Optimizations for Language-Directed Computational Steering

Jeffrey Vetter*
Department of Computer Science
University of Illinois
Urbana, IL, 61801

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA, 30332

Abstract

Two important requirements for interactive computational steering are low steering latency and minimal application perturbation. The key insight of this research is that precise, interpretable descriptions of steering commands enable runtime optimization for steering perturbation and latency. Within our structured approach for specifying steering requests, we define three optimizations for latency and perturbation while maintaining the fidelity of the user's request. To validate our approach, we have constructed an operational prototype for computational steering. Our empirical evaluations demonstrate the opportunities for optimization inherent in language-directed computational steering.

1. Computational Steering

Interactive computational steering is one way to increase the utility of high performance simulations for scientists: they can interpret what is happening to data during simulations and steer calculations in close-to-real-time. Two important requirements for interactive computational steering are low steering latency and minimal 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 this work, we focus on techniques to control steering latency and application perturbation in light of seemingly arbitrary user requests.

The key insight of this research is that a language-based description of computational steering enables optimization for perturbation and steering latency. Within our language-directed approach, we define three optimizations for latency and perturbation while maintaining the fidelity of user requests. To explore these issues, we have constructed an operational prototype for computational steering that capitalizes on this language-directed approach. We empirically evaluated the prototype with multiple high performance applications.

We provide a comprehensive review of related work in [11]; however, the research most relevant to this work includes the systems VASE [5], Falcon [4], AutoPilot [9], SCIrun [8], Cumulvs [3], and various customized environments for steering [2, 6, 7, 1]. Also, Taylor and associates [10] reveal performance problems of interactivity in a complex visualization environment. This work, in contrast to our earlier work [13], demonstrates that language-directed steering can accommodate a host of optimizations to improve latency and perturbation, and can provide a general optimization framework.

Section 2 describes our approach to computational steering including key design issues. Section 3 presents an evaluation of the three optimizations. Section 4 concludes the paper.

2. System Description

Figure 1 illustrates *Magellan*—our prototype computational steering system—where the major components are steering servers. The application remains intact except for user-inserted instrumentation.

The basic components of the steering server, as illustrated in Figure 1, are its interpreter, optimizer, ap-

*Email: jsv@cs.uiuc.edu, schwan@cc.gatech.edu. 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.

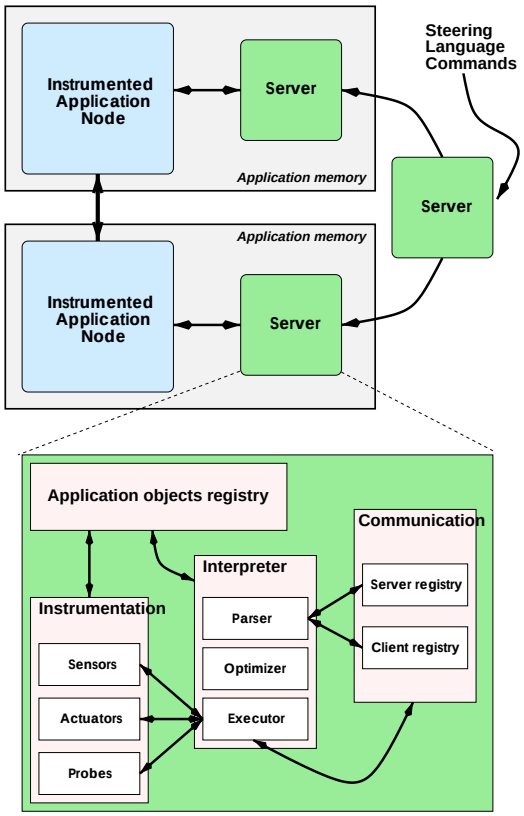

Figure 1. Software architecture of Magellan

plication object registry, communication support, and controls for monitoring and steering mechanisms. The interpreter receives steering requests from clients and other servers. Then, it parses and optimizes these requests. Finally, it uses these requests to control application instrumentation for steering. The application object registry contains information about all objects available in the application for steering; the application declares these objects at runtime via a registration call.

Magellan servers are multi-threaded, asynchronous interpreters that accept steering commands in our mini-language at runtime. The advantages of this language approach are twofold. First, the interpreter accepts complex steering commands at any point during runtime; this provides reasonable flexibility. Second, the interpreter can optimize these commands using knowledge about all steering commands, the application instrumentation, and the current server configuration.

When the interpreter receives a command, it takes three steps. First, the interpreter parses the command into tokens and builds a syntax tree.

Then, the server optimizes this AST. These optimizations, discussed later in Section 3, annotate this tree and possibly, perform minor reorganizations of the nodes. Finally, the interpreter goes to the execution phase where it walks this syntax tree interpreting each node and interacting with application instrumentation and the application object registry as necessary.

During the optimization phase, the interpreter may make multiple passes over the syntax tree to try alternate scenarios and to test various optimizations. Once the optimizer has completed, the syntax tree is annotated with additional information about how to proceed with the execution phase. The optimizer can also insert additional nodes into the tree and perform modest reorganization of nodes.

Optimization is especially important when steering commands are combined with control structures [13]. Control structures tell the interpreter that the command will be applied when some condition is satisfied. Instead of executing the command once, the command might be executed thousands of times. Therefore, the advantages of optimizing one steering command grows proportionally. By optimizing the perturbation and latency of conditional checking, the system substantially improves over the non-optimized case.

Magellan interpreters control applications through instrumentation, which is required by computational steering because it needs more information than is provided by techniques for debugging, monitoring, or executable editing. A steering system not only needs to know *what* data to access, but *how* and *when* it can safely alter the data. User-inserted annotations provide this additional semantic information about application data and control that are generally lost when source code is compiled. These instrumentation points are dynamically controlled by the steering server through instrumentation control mechanisms. Effectively, these annotations tell the steering server *what* data is available for steering and *how* it can be safely steered.

Figure 2 illustrates the instrumentation point for the Barnes-Hut application. This instrumentation point does not define *how* the instrumentation point will extract or change data, but it does allow the steering server to dynamically attach an instrumentation mechanism to this instrumentation point.

Our approach has seven possible instrumentation mechanisms that can be bound to an instrumentation point. They are probes, synchronous probes, tracing sensors, sampling sensors, snapshot sensors, queuing actuators, and idempotent actuators. These application instrumentation mechanisms provide a variety of features and result in an array of perturbation and latency characteristics, which are evaluated in [14].

```
for (pp = Local[ProcessId].mybodytab;
     pp < Local[ProcessId].mybodytab
       +Local[ProcessId].mynbody;
     pp++) {
   p = *pp;
   MULVS(dvel, Acc(p), dthf);
   ADDV(vel1, Vel(p), dvel);
   MULVS(dpos, vel1, dtime);
   ADDV(Pos(p), Pos(p), dpos);
   ADDV(Vel(p), vel1, dvel);
   bhIP(ts,&p->mass,    /* instrumentation */
      &p->pos[0], &p->pos[1], &p->pos[2]
      &p->vel[0], &p->vel[1], &p->vel[2]
      &p->acc[0], &p->acc[1], &p->acc[2]); }
   for (i = 0; i < NDIM; i++) {
      if (Pos(p)[i]<Local[ProcessId].min[i]) {
        Local[ProcessId].min[i]=Pos(p)[i]; }
      if (Pos(p)[i]>Local[ProcessId].max[i]) {
        Local[ProcessId].max[i]=Pos(p)[i]; }}}
```

Figure 2. Example of application-specific instrumentation in the Barnes-Hut application.

Given this design, Magellan has four interesting capabilities. First, it is possible to choose from multiple instrumentation strategies for specific application components or for certain steering commands. Second, any number of steering servers may be created for each application, depending on the size and structure of the target application. Third, servers themselves are lightweight entities, executing as threads within the application's address space, thereby providing efficient access to the instrumentation residing in the application. Fourth, any number of steering clients may be created, serving as remote steering agents, as analysis engines for steering purposes, or as visualization engines.

3. Experimental evaluation

This experimental evaluation focuses on demonstrating the advantages of our language-directed approach to steering with three different optimizations. As mentioned in Section 1, we use three different scientific applications for this evaluation. Basically, we apply an application-specific steering command to the Magellan server while it is attached to each application and measure the effects of command optimization on perturbation and latency.

Our optimizations focus on two metrics: application perturbation and steering latency. We define *application perturbation* as the change in the runtime of the application from its normal, optimized execution runtime. We define *steering latency* as the time from when the steering system initially monitors an application data value until it makes the change in the application in response to this latest monitored value. Put simply, steering latency is the closed-loop time that a steering system can receive new data with a monitoring mechanism, decide on some steering action, and then, change the application with a steering mechanism. Different combinations of mechanisms produce different application perturbations and steering latencies, which provides the opportunities for optimization.

Clearly, our steering system could optimize metrics other than latency and perturbation given methods to measure them and predict their characteristics. However, these two metrics are critical to a steering system's execution. Steering latency dictates how quickly a steering system can respond to changes in the target system. Application perturbation induced by the steering system must not undermine the user's original goals for high performance.

The following three applications—Barnes-Hut, Ocean, and a 3-D, 27-point stencil calculation—represent a range of computational techniques for scientific computing and are from different application domains. We selected these steering maneuvers because they exercise major components of each application as detailed in [12].

Barnes-Hut [15] simulates evolution of galaxies using hierarchical N-body methods. Steering in this example moves a cluster of bodies from one region of the simulation to another while the simulation executes. Ocean [15] simulates eddy currents in an ocean basin using finite differencing computational fluid dynamics and a square grid layout. Steering alters the simulation parameters for Laplacian integration between phase 8-9. The heat diffusion application [13] uses a 27 point 3D stencil time-stepped calculation to simulate the diffusion of heat. Steering allows users to interactively introduce heat into the application.

3.1. Optimizations

Optimizations exploit knowledge of both the system architecture and the language description of steering requests. We identify three optimizations in the context of the Magellan architecture and language. First, the *spatial locality* optimization relocates steering requests to improve steering latency. Second, the optimization for *instrumentation mechanism selection* allows the interpreter to choose the mechanism that efficiently satisfies the user request with respect to latency or perturbation. Finally, *concurrent prefetch* overlaps high latency steering operations.

These optimization procedures are heuristics and, therefore, do not always provide *optimal* solutions. Further, each optimization is not applicable to every steering command. In most cases, information pro-

vided by the application about each application object, which is stored in the application registry, limits the types of optimizations applicable to each steering command. Effectively, the optimizer uses this registry information to prune its search.

3.2. Spatial locality

Magellan analyzes steering commands for spatial locality. This optimization allows the system to migrate commands close to the server with the necessary information to make decisions. The spatial optimization works on the AST as illustrated in Figure 3. The algorithm, first, determines the location of all the terminal nodes in the AST, and it creates the set `location_set`. Then, the code checks the cardinality of this set. If the cardinality is one and the location is the current server, then the server just executes the command locally. If the cardinality is one and the location, `M`, is different than the current server, the server forwards the command to the server `M` because that server can execute the command locally. Otherwise, if the cardinality is not one, then the command must be executed locally. With cardinality zero, the command does not involve application objects and can be executed locally. If the cardinality is greater than one, then the current server must request the values from disparate servers, and compute the expression locally.

This optimization exploits knowledge of the system architecture because it realizes that a server closer to the data will generally have a smaller steering latency; and, then, it will be able to execute the commands at a much higher frequency.

To judge the impact of application perturbation caused by application instrumentation, we performed steering on the example applications with the same instrumentation and steering commands, while injecting additional steering latency for each command. Explained another way, we increased the steering latency which caused the instrumentation points within the application to execute longer. Across tests, each application uses exactly the same steering maneuver but with additional latency added to demonstrate the effect of perturbation on application runtime. With Magellan, the latency of the steering action is less than 1 microsecond. The induced latencies on instrumentation essentially simulated steering systems with high latencies.

Figure 4 illustrates the effects of high server latencies on application runtimes while steering each application. As expected, when these latencies increase from 1 to 10000 microseconds, the perturbation on the target application increases. Interestingly, however, most

```
spatial()
  location_set = 0
  n = number of terminal nodes in AST
  for i = 1 to n do
    location_set = location_set
      || Location(terminal(n))
  if ( cardinality(location_set) == 1 )
    M = Member(location_set) ## only one member
    if ( M== this_server )
      execute command locally on this_server
    else
      forward command to server M
  else ## cardinality(location_set) > 1 or == 0
    execute command locally on this_server

mech-selection(n, types, new_latency_min, new_pert)
  mech_set = Mechanisms(terminal(n)) && types
  latency_min = MAXDBL
  foreach i in mech_set
    if( Latency(terminal(n),i) < latency_min )
      Set_mechanism(terminal(n), i)
      new_latency_min = Latency(terminal(n),i)
      new_pert = Pert(terminal(n),i)

prefetch()
  location_set = 0
  n = number of terminal nodes in AST
  for i = 1 to n do
    location_set = location_set
      || Location(terminal(n))
  if ( cardinality(location_set) > 1 )
    for i = 1 to n do
      Set_prefetch(terminal(n))
```

Figure 3. Optimization pseudo-code.

of the perturbation is relatively small until the latency exceeds 30 microseconds. As these latencies increase beyond 30 microseconds, the application runtime increases drastically.

3.3. Instrumentation mechanism selection

The server can use a variety of mechanisms to monitor and steer the target application; each of these mechanisms has different perturbation and latency characteristics [14]. When each application object registers with the server, it identifies a set of possible techniques the server can use to monitor and steer it. During the optimization phase, the server's interpreter weighs the costs of different mechanisms and annotates the AST, so that during execution, the server makes efficient mechanism choices. Given cost functions for these mechanisms from [14], the optimizer uses the data size and any constraints provided by the application to predict perturbation and latency.

The interpreter calls the mechanism selection optimization, as illustrated in Figure 3, to find efficient monitoring and steering mechanisms for each terminal node in a AST. This example focuses on latency, but the perturbation optimization is similar. The algorithm begins by retrieving the allowable set of mech-

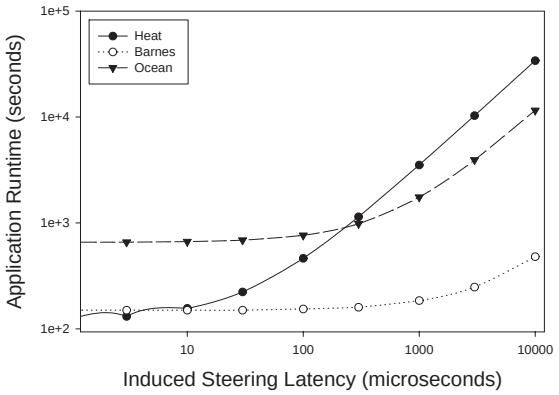

Figure 4. Average application perturbation due to increased steering latency.

anisms for the application object. This set is defined by the application when the application registers the object with the server. The server, then, stores this information in the registry. The algorithm determines the minimal latency for each of the possible mechanisms in the set. When the algorithm locates a new minimal latency, it marks the terminal node with that mechanism and records the perturbation.

Table 1 illustrates this relationship of mechanism selection, perturbation, and latency on the Barnes-Hut application. This experiment shows the resulting perturbation on Barnes-Hut when the steering system is forced to use one particular mechanism to achieve the steering request. In other words, the optimization has been disabled, and the choice of monitoring mechanism is fixed for the entire application execution. These results illustrate the dramatic range in the effects of different monitoring mechanisms on application runtime and latency. The tracing sensor, for instance, increases the runtime by almost 4 times to 469 seconds. Meanwhile, the monitoring latency–the total time from when an event occurs in the application until the server learns of that event–also increases to a maximum of 58 microseconds. The synchronous probe has the smallest latency at 6.3 microseconds. Magellan, with its optimizer, can select among these different mechanisms, which improves latency and perturbation within the constraints of the application and steering command.

Mechanism	**Runtime** (sec)	**Latency** (microseconds)
Sync Probe	147	6.3
Snap	345	24.4
Trace	469	58.6

Table 1. Effects of mechanism selection on Barnes-Hut application runtime and latency.

3.4. Prefetch

As described earlier, the locality optimization improves latency by forwarding requests to servers that have local access to all components of the steering command. In some cases, components of one steering command require more than one non-local value and all values are not stored on one server. This type of command cannot be forwarded; hence, the current server must execute the command. When this case arises, the server executing the command must retrieve each value from remote servers, possibly incurring a stall while retrieving the values. Normally, these requests would occur serially. However, it is possible to overlap the latencies required for retrieving each of these non-local values–essentially, prefetching them concurrently.

The prefetch optimization, in Figure 3, is closely related to the locality optimization. If the cardinality of the `location_set` is greater than one, then the server must execute the command locally and fetch the respective values from at least one remote server. The server annotates the AST by setting a prefetch flag on the terminal nodes of the expression's AST. The interpreter can have up to 8 outstanding prefetches for any expression during execution of any one steering command. When the execution phase begins, the interpreter begins walking the AST and finds the prefetch flags set. It requests the data from the remote server, marks the prefetch outstanding in a prefetch table, and continues to the next terminal node. When the interpreter has completed walking the AST. It waits until all the prefetches are complete, and then, walks the AST again, using the prefetched values. If terminal nodes are local, or they are constants, the prefetch attribute is ignored.

Figure 5 shows the benefits of concurrently prefetching a simple integer value, such as CPU utilization, as the communication latency increases. As the number of values that are eligible for prefetching increases, the advantages to prefetching increase due to latency hiding. Fetching these values serially when communication costs are higher than about 30 microseconds

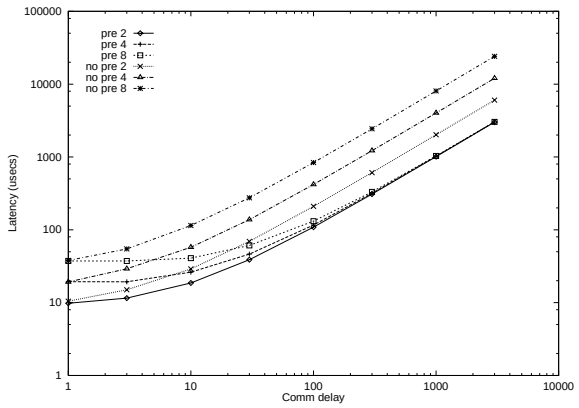

Figure 5. Latency required for prefetching multiple items versus communication delay.

drastically increases retrieval latency, which, in turn, increases steering latency. Note that one caveat for this system is that the values can arrive in various orders, but the execution of the command cannot begin until all the outstanding prefetches are satisfied. A more advanced implementation would allow the execution to begin when prefetches for partial subexpressions were complete.

4. Conclusions

This paper presented three optimizations that empirically demonstrate the viability of a language-directed approach to computational steering. These three optimizations—spatial locality, mechanism selection, and data prefetching—can improve application perturbation and steering latency. While our contribution is not a new language, this language approach has several advantages over more traditional approaches to computational steering. It allows the steering system to determine at runtime how to fulfill seemingly arbitrary user requests which lead to improved latency and perturbation.

References

[1] D. M. Beazley and P. S. Lomdahl. Lightweight computational steering of very large scale molecular dynamics simulations. In *Proc. Supercomputing 96*, 1996.

[2] 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.

[3] 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(3):224–35, 1997.

[4] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring and steering of parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[5] D. J. Jablonowski, J. D. Bruner, B. Bliss, and R. B. Haber. VASE: The visualization and application steering environment. In *Proc. Supercomputing '93*, pages 560–9, 1993.

[6] E. Kraemer and J. Wallis. Visualization and interactive steering of simulated annealing. In *Proc. SPDP'96 Workshop on Program Visualization and Instrumentation*, 1996.

[7] J. Leech, J. F. Prins, and J. Hermans. SMD: visual steering of molecular dynamics for protein design. *IEEE Computational Science and Engineering*, 3(4):38–45, 1996.

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

[9] 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)*, pages 172–9, 1998.

[10] V. E. Taylor, M. Huang, T. Canfield, R. Stevens, D. Reed, and S. Lamm. Performance modeling of interactive, immersive virtual environments for finite element simulations. *Int'l Jour. Supercomputer Applications and High Performance Computing*, 10(2-3):145–56, 1996.

[11] J. S. Vetter. Computational steering annotated bibliography. *SIGPLAN Notices*, 32(6):40–4, 1997.

[12] J. S. Vetter. Experiences using computational steering on existing scientific applications. In *Proc. SIAM Conf. Parallel Processing*, 1999.

[13] J. S. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proc. Int'l Parallel Processing Symp.*, pages 128–32, 1997.

[14] J. S. Vetter and K. Schwan. Techniques for delayed binding of monitoring mechanisms to application-specific instrumentation points. In *Proc. Int'l Conf. Parallel Processing (ICPP)*, pages 477–84, 1998.

[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*, pages 24–36, 1995.