# High Performance Computational Steering of Physical Simulations [1]

Jeffrey Vetter[2] Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
{vetter,schwan}@cc.gatech.edu

## Abstract

*Computational steering allows researchers to monitor and manage long running, resource intensive applications at runtime. Limited research has addressed high performance computational steering. High performance in computational steering is necessary for three reasons. First, a computational steering system must act intelligently at runtime in order to minimize its perturbation of the target application. Second, monitoring information extracted from the target must be analyzed and forwarded to the user in a timely fashion to allow fast decision-making. Finally, steering actions must be executed with low latency to prevent undesirable feedback. This paper describes the use of language constructs, coined* ACSL, *within a system for computational steering. The steering system interprets* ACSL *statements and optimizes the requests for steering and monitoring. Specifically, the steering system, called* Magellan, *utilizes* ACSL *to intelligently control multithreaded, asynchronous steering servers that cooperatively steer applications. These results compare favorably to our previous Progress steering system.*

## 1 Introduction

*Computational steering* is the online management of the execution of long-running, resource-intensive applications for the purpose of either application exploration or performance improvement. *Exploratory steering* allows scientists to track applications and modify them at runtime[7], typically giving them structured access to various application components. As a result, it becomes straightforward to experiment with various application parameters, alternative input data or solution methods, etc. *Performance steering* allows scientists to change applications to improve application performance. Manual load balancing is an example of interactive performance steering. In general, however, performance steering is often performed by algorithms[4]. *Algorithmic steering* replaces the human in the steering loop with an algorithm written with some language.

Interactivity within high performance computing is not a new topic. For years, scientists have interacted with their applications with customized solutions. Usually, they are effective because of their application-specific customization. However, these interfaces are not easily leveraged across different domains and applications precisely because of their customization. The visualization and user interface issues tackled in such work[7, 5, 1] are complementary to this research. Instead, this research focuses on the design of the steering system infrastructure.

Debuggers[8] furnish another method for interacting with high performance applications. However, debuggers are written to permit highly invasive interactions like inspecting and changing arbitrary application state. The perturbation they impose on target applications may be unacceptable to many potential users for interactivity. In addition, compilers often degrade performance by disabling optimizations. Other ways of interacting with programs are provided by tools and libraries enabling interactive visualizations, data exploration, and tracking.

Computational steering may be decomposed into three components[11]. First, a *steering agent* is the *decision maker* because it receives and interprets data generated by the steering system and then issues steering commands; this agent might be implemented by a decision algorithm, or it may consist of a scientific visualization and a human user. Second, the *steering system* gathers data from and controls the application. Third, the application is *instrumented* to cooperate with the steering system.

This papers uses a heat diffusion application to illustrate techniques and mechanisms. The heat diffusion simulation

[1] This paper is also available as Technical Report GIT-CC-96-21, College of Computing, Georgia Institute of Technology (http://www.cc.gatech.edu/tech_reports).

[2] NASA financially supports Vetter via Graduate Student Researchers Program grant #NGT-51154. Los Alamos National Lab also provided support for this research.

is a 27 point, 3D stencil calculation which has representative communication and computation properties of many simulations.

The principal contribution of this paper is the development of **a language-based approach to computational steering**. Namely, we posit that language-based specifications are an essential component of future systems for computational steering because they enable steering with optimizations for a variety of criteria. This paper demonstrates that a language-based approach to steering may be implemented efficiently, without limiting steering interfaces and without negative implications on system flexibility and extensibility.

Advantages derived from the proposed language-based approach include: (1) it is equally easy to perform computational steering via user interfaces or algorithms, (2) steering specifications remain independent of both the underlying computer architecture and the programming paradigm, and (3) language-based specifications permit the steering system to make the appropriate choices among the alternative ways of extracting and analyzing system information and then, performing steering.

The use of language-specifications to address program configuration and adaptation problems is not new. [3] reviews language approaches to adaptive and reactive systems, database systems, program monitoring, configuration management, scalable instrumentation, and performance visualizations. Further, [3, 1, 8] provide thorough descriptions of research on computational steering.

In comparison to such work, this research focuses on understanding and evaluating the tradeoffs in runtime performance resulting from alternative manners in which steering commands are processed into online monitoring and steering actions.

§2 describes the new language constructs for computational steering named ACSL. §3 describes the prototype of the Magellan system, which uses ACSL in its implementation of computational steering. §4 evaluates Magellan performance as described in the introduction. Conclusions and future research appear last in §5.

## 2 ACSL Steering Language Constructs

This section introduces computational steering in terms of language constructs coined ACSL for *advanced computational steering language*. An ACSL program consists of a set of commands issued by the steering agent that are translated into operations and applied to steerable components that each application exports. Since each ACSL construct describes a range of steering commands, the steering system has the ability to analyze such commands and then compare alternative ways in which commands may be executed (similar to query optimization in relational database languages).

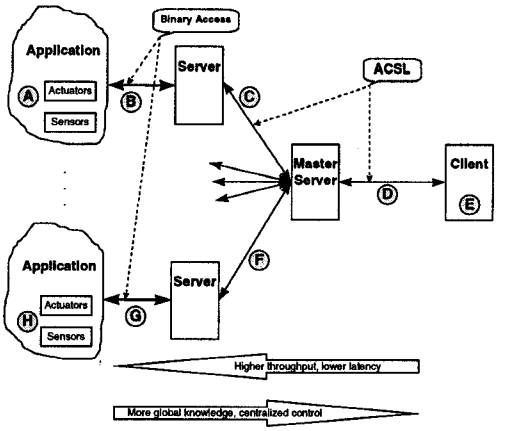


**Figure 1. Magellan software architecture.**

ACSL commands may be compiled into explicit steering code that is known *a priori*, or they may be interpreted at runtime.

Figure 1 depicts the software architecture of the Magellan system for computational steering. ACSL commands are issued by the steering agent at the client. Every server in the hierarchy is capable of feedback, not only the feedback loop through the client. ACSL provides communication among servers and between servers and clients. Servers communicate with applications directly using binary access.

The ACSL approach to steering is to specify the steering command with a language construct, as shown in Figure 2. This language construct is interpreted by the steering system. The steering system then optimizes the request to provide an equivalent steering action with lower latency, higher throughput, and/or smaller performance degradation. Specifically, concerning load balancing commands, the steering agent may send a load balancing command given in Figure 2 to the steering system at some point during the application's execution. When the steering system interprets this command, it first enables monitoring for thread loads on all the threads because that data is needed for analysis as specified in the `when "thread.*.load" < 0.8 do` clause. When the specified load of 0.8 is not met, the steering system changes the specific application thread satisfying this clause executing in `lines 3-4`. The effect of such command interpretation by the steering system is that the feedback path is *short-circuited*. Instead, monitoring information is fully interpreted by the steering system, and appropriate actions are also issued by the server, without involving the steering agent. This shortcut is made possible by the runtime interpretation of ACSL commands by the steering system. The steering system determines from this

```
when thread.*.load < 0.8 do
{
  $1.start actuate $1.start -= dimLength * 0.05;
  $1.end actuate $1.end += dimLength * 0.05;
}
```

**Figure 2. ACSL code sample.**

specification that all of the components for executing the request reside locally in each thread and that each thread can also locally execute the appropriate event-action command. The system executes the action whenever the when condition (line 1) is satisfied. Additional details on the steering system's implementation appear in §3 below.

ACSL language constructs build on earlier research[9, 11] which decomposes computational steering into several phases. This section defines the primary functionality of ACSL. Some ancillary language constructs exist in the actual implementation[10].

ACSL assumes the export of monitoring and steering objects from an application[9]. Applications are responsible for registering at runtime the objects available for steering and monitoring. Objects can be almost any application component from scalars to vectors as well as basic structures and functions. These objects are then available for the steering system to inspect and manipulate. ACSL provides operations that use these objects as the target for their monitoring and steering commands. ACSL is specified as a LALR(1) grammar. A typical ACSL steering command has the form `"<object name>" <action> [options] [data|expr];` Commands can be nested and they can be used in conjunction with control structures to provide complex steering.

ACSL commands build on basic monitoring and steering mechanisms. These mechanisms are primarily sensors, probes, and actuators. *Sensors* are monitoring mechanisms that provide consistent information about application execution in the form of events[6, 11]. Sensor events are snapshots of application state created by an application thread which has encountered an instrumentation point. Sensors can be enabled and disabled or they may have their sampling frequencies changed[6]. ACSL provides `"<object name>" sensor [id] <enable|disable|freq <rate>>;` to actually control sensors for a particular object.

*Probes reads* are monitoring mechanisms while *probe writes* are steering mechanisms. Unlike sensors, probes have no instrumentation points in the application. A probe read simply retrieves a value for an object[6, 11]. Probes cost the application performance nothing except possibly some cache and processor disruptions. Probe writes, the opposite of probe reads, simply overwrite a particular data item within an application. ACSL provides `"<object name>" probe <read|write <data|expr>>;` to actually to probe read or write a particular object.

*Actuators*[9] are another steering mechanism, and they are similar to sensors because they have instrumentation points in the application code; however, they change the application object instead of record its state as sensors do. An actuator requires that the steering agent specify data or an expression as the new value for an object. Objects are changed only when the application specifically approves such changes. ACSL provides `"<object name>" actuate <data|expr>;` for actuators. Actuator requests are buffered until the application accepts the request to change an application object. Actuators can be far more complex because they can evaluate conditions and accept or reject steering requests. Also, they can execute functions as pre-conditions and post-conditions to the requested changes.

ACSL commands manipulate these sensors, probes, and actuators, and they may be composed into simple programs using control structures, as shown in Figure 2. These control structures are based on the event-action paradigm, where some code block is executed when a monitoring event meeting some criteria is received by the steering system. ACSL provides `when <expr> do {code block};` as the basic event-action control structure. `expr` can be a combination of objects and basic numerical expressions as well as relational operators.

In addition, while the current implementation of ACSL processes language statements to control the steering system's operation, ACSL statements may also be bound to graphical user interfaces, if desired, by having parameterized steering actions[5] send ACSL commands to a steering system and update their glyphs based on the ACSL commands that it receives.

## 3 Magellan Steering System

The Magellan prototype is derived from the Progress system[9], but offers additional functionality as well as the ACSL language specification described in §2. ACSL combined with local configuration information and requests from the steering agent allows Magellan to intelligently determine what it needs to accomplish specific steering actions and then manages the steering system during execution per the goals set forth in §1. In addition, Magellan offers high performance, extensibility, flexibility, and portability.

The Magellan architecture, depicted in Figure 1, consists of two primary components: steering servers and steering clients. Steering clients provide a convenient, efficient mechanism to interact with the steering servers. All user interfaces and external programming tools access the system through a steering client, not the steering server.

A server monitors and steers an application program in three basic ways: sensors, probes, and actuators. Sensors and probes are defined elsewhere[11, 6, 2]. Actuators[9] are instrumentation points within an application that know how to change an object without disrupting application execution. Assume the steering agent issues the steering command `"tau" acutate 4e4`. This step is `E` in Figure 1. The agent has received information from the monitoring system and *decided* to change `tau` to 4e4. This command is sent to the steering server at `C` or `F` wherever `tau` lives. If `tau` is a replicated object, then commands are distributed to all the necessary servers. When the server receives this command, it retrieves a registry entry describing `tau` including its type, memory address, and the address of a shared memory buffer for the actuator requests. Pending actuator requests are stored in this shared buffer until an application thread polls the buffer for steering requests (step `G` in Figure 1). Once the server places the request in the buffer, it is finished with the actuator command. When the application thread encounters an instrumentation point (step `H` in Figure 1), then it extracts the data from the buffer, checks any conditions on the data with a user defined condition, and then, updates the object data value with the requested value.

# 4 Evaluation

Performance tests focus on the advantages of ACSL over other ad-hoc methods of steering. Algorithmic clients were chosen to create consistent and repeatable performance results.

The heat diffusion simulation is a time-stepped simulation. At initialization time, all interesting global data is registered with the steering system. Then, a number of sensors and actuators are placed into its main threads. This simulation uses a load balancing algorithm that adjusts the columns allocated to each thread along the z-axis.

Each simulation was executed on a DEC Alphaserver 8400. The system has 12 Alpha 21146 processors operating at 300Mhz. Each processor accesses 4GB of shared memory through a memory interconnect that has a sustainable bandwidth of 1.6GBps. A reference cost of a null procedure call on this system is 18 *nano*secs.

**Instrumentation Costs:** This section presents experimental latencies in Magellan for sensors and actuators. Probes are not measured.

The cost of a disabled sensor is just 111 nsecs, about 6 times the cost for a procedure call. Second, the cost of an enabled sensor must be small. An enabled sensor executes in 3138 nsecs, which is 28 times the cost of the disabled sensor.

Actuators[9, 11] are the opposite of sensors. The disabled actuator costs 151 nsec, 8 times a procedure call. The enabled actuator is similar in execution time to the enabled sensor costing about 2905 nsecs, only 19 times the disabled actuator when using 4 bytes for data size.

**Application performance:** Application performance was measured on the heat diffusion simulation (§1). Heat diffusion runs on 8 processors for 127 seconds with work evenly distributed between processors. Disabled steering and monitoring instrumentation increased the runtime approximately 1%.

## 4.1 Performance improvements due to ACSL

Recall the hypothesis of ACSL in §1. This section compares a simple method of steering with the prototype system Magellan which uses ACSL for steering. Equation 1 helps to quantify the improvements in steering response due to ACSL over other less complex steering methods. (Eq. 1) $\Delta_{steercycle} = \Delta_{monitor} + \Delta_{agent} + \Delta_{steer} = \Delta_{me} + \Delta_{mt} + \Delta_{ma} + \Delta_{agent} + \Delta_{ss} + \Delta_{st} + \Delta_{se}$. The cost for one cycle of steering in the original steering feedback model[11] includes generating monitoring data within the application($\Delta_{me}$), transporting it to the steering agent($\Delta_{mt}$), digesting the data($\Delta_{ma}$), deciding on a steering command($\Delta_{agent}$), issuing the steering command($\Delta_{ss}$), transporting it to the steering system($\Delta_{st}$), and finally executing the steering command($\Delta_{se}$). Equation 2 provides the frequency response of a steering system. (Eq. 2) $\omega_{steering} = \frac{1}{\Delta_{steercycle}}$. As any of the three components ($\Delta_{monitor}, \Delta_{agent}, \Delta_{steer}$) increases, the frequency of steering decreases inversely.

**$\omega_{steering}$ for traditional steering system:** Previous results from Falcon and Progress are estimated on the new system by calibrating them with empirical measurements on Magellan. Gathering data with a monitoring system such as Falcon[2], a steering system can expect a $\Delta_{me}$ of about 3 $\mu$secs. Further, a steering system like Progress[9] can arm actuators in a similar amount of time. Transport times for both monitoring and steering depend greatly on the underlying protocol and operating system. TCP-IP, for example, have a single one-way message send of 1 msec over traditional 10Mbs Ethernet on Unix. The response time for the steering agent is directly a function of the complexity of the steering agent. Assuming a human in the loop, then it is safe to ignore the actual costs of the interface technology and estimate the human user response time of about 100 msec. Given this estimate, the final $\omega_{steering}$ is $\frac{1}{((2*3)+(2*1000)+100000)\mu sec} \approx 10$. Note that the costs for this architecture are $\Delta_{agent} \gg \Delta_{mt}, \Delta_{st} \gg \Delta_{me}, \Delta_{ma}, \Delta_{ss}, \Delta_{se}$ which leads to hierarchical costs and limitations for computational steering. This conservative estimate demonstrates that the frequency of interactive computational steering with a traditional system approaches 10 hz. By forcing all monitoring events and steering commands to flow through the entire loop, first generation steering systems limit the fre-

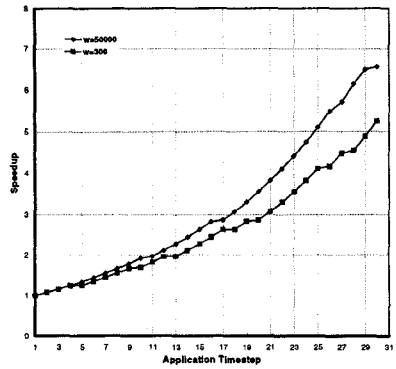


**Figure 3. $\omega_{steering}$ for heat diffusion.**

quency with which they can respond to applications.

**$\omega_{steering}$ for Magellan steering system:** ACSL vastly improves $\omega_{steering}$ by moving the steering commands as close to the application as possible. ACSL allows the steering system to interpret steering commands and disseminate them throughout the steering system with several benefits. First, because the steering command is "close" to the application, the steering system can receive monitoring information at higher rates and execute steering decisions at a higher frequency. Second, because the monitoring information is digested early in the process, the steering infrastructure is not overwhelmed with data. Third, flexibility is maintained because ACSL does not force migration of steering commands to the system, it only does so when appropriate. For Magellan on the heat diffusion code, $\Delta_{steercycle}$ was measured at 12 $\mu secs$ when steering is performed directly at the steering server. This time gives a $\omega_{steering} \approx 83333$. Forcing Magellan to route data through the client raised $\Delta_{steercycle}$ to 3 msecs dropping the steering frequency to $\omega_{steering} \approx 333$.

Figure 3 demonstrates the importance of steering frequency using performance steering results. The heat diffusion application is executed with steering changing the load as described earlier. Two different $\omega_{steering}$ settings are used for the application. Although both of the applications eventually achieve a reasonable load, the higher $\omega_{steering}$ setting of 50000 achieves the balance about 10 timesteps before the lower setting.

## 5 Conclusions

ACSL provides language constructs for computational steering that dramatically increases the frequency of the steering rate for an application as well as decrease the data deluge. This increase is brought about by the ability of the steering system to interpret steering commands stated with ACSL at runtime and then translate such commands into specific steering and monitoring instructions so that certain performance criteria are optimized. The prototype described in this paper, called Magellan, which uses ACSL specifications demonstrates the viability of this approach by increasing the steering frequency for a sample application by a factor of approximately 250, from a steering frequency $\omega_{steering}$ of 333 to almost 83333. Such improvements are attained in part because the system accomplishes steering 'close' to the application and well as limiting the amounts of data transferred across the steering infrastructure. ACSL is not incongruent with graphical user interfaces. GUIs can use the language to control the steering system and even realize a performance improvement due to control of the *data deluge.*

## References

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

[2] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter. Falcon: on-line monitoring and steering of large-scale parallel programs. In *Proc. Frontiers '95*, pages 422–9, 1995.

[3] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *SIGPLAN Notices*, 29(9):140–8, 1994.

[4] B. Mukherjee and K. Schwan. Experiments with a configurable lock for multiprocessors. In *Proc. 22nd Int'l Conf. on Parallel Processing*, pages 205–208, 1993.

[5] J. Mulder and J. van Wijk. 3d computational steering with parametrized geometric objects. In *Proc. Visualization '95*, pages 304–11, 466, 1995.

[6] D. Ogle, K. Schwan, and R. Snodgrass. Application-dependent dynamic monitoring of distributed and parallel systems. *IEEE Trans. on Parallel and Distributed Systems*, 4:762–778, 1993.

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

[8] M. Simmons, A. Hayes, J. Brown, and D. Reed, editors. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society Press, 1996.

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

[10] J. Vetter and K. Schwan. High performance computational steering of physical simulations. 96-21, College of Computing, Georgia Tech, 1996.

[11] J. Vetter and K. Schwan. Models for computational steering. In *Proc. Int'l Conf. on Configurable Distributed Systems*, 1996.