CS_LITE: A LIGHTWEIGHT COMPUTATIONAL STEERING SYSTEM

Silvia M. Figueira
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566
sfigueira@scu.edu

Sonia Bui
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5118
sbui@nps.navy.mil

Abstract

Computational steering environments (CSEs) allow users to interact with applications executing in batch mode. Interaction with an application includes viewing intermediate results of computations and changing the values of program variables during program execution. Currently, most CSEs available are too application-specific and too complicated for the user to learn and use. In this paper, we show that computational steering can be done in a simpler, user-friendlier way. Through the use of a C library, CS_Lite provides users with an easy way of incorporating computational steering in their programs. Users need only to add CS_Lite function calls to their existing programs. In addition, a web-based graphical user interface simplifies the steering process and allows for more portability of CS_Lite through a JavaScript 1.2 (or higher) enabled browser. With CS_Lite, users will be able to steer primitive C data types: integers, floating point numbers, and character strings. For more complex data types, a file-transfer utility is provided. CS_Lite also recovers from server failure through the use of redundant servers.

Keywords

Computational Monitoring and Steering, Clusters of Workstations.

1 Introduction

Large-scale scientific computations, such as climate modeling and molecular dynamics simulation, are generally written as parallel programs, in which many computers work together to solve a particular problem. Traditionally, these parallel programs are batch-oriented. The results from running a parallel program are not available to the user until after the program has finished executing, which could potentially take many hours, days, or even weeks. Typically, users can only see the results and discover a scientific revelation, or a flaw in the program, after the end of the execution.

Computational steering environments (CSEs) were developed to solve this problem of non-interaction between users and executing programs. CSEs allow users to see intermediate results while the program is executing, instead of waiting until the end of the batch process. This displaying of results beforehand is called monitoring. In addition, CSEs also perform steering, which enables users to respond to the program current status (obtained from monitoring) and change the values of program variables without stopping program execution. The new values of the altered program variables are then used for the remainder of the program execution. The monitoring and steering functionality of CSEs have greatly helped scientists follow their programs' execution and gain insights into their research [8].

A large number of CSEs have been developed for scientific research in a variety of fields [2, 8, 11, 4, 12, 9, 10, 1]. However, most of them are very application-specific. A particular CSE may be well suited for doing simulations of wake-vortices but may be otherwise poorly suited for other types of scientific computations. A reason for this tight coupling between a CSE and a specific application is that the scientists working on the application are the same people who write the CSE for that application. Consequently, the CSE is optimized for the needs of that certain application and may not even work for other types of applications, e.g. [1]. This is a problem that these application-specific CSEs face, the inability to be used for diverse types of scientific computations [11].

In addition, many CSEs are extremely complex to learn and to use. It is often the case that the people who know how to use the CSE are the people who wrote it themselves. New users of CSEs face a steep learning curve [8]. Again, if the CSE is tailored for an application that is different than the application that the user is working on, then learning the semantics and peculiarities of that CSE are even more daunting for the user. Unfriendly, complicated user interfaces of some CSEs also do not help people trying to work with the CSEs.

Another reason why CSEs are difficult to use is that the monitoring and steering mechanisms are not intuitive

for the user. A majority of CSEs require complex program annotation (the adding of pieces of code to the program) in order to provide the program monitoring and steering functionality. One of the earliest computational steering systems, Falcon [4], suffered this problem of overly complicated monitoring and steering mechanisms. Some of the scientists using these CSEs have little formal education in computer science, so the task of program annotation may be long, complicated, and prone to error. Non-intuitive monitoring and steering methods are a huge problem plaguing CSEs.

This paper describes a simpler, more general, and easier to use computational steering environment (CSE). We call it CS_Lite because it is a lightweight computational steering system, a CSE without the added complexity. CS_Lite is a functional and useful tool, which can be used in clusters and Grid environments. CS_Lite could be easily incorporated into MPI libraries, such as MPICH [6] or LAM/MPI [5], to extend their capabilities.

The following components are part of CS_Lite:

1. Steering and monitoring mechanisms:
   These mechanisms describe the means in which the user program is to be monitored and steered. In other words, how a variable's value could be observed or altered during program execution.
2. Web-based interface:
   The graphical user interface or GUI consists of a web-browser that is widely available and familiar to any user. This interface helps providing a friendly environment for monitoring and steering.

This paper is organized as follows. Section 2 provides an overview of the system. Sections 3, 4, and 5 discuss the monitoring and steering mechanisms provided. Section 6 presents the graphical user interface. Section 7 concludes with a summary and a discussion about future work.

2 System Overview

CS_Lite has a client-server architecture, similar to the POSSE computational steering system [8]. CS_Lite uses two servers: one to handle monitoring and another to handle steering. The client is the user's program. In the case of a parallel program, there are multiple clients, each representing a node in the cluster in which the program is executing. In the monitoring process, each client, when they have information to send, checks the connection to the monitoring server and sends the data. For steering activities, the client makes a connection to the steering server and waits for the server to supply it with the new value for the variable being altered. The user inputs the new data for each parameter through a GUI. The user input is then read by the steering server, which sends the new value back to the client that requested the value for a parameter.

The clients executing the program communicate synchronously with the steering server and vice versa. We decided to make the network communication synchronous to ensure that the messages passed between clients and servers (the output results from the program and the new program variable's values) actually reach the intended receiver. Synchronicity also ensures the data integrity of the steered variable's values. For example, a client requesting a value for a variable will not get an outdated value from the steering server.

All the communication in the CS_Lite application is one-to-one: one server receives one message from one client at time. During monitoring, the monitoring server receives monitoring data one at a time, one for each *cs_send* call from a node. In the steering process, the steering server handles one request for a new value of a parameter at a time - reading the request, sending it to the GUI, receiving the user's input, and sending the new parameter value back to the client. In the case of parallel programs, CS_Lite broadcasts are not needed because the new parameter value can be propagated among the nodes, e.g., using MPI [7] collective operations, after one of the nodes receives the new value from the server.

The actual passing of information between the nodes was implemented using both TCP and UDP. For monitoring, which is the sending of output data from the clients to the monitoring server, we use UDP. For steering, which is the requesting for data from the client to the steering server and the sending of the new data from the steering server back to the client, we use TCP.

The reason why we use UDP for monitoring is because UDP is much faster for sending data. Even though UDP is unreliable, the occasional loss of output data is not detrimental for the user because the user can always add additional monitoring outputs through subsequent *cs_send* calls to the code. Also, all data that does arrive is written to a data log, so that the user can refer to previous data and interpolate the values for missing data. Data loss does not interfere with the flow of the program, so the user program continues executing even if a monitoring data packet has been lost.

However, CS_Lite adds a little more reliability in the UDP data transfer because it actually contacts the TCP steering server to see if the server is up before sending the message. We assume that the user will run the monitoring and steering servers on the same machine. If the machine is down, then it is most likely that both servers are down.

In addition, UDP allows for better scalability than TCP. If TCP was used, the connections between the server and client nodes would have to remain open for the duration of the program execution, which may become a problem if the number of nodes increases.

For steering, we use TCP, despite the scalability problems, because unlike monitoring, it is crucial that the client node receives the new steering parameter from the server. The user's program may be very dependent on receiving the new value of the parameter. TCP, being

reliable, ensures that the new value has been sent and received by the client node. Therefore, each time the client needs a value for a variable from the server, the client initiates a TCP connection and requests data from the server. The TCP connection is then closed after the client receives the data from the server.

The structure of the UDP packets used in the monitoring messages between the client and servers is as follows: 4 bytes for the node *id* ($0 \leq id < N$, where N is the number of nodes), 4 bytes for the message number of the packet (unique for each node or client), and 1,016 bytes for the monitoring data message (the value of the parameter). The structure of the TCP packets used by the client during the steering process is as follows: 4 bytes for the node *id*, 4 bytes for the message number (again, unique for each node/client), 1 byte for the data type of the steering parameter (integer, float, or string), and 1,015 bytes for the message the user may want to add to identify the name of the parameter.

The functions were written in C, since C is still the most used language in parallel and distributed computing. The GUI, or graphical user interface, is web-based so that the user can use CS_Lite from any machine. The various windows for monitoring and steering can be easily created using HTML. The interaction between these webpages and the CS_Lite monitoring and steering servers is achieved through CGI [3]. The CGI programs, one for monitoring and one for steering, are also written in C. Type checking of user input from the webpage is done through JavaScript. JavaScript can also update the webpages automatically by calling the CGI program after an elapsed time.

The design of CS_Lite was greatly influenced by that of POSSE [8], Portable Object-oriented Scientific Steering Environment. POSSE is a general use CSE, with a client/server architecture, POSIX threads, and remote steering on a GUI. CS_Lite differs in that it does not provide a visualization component that POSSE provides, and CS_Lite was written in C instead of C++. But POSSE is still more complicated to use than CS_Lite in that POSSE requires users to write their own servers to do the monitoring and steering. CS_Lite provides the monitoring and steering servers for the users. Also, POSSE does not automatically provide recovery of server failures and its GUI, although cross-platform, is not as portable as a web-based GUI. The graphics utilities for the POSSE GUI need to be installed on each machine using it. Not all machines have supporting graphics libraries, but almost all machines have web browsers. The most common web browsers can even be downloaded for free off the Internet.

The idea of web-based remote steering came from another CSE, DISCOVER (Distributed Interactive Steering and Collaborative Visualization Environment) [9]. But, in all other aspects, DISCOVER is very different from CS_Lite. DISCOVER is Java-based and meant for use in distributed systems. The web interface allows for geographically distributed scientists and researchers to collaborate on programs that use DISCOVER. The web interface may also be applicable in a smaller context where the machines are all on the same LAN.

3 Monitoring and Steering Mechanisms

A major component of CS_Lite is a channel of communication between the steering computer (the machine where the user will be monitoring and steering the program) and the cluster of workstations (which communicate with one another through MPI), which are executing the batch program. Note that the steering computer may reside outside of the cluster, so using MPI alone to communicate with the machines in the cluster is not sufficient. The steering computer and the cluster of workstations are connected by anything, from a LAN to a WAN link. Communications between the steering machine and the nodes of the cluster need to go through regular sockets.

As in MPI, the user must add CS_Lite setup functions to the program before any monitoring or steering can be done. This process is called program annotation. While other CSEs use complicated ways of annotating programs, CS_Lite will only require the user to make some function calls. Two CS_Lite functions handle the server setup: *add_server* and *set_udp_port*. The first function adds a host name and TCP port number to a linked list of steering servers, with which the client program will be communicating for steering. The second function indicates the UDP port for the monitoring server. It is required that the TCP and UDP host names are the same, only the port numbers are different. These functions should be called once for each possible machine to be used as a server, and more than one should be specified for fault tolerance.

When the user wants to run the CS_Lite program, the user must first run both the CS_Lite monitoring and steering servers on the user-specified host with the user-specified port number. Then the user runs the client program containing the monitoring and steering functions that communicate with these servers at the given host names and port numbers.

Because there is a list of servers, recovery from the failure of a host is possible. The user needs only to have provided several servers and port numbers (calling the two setup functions more than once). When the server goes down, the client program will try to connect to the next server on the list. The condition for the server to be considered unavailable is an elapsed time span, currently 30 seconds, after the client tries to contact the server. The Unix function *alarm* allows the client program to start connecting to another server after the set time of the alarm has passed.

The shutdown function, *cs_shutdown* shuts down the monitoring and steering servers by sending each a special packet that indicates that the client program has finished

executing all monitoring and steering functions. Leaving the servers running continuously is potentially dangerous, because a malicious intruder could discover the open port and try to overflow the buffer, which could lead to other damaging results. This function attempts to prevent unauthorized clients from communicating with the servers by shutting them down as soon as they are no longer needed.

To monitor and steer, two other function calls are required. To monitor the value of a variable at a particular point in the program, users just have to add a call to *cs_send_x* to the program. Similarly, to change the value of a variable in the executing program, users need to write in their program the following instruction: *variable = cs_receive_x()*.

These setup, shutdown, monitoring, and steering functions were written in C and incorporated into a library that users need to include with their program. This is similar to using MPI [7] library functions in parallel programs. The use of simple function calls to provide monitoring and steering abilities for programs is more intuitive for a user to use and understand. This stands in stark contrast to the complicated and confusing mechanisms that users of other CSEs must employ to make programs ready to be monitored and steered.

The other major component of CS_Lite is the graphical user interface. After annotating the program code to include monitoring and steering functionality, the user is ready to execute the program. As the program is executing, the CS_Lite monitoring and steering functions will be called. To monitor or steer the program variables, the user needs to open a web browser to the monitoring or steering page. The monitoring webpage will display the values of the variables being monitored. As more values are sent to be monitored, the webpage is updated to reflect the additional monitoring data. The steering webpage will display, for each *cs_receive_x* call, the request for a new value of a particular parameter and provide a text box for the user to input the new information. The user enters the information in the text box on the webpage and the information is sent back to the executing program. Within the executing program, the value of the variable is set to the new value specified by the user. The GUI is explained in more detail in Section 6.

4 Monitoring Functionality

The function *cs_send_x* sends an output from the client to the monitoring server. Note that *x* stands for the data type of the variable, either *int* (integer), *flt* (float), or *str* (character string). Adding new *cs_send* functions for different data types is straightforward. All the data type specific *cs_send_x* functions call the same generic function that puts the monitoring data packets together and sends them to the monitoring server.

All the low-level details of sockets are abstracted away from the user. All users need to do to incorporate monitoring into their program is to add simple function calls to the CS_Lite library. Each of these monitoring functions takes just two parameters, one for the node *id* of the client, and the actual data to be sent to the server. This interface is very intuitive and simple to use.

Figure 1 shows a piece of a program annotated for monitoring. This code will cause all the values in *matrix* (within *x* and *y*) to be shown by the monitoring server.

```
for (i = 0; i < x; i++)
    for (j = 0; j < y, j++)
        cs_send_int (matrix[i][j], myid);
```

Figure 1: Monitoring.

The monitoring server is implemented using POSIX threads. Two threads are needed, a *reader* thread and a *writer* thread. The writer thread receives the monitoring data from the clients and writes the received data in a special buffer (a linked-list). The writer thread handles one *cs_send_x* call at a time. The reader thread reads the contents of the buffer and writes the contents to a special monitoring log. This log will later be used by the GUI.

Threads are needed since the client program may be executing on more than one node, and several of these nodes may need to communicate with the monitoring server at the same time. If the server was not multi-threaded, messages from the clients may be lost because the server is busy opening and writing to the monitoring log file.

Since the reader and writer threads share the same buffer, the buffer needs to be protected. A POSIX-provided mutex was used to protect the monitoring data buffer. To prevent busy waiting (the reader thread constantly polling the status of the buffer), we used POSIX condition variables to signal when a thread has finished using the mutex.

The use of POSIX threads, mutex semaphores, and condition variables allows more flexibility in the machines that can execute the program. In fact, Unix, Linux, or any machine that uses POSIX threads can be used.

The GUI displays the monitoring results in a friendly format for the user. The monitoring GUI uses a web browser to display the contents of the monitoring data log file. The user simply needs to open the browser to the monitoring webpage, click the button, and the log is displayed on the page. The page also refreshes automatically every 20 seconds to show subsequent updates of the monitoring log (the result of subsequent *cs_send_x* function calls).

5 Steering Functionality

For steering programs, CS_Lite provides the functions

cs_receive_x, which sends requests for new values for variables from the client program to the steering server. The *x* stands for *int*, *flt* or *str* meaning that integers, floats, and strings can be steered, i.e., the values altered by the user. The steering server will send back to the client the new value for the variable, entered through a GUI.

The function *cs_receive_x* has three parameters: the node *id*, the name of variable (in a string), and the current value of the variable. This information is sent to the steering server, which outputs it before reading the new value input by the user.

Figure 2 shows a piece of a program annotated for steering. This code requests a value for *x* and a value for *y*, which are used to determine the values of the matrix to be used in the *calc* function. These values are requested from the steering server, which obtains them from the user.

```
x = cs_receive_int (x, "x", myid);
y = cs_receive_int (y, "y", myid);
for (i = 0; i < x; i++)
      for (j = 0; j < y, j++)
            matrix[i][j] = calc (i, j);
```

Figure 2: Steering.

Currently, more complex data types cannot be steered (or monitored) directly by CS_Lite functions. Since complex data types, such as matrices, are usually stored in a file, CS_Lite does provide a bare-bones file transfer utility, *cs_receive_file* that sends files from the steering server to the client program. But the user needs to add the necessary file input/output code needed to handle the file. One reason why CS_Lite does not directly handle data from files is because the format of the file varies depending on the user's preferences or particular application. By providing the minimum file transfer utility, CS_Lite gives the user more flexibility for handling different types of files (and data types). When received, the file is stored as a local file in the same directory where the client program is stored. The name of the file is provided by the user and is passed as a parameter to the *cs_receive_file* function. Note that, a program may receive different sets of data into different files. Note also that the format of the file is defined by the user.

Figure 3 shows a piece of a program annotated for receiving data through a file. This code requests a value for *x* and a value for *y*, which are used to determine the dimensions of the matrix into which the data should be read. Then, the code requests the file, which will be stored in "file.dat". After the file is received, the file is opened, and the data is read into the matrix, row by row. All these values are requested from the steering server, which obtains them from the user.

Like the monitoring server, the steering server uses two POSIX threads. One thread is similar to the monitoring writer thread, in that it receives client requests and writes them to a linked-list buffer. The other thread reads the request from the buffer and writes it to a special request/response file that the GUI will use. Thread synchronization/communication is also achieved through POSIX condition variables and mutex semaphores.

```
x = cs_receive_int (x, "x", myid);
y = cs_receive_int (y, "y", myid);
cs_receive_file ("file.dat", myid);

fp = fopen ("file.dat", "r");

for (i = 0; i < x; i++)
{
      if (fread (matrix[i], sizeof (int), y, fp) <= 0)
      {
            printf ("error reading data from file\n");
            exit (1);
      }
}
```

Figure 3: Obtaining data through a file.

The steering GUI is also web-based like the monitoring GUI. To steer the parameters, the user opens the web browser to the steering page and clicks the button to start the GUI application. The GUI reads requests from the request/response file. If a request is in the file, the GUI prints it to the browser and provides a text box where the user can input the new data value. The GUI checks for the correct data type of the user input. If the data is of the correct type, the GUI writes the response back to the request/response file. The steering server reads the user's response from this file and sends it back to the requesting client.

6 Graphical User Interface

The GUI is a web browser, which most users are familiar with. Because the steering GUI and the monitoring GUI are both web-based, remote steering and monitoring of program variables is possible. The user can start the client program on one machine, the steering and monitoring servers on another machine, and monitor and steer the program from yet another machine. All that is needed is a web browser and an Internet connection.

To monitor program variables, the user needs only to open the browser to the CS_Lite monitoring webpage and click the button to view the monitoring data log. The CGI monitoring program is called when the button is clicked. The CGI program then reads the monitoring data log, which is displayed in the browser and automatically updated every 20 seconds using JavaScript.

To steer program variables, the user opens the browser to the steering webpage and clicks a button that starts the CGI steering program. The CGI program reads

the request file for a new request and displays the next request to the browser. Information about the node that requested it, the data type of the variable and the name of the variable are displayed, along with a text box. The user enters the new value of the variable into the text box and clicks the submit button. On the submit, JavaScript checks for the correct data type of the user input. If it does not match the request, an alert box indicates that the user entered the wrong data type. Otherwise, the correct data is submitted to the CGI program, which writes the new value of the variable that the user entered back into a response file. JavaScript then redirects the user back to the steering page. If no new requests are pending, JavaScript waits every 5 seconds to call the CGI steering program to read the request file for new requests.

7 Conclusion

In summary, the CS_Lite computational steering environment is much simpler and easier to use than the CSEs available. Users need only to add function calls to their programs to allow monitoring and steering of parameters. A user may run the monitoring and steering servers on a machine and the program annotated with CS_Lite code on another machine. The user, on another machine that may be separate from the machines running the servers and the user's program, both sees the monitored results coming from the nodes and enters the values of steering parameters to the nodes that requested them through a web browser. Because CS_Lite is easy to use, it makes it easier for programmers to monitor their programs and maybe improve the development process.

It is important to mention that CS-Lite can be easily extended by adding new functions to deal with other types, such as characters or doubles.

There are some improvements that can be made to CS_Lite. The most significant improvement would be including security into the system. Currently, all communications between the servers and the client program are unprotected and not encrypted. Incorporating an encryption scheme or using secure sockets would provide better security. In addition, the use of CGI scripts in the web-based GUI carries security risks, and ensuring that the CGI scripts are secure would be another improvement.

Another area of improvement would be to have a version of CS_Lite that is Linux compatible. Because most parallel computing clusters use the Linux operating system, a useful tool for parallel programs like CS_Lite should be available for Linux machines. Since CS_Lite was developed using standard Unix features, the transition to Linux should be a simple process.

CS-Lite is available for any user interested in monitoring/steering parallel and distributed applications. The library can be downloaded from the web: http://www.cse.scu.edu/~sfigueira/projects/projects.html.

References

[1] D. M. Beazley and P. S. Lomdah, "Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations," in the electronic proceedings of *Supercomputing'96*, November 1996.

[2] G. Eisenhauer and K. Schwan, "An Object-Based Infrastructure for Program Monitoring and Steering," *2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[3] M. Felton, "CGI: Internet programming with C++ and C," Prentice Hall, 1997.

[4] 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," *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 433-429, February 1995.

[5] LAM/MPI Parallel Computing, http://www.lam-mpi.org/, June 2003.

[6] MPICH-A Portable Implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich/.

[7] Message-Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[8] A. Modi, L. N. Long, and P. E. Plassmann, "Real-Time Visualization of Wake-Vortex Simulations using Computational Steering and Beowulf Clusters," in *Proceeding of the Fifth International Conference on Vector and Parallel Processing Systems and Applications (VECPAR)*, pages 787-800, June 2002.

[9] R. Muralidhar, S. Kaur, and M. Parashar, "An Architecture for Web-based Interaction and Steering of Adaptive Parallel/Distributed Applications," in *Proceeding of the 6th International Euro-Par Conference*, pages 1332-1339, August 2002.

[10] S. G. Parker, M. Miller, C. D. Hansen, and C. R. Johnson, "An Integrated Problem Solving Environment: The SCIRun Computational Steering System," in *IEEE Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, 7: 147-156, 1998.

[11] B. Reitinger, "On-line Program and Data Visualization of Parallel Systems in a Monitoring and Steering Environment," Dipl.-Ing. Thesis, Johannes Kepler University, Linz, Austria, Department for Graphics and Parallel Processing, http://www.gup.uni-linz.ac.at/thesis/diploma/bernhard_reitinger/main/main.html, January 2001.

[12] R. van Liere and J. J. van Wijk, "CSE: A Modular Architecture for Computational Steering," in M. Goebel, J. David, P. Slavik, and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualization '96*, pages 257-266. Springer-Verlag Wien, 1996.