Progress: a Toolkit for Interactive Program Steering[1]

Jeffrey Vetter[2]

Karsten Schwan

Technical Report GIT-CC-95-16, August 1995

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
vetter@cc.gatech.edu, 404/853-9389, Fax: 404/853-9378
schwan@cc.gatech.edu, 404/894-2589

Abstract

Interactive program steering permits researchers to monitor and guide their applications during runtime. Interactive steering can help make end users more effective in addressing the scientific or engineering questions being solved with these programs, and it may be used to improve the performance of complex parallel and distributed codes. **Progress** is a toolkit for developing steerable applications. Users instrument their applications with library calls and then steer parallel applications with Progress' runtime system. Progress provides *steerable objects* which encapsulate program abstractions for monitoring and steering during program execution. Once created, steering objects are known to and manipulated by Progress' two components: (1) a server executing in the same memory space as the target program and capable of inspecting and manipulating program state, and (2) a potentially remote client providing command and graphical interfaces. Developers instrument their applications with the Progress toolkit library to create and maintain these steering objects. The server maintains information about the steering objects and performs steering actions on the application. This toolkit provides sensors, probes, actuators, function hooks, complex actions, and synchronization points. Progress' server is built on a Mach-compatible Cthreads library; it is a general toolkit for use with a variety of multithreaded, C programs executing on multiprocessors. Progress has been applied to several large-scale parallel application programs, including a molecular dynamics code and an N bodies simulation. It is currently being used with a complex global atmospheric modeling code.

Keywords: steering, dynamic, visualization, monitoring, environments.

1. Introduction

If high performance computing continues to remain 'non-interactive' [McCormick88], end-users and program developers alike will not capitalize on new techniques for interactive data visualization and program animation [Jablonowski93,Stasko90], remote and collaborative work, interactive debugging and monitoring [Gu95,Ogle93], and on-line program adaptation [Mukherjee93]. For example, when run in ' batchmode' , erroneous or uninteresting results produced by large-scale scientific or engineering simulations are not apparent until after the computations complete, sometimes days or weeks after program initiation. *Program*

[1] This report was originally submitted to the 24th International Conference on Parallel Processing 1995 (ICPP 95). It was accepted and published as a shorter version.
[2] Vetter is financially supported by a NASA Graduate Student Researchers Program (GSRP) grant.

steering provides end-users with the capability to monitor and guide their applications during runtime. The goal of our research in program steering is the exploration of the opportunities, limitations, and challenges inherent in the development and use of interactive high performance programs, on parallel and distributed execution platforms. More specifically, the **Progress** toolkit presented and evaluated in this paper provides facilities through which existing high performance multiprocessor programs are extended for increased interactivity [McCormick88, Jablanowski93, Eisenhauer94]. Once these extensions have been performed on a target application, a user can 'connect' to her high performance program, extract application-specific data regarding the program' sexecution state, perform steering actions, if desired, and then 'disconnect' from the program. The runtime overheads of these tasks depend only on the desired degrees of interactivity, ranging from minimally perturbing program execution in the absence of steering to being highly intrusive when Progress calls explicitly stop and restart programs.

High performance of Progress-instrumented programs is attained by use of *selective* and *application-dependent* runtime monitoring and program steering. Namely, programmers encapsulate program abstractions in *steering objects* that explicitly identify those program components as steerable. Steering objects are created, catalogued, and manipulated at execution time by Progress' runtime system. The runtime system consists of two distinct components: a server and a client. The server is an additional thread associated with the multithreaded multiprocessor application program. This thread responds to runtime application events, performs steering actions in response application events, executes commands from the client, and communicates with the client to provide consistent object information. The client provides command, recording, scripting, and user interface facilities, and it interfaces to existing program animation or data visualization facilities offered by the Falcon monitoring system [Eisenhauer94FAL] and by data visualizers [Ribarsky94] employed by Progress users.

This paper explores the requirements and opportunities of on-line program steering. Its contributions focus on the usage of steering during the execution of high performance parallel programs, similar to ongoing research on dynamic program monitoring [Eisenhauer94FAL,Hollingsworth93]. Stated briefly, the hypothesis we explore in this research is "program steering should be dynamically initiated, enabled and disabled, used selectively, and changed in scope and functionality." The Progress toolkit provides facilities with which we test this hypothesis.

1.1 Interactive Program Steering

Program steering is defined as the runtime manipulation of an application program and its execution environment. The goal of this manipulation is either performance improvement or increased functionality, such as focusing program execution toward more interesting data domains or improving resource usage through manual load balancing. An essential characteristic distinguishing program steering from work on interactive data visualization and navigation is the latter' slack of feedback from data manipulation to the program producing the data. *Program steering* affects the programs producing data, whereas data navigation is usually performed after program execution has completed. Similarly, a distinction of program steering from research in on-line program adaptation [Mukherjee93, Bihari91] is that steering enables both algorithms and users "in the loop" when application programs take actions in response to changes in program state or output. Our hope is to utilize this interactivity to improve productivity in the specific scientific and engineering processes being undertaken. For example, in the interactive atmospheric modeling code being developed by our group, multiple researchers will be able to inspect and manipulate data objects in shared 3D data visualizations. One such manipulation concerns changes in the concentration of atmospheric constituents, so that researchers can play “what if” games concerning the model’s global effects. Another manipulation allows alterations to the vertical constituent movement in the simulation' stransport model. This movement remains a poorly understood phenomenon, and model outcomes are significantly affected or even invalidated by the settings of simulation parameters controlling movement and the computational methods used for describing it. Steering improves the application by providing constant, selective feedback to the end user. The user can terminate the program, stall the program for inspection, schedule steering actions for execution, and effect a host of other modifications.

A concise breakdown of interactive program steering has three components: data collection, data interpretation, and steering. Data collection has a rich history of research in monitoring and data analysis. Data interpretation also has an extensive past with visualization and other techniques such as filtering, clustering, and queries [Bates86,Snodgrass88]. Steering, however, is rather immature. We review several systems in the Section 4 (Related Work); a complete review of systems related to this topic is available in [Gu94].

To summarize, *program steering* permits users to control program execution in terms of program abstractions familiar to them. Such abstractions may be encapsulations of computations producing program output like ' constituentconcentration' ,or they may address program resource usage like ' averagethread wait times' (typically of interest to program developers). In either case and in contrast to traditional research on program monitoring and debugging, steering must be based on abstractions specific to each application program, using runtime support that minimally perturbs program performance. This implies that steering cannot rely on automated methods for code inspection that may require disabling compiler optimizations, and it cannot require default instrumentation at any level of program abstraction. Furthermore, since steering is on-line, Progress cannot utilize existing post mortem methods for trace analysis as used in program debugging and in off-line monitoring[LeBlanc87, Mal91]. This prompts us to adopt the Falcon system's[weiming] event- and view-based models of program monitoring first advocated by Bates [Bates86] and Snodgrass for the collection of program state in Progress. Similarly, the Progress toolkit assumes that users employ visualization and animation toolkits [Ribarsky94,Kra93] for constructing the application-specific displays of program behavior required for steering. The Progress prototype targets the actual tools that software developers need to create generally 'steerable' applications and the additional functionality required from the operating system, and the monitoring system.

Computational fluid dynamics is a interesting target application area for steering. Biomedical researchers at GT are developing a parallel version of a spectral element fluid dynamics simulation program. CFD is an attractive area for program steering because most computations meet the time requirements with average simulations lasting days or weeks. Intermediate analysis of results could both pinpoint errors in initial conditions and allow modifications to error convergence limits. CFD data is usually visualized for final analysis but with on-line, interactive systems, various data might be visualized and modified throughout the simulation.

MD [Eisenhauer94] is a molecular dynamics simulation that explores the statistical mechanics of complex liquids. The dominant computational requirement is the calculation of long-range forces between particles. MD is an interesting steering target because load balancing is difficult and standard heuristics are not effective in managing the load. With the aid of a steering system, however, a human user may manage resources manually resulting in a low cost, effective simulation.

Weather and climate modeling presents a fascinating area for experimenting with Progress due to the enormous data sets garnered from satellites and other remote sensing stations. These data sets sometimes require repeated processing. End users can easily experiment with alternative model parameters, conveniently evaluate and re-evaluate the behavior of specific processes being modeled, and affect or change model execution to improve performance. Also, selecting certain data sets for evaluation or 'focusing' the calculation is useful with large data sets.

1.2 Paper Outline

This paper describes the design objectives for Progress, its architecture and implementation, and the evaluation of Progress with a real-world application. Section 2 describes the construction of the Progress steering system. Section 3 evaluates Progress with an application. Section 4 reviews related work. Section 5 concludes this paper with a review of research goals and future research.

2. The Progress Steering Toolkit

Progress is an acronym for *Pro*gram and *Re*source *S*teering *S*ystem. The goal of ProgReSS is “to provide applications developers with a complete set of tools and facilities for creating steerable parallel applications.” The Progress steering toolkit has several concepts and stages. First, the application developer must understand the steering object model to properly instrument the target application. The Progress object model encapsulates components of the application that the user might wish to steer or monitor. Second, the application developer actually instruments the application with calls to the Progress library to create and maintain these steering objects throughout the application’s lifetime. The developer understands the application’s operation sufficiently to allow external changes to its state while the application is executing. And finally, the end-user controls the application with the steering runtime system. The steering system is composed of a steering server and steering client. The steering server executes as a separate thread in the same memory space with the application. The client presents a graphical user interface the end-user for control of the remote steering server.

Progress is not intended as an advanced remote visualization system, or a parallel debugger. Remote visualization systems focus on the visualization of application output or performance visualizations. While interactive program steering will combine visualization of data and control of the application, its goal is not primarily visualization of data. Progress admittedly does not attempt to duplicate the functionality of advanced parallel debuggers. Parallel debuggers are far more general and flexible in their exploration of program information.

2.1 Design Requirements

Our design requirements for Progress include three fundamental notions: basic steering should be possible with many dynamic applications, the steering library should provide functionality for steering beyond updates to simple variables, and the user interface should allow end-users the capability to explore their executing programs. While designing Progress, it became obvious that certain types of programs are more ‘steerable’ than others. There is a distinct range of options facilitated by steering and not all these options are feasible with every program.

The library should support complex steering operations. Steering should provide tools for changing an executing application that protect application integrity and allow changes to application state that do not degrade application performance. As opposed to changing one integer variable in a SPMD application, steering should allow a variety of changes to all types of data within the application. These tools should also provide a level of abstraction that is effective for an end-user and useful for steering. Steering, as opposed to debugging, should not allow access to every program variable and the ability to arbitrarily interrupt program flow. Steering concentrates on observing and changing parameters of a correct application which is contrary to debugging in that debugging focuses attempts to locate faults in an application [McDowell89]. Types should not be limited to standard data types provided by the language. User-defined types including structures and arrays must be accessible to allow complex steering operations on arrays and structures of data.

Because the interface to the steering system might be predominately used by non-computer scientists, the interface to the steering system must follow typical guidelines for user interface design. The end user, not the application developer, controls the steering system through this steering user interface. This interface must provide various types of steering actions consistently on a continuous range of steerable programs. One steering interface for all steering. The steering interface to a CFD application and an atmospheric modeling application are the same; however, the steering objects accessible from this interface are application specific. Operations on steering objects are consistent between different applications. From the steering interface, the user should be able to selectively monitor and modify steering objects within the application. Also, the interface must provide ways of interpreting and analyzing large amounts of data produced by the steering system [Kra93].

2.2 Steering Object Model

An object model allows the steering system (and the user) to concentrate on only those components of the application that the developer explicitly declared prior to execution. This abstraction is necessary to limit amount of information that the steering system must contend with as well as the end-user. This object model, distinct from other notions of object-orientation, provides a convenient mechanism for naming and manipulating program components. Progress does not support inheritance and other characteristics of object oriented languages; Progress' object model encapsulates and identifies components of the program for monitoring and steering. This concept is identical to LeBlanc's instant replay mechanism [LeBlanc87], where the user identifies objects within the code. Replays guarantee to simulate interactions between these objects, but not between *every* program component.

An object becomes known to the steering server (and the user) when the object is registered. An object is a convenient way of describing data within the program that is of interest. A registered object can be accessed in three ways throughout the Progress system. Information about each object is stored in the application as a handle, in the server's object registry, and in the client's object registry. This object registration delimits the data of interest within the application, classifies the data type, and assigns a unique ID number to that object. Registration also enters a record into the steering server's registry of steering objects. The steering server uses this registry record to control the steering object within the application. The application receives a handle to the object so that it can perform operations on the object. The application may want to unregister an object, synchronize itself with the steering server and the end-user, or check an object to determine if any synchronous modifications are scheduled. After registration, the steering server can control steering objects. The steering server controls the object by monitoring the object or steering (modifying) the object. These operations are described fully in Section 2.3. At application termination or unregistration for a steering object, the server's registry record is removed and future references to this object are either ignored or produce an error.

2.3 Steering Object Operations

Once a steering object is defined within the application, the steering server manipulates the object with several different operations. These operations allow a variety of synchronous and asynchronous accesses to the application. Synchronous and asynchronous access are important to differentiate because, in some cases, asynchronous access to steering objects may produce inconsistent views of the object or actually invalidate the application results! To perform synchronous operations, the developer must instrument the application. The operations available on all steering objects are probe read, probe write, sense, and actuate. Operations available on specialized steering objects are synch points, function execution, and scripts.

Probe reads and writes to the steering object. A probe is the simplest of the steering operations because once an object is registered the steering server can just read the object's memory. After registration the developer does not have to introduce any additional code into the application. The steering server performs probes without respect to the applications control flow; therefore, probes are considered asynchronous. Probes are particularly useful for inspecting stalled programs or updating non-critical variables in the application.

Sense captures an object's state within the application and forwards it to the monitoring system for analysis. When the application encounters a 'Sense' call in its thread of execution, it copies the object state to an event record, and places the record into a buffer destined for the monitoring system. Because sense is executed within the control flow of the application, sense is synchronous. The steering server can enable or disable 'Sense' for each particular steering object. Both probes and sensors are investigated in [Ogle93] as part of an application specific monitoring system. In Progress, these mechanisms monitor steering objects instead of language specific application components.

Actuate performs a modification on the steering object. Actuate is a new steering mechanism that is analogous to the sense mechanism because it is synchronous with the application's control flow and it performs an opposite action on the application. When the application thread executes this actuate call, actuate checks its buffer to determine if any changes are intended for its steering object. These changes are 'programmed' for a steering object by the steering server. when the application executes an actuate call, actuate checks to decide if any changes to it's steering object are necessary. Actuate is synchronous with the control of the application because the modification is not performed on the application until the application encounters an actuate instrumentation point. If no modifications await the object, the actuate call exits. If a change is waiting in the buffer, then the actuate call modifies the object as specified.

	Synchronous	**Asynchronous**
Monitor	Sense	Probe Read
Steer	Actuate	Probe Write

Table 1 - Summary of Primary Operations for Steering Objects

For enhanced functionality, Progress provides a set of specialized objects. These objects have particular functionality that helps a user to steer an application. These operations are synch points, function execution, and scripts. **Synch Points** stall an application so that the end user can explore the state of the application and the steering client views a consistent snapshot of the application state. Synch Points are activated by the user so that the application stalls. The application threads spin waiting for a release command from the server. Because the threads are essentially halted, the user can interactively inspect and modify program state without fear of corrupting an executing application. The user can also be certain that the information stored in the object registries, on both the client and server, are consistent with the actual application state.

Functions allow the developer to register application functions with the steering server so that the server can execute application functions. Usually, these functions are complex operations that either gather information from that application or update some application state. Functions are regular C functions registered as steering objects with the server. Either the application or the server thread might execute these registered functions. The application could just call the function as it would any other function as could the server thread; however, for this function to be known to the registry and the client, the application must register the function. Registration publishes the name of the function to the steering system including the end user. When the calling thread is the server thread, functions can have one parameter and their return value is ignored. Functions are useful for accomplishing tasks other than just reading or writing a data value. These functions can alter application specific data structures within the executing application. For example, a user might stall an application and execute a registered function to remove an element from a queue. The function knows how to update the data structure, while updating all the variables for the queue data structure per se is tedious and error prone. In certain cases, these functions must synchronize with the application to prevent corruption of the application. Alternatively, a function might calculate a global value and store it in a steering object where it can be accessed asynchronously with a probe.

Scripts provide users with the functionality of combining other steering operations in a language form for repeated execution. Scripts are different from functions because they are executed either on the client or in the steering server; the values they use and actions they execute are derived only from the registry - scripts cannot access program data other than registered objects.

2.4 Runtime System

The runtime system is composed of a server and a client. This separation is convenient because the server runs on the same machine as the application, and the client, which presents a graphical user interface to the user, usually executes remotely on a separate machine. With the server on the same machine as the application, the server can access and control the application with low latency [Gu95]. Because the client is remote, latencies introduce network delays. These network delays are closer to human interactive response times; however, they are burdensome for the server.

This architecture is also advantageous because the server must exist through the application's lifetime while the client does not. The client is transitory and can connect to the server many times throughout the server's (application's) existence. Because the client may use visualizations for data interpretation, the end-user may choose to run the client on a high-performance graphics system.

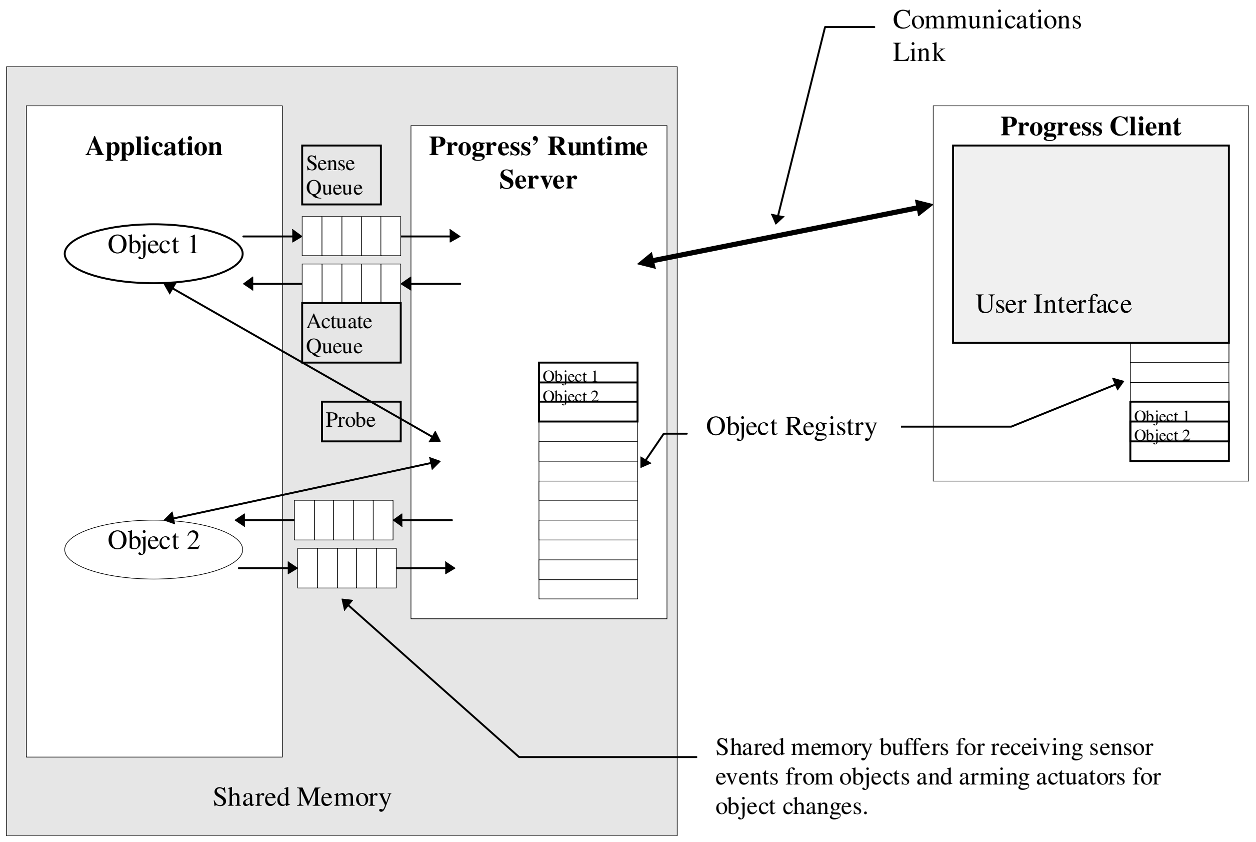


Figure 1 - Progress Architecture

2.4.1 Steering Server

The steering server is a separate thread that executes in the same memory space as the application. The server architecture (Figure 1) allows the application to execute normally. In fact, the application can execute entirely without interference from the server. The server thread has three basic tasks: interact with the steering client, gather monitoring output from the application, and steer the application via the steering objects. This server continuously maintains a registry of steering objects within the executing application. Each object that the application registers is stored by name in the registry. Registry records contain enough information to access the application's object at anytime and interpret information generated from the object. The registry is also used to route information from the application to the client and visa-versa.

The server communicates with the client through UNIX sockets. When the server starts, it allocates a socket, publishes the socket number, and listens to that socket for pending client connections. Once the client connection contacts this server socket, the server creates a new stream connection for that client-server pair. Thereafter, the message protocol allows bi-directional messages between the client and server. The server polls the client's socket for incoming messages. When a message is received, the server decodes the message and executes any action required. The server assumes that incoming messages from the client will be relatively infrequent compared to the other server tasks. The server is required to execute actions requested by the client. Occasionally, those actions require extensive time.

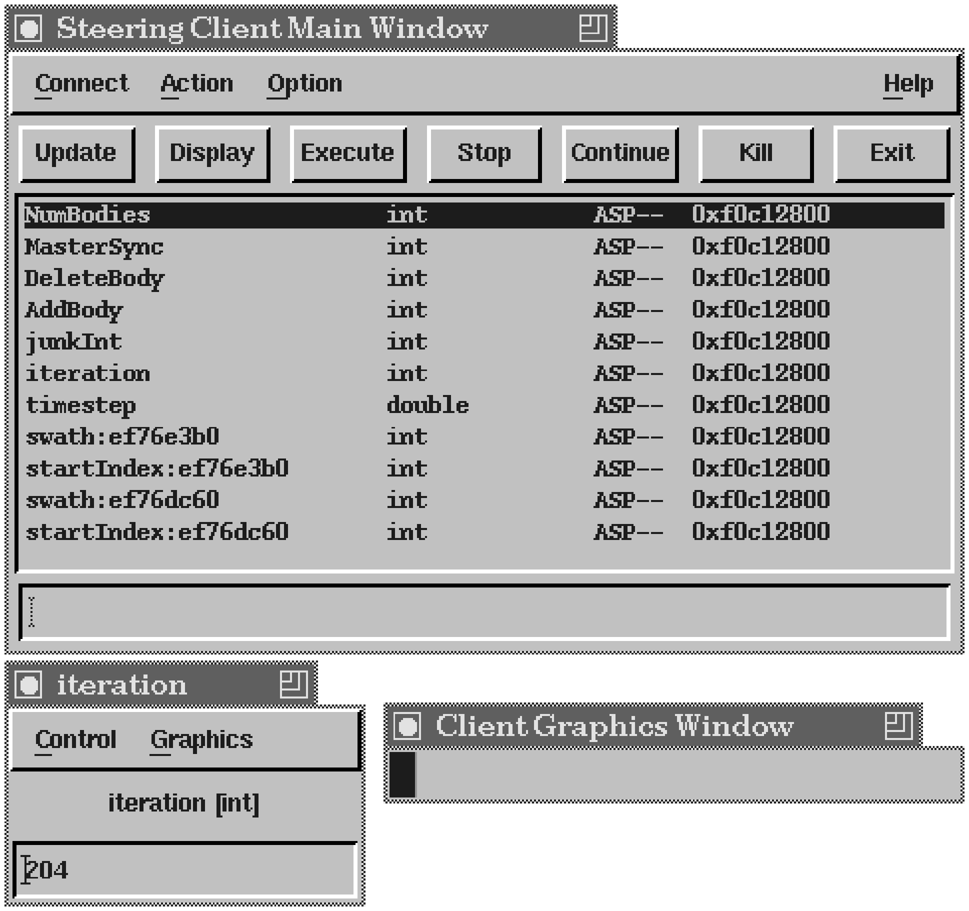
The server monitors the application by receiving sensor events and probing objects. The server does simple analysis to filter irrelevant events out of the stream. The server controls monitoring so that the end-user can selectively observe different steering objects throughout the application's lifetime. Based on the commands from the end user, the server enables and disables sensors and probes objects to gather information for the end user. This monitoring has two levels of application specific filtering. First, the application developer registers only those objects that may possibly be of interest to an end user. This level of filtering discards all temporary or uninteresting data values in the application. The developer selects data that the user may want to observe or control. Second, the end user elects which steering objects to monitor through the client user interface. The end user only receives information about objects that are registered with the server *and* that he has selected through the client interactively. The user cannot arbitrarily access components of the application that are not registered with the server.

The server steers the application through steering objects using the steering operations detailed in Section 2.3. Several actions can trigger a steering operation. The user can manually request a steering operation on a steering object, or the server can execute a steering operation in response to an event received from the application.

2.4.2 Steering Client

The steering client is a remote application to control the resident steering server. The client is a single threaded Motif application that communicates with the steering server through a customized message protocol. The graphical user interface for the steering client is presented in Figure 2. The client has three main tasks: interact with the user, communicate with the steering server, keep relatively consistent state information about all the steering objects. The client receives all of its information from the steering server; it receives periodic updates to its registry from the server, based on the frequency of activity at the server. The client has a mirror registry of the server's registry. The interface allows the user to selectively monitor and modify steering objects. Each of the steering actions available through the steering server can be initiated from the steering client albeit the latency is higher. A user can enable monitoring of a steering object through sensors, probe an object for its current state, or modify the object through a probe write or an actuator.

An additional task of the steering client is easy manipulation of possibly large datasets resulting from monitoring. If a user selects monitoring for an object at a high frequency, simple textual display of the results fails [Kra93]. However, because the structure of the steering object is known to the client, the client can map the objects updates onto a simple graph or other visualization.



```
NumBodies                int       ASP--  0xf0c12800
MasterSync               int       ASP--  0xf0c12800
DeleteBody               int       ASP--  0xf0c12800
AddBody                  int       ASP--  0xf0c12800
junkInt                  int       ASP--  0xf0c12800
iteration                int       ASP--  0xf0c12800
timestep                 double    ASP--  0xf0c12800
swath:ef76e3b0           int       ASP--  0xf0c12800
startIndex:ef76e3b0      int       ASP--  0xf0c12800
swath:ef76dc60           int       ASP--  0xf0c12800
startIndex:ef76dc60      int       ASP--  0xf0c12800
```

```
204
```

Figure 2 - Graphical User Interface for Steering Client

The client's user interface is build with Motif to present steering objects in a consistent and general manner. This same interface is used with all steering applications whether the application is a atmospheric modeling simulation or a CFD application. Steering objects are presented to the user in a consistent fashion for investigation and modification.

Figure 2 shows the main screen of the Progress client. The main menubar is at top. Next are two boxes: one list box and one text box. The list box displays all registered steering objects in the application. All objects are listed here whether they are regular steering objects or more specialized steering objects. The text box is an event log. As objects produce events, they are logged here in the event log. This log information can be captured in an ASCII file for post-mortum review. The command line allows the user to enter complex expressions and other statements not easily entered with the graphical user interface. Below the command line, the message log displays messages specific to the operation of the client, but not the server. A popup window (not shown), the object information box, provides object specific information about the selected object.

This box displays the object ID, object type, object name, etc. and any attributes particular to the object type. Menu sensitivities change as different objects are selected in the object list box. For example, the 'enable' and 'disable' menu options are disabled when a function is highlighted in the object list box because these menu options do not work with functions.

2.5 Instrumenting an application

Instrumenting the application builds entirely on the steering object model (Section 2.2). Instrumentation creates the steering server, and registers and manages the steering objects. The application registers these steering objects with the steering server.

```
1   main()
2   {
3       int i;
4       pSteeringServer steeringServer;
5       pSObject sobj;
6                   .
7                   .
8       CreateSteeringServer( &steeringServer );
9                   .
10                  .
11      RegisterSteeringObject( steeringServer, &sobj, &i, A_INTEGER );
12                  .
13      i = 1;
14                  .
15      while( ++i < 10 )
16      {
17              Sense( sobj );
18      }
19                  .
20      while( i != 0 )
21      {
22              Actuate( sobj );
23      }
24                  .
25      printf( "%d", i );
26                  .
27      DestroySteeringServer( steeringServer );
28  }
```

Figure 3 - Steering Server Calls

Figure 3 outlines the necessary calls to the Progress toolkit for a simple application. CreateSteeringServer creates data structures, initializes the state, and spawns that steering server for this application. The steering server's internal structure is described in Section 2.4.1. Once the steering server is initialized, the application can register steering objects. RegisterSteeringObject creates an record in the application containing information about the object, and it also enters a record in the steering server's object registry. The registration requires a pointer to the application state of interest (e.g., integer i on line 11) and the data type of that application state (e.g., A_INTEGER on line 11). After the registration, the server can access the object with steering operations and the steering client is notified of the registration, so that the user can interrogate the new object and possibly steer it.

The ‘Sense’ call uses the information stored in the steering object record to determine its operation. First, the call checks whether sensing is enabled for this object. If sensing is disable, the call returns without further action. Otherwise, it continues. The call, using the information in the steering object record, copies the object state to an event record and places it in a shared memory buffer. The buffer was previously created by the steering server and the steering object record contains the name of this buffer. In this example (Figure 3), the event record would contain an object identifier, a timestamp, a thread id, and the value of the steering object (integer i=1). Once the event record is inserted into the buffer, the sense call returns to the application. Notice that the sense call only requires a steering object handle. The Sense call gathers all the necessary information about the object from the steering object handle. This design is convenient for several reasons. First, if the frequency of state updates for a steering object is too low, then the developer can insert more ‘Sense’ calls using the same steering object handle. Second, if multiple objects are to be monitored, then a sense call for each object must be inserted with a handle to each object. Third, if a user wants to disable monitoring for a particular object, then they disable monitoring for that object, not each sense call. Because the event records generated by each sense call update the state of an object at the server and eventually the client, it is advantageous to enable and disable monitoring on a per object basis rather than a per sensor basis. The object model allows the user to focus on objects of interest within the program and only those objects. The user does not have to enable and disable multiple sensor insertion points for each object they wish to observe; they merely enable or disable all sensors for each object.

The ‘Actuate’ call is a steering operation that modifies the steering object. Actuate uses information stored in the steering object record to determine if any modifications to the object state are required. If the actuator is not armed, then the call just returns to the application. If the object is to be updated, then the actuate call retrieves a record from a shared memory queue. This record contains all the necessary information to update the steering object. When the actuate call is executed, it checks this shared memory queue for entries. If the queue contains an entry, then the actuator retrieves the record and updates the steering object state appropriately. The actuator is armed by the steering server. The server arms an actuators on a per object basis. In Figure 3, the actuate call forces the value of i to equal 0, otherwise the program loops forever. The external steering server places an record into the actuate queue for ‘sobj.’ Then, when the application executes the actuate call on sobj, the application retrieves the record and updates the object appropriately. In our example, the actuate call is constantly checking the sobj actuate queue to determine when a change must occur. Because the steering server places only one record in this object’s actuate queue, the majority of calls to actuate just return without modifying the sobj (or the integer i).

However, on the last call to actuate, the server has placed a record in the actuate queue to change the object to 0. On this last call, the object is updated to 0 and the loop terminates on its next loop test (*i != 0*).

Actuate operates on a per object basis. Many actuate calls can service one steering object. Placement of the actuate calls throughout the source code identifies time frames when the steering object can be updated without corrupting the application during execution. A higher frequency of these actuate calls allows steering objects to respond more rapidly to user steering requests. As with sense, each individual actuate call cannot be disabled.

Actuators also have pre- and post-conditions [Bihari91]. Preconditions allow a binary test to check a state within the application before firing the actuator. Post conditions execute a function after the actuator has modified the application state. Postconditions usually update application state based on the change performed by the actuator. An actuator might even return a steering object to its previous value because the postcondition failed with the new value.

3. Evaluation

To evaluate Progress, we use N-body because it is a well-understood and concise example with which we can describe the functionality and performance of our toolkit. Additional functionality of Progress is evaluated by demonstrating new techniques of interacting with the executing application. Performance is important because this toolkit cannot prohibitively degrade performance of the application. Users will not tolerate excessive performance penalties.

3.1 N-body

The numerical N-body of gravitation [Greengard90] simulates dynamical behavior of large stars with only gravitational forces acting between them. This simulator uses a straightforward N^2 algorithm for calculating pairwise gravitational forces and updating the velocities and positions of the bodies. The N-body application is implemented as a collection of worker threads calculating gravitational forces and new positions for each of the bodies in synchronous timesteps. N-body is built on a user-level threads library. Multiple threads divide the work by allocating each thread a group of bodies to update. Each worker thread reads the positions and parameters of its neighbors, and then updates the position and velocity for each body assigned to it.

Integrating N-body with the Progress system required two distinct steps. First, the developer added appropriate calls to create and initialize the steering server. Second, registration calls for all steerable objects were inserted into the source code. Third, all synchronous objects had the instrumentation code inserted in the application to identify points where these objects could be accessed.

Creating and destroying the steering system only required adding two calls to the N-body source code. Straightforward steering objects provided information on the iteration number, timestep, body count, and various other parameters. These objects were registered with the server through the code, usually near their declaration. At various points in the master thread, the sense calls produce one event to update these objects, if monitoring is enabled. These activation points are not limited to one location. For example, the timestep variable controls the outermost loop in the simulation. In the master thread and the slave threads, these loops are identical with barriers synchronizing the loops. Timestep is the variable loop variable. The timestep object is registered once in the master thread's loop and it has only one sense call within the master thread's loop. This thread generates one event per loop for the entire application. Because the timestep only changes once per loop, then updating it more often with additional sense calls would be repetitive and inefficient. Also, if the slave threads had sense calls within their loop, the timestep object would generate one event per thread per sense call. In other words, at every timestep the number of events generated updating the timestep object would be equal to the number of threads in the application including the master and its slaves. However, all of these events would be identical updates to the timestep object because the timestep loop is synchronized across all threads. Thus, these multiple updates are redundant.

Prior to the steering integration, user interaction was limited to file I/O. The user created an input file with various parameters. N-body then read this file and began processing, occasionally, producing an output message for feedback to the user. At various intervals, the application dumped body position and velocity to a file. While this type of interaction could be customized to provide interactivity, there are no generic methods or toolkits to facilitate this type of selective, application-specific interaction.

Progress allows far more interaction with the simulation than file I/O. With Progress the user interactively explores the intermediate results of the simulation. For example, one particular body's velocity can be traced during the simulation using a sensor to determine if the parameters are realistic. Another sensor can report the timestep value. Yet another sensor can track a load average for that thread. If the user notices an error in the parameters, then he can stall the application and inspect all other objects. Furthermore, if the user decides to take corrective action, the user can update the parameter with a probe write or an actuator, and then, allow the application to continue.

Users may add new bodies and delete existing bodies to the executing simulation with the aid of Progress. As the application is advancing through its timesteps, the Progress server manipulates the application so that a new body with specific parameters is added to the simulation. In our N-body simulation, two application procedures were created that add one body to the current simulation and delete one body from executing simulation. The results of these procedures are cumulative. Executing AddBody twice will add two bodies to the executing simulation. These two application procedures are necessarily application specific. They have access to the necessary variables and data structures so that the simulation can be updated properly. Additionally, this AddBody procedure can register these new bodies with the steering server so that the user can alter their parameters. Once the procedure is complete and the body parameters are adjusted, the user can continue the simulation.

3.2 Performance

The utmost concern of Progress' designers was the performance degradation due to instrumentation with the Progress library. Obviously, the user can chose to degrade performance by controlling the program; however, this option remains with the user. As illustrated in ***Table 2***, the application's performance degradation due to the addition of Progress is minimal when compared to other common debugging and profiling techniques. The steering system did not interact with the application during the test. These tests did varied proportionally with longer execution times and they did not vary considerably from architecture to architecture (KSR, SGI, Sparc). The measurements in ***Table 2*** were gathered on an SGI 8-node multiprocessor with the standard SGI compiler and multiprocessor library.

Measurement	**Time (sec)**
N-body optimized	55.55
N-body w/ Progress	57.23
N-body w/ gprof	69.73
N-body w/ -g compiler option	72.11

Table 2 - Application Performance

Latency is important because the information presented at the client, and used for decision making at the server must be close to real time. Unusual feedback might occur if the information is too far out of date, impeding any performance gains due to steering and limiting the usefulness of the steering system. Also, the server must be fast enough to execute steering actions. In ***Table 3***, the performance of several measurements are outlined.

Table 3 describes the latency for several type of steering operations under the following conditions: a total of 100,000 operations are executed by the server and they are measured from this beginning of the operation until they complete including the object registry query time. Additional filtering or processing on the operations are not used. For the synchronous operations, the server time is the time the server requires to place a record into the objects actuate buffer. Enabling and disabling sensors is essentially a probe write to a memory location that the sensor checks during each call this is why the time is close to that of the probe read/write.

4. Related Work

Several steering systems exist [Gu94] as well as research in dynamic applications and adaptable systems. Because Progress focuses on interactive systems, we limit our review to systems that allow interactivity. Functionality and generality of these systems vary but they are consistent with Progress' goals of performing interactive program steering. As discussed earlier, interactive program steering implies that human users have the option of interpreting program data and providing feedback to the program during its execution. Other research on dynamic systems discusses feedback and adaptation; however, the feedback is usually the product of an algorithm.

Tuchman, et al. [Tuchman91] created the *Vista* system for simulation-time visualization of data. Vista

Action	**Server Time** (μs)
Probe Read/Write	643
Actuate	627
Sensor Event	181
Enable/Disable Sensor	651

Table 3 - Progress Performance

provides a window into the application by showing program data automatically during execution. The system architecture is designed for a distributed or remotely executing application. The Vista model allows a trace file to replace the executing application, providing a visualization ' databrowser' for data from past simulation runs. Data from the executing application are interactively selected and displayed. Vista did not concentrate on steering the application; however, the interactive selection and display of application data is similar to Progress' interactive selection and control of steering objects. Progress goes further by allowing the user to propagate changes from the user interface back into the executing simulation.

Program directing is investigated in [Sosi92]. Program directing is synonymous with program steering. *Dynascope* monitors a program, presents the data to a user or program, and allows for possible feedback actions. Dynascope provides basic monitoring and controlling in distributed environments. The system is integrated with existing programming tools and uses a few generic operating system and networking primitives. Dynascope provides a complete set of tools for interacting with an application, including feedback; however, Dynascope did not concentrate on high performance, parallel applications. The tools available for interacting with the executing program gave the operator access to the entire program and Dynascope did not allow the developer to clearly define how and where a user could steer the application. Progress focuses both the developer's and user's attention on the steering objects allowing efficient and meaningful access to program components.

The *VASE* system [Jablonowski93] presents an abstraction for a steerable program and offers tools that create and manage collections of steerable codes. VASE annotates existing Fortran code to create a high-level model of the application; therefore, users do not have to work at the source code level. Software developers must annotate the existing code, however. Once the source code is annotated, VASE coordinates the execution of these codes in the distributed environment. VASE supports only the SPMD model of parallel execution. A powerful 'C'-like scripting language provides flexibility for data selection and steering during execution. The SGI Iris Explorer renders output data visualizations. Progress resembles VASE in several ways; however, there are differences. Both VASE and Progress provide a user interface for interacting with a remotely executing application. They both also provide a technique for abstracting uninteresting details from the steering process. VASE, however, concentrates of abstracting blocks of code and control flow, whereas Progress focuses on abstracting important data (steering objects) and time windows for accessing those data items (sense,actuate). Both of these system recognize the difference between the application developer and the application user. The application developer is responsible for instrumenting the application code so that the end user can control the application with a general steering interface.

DYNA3D and AVS (Application Visualization System from AVS Inc.) are combined with customized interactive steering code to produce a time-accurate, unsteady finite-element simulation in [Kerlick93]. Rudimentary steering is demonstrated in a distributed environment consisting of a supercomputer and multiple graphics workstations. Although steering was demonstrated, [Kerlick93] did not present a general toolkit for steering any application program. Progress attempts to define a library and runtime system that will work with a variety of MIMD applications.

[Parris93] describes challenges for a real-time visualization of a complex physical simulation. The goal of this real-time visualization is a virtual world where human users interact with the visualization in a 3D environment. The implementation spans a network of several specialized computer systems. [Parris93] is interesting in the context of Progress because of the feedback techniques used. However, the results of this work were customized for a particular high performance graphics system and network of computers, and it did not address the creation of a general toolkit for steering common high performance parallel applications.

5. Conclusions

Progress is a prototype steering toolkit for the specific purpose of evaluating the necessary components of a general steering system and the essential functionality required by interactive steering. Progress has successfully provided a testbed for interactive steering, and we have outlined a set of general features for inspecting and modifying executing applications: the steering object model and their respective object operations. Progress' steering object model provides a useful technique for identifying components of the application to export to the end-user. Probes, sensors, actuators, synch points, and functions furnishes a developer with numerous mechanisms to allow an end-user to observe and modify his application at runtime.

Two improvements to the existing Progress system are essential. First, the object registry system must allow complex user defined types including arrays and structures. Existing technology forces our toolkit to define these user types are runtime, rather than compile time. The developer must add additional toolkit calls to the application to describe any user defined types that he may want to register as steering objects. Arrays are extremely valuable because the user may want to adjust an entire array of values and rather than registering each element of the array with the server, the developer could just register the array itself. Structures are also important because to allow a user access to the fields within a structure, the developer must now register each field individually instead of just registering the entire structure.

Second, visualizations of steering objects at runtime is necessary to interpret the massive amounts of information that the user might select [Appelbe91,Bem93,Cou93]. Eventually, the goal of Progress is direct manipulation of graphical models of the simulation with appropriate feedback into the executing simulation. High performance graphics systems could possibly provide complex graphics in real-time. For example, one sensor in the application captures three variables: timestep, convergence error, and data region. The Progress runtime periodically forwards this event to the graphics system over the network. At the graphics system, a user binds these three values to a 3D surface graph. Using this visualization, the user easily locates convergence problems with portions of the data regions. As the simulation executes, the user could view an animation of the convergence errors and their respective data regions in the simulation.

6. References

[Appelbe91] William F. Appelbe, John T. Stasko, Eileen Kraemer. "Applying Program Visualization Techniques to Aid Parallel and Distributed Program Development." Report GIT-CC 91/34, College of Computing, Georgia Institute of Technology. July 1991.

[Bates86] Peter Bates. "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior." *ACM/ONR Workshop on Parallel and Distributed Debugging* (1988), pp. 11-22.

[Bem93] Thomas Bemmerl, Peter Braun. "Visualization of Message Passing Programs with the TOPSYS Parallel Programming Environment." *Journal of Parallel and Distributed Computing*, 18(2):118-128, June 1993.

[Bihari91] Thomas E. Bihari, Karsten Schwan. "Dynamic Adaptation of Real-Time Software." *IEEE Transactions on Computer Systems*, 9(2):143-174, May 1991.

[Cou93] Alva L. Couch. "Categories and Context in Scalable Execution Visualization." *Journal of Parallel and Distributed Computing*, 18(2):195-204, June 1993.

[Eisenhauer94]Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. "Falcon -- toward interactive parallel programs: The on-line steering of a molecular dynamics application." In *Proceedings of The Third International Symposium on High-Performance Distributed Computing*, San Francisco, CA, August 1994.

[Greengard90] Leslie Greengard. "The Numerical Solution of the N-Body Problem." *Computers in Physics*, March/April 1990, pp. 142-152.

[Gu94] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. "An Annotated Bibliography of Interactive Program Steering." *ACM SIGPLAN Notices,*29(9):140-148, September 1994.

[Gu95] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs." *Proceedings of FRONTIERS'95*, February 1995.

[Hollingsworth93] Jeffrey K. Hollingsworth, Barton P. Miller. "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems." *Proceedings of International Conference on Supercomputing* (1993).

[Jablonowski93] David Jablonowski, John Bruner, Brian Bliss, and Robert Haber. "VASE: The Visualization and Application Steering Environment." In *Proceedings of Supercomputing 93*, pp. 560--569.

[Kerlick93] David Kerlick and Eliabeth Kirby. "Towards Interactive Steering, Visualization and Animation of Unsteady Finite Element Simulations." In *Proceedings of Visualization 93*.

[Kilpatrick91] Carol E. Kilpatrick, Karsten Schwan. "ChaosMON -- Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems." *ACM/ONR Workshop on Parallel and Distributed Debugging* (1991).

[Kra93] Eileen Kraemer, John T. Stasko. "The Visualization of Parallel Systems: An Overview." *Journal of Parallel and Distributed Computing*, ():, May 1993.

[LeBlanc87] Thomas J. LeBlanc, John M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay." *IEEE Transactions on Computers*, C-36(4):471-481, April 1987.

[Mal91] Allen D. Malony, David H. Hammerslag, David J. Jablownski. "Traceview: A Trace Visualization Tool." *IEEE Software*, 8(5):29-38, September 1991.

[Malony92] Allen D. Malony, Daniel A. Reed, Harry A. G. Wijshoff. "Performance Measurement Intrusion and Perturbation Analysis." *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433-450, July 1992.

[Marinescu90] Dan C. Marinescu, James E. Lumpp, Thomas L. Casavant, Howard Jay Siegel. "Models for Monitoring and Debugging Tools for Parallel and Distributed Software." *Journal of Parallel and Distributed Computing*, (9):171-183, 1990.

[McCormick88] B. H. McCormick, T. A. DeFanti, M. D. Brown. "Visualization in Scientific Computing." *ACM SIGGRAPH Computer Graphics*, 21(6):, November 1988.

[McDowell89] Charles E. McDowell, David P. Helmbold. "Debugging Concurrent Programs." *ACM Computing Surveys*, 21(4):593-622, December 1989.

[Mukherjee93] Bodhisattwa Mukherjee and Karsten Schwan. "Improving Performance by Use of Adaptive Objects: Experimentation with a Configurable Multiprocessor Thread Package. *Proc. of Second International Symposium on High Performance Distributed Computing (HPDC-2),* July 1993, pp. 59-66.

[Ogle93] David M. Ogle, Karsten Schwan, Richard Snodgrass. "Application Dependent Dynamic Monitoring of Distributed and Parallel Systems." *IEEE Transactions on Parallel and Distributed Systems*, 4(7):762-778, July 1993.

[Parris93] Mark Parris, Carl Mueller, Jan Prins, Adam Duggan, Quan Zhou, Erik Erikson. "A Distributed Implementation of an N-body Virtual World Simulation." In *Proceedings of The Workshop on Parallel and Distributed Real-Time Systems* (April 1993), pp. 66--70.

[Ribarsky94] William Ribarsky, Eric Ayers, John Eble, Sougata Mukherjea. *"Using Glyphmaker to Create Customized Visualizations of Complex Data*." IEEE Computer, June 1994.

[Schwan88] Karsten Schwan, Rajiv Ramnath, Sridhar Vasudevan, David Ogle. "A Language and System for the Construction and Timing of Parallel Programs." *IEEE Transactions on Software Engineering*, 14(4):455-471, April 1988.

[Snodgrass88] Richard Snodgrass. "A Relational Approach to Monitoring Complex Systems." ACM Transactions on Computer Systems, 6(2):157-196, May 1988.

[Sosic92] R. Sosic. "Dynascope: A Tool for Program Directing." In Proceedings of SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 27(7):12-21, July 1992.

[Stasko90] John T. Stasko, Charles Patterson. "Understanding and Classifying Systems for the Visualization of Computer Data Structures, Programs, and Process." Report GIT-CC 90/66, College of Computing, Georgia Institute of Technology. 1991.

[Tuchman91] Allan Tuchman, David Jablonowski, George Cybenko. "Run-Time Visualization of Program Data." *Proceedings of Visualization '91* (1991), pp. 255-261.