

# Experiences using Computational Steering on Existing Scientific Applications\*

Jeffrey S. Vetter<sup>†</sup>

## Abstract

Computational steering remains an impressive yet challenging opportunity for computational scientists. In this paper, we examine our practical experiences when we used computational steering on four existing scientific applications. In particular, this paper enumerates the concrete steps required for specific computational steering on each individual application. We then provide an evaluation of steering effects on application performance, and also present necessary modifications to the application. Finally, we provide a summary of common issues distilled from these experiences.

## 1 Computational steering

Simulations are playing an increasingly critical role in all areas of science and engineering. As the uses of these simulations expand, the demand grows for high performance computing of increasing power, flexibility, and utility. Interactive computational steering is one way to increase the utility of high performance simulations for scientists because it allows them to drive the scientific discovery process and interact with their data. They can interpret what is happening to data during simulations and steer calculations in close-to-real-time: they can change parameters, data-sets, resolution, and representation; then, see the effects.

In this work, we explore some of the challenges for creating an efficient software infrastructure for computational steering. Specifically, we consider two important requirements for interactive computational steering: steering latency and application perturbation. High latency results in poor decision quality and low steering frequency. High application perturbation conflicts with the original intent of high performance applications.

In addition, our approach assumes three basic constraints. First, we attempt to minimize the changes to the target application and system. Literally, we try to minimize the number of changes to the source code and to preserve the application's original execution environment. Second, we strive to minimize the differences in the infrastructure across applications. Our objective is to develop a general set of mechanisms to change applications even though applications exhibit a variety of designs and behaviors (e.g., data decompositions versus functional decompositions). These mechanisms used in conjunction with specific policies and metaphors provide application-specific steering. Finally, we try

---

\*A portion of this work was completed while Vetter was a PhD candidate at the Georgia Institute of Technology. The Georgia Tech work was funded in part by a NASA Graduate Student Researchers Program Fellowship for Vetter, by NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846, and by Los Alamos National Lab. At Illinois, this work is funded in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049 (SIO Initiative), F30602-96-C-0161, and DABT63-96-C-0027, by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202, and by the Department of Energy under contracts DOE B-341494, W-7405-ENG-48, and 1-B-333164.

<sup>†</sup>Department of Computer Science, University of Illinois, Urbana, IL.

to optimize both the performance of the application in light of infrastructure requirements and the responsiveness of the infrastructure itself.

**Key insight and contribution.** The primary contribution of this work is an analysis of how to use language-directed computational steering [12] on a group of existing applications. We also delve into application-specific issues about steering these applications. Essentially, this paper has four contributions. First, we present well-defined steering maneuvers for each application. Second, we furnish specific details of how we modified four individual applications for steering. Third, we empirically evaluate each application for performance and steering latency. Finally, we distill key issues from these experiences of transforming existing applications into steer-able form.

**Related work.** Many successful projects have tackled the concept of steering including SCIRun [9], CUMULVS [4], VASE [6], FALCON [5], Autopilot [11], and others [14]. Additional endeavors have focussed on steering of particular application domains or of one customized application. Some representatives are SMD [8], the work by Beazley and Lomdahl [1], a case study of seismic tomography [3], and a study of simulated annealing [7]. For more related work, the reader should consult these reviews [14, 10, 2]. With respect to this related research, this work focuses on developing efficient, general techniques to allow a range of existing applications to capitalize on computational steering.

**Paper organization.** Section 2 sketches an overview of language-directed computational steering. Section 3, then, details an evaluation of using steering on several applications with our steering framework. Section 4 presents common issues distilled from the evaluation. Finally, Section 5 summarizes the paper and furnishes some future research directions.

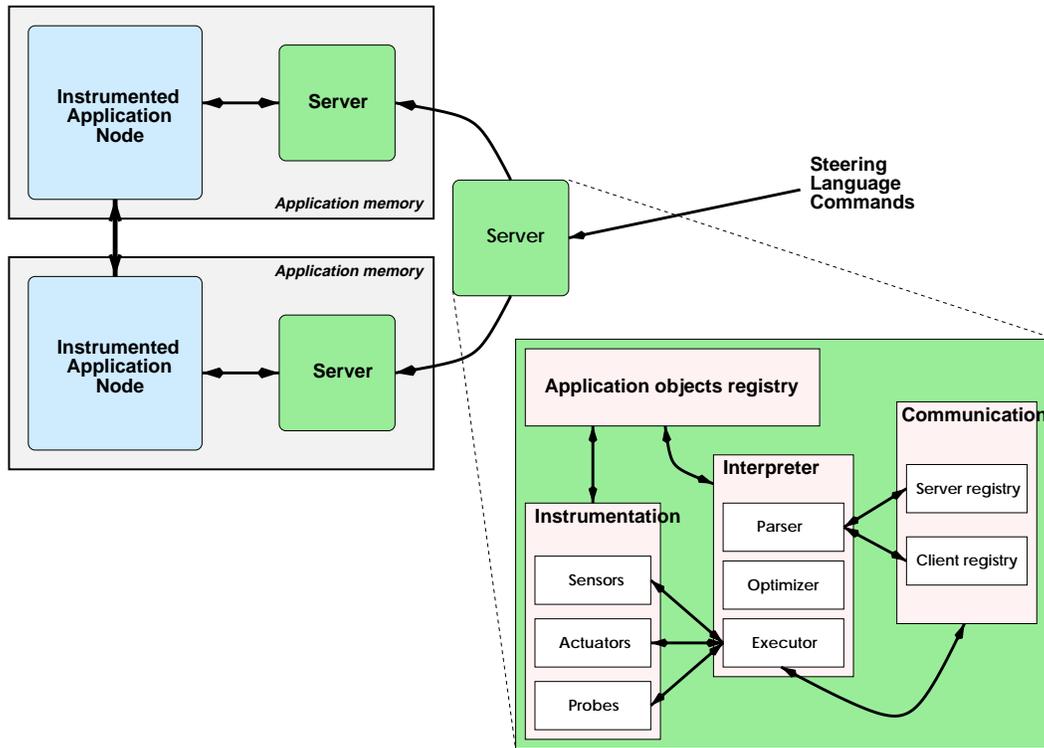
## 2 Steering overview

Figure 1 illustrates *Magellan* [12]: a system for language-directed computational steering. The Magellan system is composed of three basic pieces: steering servers, a steering language, and application instrumentation. The primary advantage of language-directed steering is the possible optimization of user requests [13]. Because the server evaluates steering requests at runtime, it can make intelligent tradeoffs that improve the overall efficiency of each request. This overview omits details on server construction and language optimization that are presented in both [12] and [13].

The servers represent the core components of this system. As Figure 1 illustrates, each server is composed of an interpreter and optimizer, an application object registry, communication support, and controls for monitoring and steering mechanisms. The interpreter receives steering requests from clients and other servers. Upon receipt of requests, the interpreter parses and optimizes them. Finally, it uses these optimized requests to control instrumentation within the application for monitoring and steering.

The application object registry contains information about all objects available from the application for steering; the application declares these objects at runtime via a registration call. Typical object information includes the object’s address, data type, size, and flags associated with the object. The server registry contains connection and management information on additional servers, if any.

The server’s interpreter accepts commands in the form of a mini-language, which is specified as a LALR(1) grammar. This language approach supplies both flexibility and

FIG. 1. *Software architecture of Magellan.*

abstraction. The servers are flexible because they port across applications without recompilation. That is, although the servers remain unchanged, the application instrumentation and language commands are different and dynamic across applications. Regarding abstraction, the language provides steering abstractions that allow the interpreters to apply general optimizations across all application steering requests.

User-inserted instrumentation identifies application-level components that are available to the steering system for observation and structured changes. At runtime, each individual instrumentation point coordinates with the Magellan server. These instrumentation points serve a very important purpose in this system: they define *what* data the server can access, *when* it can safely observe that data, and *how* it can make structured changes to application components. Other techniques, such as debugging, executable editing, and profiling do not provide this semantic information about the application. Note, however, that the use of application-specific instrumentation does not preclude use of other techniques such as profiling or executable editing. This instrumentation can also provide pre-conditions and post-conditions that call application functions. These functions allow the instrumentation to engage in more complex behaviors (e.g., using the application’s mutex locks to protect data) than normal instrumentation.

To make these abstractions more concrete, we show application instrumentation for Barnes-Hut in Figure 2 and one possible steering command in Magellan’s language form in Figure 3. When these two are combined with a runtime Magellan server, the result is a dynamically steer-able application.

```

1 /* advance my bodies */
2 for (pp = Local[ ProcessId ]. mybodytab;
3      pp < Local[ ProcessId ]. mybodytab
4      + Local[ ProcessId ]. mynbody;
5      pp++) {
6     p = *pp;
7     MULVS(dvel, Acc(p), dthf);
8     ADDV(vell, Vel(p), dvel);
9     MULVS(dpos, vell, dtime);
10    ADDV(Pos(p), Pos(p), dpos);
11    ADDV(Vel(p), vell, dvel);
12    /* instrumentation point -- bhIP */
13    bhIP(ts, &p->mass,
14         &p->pos[0], &p->pos[1], &p->pos[2]
15         &p->vel[0], &p->vel[1], &p->vel[2]
16         &p->acc[0], &p->acc[1], &p->acc[2]);
17    for (i = 0; i < NDIM; i++) {
18        if (Pos(p)[i] < Local[ ProcessId ]. min[i]) {
19            Local[ ProcessId ]. min[i] = Pos(p)[i]; }
20        if (Pos(p)[i] > Local[ ProcessId ]. max[i]) {
21            Local[ ProcessId ]. max[i] = Pos(p)[i]; }
22    }
23 }

```

FIG. 2. *Application-specific instrumentation in Barnes-Hut.*

### 3 Evaluation

For this evaluation, we selected three different applications from the Splash2 suite and one message-passing PDE computation. Although, we selected these applications because each has a different goal, we also wanted applications with different underlying computations and data structures. For example, each application decomposes the data domain, but some of the decompositions are quite different. These differences are important for computational steering and they impact the potential set of steering maneuvers available with each application. In regard to maneuver selection, we focussed on steering maneuvers that accessed primary components of the application and seemed reasonable.

The first three applications are part of the Splash2 benchmark suite [15]: Barnes-Hut, Water-Spatial, and Ocean. All three applications use threads for parallelism and they assume globally shared memory. The fourth application—a conjugate-gradient heat diffusion simulation—is a message passing application built on FORTRAN 90 and MPI. The accompanying Splash2 measurements were performed on a dual-processor Sun Ultrasparc-60 using Sun OS 5.6. The CG Heat measurements used a cluster of Ultrasparc-30s connected with 100Mb Ethernet. All applications and steering system components were compiled with optimization `-O`.

#### 3.1 Barnes-Hut

Barnes-Hut simulates evolution of galaxies using hierarchical N-body methods. Steering in this example moves a cluster of bodies bounded by  $(0, 0, 0)$  and  $(5, 5, 5)$  to a new location 5.0 units along the  $+x$  axis during the third timestep.

Internally, Barnes-Hut uses threads to distribute work across the data domain where

```

1 when((( bhx > 0) && (bhx < 5.0)) # constrain to a 3-D region
2   && ((bhy > 0) && (bhy < 5.0))
3   && ((bhz > 0) && (bhz < 5.0))
4   && (ts == 0.075))      # during timestep 3
5 {
6   bhx = 5 + bhx;         # change x displacement
7 }

```

FIG. 3. *Magellan language command for Barnes-Hut steering maneuver.*

each thread receives a linked list of bodies it has been assigned. A global timestep synchronizes all threads. Between these synchronizations, each thread can process bodies in its list with limited regard to other threads. Figure 2 shows the update loop with an application-specific instrumentation point. These lists contain pointers to structures that represent all the properties for each body in the simulation including mass, position, velocity, and acceleration. A `for` loop walks the list and advances the bodies to their new position using their calculated velocity and acceleration. At the end of the update for each body, the thread updates its local bounding box with the new parameters of the most-recently computed body.

To accomplish the aforementioned steering, at compile time, we insert an instrumentation point immediately after the update to each body but before the update to the bounding-box. Figure 2 shows this instrumentation point as `bhIP`. Note that as each body is processed by the `for` loop, this one instrumentation point gives the steering infrastructure access to all bodies and all properties of each body. Hence, the steering infrastructure can manipulate each body when the user issues interactive requests. Figure 3 illustrates the Magellan command to accomplish our steering maneuver.

The changes to the target application were limited to about 20 lines of source code and to linking the steering library with the application. The original application contains about 3044 lines of source code. The instrumentation points required declarations in both the application and the steering system; this aspect involves a short description (about 20 lines) in an instrumentation definition script. This instrumentation can be generated manually; however, a translator makes the process more convenient and less error prone. Regarding performance, the latency of instrumentation execution directly impacts the runtime of Barnes-Hut. In our experimental evaluation, the original, uninstrumented Barnes-Hut ran in 85 seconds as illustrated in Figure 4. The addition of instrumentation and library, but no steering, increased the runtime to 87 seconds. Then, with the steering maneuver active, we introduced additional execution cost into the instrumentation point, which increased the execution time to 88 seconds.

This example illuminates the advantages of using application-specific instrumentation. Clearly, a utility could be developed to allow these changes but the utility must have intimate knowledge of the application structure and execution. For this example, instrumentation is straightforward and convenient; and, the runtime is not adversely affected by the mere presence of instrumentation.

### 3.2 Water-spatial

Water-Spatial [15] evaluates forces and potentials that occur over time in a system of water molecules. It imposes a uniform 3-D grid of cells on the problem domain, and

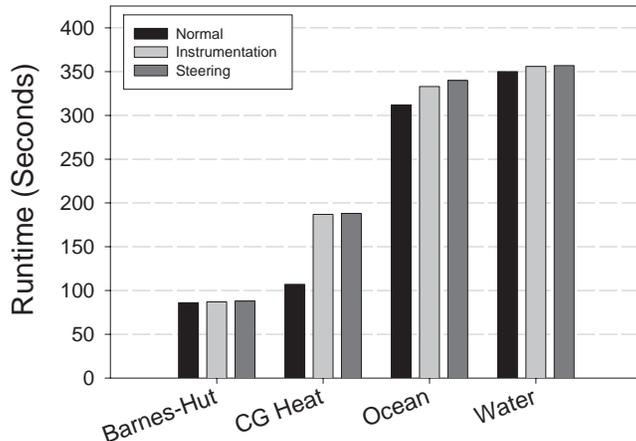


FIG. 4. *Impact of steering on application runtime.*

forces and potentials are calculated using an  $O(n)$  algorithm. A predictor-corrector method integrates the motion of the water molecules over time. Steering for this example increases the velocity by 10% for hydrogen and oxygen molecules in a cuboidal region bounded by  $(0, 0, 0)$  and  $(10, 10, 10)$ . Water is similar to Barnes-Hut because it stores data as linked lists; however, data for each molecule is a structure that is much larger at 500 bytes than the corresponding elements in Barnes-Hut that are 112 bytes. Similar to Barnes-Hut, we choose an instrumentation location in a loop. As the loop walks the list, the instrumentation gathers information from each body and makes changes to the bodies values, if requested.

Water required changes similar to Barnes-Hut. Changes to the source code were limited to 17 lines of code that included initialization and termination calls for the steering server and instrumentation points. We also had to create a global variable containing the current application timestep that merely reflected the application’s local, timestep variable. Figure 4 reveals that the runtime of the application increases only moderately from 350 seconds to 356 seconds and 357 seconds for instrumentation and steering, respectively.

### 3.3 Ocean

Ocean [15] is a multi-grid application that simulates eddy currents in an ocean basin using finite differencing computational fluid dynamics and a square grid layout. It uses a Red-Black Gaussian solver that generates many transient parameters during multiple phases that are used to integrate the simulation over each timestep. Threads provide concurrency and each thread is allocated one square portion of the grid. Threads are loosely synchronized between timesteps. As this simulation evolves, it maintains error values to track the success (or failure) of the solution. Although this computation does have global timesteps and a square-grid decomposition, the control flow of the multi-grid calculation is data-dependent. Steering alters the simulation parameters, a subregion of one  $512^2$  grid, for Laplacian integration during one timestep.

Although the data structures of Ocean are essentially arrays that are easy to access, the control flow of this multi-grid application is data dependent. Further, the arrays can become

```

1      do l=1, numcell
2 c      Instrumentation point -- cg_ip
3          i=mod((l-1), imax)+1
4          j=mod((l-1)/imax, jmax)+1
5          k=(l-1)/(imax*jmax)+1
6          i=i+ioff
7          j=j+joff
8          k=k+koff
9          call cg_ip(time, i, j, k, tev(l))
10
11         mtrxd(l)=mass(l)*cv(l)
12         vctr(l)=mass(l)*cv(l)*tev(l)
13         tevold(l)=tev(l)
14     enddo

```

FIG. 5. *Application-specific instrumentation in CG-Heat.*

quite large (e.g.,  $512^2$  doubles), resulting in large data volume for the steering system. In addition, the solver must be instrumented in both the **BLACK** and **RED** portions of the solver to cover the entire grid. As in the Barnes-Hut instrumentation example and because the solver maintains an error value, the instrumentation point allows the steering system to change the value immediately before the calculation of this new value, and this preserves the integrity of the multi-grid error calculation. Also, because the multi-grid calculation is data dependent, one instrumentation point within the code might not encounter grid regions the same number of times during one timestep. Each instrumentation point must also indicate the location within the grid—an x and y coordinate—being accessed along with the current application timestamp.

Figure 4 shows that the runtime of the application increases from 312 seconds to 333 seconds and 340 seconds for the instrumentation and the steering, respectively. The changes to the source code included server initialization and termination; however, due to the nature of the multi-grid solver, we had to replicate the instrumentation point in four places, so that the entire grid would be covered during one timestep. This instrumentation added about 40 lines of code to the application. The perturbation is primarily a result of the increased number of hits on the instrumentation points. During this run, Ocean’s instrumentation points are hit over 165 million times, far more hits than the other applications. Moving the instrumentation or targeting the instrumentation on a larger granularity of application data would almost certainly reduce perturbation.

### 3.4 CG heat diffusion

The heat diffusion simulation solves the implicit diffusion PDE using a conjugate gradient solver over each timestep. This code is built on FORTRAN 90 and MPI, and it serves as our example for distributed memory codes. We ran this code on a cluster of four Ultrasparc-30s. Steering in this example changes the heat quantity in sub-regions of the 3-D space.

This code required slightly more modifications than the previous codes. As Figure 5 shows, we had to add calculations that decoded each cell’s position preceding the instrumentation point. We also had to move 4 variables into the common block, so that they were available throughout the application to calculate the global position of each heat cell. Because we include this global position in the instrumentation point, the master steering

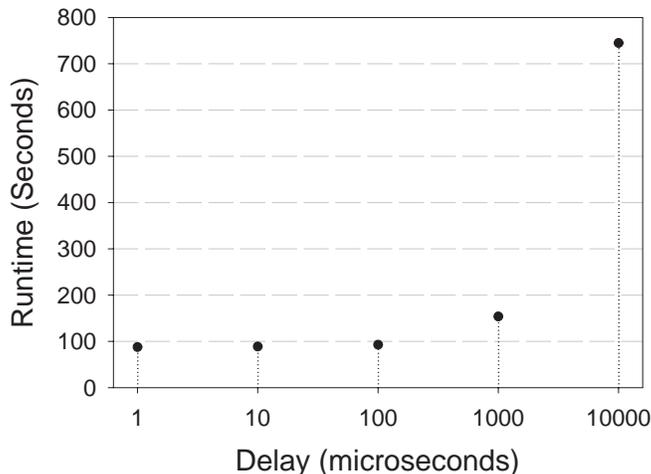


FIG. 6. *Result of steering latency on Barnes-Hut runtime.*

server has a global view of the 3-D space. This global view simplifies the implementation of steering requests when they span data regions involving more than one MPI node.

As illustrated in Figure 4, the runtime of the CG heat application suffered more than the previous applications when steering was enabled. This degradation occurs because the configuration of the experiment platform changes from a dual-processor—as used in the preceding examples—to a network of single-processor systems. Every time an instrumentation point is called and steering is enabled, the application transfers control to the steering server. In a single processor system using kernel-level pthreads, this translates into a context switch. The dual processor system did not suffer the same consequences because both the server thread and application thread were running concurrently on separate processors and they communicate via application memory.

Regarding MPI, our version of MPI was MPICH 1.1 and it is not thread-safe; however, our steering communication does not use MPI. Our servers use raw TCP/IP connections that are outside the domain of the application’s MPI library calls.

### 3.5 Latency impact on application perturbation

In Magellan, the steering server can optimize steering requests to improve latency by migrating steering requests *close* to the application data. This migration usually translates to a lower perturbation that each instrumentation point injects into the application. Figure 6 shows how a higher-latency (delay) in the steering system translates into an increased runtime for Barnes-Hut. The steering configuration remained the same as in the earlier example in Section 3.1, but we added a delay to the steering action. Interestingly, the runtime did not increase dramatically until the delay increased over 100 microseconds. Because Magellan has at least one server within the application’s address space, many steering operations can occur without crossing expensive process boundaries or within several microseconds.

## 4 Experiences

Several issues appeared repeatedly as we applied our steering infrastructure to these applications. From these experiences, we distill the following.

1. *Each application requires well-placed instrumentation to allow reasonable examples of steering.* As illustrated these applications, the opportunities to observe or change application data are sometimes non-trivial. However, once the instrumentation is in place, a user can issue a variety of dynamic commands to steer the application.
2. *Steering infrastructure is isomorphic to the application structure.* Potential steering opportunities reflect application structure and instrumentation placement. The two different types of applications presented here, namely shared memory and distributed memory, have different requirements for a steering system architecture even though the instrumentation remains consistent.
3. *Software instrumentation can offer some easy and beneficial opportunities for steering and tracking.* All of the examples presented in this paper are straightforward examples of steering scientific applications without massive investments required to change the code. Application-specific software instrumentation allows developers to insert precise opportunities for the steering system to observe or change the target application.
4. *The presence of a steering infrastructure does not prohibit high performance.* Most users have considerable investments in code optimization. A primary concern of these users is that the addition of steering infrastructure to a code will reduce performance by disabling optimizations and perturbing the application. With our infrastructure and evaluation, we have demonstrated that these applications do experience reasonably small performance degradation. However, our examples do not suffer from extreme performance problems in light of the additional interactivity provided by the steering infrastructure.
5. *The steering maneuvers for our applications use the application's notion of time to synchronize their steering changes.* Most applications have internal synchronization mechanisms that control global time. By adding a conditional that uses this application time to our steering maneuvers, we can limit and coordinate the steering changes without relying on complex external synchronization mechanisms.

## 5 Conclusions and future work

In this paper, we examined our practical experiences when we used computational steering on four existing scientific applications. This perspective used a language-directed steering system named Magellan. In particular, we enumerated the concrete steps required for specific computational steering on each individual application. These steps included adding instrumentation to the application and developing steering commands, which control the instrumentation. We evaluated the effects of steering on application performance and steering latency. Application performance does incur a small overhead from the addition of the steering infrastructure; however, this degradation is reasonably small. Finally, we provide a summary of common issues distilled from these experiences. Most important among these issues is the dependence of the steering system on the application structure and associated instrumentation placement.

We are currently investigating a variety of visual metaphors and algorithmic policies for controlling target applications. In addition, the current system does not provide solid support for managing invalid steering requests either within the instrumentation or the servers. We are also trying to improve the server infrastructure with aggressive memory management, more language optimizations, and load balancing between the server and the multiple application nodes.

## Acknowledgments

We would like to thank both the Pablo group at University of Illinois, led by Dr. Dan Reed, and the Falcon group at Georgia Tech, led by Dr. Karsten Schwan, for their discussions and insights. We are grateful to Los Alamos National Laboratory and Stanford University for providing the heat diffusion code and the Splash2 suite, respectively.

## References

- [1] D. M. Beazley and P. S. Lomdahl, *Lightweight computational steering of very large scale molecular dynamics simulations*, in Proc. Supercomputing 96, Pittsburgh, 1996.
- [2] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang, *Toward visual programming languages for steering scientific computations*, IEEE Computational Science & Engineering, 1 (1994), pp. 44–62.
- [3] J. Cuny, R. Dunn, S. Hackstadt, C. Harrop, H. Hersey, A. Malony, and D. Toomey, *Building domain-specific environments for computational science: a case study in seismic tomography*, in Proc. Europar, 1996.
- [4] G. A. Geist, II, J. A. Kohl, and P. M. Papadopoulos, *CUMULVS: providing fault tolerance, visualization, and steering of parallel applications*, Int'l Jour. Supercomputer Applications and High Performance Computing, 11 (1997), pp. 224–35.
- [5] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter, *Falcon: On-line monitoring and steering of parallel programs*, Concurrency: Practice and Experience, 10 (1998), pp. 699–736.
- [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, 1993, pp. 560–9.
- [7] E. Kraemer and J. Wallis, *Visualization and interactive steering of simulated annealing*, in Proc. SPDP'96 Workshop on Program Visualization and Instrumentation, New Orleans, 1996.
- [8] J. Leech, J. F. Prins, and J. Hermans, *SMD: visual steering of molecular dynamics for protein design*, IEEE Computational Science and Engineering, 3 (1996), pp. 38–45.
- [9] S. G. Parker and C. R. Johnson, *SCIRun: a scientific programming environment for computational steering*, in Proc. Supercomputing 95, San Diego, 1995, pp. 1–1.
- [10] S. G. Parker, C. R. Johnson, and D. Beazley, *Computational steering software systems and strategies*, IEEE Computational Science & Engineering, 4 (1997), pp. 50–59.
- [11] R. Ribler, J. Vetter, H. Simitci, and D. Reed, *Autopilot: adaptive control of distributed applications*, in Proc. Seventh IEEE Int'l Symp. High Performance Distributed Computing (HPDC), Chicago, 1998, pp. 172–179.
- [12] J. Vetter and K. Schwan, *High performance computational steering of physical simulations*, in Proc. Int'l Parallel Processing Symp., Geneva, 1997, pp. 128–132.
- [13] ———, *Optimizations for language-directed computational steering*, in Proc. Int'l Parallel Processing Symp., San Juan, 1999.
- [14] J. S. Vetter, *Computational steering annotated bibliography*, SIGPLAN Notices, 32 (1997), pp. 40–44.
- [15] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, *The SPLASH-2 programs: characterization and methodological considerations*, in Proc. 22nd Annual Int'l Symp. Computer Architecture, 1995, pp. 24–36.