

Models for Computational Steering

Jeffrey Vetter*

Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

This paper describes a conceptual model for computational program steering. By exploiting previous results attained in program monitoring, debugging, and configuration, the abstractions and mechanisms derived from this model are shown suitable for both algorithmic and human interactive steering. A characterization of existing systems for computational steering using the model's abstractions demonstrate such generality. The model is illustrated through its use for steering a large-scale scientific code. Mechanisms for computational steering derived from the model are presented that include sensors, probes, and actuators.

1 Introduction

Computational steering, or steering, is the run-time control of an application and of the resources it uses for purposes of experimenting with application parameters or improving application performance. Examples of performance-motivated computational steering include the runtime adjustment of performance-relevant parameters in real-time applications [1], the adjustment of operating system mechanisms to improve application performance[2, 3], and the improvement of load balance in scientific applications [4]. Steering performed for purposes of experimentation with target applications alter models and applications attributes[5, 6, 4, 7]. For example, the steering interface developed for the global atmospheric modeling code described in [8] permits space scientist to run the code with alternative settings for atmospheric transport, to restart the code from a previously defined checkpoint, and to gradually inject certain chemical constituents into user-identified atmospheric regions.

The model of computational steering presented in this paper seeks unifies both algorithmic and human-interactive steering. Human-interactive steering assumes that a human interprets monitoring data from the target system and then inputs steering decisions, which are enacted by the steering system. Since this steering involves human responses, the speed for interactive steering is constrained to human response times. Algorithmic steering, on the other hand, utilizes algorithms that make decisions based on monitoring information and other information sources, such as history files. Its response time depends only on al-

gorithm complexities and on monitoring and steering delays.

This paper argues that steering systems should be constructed to support both interactive and algorithmic steering, especially since users will typically wish to employ both modes during the course of application development and use. For example, the space scientists working with our group wish to use the interactive steering facilities to comprehend the nature of the multidimensional data produced by their application and its relationship to specific model parameters (e.g., vertical advection). Such human interactive steering can take advantage of advanced visualization environments like virtual reality techniques, data walls, etc.[3]. However, once scientists have discovered suitable model parameters for the input data being used, interactive steering could be automated with algorithmic steering. Therefore, we propose that any steering system must support an evolutionary approach to computational steering. In effect, the system should make it straightforward to redirect online monitoring data from human interactive tools to algorithmic steering tools and visa versa.

The desire to steer programs, interactively and algorithmically, distinguishes our work from most previous research in scientific visualization and user interfaces. Furthermore, the interfaces employed in our work must interact on-line with systems generating data rather than explore post-mortem data, as generally supported by commercial visualization tools (e.g., SGI Explorer, AVS) and explored in research on visualization[9, 10]. Nonetheless, we view computational steering as an extension of visualization and tracking[9] based on technologies first developed for configurable and adaptable systems[11, 1]. In contrast to the visually-based computational steering systems described in [12]. Our work separates user interfaces and scientific visualization issues from computational steering. One result of this separation is our system's ability to work with a variety of user interfaces. Another result is its ability to cross machine boundaries and offer the performance required by on-line steering systems.

In the remainder of this paper, we first describe sample application programs and their computational steering. Using these examples as illustration, we develop a model that describes computational steering as consisting of 8 distinct phases. In Section 3, the model is used to identify some implementation issues

*Vetter is financially supported by NASA Graduate Student Researchers Program grant #93-236.

of steering; in Section 4, other steering systems are described within the context of the proposed model. Section 5 outlines steering mechanisms derived from the model. A summary of results and future research appear in Section 6.

1.1 Steering Scientific Applications

This research is motivated with two sample high performance programs. First, the on-line steering of a **molecular dynamics simulation**[4] is shown to improve program performance by removing load imbalances due to inappropriate data decompositions. This long-running, time-stepped simulation models the behavior of complex hydrocarbon chains residing on an inert substrate. When parallelizing this computation by allocating chains to different processors, the resulting computational loads are not easily estimated, since the bond force computations for each molecular chain depend not only on the molecule's complexity, but also on its distance to other molecules. Second, the initial decomposition of the data domain into 'equal' horizontal or vertical slabs change during runtime. Third, it is difficult to anticipate the computational loads induced by global computations of total energy and by slabs' accesses to neighboring slabs' data.

MD is steered by implementing movable boundaries between its different data domains and then moving these boundaries at runtime such that computational loads are balanced. These movements are in response to load balance information captured over some number of time steps with the monitoring system. Experimental results described in [4] demonstrate that significant performance improvements are attained even with human interactive steering.

Clearly, it is possible to automate the relatively straightforward task of domain boundary adjustment when steering the MD application. A more complex example of program steering is demonstrated with global atmospheric modeling code.

As shown in Figure 1, this ad-hoc steering interface presents to end users modeling outputs along with the corresponding empirical measurements from satellite data.

It requires human judgment to evaluate and correct divergence between both sets of data. This divergence between both sets of data requires that the model be stopped and rewound to a previous time step, then restarted using different settings for certain model parameters (e.g., vertical advection in constituent transport). The atmospheric model requires human intervention during many initial model runs. The purpose of interactive steering in this scenario is to abort useless model runs early, to discover suitable parameter settings, and to further scientists' understanding of the models they use. In addition, end users actually use models like these in 'what if' games. For example, if they inject certain chemical constituents into the atmospheric model through steering, then they can view the evolving distribution so they can better understand their model of the global phenomena. In our current work with space scientists, interactive program execution is being used to understand the distribution and transport of fluorocarbons and their derivatives

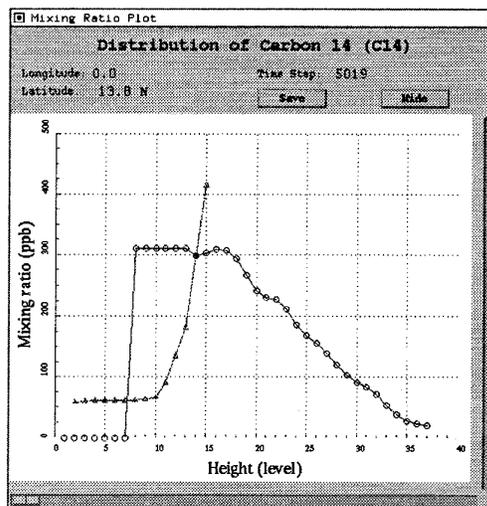


Figure 1: Sample Graphical User Interface.

in the earth's atmosphere.

2 A Model for Computational Steering

The diverse examples of program steering described in the previous section appear to indicate the need for diverse abstractions and mechanisms for computational steering. This is not the case. Instead, each of the computational steering tasks described previously may be modeled as a closed-loop feedback system [13]. On-line monitoring and tracking[9] provide information about the target system's current state and the remainder of the system permits users to make on-line decisions in order to increase performance, to experiment with certain program parameters, and to attempt relevant 'what-if' scenarios.

Consider the model of a traditional feedback system depicted in Figure 2. Computational steering maps onto this model as follows. First, **monitoring** provides inputs to the **steering agent**¹. In MD, monitoring furnishes on-line information such as processor loads, the number of molecules assigned to each domain, and data about the cutoff radii experienced by certain molecules. The steering agent then decides what **steering actions** to exercise based on both inputs from the monitoring system and, possibly, inputs from other external sources. External sources provide information not available from the monitoring system, such as inputs from human users or history files that contain information about previous executions. The steering system provides mechanisms to the steering agent for changing the executing system, which in the

¹Agent describes a decision-maker that interprets the input from the monitoring system and dictates steering. Do not confuse 'steering agents' with 'agents' commonly used in Internet or cognitive science vernacular.

case of MD, facilitates the runtime configuration of domain boundaries.

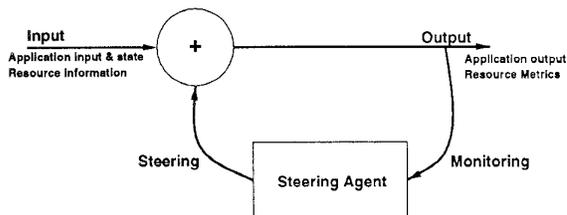


Figure 2: Simple feedback system.

The simple feedback model presented in Figure 2 is the basis on which we develop this paper’s abstract model for computational steering. In this model, the interchangeable steering agent may be a human user, an algorithm, or a hybrid. The steering agent implementation need not depend on the monitoring system. For example, in MD, the steering agent is a Motif-based user interface that receives load information and, then issues steering actions to change the domain boundaries in the running application.

As evident from Figure 2, the performance of any steering system depends on the latency of monitoring, the latency of the steering agent, and the costs of enacting steering decisions. All of these phases jointly determine the steering system’s response time. This response time varies across systems and limits the types of steering possible on those systems. High performance also requires low perturbation of the system’s execution. Correctness demands that the steering system provide mechanisms to protect the target system from introducing irregular behavior or producing invalid application results.

The model extends this analogy with feedback systems further by proposing that a steering system consists of 7 different phases: event generation, event transport, event analysis, steering agent, steering action synthesis, steering action transport, and steering action execution. The first three steps are common to event-based monitoring. The last four steps are specific to the proposed steering model.

2.1 Monitoring

Monitoring is the extraction of dynamic information from a computational system[14]. This model uses an event-based approach to monitoring[15, 16, 17, 18], where each primitive event contains 4 basic components[16], represented by the tuple $\langle \text{event class}, \text{processor ID}, \text{timestamp}, \text{state} \rangle$. Primitive events are immutable. *Event class* is an attribute that describes the type of the event. Event classes differ in the set of attributes they capture. In MD, sample event classes include processor load, domain boundary locations, number of molecules in a domain, and attributes of individual molecules. Event instances of the same class are differentiated by their two attributes *processor ID* and *timestamp*. The *processor ID*, or location, identifies the processor that generated the event. The *timestamp* reports the time when

the event occurred relative to the producing processor’s clock. Finally, the *state* attribute describes some state within the application relative to observations by the producing processor. *State* for MD’s *processor load* event class is a single floating point number that describes the percentage utilization of one processor. *State* for the number of molecules within a domain is an integer. Complex states contain multiple attributes, such as cache performance data for an individual processor (e.g., number of hits and total accesses).

The state of a target system is described by a stream of event instances [16]. These streams are processed in order to collate and present interesting results to steering agents. A steering agent does not share state with the executing processor other than through the channels through which events are received[18]. Each processor has its own reliable channel to the steering agent. Each channel retains event orderings with variable but finite event delivery times.

The event timeline as in Figure 3 provides a common and useful medium for discussing this model. Time increases from left to right and distinct processor execution flows along each arrow. Distinct processors exchange events and actions, similar to messages. For the purposes of steering, the processors do not have access to any shared global state. All state information is exchanged through event and action messages. PX represents a processor that is ‘external’ to the target system being monitored and steered. P1 is one processor within the target system.

The 3 phases of monitoring. Figure 3 illustrates the different processing phases experienced by each single primitive event, which are: event generation, event transport, event analysis, and possibly, event presentation[16, 18, 14]. Event presentation is shown for completeness but later it is omitted because the steering agent subsumes this phase.

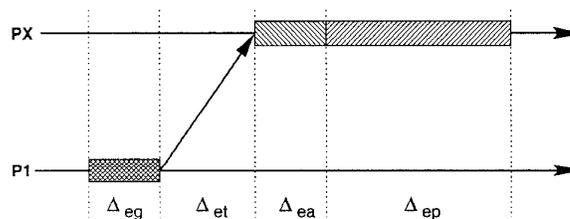


Figure 3: Primitive Event Timeline.

Event generation occurs when a processor retrieves some local state, packs the state into an event tuple along with other necessary information, and places it in the channel. For example, a primitive *processor load* event in MD is created at runtime by allocation of an event structure, followed by filling its *event class*, *processor ID*, *timestamp*, and processor load attributes. Total latency of event generation is Δ_{eg} , where individual processors serialize multiple event generation phases, but multiple processors can generate events concurrently. If the channel cannot

accept the event due to unavailable resources, then the generating processor must wait for the resources. If a processor must block, the wait time is included in Δ_{eg} .

The latency of **event transport** is captured by the time, Δ_{et} , which measures the event's travel time from the originating processor to the steering agent's processor. Channels are error-free, so that Δ_{et} fully describes the event's transmission time across the channel. However, since events are queued in the channel until the steering agent receives them, the event transport time, Δ_{et} , also includes buffering costs due to a busy target processor. Furthermore, this cost is different than the possible waiting cost in Δ_{eg} because Δ_{eg} only includes the time of waiting on a channel to become available and not while the event is in the channel.

During the event analysis phase, PX inspects event attributes. Event analysis time within the steering agent is denoted with Δ_{ea} in Figure 3. During this phase, PX can combine multiple primitive events to form composite events, it can discard events, it can forward primitive events to the next phase for display, or it can store events for future analysis. A **composite event** contains information from 2 or more primitive events to convey summary information or to describe some aggregate state extracted from these multiple events. For example, a composite event in MD states that two processors in the simulation have less than 50% utilization (see Figure 3). The implementation of composite events is straightforward if its constituent primitive events arrive in a loosely synchronized fashion, at similar rates. Otherwise, obvious problems arise regarding the time interval for which primitive events are retained in the steering agent for possible combination with other events [16]. The reliability of such information in composite events must also be guarded. The event analysis time, Δ_{ea} , contains all latencies due to event analysis. Event presentation follows event analysis.

Event presentation with latency Δ_{ep} in Figure 3 is the presentation of information contained within a primitive or composite event. Event presentation differs from analysis because it does not manipulate event contents, it only presents them in a form suitable for steering algorithms, for human end users[10], or for later use (e.g., in trace files or in temporal databases). For example, a simple bar-graph view of load balance in the MD application simply checks each primitive event received from event analysis to determine whether any of its attributes are bound to the graphical view. It then updates the particular bar-graph for each processor if that is the case.

From Figure 3, it is apparent that the total latency of monitoring, Δ_m , is the sum of phases one to three ($\Delta_m = \Delta_{eg} + \Delta_{et} + \Delta_{ea}$). Phase five is not included in this calculation because its needs are determined by the complexity of the display system, steering algorithms, and human-end users. Notice that in the a system that has separate processors for the application and the steering agent, the application perturbation is depends only on Δ_{eg} . Once an application processor generates an event, it can continue.

2.2 Steering

Concepts for steering are not as well-defined as those for monitoring and debugging. This model offers several new abstractions to assist in the discussion of steering and provide metrics for comparing systems. *Steering actions* are the basis for this model. We propose that this abstraction for steering permits a flexible and complete model for changing executing systems when combined with event-based monitoring.

Steering actions are a natural generalization of monitoring events [1]. A primitive steering action is represented with the 4 tuple $\langle \textit{steering action class}, \textit{processor ID}, \textit{condition}, \textit{state} \rangle$. The action tuple contains four entries: *steering action class* defines the action type, where tuples with the same *action class* differ in their entries, *condition*, and *processor ID*. *Processor ID* identifies the action's target processor. *Condition* generalizes the notion of *timestamp* in monitoring events. Generally, timestamps are nothing more than conditions that use only the total ordering of time. With conditions, the activation times of steering actions' may be described in terms of timestamps or in terms of program states that must be observed by the target processor before the action is performed. Our model assumes that such conditions are some locally observable system state, the conditions are limited by the well-known problems with global conditions in systems with distributed program state[19, 20]. A *null* condition indicates immediate action execution. When a target processor receives a steering action with a *null* condition, it executes the action independent of the local program state. Last, *state* denotes the action parameters associated with *action class*, where **absolute state** resets some target program *state* to a new value, whereas **relative state** updates the target state relative to its current *state* (e.g., 10%).

Our model classifies actions with respect to the application as either synchronous or asynchronous. Steering actions scheduled to execute immediately upon receipt by the processor without condition are **asynchronous** because they are executed irrespective of the current processor state. Asynchronous actions perform their modifications sometime after they are synthesized. The steering agent has no control over the time or the state of the target system when the asynchronous action is executed. These actions are useful for updating non-critical program variables, such as the advection parameters in the atmospheric modeling code. These actions are also useful for modifying suspended systems. Conversely, a **synchronous** action occurs only when some predetermined *condition* has been observed. This conditional prohibits the actions execution until the *condition* is evaluated by the target processor as **true**. For example, the injection of a specific chemical constituents at a certain timestep of the atmospheric modeling code are synchronous because the required modifications are not executed until the system is between timesteps and hence, the condition is true.

As apparent from the previous descriptions, steering is defined such that its execution does not rely on the system's global state. Specifically, the steering agent communicates with each individual processor

via their dedicated channels (Section 2.1). Furthermore, a synchronous action is executed by the target processor rather than the steering processor so that state changes may be performed in conjunction with the target processor's behavior. Discussion of the interchangeable steering agent begins in Section 2.3.

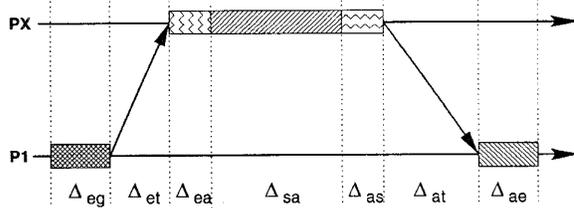


Figure 4: Steering Action Timeline.

The 3 phases of steering. As with monitoring, steering may be defined with three phases: steering action synthesis, steering action transport, and steering action execution. Figure 4 depicts the timeline for the generation of a primitive event by the monitoring system, followed by the steering system's generation of a **primitive steering action**. Specifically, the steering agent accepts primitive and composite events from the monitoring system at the completion of the event analysis phase, after time Δ_m has elapsed (ignoring event presentation). With execution latency Δ_{sa} , the steering agent decides what steering actions to perform, if any. Its latency may be quite high when such decision making includes event presentation with some user interface, followed by decision-making by human users.

Action synthesis generates primitive steering action tuples. These tuples target all relevant processors. Steering action tuples are generated from data available to the steering agent from the monitoring system and from external sources (e.g., history information). For example, the steering action for changing domain boundaries in MD targets an appropriate processor, sets its condition to *two boundary-adjacent processors are finished with a timestep*, and generates *state* to specify the new relative boundary value (e.g., -10% along the X-axis).

Transport of the steering action utilizes the buffered channels described in Section 2.1 to transmit steering actions to processors. Steering action synthesis first packages the steering action tuple, then sends it via the channel, with total latency Δ_{at} , to the target processor. The transport protocol does not generate acknowledgments. Notice that the steering agent cannot predict when a processor will receive a tuple or when the action has completed. Subsequent receipt of appropriate monitoring events or an extension to the protocol may confirm the completion of an action. Buffering time is included in Δ_{at} .

Upon receipt of an action tuple, the **action execution** begins on the target processor with a possible delay caused by its condition. This delay might require the processor to buffer the action locally. This delay

is included in the latency for action execution Δ_{ae} . For example, a synchronous version of the boundary change in MD may require that the target processor first determine whether the boundary value is in current use (e.g., using a critical section) before updating it. As stated previously, the new boundary values are stored in the action's *state*. When the target processor has executed the action, it discards the tuple. The latency for individual action execution is Δ_{ae} which includes delays incurred by unsatisfied conditions.

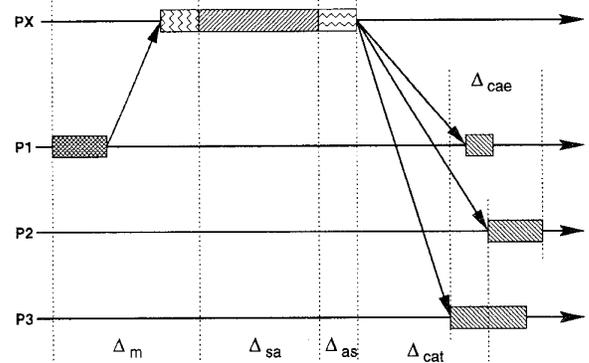


Figure 5: Composite Steering Action Timeline

Composite steering actions, illustrated in Figure 5, allow creation of a hierarchy of steering actions. During the action synthesis phase, the steering system can expand a composite steering action into many primitive steering actions. All of the actions are placed on the channel to the target processors. Several metrics change for composite actions. Action synthesis remains the same for both primitive and composite steering actions. First, action transport becomes **composite action transport** with latency Δ_{cat} . Δ_{cat} is the maximum transport time for any one of the primitive steering events that is part of the composite steering action. **Composite action execution** Δ_{cae} is the difference from the start time of the earliest primitive action execution until the end time of the last primitive action execution. The variance in both the start time and execution time of any one of the primitive actions could effect Δ_{cae} . Once the actions are received, they are executed on their target processor. The purpose of composite actions created during action synthesis is to simplify and coordinate complex steering maneuvers executed by multiple processors. An example of a composite action in MD is a concurrent shift in all domain boundaries by some relative amount. Each processor receives one action, but that action is tailored for the target processor.

With large delays in communication channels, the state retention problem for composite actions is the same as the one for composite monitoring events[16], where storage requirements depend on the number of pending primitive actions. This problem is discussed further in Section 2.5.

Therefore, the total latency of the steering system is the sum of the separate steering phases: $\Delta_s = \Delta_{as} + \Delta_{at} + \Delta_{ae}$.

2.3 Steering Agents

Steering agents (see Figure 2) make decisions based on the event stream from the monitoring system and possibly inputs from an external source. The existence of an external source is a critical element of the interactive steering model, because it permits the dynamic use of human knowledge and execution history. A steering agent can use a variety of methods such as graphical interfaces or expert systems to decide what steering actions to enact. For example, in the case of MD, once human users have gained better insights into the correlation between processor loads and relative domain sizes for certain data sets[10, 21], they may then devise and experiment with algorithms that automate the load balancing task, or they may simply choose to replay steering actions scripted during previous program runs. In all such cases, interfaces to the steering system remain the same, consisting of event tuples provided by the monitoring system and steering actions affecting the executing program.

2.4 Steering System Response Time

The latency of entire steering system combines the monitoring latency Δ_m with the latency of the steering agent Δ_{sa} and the latency of the steering system Δ_s to give a total system latency of $\Delta_{steering} = \Delta_m + \Delta_{sa} + \Delta_s$. The entire perturbation on the executing application is only the two phases of event generation Δ_{eg} and steering action execution Δ_{ae} . The remainder of the monitoring and steering phases operate concurrently on target systems when sufficient resources are reserved for the monitoring and steering system. Given this model of monitoring and steering, the application is minimally perturbed. Transport, analysis, and steering action synthesis execute concurrently to the application.

2.5 Model Issues

The model presented in Section 2 does not capture extended steering issues, such as coordination of composite steering actions, recognition of action completion, and enhancement of steering with properties typically associated with general feedback systems.

The execution of composite actions is complicated by several factors, including delays in the execution of primitive actions, potential deadlocks caused by circular dependencies among composite actions. The model does not offer general solutions to any of these topics, but instead, posits that a basic steering system should provide mechanisms and guidance but should not offer specific solutions to these problems. This model permits target system specific solutions to these issues, as shown possible in [1], where ‘adaptation transactions’ are efficiently implemented for pipelined program structures.

Recognition of action completion is difficult with this model because it assumes distributed state[19]. Because processors execute and then discard steering actions, the steering agent cannot trivially infer when

a steering action has completed. Conditions on steering actions exaggerate this problem because a processor may hold a steering action for an arbitrary amount of time. In addition, completion checking cannot trivially rely on action orderings since the orderings may change when processors remove steering actions from the channel and then store them locally. Specifically, if such a solution relies on maintaining action orderings, then such orderings may be implemented using additional conditions formulated by the steering agent and enforced by processors. In addition, processors could implement acknowledgments, some of which may be piggybacked on future monitoring messages. Alternatively, a steering agent might simply rely on future monitoring output to verify that certain steering actions have been taken.

The steering model is easily analyzed with several general characteristics of feedback systems [13].

Sensitivity of the target system to steering may be evaluated by analyzing the steering of particular system components. Sensitivity ranges from not changing the system output to deadlocking the system. The sensitivity of each component is measured by perturbing each possible system component with the steering system and evaluating the change in system output.

A system being steered should remain stable, which implies that its outputs should remain bounded given bounded inputs. **Stability** checking should be performed with algorithms specific to certain systems and data sets. The addition of steering should not cause the target system to ‘thrash’ or oscillate. Usually, this requirement forces the steering system to have low latency so as to prevent positive feedback through the steering system.

3 Related Research

This section maps a selected group of those systems onto the model developed in Section 2. Several prototype computational steering systems exist; they are discussed extensively in two separate articles [22, 12].

VASE [6] established an interactive steering system for SIMD computers that provided tools for code annotation, run-time control, and visualization. A SIMD model of computation relaxes several of the constraints on steering actions mentioned in Section 2.2. First, because global state is shared, any processor including the steering agent can determine not only the data for another processor but it also knows the state of computation at each processor. Therefore, steering actions could be performed by the steering agent alone while other processors idle. Conditions are unnecessary because the steering agent knows the state of each processor and can determine if any condition is true.

Dynascope[23] developed a steering (directing) system using debugging facilities provided by the operating system. Dynascope lacks important time information about when to change state. Debugging mechanisms do not enforce system integrity with conditions as steering action tuple do and this lack of conditions allows arbitrary changes to the system which might invalidate system execution.

Several systems use dataflow architectures for steering and visualization [5, 12]. Dataflow systems are

the easiest to map to this feedback model. Computation is usually contained within modules whose input and output are connected to other modules for visualization and control. This dataflow network forces an ordering similar to the proposed model ordering on the computation and explicitly specifies inputs and outputs. Changes are fed into the input ports of the computation modules. As the module executes, new inputs are integrated into the calculations. This large granularity for steering has several disadvantages. First, if the computation is not decomposed thoroughly through modules, then any feedback requires recomputing the dataflow network. Thus, even the smallest change in the inputs to a module cause recomputation. This sensitivity to every system component reduces performance because any action forces a total recomputation. Second, it is unclear how modules incorporate changes into their execution other than restarting the computation with new parameters. Third, due to the limited number of inputs a module can accept, the possible number of parameters to steer is limited.

Progress[7] and Falcon[15] provide the ability to monitor and steer fine-granularity items within the executing system. Falcon is an event-based monitoring system that provides low-latency access to system data at runtime. Progress is an application-specific, low-latency steering system that provides actuators and probes to allow structured changes to executing systems. These actuators provide conditions as described by steering action tuples that allow the target processor to validate and coordinate changes before their execution.

4 Mechanisms for Steering

This section presents several mechanisms that provide the functionality outlined by the proposed model. Many of the monitoring mechanisms have been presented by earlier research; however, we list them here to solidify the model's usefulness.

4.1 Monitoring Mechanisms

The results of any monitoring mechanisms are events as described earlier. Two possible types of monitoring mechanisms produce events: sensors and probes. Both of these devices are described elsewhere [17].

Sensors. Sensors are instrumentation points placed throughout the application's source code; therefore, sensors are synchronous mechanisms with respect to the application. Each sensor captures information about the state of the application or a system resource as an event and forwards these events to the steering agent. The generation of events by sensors is totally dependent on the frequency that the applications processors encounter these sensor instrumentation points. Advanced techniques for sensor control use online control, filtering, and sampling sensors to control event frequency. Sensors can do minimal calculation; however, they must be very efficient[15];

Probe reads. Probes, as opposed to sensors, are not part of the application code. Probes allow some external processor to 'snoop' into an executing application's data and resources. Because probes don't

synchronize with the application execution, they are considered asynchronous. The result of a probe is the same as a sensor: an event tuple. Probes are useful for monitoring data differently than sensors because they have no application overhead and they allow instant data investigation.

4.2 Steering Mechanisms

Steering mechanisms are the converse of monitoring mechanisms. Two such steering mechanisms, actuators and probe writes, provide the flexibility and completeness to steering furnished to monitoring by sensors and probe reads. Both actuators and probe writes are controlled by steering actions.

Actuators. Actuators, like sensors, require instrumentation points in the application source code [7]. An actuator is controlled by a steering agent; however, the actual execution of the steering action is performed by the application's processor when it encounters an actuator instrumentation point. With respect to the application, the change is synchronous with the application because the application can identify the actuators as the only regions of code that allow modifications from the steering system. With this limitation, actuators provide a 'safe' mechanism to change state within the application. The application can expect changes at various points throughout its execution and accept those possible changes. Conditions for steering actions limit those changes. The condition can limit the action to a specific segment of the code, or it can specify some state in the application is true prior to performing the steering action. Actuators rely on the application for their execution. If the application execution does not include an actuator instrumentation point, then the changes requested by the steering agent will not occur.

Probe writes. Probe writes allow identical functionality to probe reads, but instead of reading some state they actually update it. Probe writes are asynchronous because the target state is updated immediately instead of waiting for the application's processor to perform the change. Probe writes have a *null* condition in their steering action tuple. Regardless of the application's state, the requested steering action updates the application state to the state required by the tuple.

5 Conclusions

This paper presents a model for computational steering that captures the essential properties of steering systems whether the steering is guided by a human or algorithm. The model is based on previous research in monitoring and debugging systems, and adaptable and configurable systems. The model defines the process of steering using a timeline of monitoring events and steering actions. A time diagram motivates the description of the monitoring and steering system as well as the role of the steering agent. The steering agent is interchangeable and it could provide either human-interactive or algorithmic steering. Event and steering action tuples capture the essential abstractions in the steering system. Existing prototype steering systems demonstrate the model's usefulness and

provide illustration of the model features and limitations.

6 Acknowledgments

We thank Vernard Martin and Daniela Ivan for their comments and proofreads, Greg Eisenhauer and Weiming Gu for their implementation work and discussion, and Thomas Kindler for his work on the atmospheric modeling software.

References

- [1] T. Bihari, P. Gopinath, and T. Walliser, "Managing beliefs, desires and time in real-time systems," in *Proc. Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pp. 114–9, 1991.
- [2] B. Mukherjee and K. Schwan, "Experiments with a configurable lock for multiprocessors," in *Proc. 22nd Int. Conf. on Parallel Processing*, pp. 205–8, 1993.
- [3] D. A. Reed, K. A. Shields, W. H. Scullin, L. F. Tavera, and C. L. Elford, *Virtual Reality and Parallel Systems Performance Analysis*. Dept. of Computer Science, University of Illinois, 1995.
- [4] G. Eisenhauer, W. Gu, K. Schwan, and N. Mallavarupu, "Falcon - toward interactive parallel programs: The on-line steering of a molecular dynamics application," in *Proc. Third Int. Symp. on High-Performance Distributed Computing (HPDC-3)*, pp. 26–34, 1994.
- [5] S. G. Parker and C. R. Johnson, "Scirun: a scientific programming environment for computational steering," in *Proc. Supercomputing 95*, (San Diego), pp. 1–, 1995.
- [6] D. J. Jablonowski, J. D. Bruner, B. Bliss, and R. B. Haber, "VASE: The visualization and application steering environment," in *Proc. Supercomputing '93*, (Portland, OR, USA), pp. 560–9, 1993.
- [7] J. Vetter and K. Schwan, "Progress: a toolkit for interactive program steering," in *Proc. 24th Int. Conf. on Parallel Processing*, pp. 139–42, 1995.
- [8] T. Kindler, K. Schwan, D. Silva, M. Trauner, and F. Alyea, *A parallel spectral model for atmospheric transport processes*. Georgia Institute of Technology, 1994.
- [9] B. H. McCormick, T. A. DeFanti, and M. D. Brown, "Visualization in scientific computing," *IEEE Computer*, vol. 23, no. 8, 1989.
- [10] E. Kraemer and J. T. Stasko, "The visualization of parallel systems: An overview," *Jour. of Parallel and Distributed Computing*, vol. 18, pp. 105–17, 1993.
- [11] J. Kramer and J. MaGee, "Dynamic configuration for distributed systems," *IEEE Trans. on Software Engineering*, vol. 11, no. 4, pp. 424–36, 1985.
- [12] M. Burnett, R. Hossli, T. Pulliam, B. Van-Voorst, and X. Yang, "Toward visual programming languages for steering scientific computations," *IEEE Computational Science & Engineering*, vol. 1, no. 4, pp. 44–62, 1994. Oregon State Univ.,
- [13] C. L. Phillips and R. D. Harbor, *Basic Feedback Control Systems*, vol. 1. Prentice-Hall, 1991.
- [14] R. Snodgrass, "A relational approach to monitoring complex systems," *ACM Trans. on Computer Systems*, vol. 6, pp. 157–96, 1988.
- [15] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu, "Falcon: On-line monitoring and steering of large-scale parallel programs," in *Proc. Frontiers '95*, 1995.
- [16] P. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," in *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 11–22, 1988.
- [17] D. M. Ogle, K. Schwan, and R. Snodgrass, "Application-dependent dynamic monitoring of distributed and parallel systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 762–78, 1993.
- [18] D. C. Marinescu, H. J. Siegel, J. E. Lumpp, and T. L. Casavant, "Models for monitoring and debugging tools for parallel and distributed software," *Jour. of Parallel and Distributed Computing*, vol. 9, pp. 171–84, 1990.
- [19] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. on Computer Systems*, vol. 3, pp. 63–75, 1985.
- [20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–65, 1978.
- [21] A. D. Malony and D. A. Reed, "Visualizing parallel computer system performance," in *Parallel Computer Systems: Performance Instrumentation and Visualization* (M. S. Bucher, ed.), New York: ACM, 1990.
- [22] W. Gu, J. Vetter, and K. Schwan, "An annotated bibliography of interactive program steering," *SIGPLAN Notices*, vol. 29, pp. 140–8, 1994.
- [23] R. Sosic, "Dynascope: a tool for program directing," *SIGPLAN Notices*, vol. 27, pp. 12–21, 1992.