proceedings of The Workshop on Parallel and Distributed Real-Time Systems* (April 1993), pp. 66--70.

[Ribarsky94] William Ribarsky, Eric Ayers, John Eble, Sougata Mukherjea. *"Using Glyphmaker to Create Customized Visualizations of Complex Data."* IEEE Computer, June 1994.

[Schwan88] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, David Ogle. "A Language and System for the Construction and Timing of Parallel Programs." *IEEE Transactions on Software Engineering*, 14(4):455-471, April 1988.

[Snodgrass88] Richard Snodgrass. "A Relational Approach to Monitoring Complex Systems." ACM Transactions on Computer Systems, 6(2):157-196, May 1988.

[Sosic92] R. Sosic. "Dynascope: A Tool for Program Directing." In Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 27(7):12-21, July 1992.

[Stasko90] John T. Stasko, Charles Patterson. "Understanding and Classifying Systems for the Visualization of Computer Data Structures, Programs, and Process." Report GIT-CC 90/66, College of Computing, Georgia Institute of Technology, 1991.

[Tuchman91] Allan Tuchman, David Jablonowski, George Cybenko. "Run-Time Visualization of Program Data." *Proceedings of Visualization '91* (1991), pp. 255-261.