Simulation Steering with SCIRun in a Distributed Environment

Michelle Miller, Charles D. Hansen, and Christopher R. Johnson

University of Utah, Salt Lake City, UT 84112, USA,
{mmiller, hansen, crj}@cs.utah.edu,
WWW home page: http://www.cs.utah.edu/~sci

Abstract. Building systems that alter program behavior during execution based on user-specified criteria (computational steering systems) has been a recent research topic, particularly among the high-performance computing community [1–5]. To enable a computational steering system with powerful visualization capabilities to run in a distributed computational environment, a distributed infrastructure (or runtime system) is required. This infrastructure permits one to harness a variety of machines to collaborate on an interactive simulation. Building such an infrastructure requires devising strategies for coordinating execution across machines (concurrency control mechanisms), mechanisms for fast data transfer between machines, and mechanisms for user manipulation of remote execution.

We are creating a distributed infrastructure for the SCIRun computational steering system [1]. SCIRun, a scientific problem solving environment (PSE), provides the ability to interactively guide or steer a running computation. Initially designed for a shared memory multiprocessor, SCIRun is a tightly integrated, multi-threaded framework for composing scientific applications from existing or new components. High-performance computing is needed to maintain interactivity for scientists and engineers running simulations. Extending such a performance-sensitive application toolkit to enable pieces of the computation to run on different machine architectures all within the same computation proves useful for some scientists. Not only can many different machines execute this framework, but also several machines can be configured to work synergistically on computations (*e.g.,* farming off compute-intensive pieces to the "big iron").

1 Introduction

In recent years, the scientific computing community has experienced an explosive growth in both the size and the complexity of numeric computations possible. This growth is due, in large part, to both the rapid development of parallel machines and the rapid performance gains of RISC microprocessor technology. One of the significant benefits of this increased computing power is the ability to perform complex three-dimensional simulations. However, such simulations present new challenges for computational scientists. How does one effectively analyze and visualize complex three-dimensional data? How does one solve the problems of working with very large datasets often consisting of tens to hundreds of gigabytes? How does one provide tools that address these computational problems while serving the needs of scientific users?

Data analysis and visualization play critical roles in the scientific process. Unfortunately, these tasks are often performed only as a post-processing step after a large-scale batch job is run. For this reason, errors invalidating the results of the entire simulation may be discovered only during post-processing. Further, the decoupling of simulation and analysis can present serious scientific obstacles to the researcher. An analysis package may provide only a limited data analysis capability and may be poorly matched to the underlying physical models used in the simulation code. To add to the problem, many of these packages are simply unable to effectively manage the vast amounts of data being generated by many simulations–especially those generated on large parallel systems. As a result, the researcher may expend significant effort trying to use a data analysis package only to walk away frustrated.

Given the limitations of the batch/post processing cycle, a better approach might be to break the cycle and improve the integration of simulation and analysis. Scientists want more interaction than is currently present in most simulation codes — a desire the scientific computing community is trying to find better ways to address. Clearly, however, the solutions will not be simple ones, since the problems encountered by computational scientists encompass a wide range of issues. To address some of these needs, we have created the SCIRun problem solving environment. In this chapter, we focus on the SCIRun system [1], addressing the issues of interaction and integration of scientific simulation and visualization in a distributed computing environment.

2 Interactivity and Computational Steering

Interactive scientific visualization and computational steering require low-latency and high bandwidth computation in the form of model generation, solvers, and visualization. Latency is particularly a problem when analyzing large datasets, constructing and rendering three-dimensional models/meshes, and allowing a scientist to alter the parameters of the computation interactively (thus "steering" the computation). However, large-scale computational models often exceed the system resources (memory and storage) of a single machine, motivating closer investigation of meeting these same needs with a distributed computational environment comprised of many machines.

2.1 The SCIRun Problem Solving Environment

To achieve execution speeds needed for interactive three-dimensional problem solving and visualization, a multi-threaded problem solving environment and computational steering system, called SCIRun [6, 7], was built to make use of a shared memory multiprocessor, notably the SGI Power Challenge and SGI Origin 200/2000. The SCIRun scientific problem solving environment is a *computational steering* system [1] that allows the interactive construction, debugging, and steering of large-scale scientific computations. SCIRun can be conceptualized as a *computational workbench*, in which a scientist can design via a dataflow programming model and modify simulations interactively. SCIRun enables scientists, for example, to interactively modify geometric models and change numerical parameters and boundary conditions, as well as to modify the level of mesh adaptation needed for an accurate numerical solution, while at the same time visualizing intermediate (and final) simulation results.

An example of the SCIRun system interface is shown in Fig. 1. The center of the picture displays a graphical representation of the dataflow network. The boxes represent computational algorithms (modules), with the lines representing data connections between the modules. Each module may have a separate user interface, such as the matrix solver interface at the left, that allows the user to control various parameters, including, for example, the maximum number of iterations, error tolerance, preconditioner, and iterative solution method. The window on the right is an interactive three-dimensional viewer that combines visualization output with data probes for exploring the data and model. When

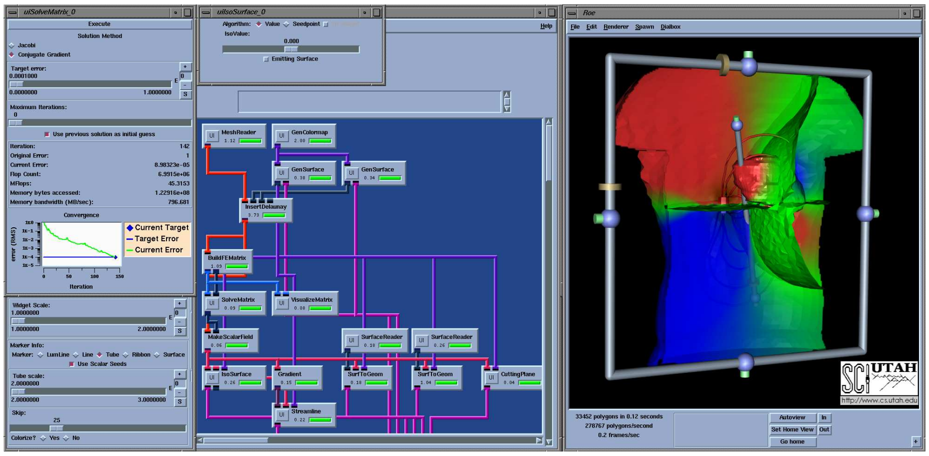

Fig. 1. An example SCIRun network, showing the dataflow programming interface, user interfaces for controlling simulation parameters, and results from an large finite element model.

the user changes a parameter in any of the module user interfaces, the module is re-executed, and all changes are automatically propagated to all downstream modules. The user is freed from worrying about details of data dependencies and data file formats. The user can make changes without stopping the computation, thus "steering" the computational process. As opposed to the typical "off-line" simulation mode - in which the scientist manually sets input parameters, computes results, visualizes the results via a separate visualization package, then starts again at the beginning - SCIRun "closes the loop" and allows interactive steering of the design, computation, and visualization phases of a simulation.

The ability to steer a large-scale simulation provides many advantages to the scientific programmer. As changes in parameters become more instantaneous, the cause-effect relationships within the simulation become more evident, allowing the scientist to develop more intuition about the effect of problem parameters, to detect program bugs, to develop insight into the operation of an algorithm, or to deepen an understanding of the physics of the problem(s) being studied.

Currently, SCIRun manages machine resources to maximize computational efficiency. In a sophisticated simulation, each of the individual components (modeling, mesh generation, nonlinear/linear solvers, visualization, *etc.*) typically consumes a large amount of memory and CPU resources. When all of these pieces are connected into a single program, the potential computational load is enormous. In order to use the resources effectively, SCIRun adopts a role similar to an operating system in managing these resources. SCIRun manages scheduling and prioritization of threads, mapping of threads to processors, inter-thread communication, thread stack growth, memory allocation policies, and memory exception signals (such as segmentation violations). These become important issues as we distribute part of the computation to another machine.

3 Related Work

The concept of steering distributed scientific computation is a relatively new endeavor. However, there are some systems that have investigated aspects of computational steering. We briefly discuss several of these below. The CUMULVS system [3, 8] allows computational scientists with distributed PVM applications to modify a fixed set of simulation parameters while using AVS to visualize the underlying computational model. The Progress and Falcon systems [2, 9] allow application programmers to develop distributed steerable applications through source code modification with the steering being assisted by a runtime system. Basically, these systems explore computational steering by exposing portions of the simulation through their interfaces. The SCIRun system provides a more holistic approach where the entire computation modeling, simulation and visualization are built and executed within a problem solving environment. Such a complete system provides greater flexibility by allowing the user to adapt computational steering at runtime rather than at compile time.

3.1 CUMULVS

The CUMULVS library [3, 8], developed at Oak Ridge National Laboratory, acts as a middle layer between PVM applications and the commercially-provided AVS visualization package [10]. CUMULVS is implemented as a user-programmed

AVS module. It has been used in distributed computational fluid dynamics applications. One of the properties of CUMULVS is that viewers can dynamically plug and unplug from running distributed simulations and visualization packages. After initializing a viewer, the application programmer can provide a list of parameters to be adjusted on-the-fly in a CUMULVS steering initialization procedure call. Separate procedure calls are used for altering scalar or vector parameters from within the application. CUMULVS supports collaborative visualization and simulation by permitting multiple viewers to view and steer a running application. Viewers can attach and detach dynamically during a simulation. An interesting checkpointing capability for rolling back and restarting a failed program run has the potential to allow cross-platform migration and heterogeneous restart of an application.

3.2 Progress and Magellan

The Progress Toolkit [2], developed at the Georgia Institute of Technology, assists application programmers in developing steerable applications. Programmers "instrument" their applications with library calls, using *steerable objects*, which can be altered at runtime through the use of the Progress runtime system. Steerable objects include sensors, actuators, probes, function hooks, complex actions, and synchronization points. Progress uses a client/server program model.

Developed by the same group, the Magellan Steering System [11] is derived from the Progress system, and extends the steering clients and steering servers model used in the initial system. This system uses a specialized specification language, ACSL, which provides commands for monitoring and steering using probes, sensors and actuators. However, application codes still must be instrumented with these commands in order to utilize the steering capabilities of this system. These systems have been used for Molecular Dynamics simulations.

Both systems are layered on top of the Falcon runtime system for on-line monitoring of running simulations. The Falcon system [9], also developed at the Georgia Institute of Technology, monitors a running program, capturing information ranging from a single program variable, much as a debugger would, to complex expressions. It also permits the monitoring of performance data, with interfaces to visualization systems, such as Iris Explorer [12]. Applications can be observed or modified through the use of a probe or actuator, respectively. However, decisions are made about which steering actions to take based on previously encoded routines stored in a steering event database located on a steering server. Clearly, codes must be instrumented to use these facilities via procedure calls and, possibly, the coercion of existing scientific codes to fit a client/server model.

4 Distributing SCIRun

Large-scale scientific computations require multiprocessors of some form due to the vast amount of data inherent in these applications/simulations. However, the computer architecture of multiprocessors varies greatly, differing in the number of processors, communication mechanism, operating system, and memory model, etc. Using abstraction to hide the underlying details of the memory model (*i.e.,* shared memory versus distributed memory) from the application allows us to take advantage of the range of machine types, from shared memory multiprocessors to massively parallel processors. However, such an abstraction creates extra layers in the software infrastructure that generally result in increased software overhead. One must strike a balance of exposing enough details to efficiently tune the application without losing portability.

To achieve our goal of computational steering of distributed simulations, we first created a mechanism for using SCIRun across multiple machines, then we can solve the problem of running a scientific code on a distributed memory architecture. This discussion focuses on the infrastructure mechanisms needed to run one SCIRun computation across multiple machines while maintaining control of the computation from the user's local workstation. The goal of SCIRun has been to maintain interactive rates for users constructing and manipulating models and simulations. To be successful with a distributed version of SCIRun, we must not impair interactive speeds by incurring excessive network latency.

To meet the demand for growing computational problem sizes, we have extended the existing communications infrastructure within SCIRun to include cross-machine communications and execution. In this way, we can preserve existing system functionality for shared-memory executions, as well as providing more architectural flexibility. In constructing a distributed system, our goal is to provide SCIRun users with the illusion of running on a single computer, while they are really utilizing several computers with potentially very different characteristics. Distributing SCIRun has been largely motivated by the large-scale computational needs and the desire to link remote computing facilities for the Utah C-SAFE ASCI Alliance Center and the NCSA PACI Alliance.

4.1 Software Architecture for Distributed SCIRun

The model for running SCIRun on multiple machines calls for the scientist to initiate the computation (*i.e.,* start SCIRun) on a graphics-capable machine, to compose an application from existing SCIRun components (*modules*), to specify which of these should run remotely, and then to start execution of the computation (*dataflow network*). Interacting with modules, viewing the rendered visualization, and interacting with three-dimensional data probes (*widgets*) within the visualization all occur on the initiating machine, which we denote as the Master. The compute-intensive dataflow subnetwork executes on the remote machine, which could be a multiprocessor machine.

We abstract away the differences between running on one or more than one machine by modifying the scheduling, inter-module communication, user interface mechanisms, and other "internal" portions of SCIRun that an application writer would typically never see. Making modifications to the infrastructure allows programs that have been designed for the SCIRun problem solving environment to function in either a multi-machine, networked environment or a single-machine, shared memory environment.

The software architecture, shown in Fig. 2, depicts this solution for a computation distributed across two machines. This architecture is based on a master/slave model. Steering and visualization takes place on the local machine, the user's computational workbench, which we call the Master SCIRun instance. The remote instance of SCIRun will be the Slave. The application writer configures a program by connecting together existing modules, choosing a subset of modules to execute remotely, and choosing the target remote machine (slave). In Fig. 2, the application program (network) consists of reading in data using a `Reader` module, then executing `RMod1`, `RMod2`, and finally the `Salmon` module to render the visualization, in that order. Modules to be run remotely have a proxy module (copy) that remains on the Master side. This proxy keeps a copy of the state and runtime information necessary for the scheduler to perform its task, while the module that performs the actual computation is instantiated on the Slave. The architecture of the Slave consists of a daemon, the *Slave Controller*, which listens on a communication channel for directives from the Master while controlling the modules on the remote side. These directives follow:

– Instantiate a module.
– Destroy a module.
– Instantiate a connection between two modules.
– Destroy a connection between two modules.
– Execute a module.
– Re-execute some portion of the remote network.

This set of directives was based on control commands in the current shared-memory version of SCIRun and completely provides the necessary control for the management of remote computation. By simply extending existing messages and functionality of the scheduling code it is possible to provide all required support for a remote machine.

Control communications is largely uni-directional, moving from the Master to the Slave, using a fire and forget communications strategy. The need for control messages from the Slave to Master is mitigated by having all user interactions initiated on the Master, with the Master then sending notification to the Slave of any changes needed.

In addition, most module-to-module communications are one-way communications. For example, Fig. 2 shows module `Reader` connected to module `RMod1`. Data flows from the output port of `Reader` to the input port of `RMod1`. In most cases, this communication happens after the module, `Reader`, has finished executing, when resulting data is passed out its output port. For remote communications, then, the communications channel must be set up before `Reader` has started to execute.

On the other hand, user interaction with remotely running modules requires bi-directional communication between the Master and Slave. Modules obtain the most current parameter values before executing in SCIRun. If modules are remote, they must request from the Master the current value. In response, the

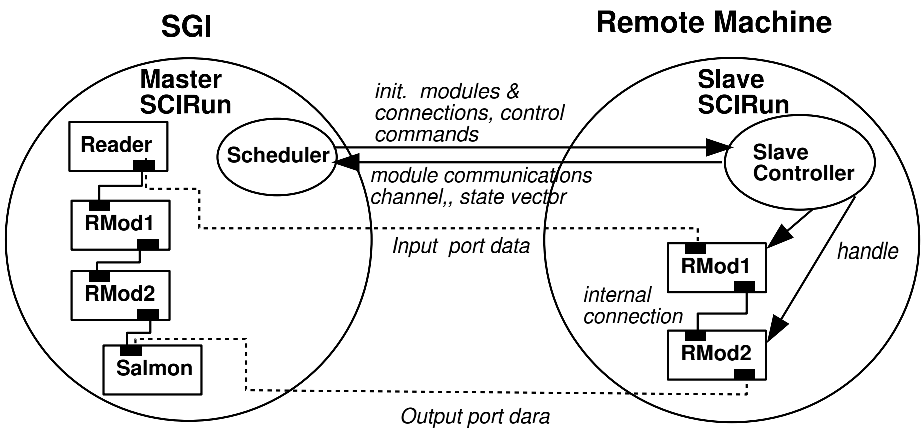

Fig. 2. Software architecture for partitioning a SCIRun dataflow network across two machines. Communications paths are shown. The Slave Controller acts as a daemon listening on a communications channel for commands from the Master SCIRun scheduler. It creates modules, connections between modules, and destroys both. It also forwards control messages, such as `execute()` and `resend()`, to the proper module. Data flows in and out of modules through the typical input and output port mechanism, although a special communication channel will be established for remote module-to-module communication.

Master sends these values to the Slave. Additionally, various GUI updates, displayed on the Master, are sent by the Slave modules running remotely, such as updating the progress bar or runtime indicator for a remote module.

Designing and implementing support for distributed computational environments in the existing SCIRun architecture can be broken down into three separate phases.

1. Design and implement distributed control infrastructure to be integrated within existing SCIRun control mechanisms (specifically the scheduler).
2. Design and implement module-to-module data communication that spans machines.
3. Design and implement user interaction and steering of remotely running modules.

4.2 Distributed Control Infrastructure

The SCIRun scheduler controls the execution order of modules and tracks and signals modules that need to run again. SCIRun is based on a hybrid form of a dataflow network where data passes from upstream modules to downstream ones. To ensure concurrency control, we use one scheduler for all machines working collaboratively on a SCIRun computation. A master-slave configuration yields a single point of control from which module scheduling decisions can be made for the entire network consisting of both local and remote modules. For each remote module, the Master SCIRun instance, running on the user's workstation, instantiates proxy modules to keep the internal module state necessary for scheduling and control. Thus, the single scheduler model is efficient and effective.

4.3 Remote Data Transfer

Since we must maintain the dataflow metaphor within one SCIRun application program (or dataflow network), we provide a mechanism for data transport from modules running on one machine to downstream modules running on different machines. We accomplish this remote dataflow by opening a communication channel, in this case a socket, for every data pipe linking two modules running across machine boundaries. The data are then bundled into messages and sent across the channel from the module's output port. These data channels are instantiated and torn-down when the user makes, or destroys, a connection between modules which cross machine boundaries. Connected modules running on the same machine, even though remote, use the current SCIRun shared memory data communication.

4.4 Remote User Interaction and Steering

SCIRun succeeds in allowing a range of intuitive user interactions with parameters of the model, choice of simulation techniques and their parameters, and the resulting visualization. When we move to a distributed environment, careful attention must be paid to maintaining close coupling between user action and system response. Further, providing visual feedback to the user about the run-time performance traits helps the user choose appropriate modules to execute remotely.

We have built a remote communication mechanism into our Tcl wrapper calls used by the GUI. These are based on a combined *push* and *pull* strategy. Modules always check the most current user-selected parameters before using them, and remote modules do the same. Similarly, a progress bar, which indicates the percentage of the computation completed, is updated on the proxy module residing on the Master by the corresponding Slave module. Finally, if a user changes a parameter in one of the steering windows, the module is rescheduled in both the single-machine and distributed versions of SCIRun.

5 Conclusions and Future Work

We have demonstrated an effective architecture for steering computations with SCIRun by combining machines that have some concept of a global address space. While this work is preliminary, we have shown that by extending the underlying control, data transfer, and user interaction mechanisms of SCIRun, the user need not be concerned with the details of inter-machine communication. Users can choose which modules in the dataflow network are to be run remotely and the target machines on which they should execute. Future work consists of detailed experiments with a variety of platforms and building in support for parallelism in a distributed memory machine. We will also investigate other communication mechanisms, as well as automatic scheduling algorithms for the distribution of the various application modules given specific network and computing infrastructure.

6 Acknowledgments

This work was supported by the NCSA PACI Alliance and the Department of Energy. Equipment was funded by the SGI Visual Supercomputing Center and the National Science Foundation.

References

1. S.G. Parker, M. Miller, C.D. Hansen, and C.R. Johnson. An integrated problem solving environment: the SCIRun computational steering system. In Hesham El-Rewini, editor, *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*, volume VII, pages 147–156. IEEE Computer Society, Jan. 1998.
2. J. Vetter and K. Schwan. Progress: A toolkit for interactive program steering. In *Proceedings of the 24th Annual Conference of International Conference on Parallel Processing*, 1995.
3. G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. Cumulvs: Providing fault-tolerance, visualization and steering of parallel applications. *SIAM*, Aug. 1996.
4. D.J. Jablonowski, J.D. Bruner, B. Bliss, and R.B. Haber. Vase: The visualization and application steering environment. In *Proceedings of Supercomputing '93*, pages 560–569. IEEE Computer Society Press, 1993.
5. D.A. Reed, C.L. Elford, T.M. Madhyastha, E. Smirni, and S.E. Lamm. The next frontier: Interactive and closed loop performance steering. In *Proceedings of the 25th Annual Conference of International Conference on Parallel Processing*, 1996.
6. S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 1–44. Birkhauser Press, 1997.
7. S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.
8. J.A. Kohl, P.M. Papadopoulos, and G.A. Geist. Cumulvs: Collaborative infrastructure for developing distributed simulations. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. Minneapolis, Mar. 1997.
9. W. Gu, G. Eisenhauer, E. Kramer, K. Schwan, J. Stasko, and J. Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings of the 5th Symposium of the Frontiers of Massively Parallel Computing*, pages 422–429, Feb. 1995.
10. C. Upson and et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics & Applications*, 9(4):30–42, July 1989.

11. J. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proceedings of the 11th International Parallel Processing Symposium*. Geneva, Switzerland, Apr. 1997.
12. Iris Explorer - http://www.nag.co.uk/Welcome_IEC.html.