Simulation Steering with SCIRun in a Distributed Memory Environment

Michelle Miller, Charles D. Hansen, Steven G. Parker, Christopher R. Johnson
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
{mmiller, hansen, sparker, crj}@cs.utah.edu
http://www.cs.utah.edu/~sci

Abstract

SCIRun is a shared memory problem solving environment (PSE) that provides the ability to guide or steer a running computation. Extending such a tightly-integrated, performance-critical framework to enable pieces of the computation to run on different memory architectures all within the same computation would prove very useful. In this way, many different machines could execute this framework, as well as permitting configurations of heterogeneous machines to work synergistically on computations, farming off compute-intensive pieces to the "big iron." In this paper, we describe a distributed version of the SCIRun problem solving environment.

1 Introduction

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 (among other things). In particular, hiding, through abstraction, the underlying details of the memory model (*i.e.,* shared memory versus distributed memory) from the application should allow 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, but not too many architecturally-specific details, lest the application lose portability.

1.1 SCIRun: User Interaction and Steering a Computation

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 [1, 2], 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 (PSE) is a *computational steering* system [3] 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 to modify geometric models and interactively change numerical parameters and boundary conditions, as well as to modify the level of mesh adaptation needed for an accurate numerical solution.

An example of the SCIRun system interface is shown in Figure 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. The window in the upper right hand corner is an interactive three-dimensional viewer that combines visualization output with data probes for exploring the data and model.

When 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.

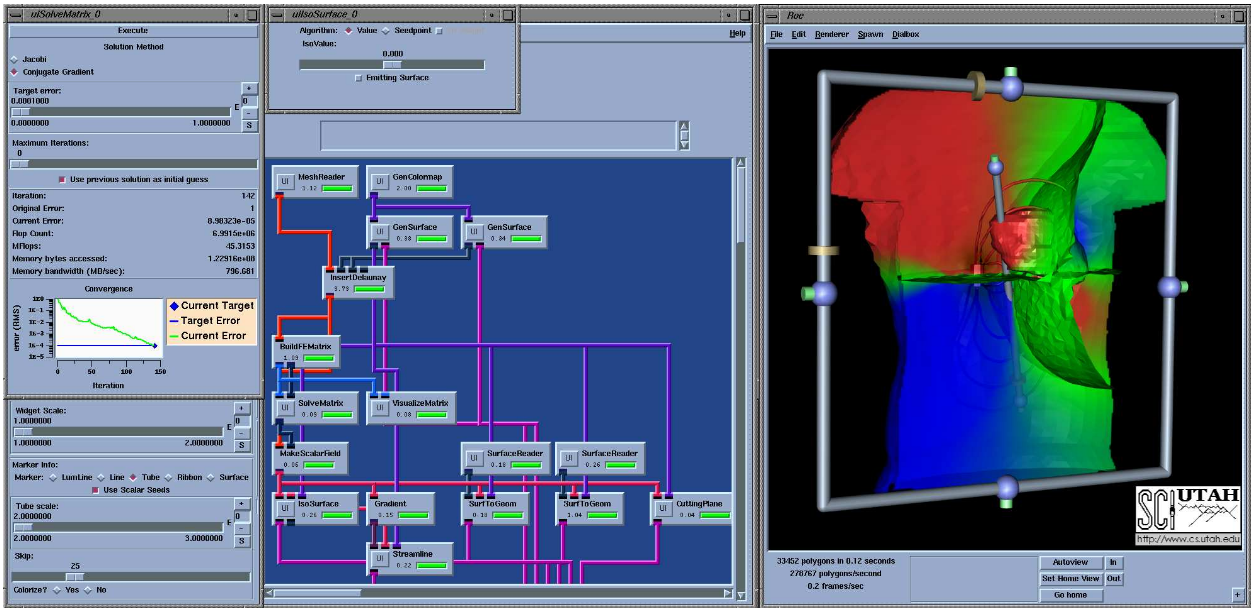

Figure 1: An example SCIRun network, showing the dataflow programming interface, user interfaces for controlling simulation parameters, and results from an large finite element model.

1.2 Interactivity Requires Efficiency

Interactive scientific visualization and computational steering requires 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, the scaling ability of shared memory systems (or distributed shared memory systems) is limited, motivating closer investigation of meeting these same needs with combined shared memory and distributed memory machines.

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.

2 Distributing SCIRun

To meet the demand for growing computational problem sizes, as well as meet the needs of the computational community who use large distributed memory machines, 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, defined as a collection of independent computers that appear to users of the system as a single computer [4], our goal is to provide SCIRun users with the illusion of running on a single computer, while they are really utilizing several computers with very different characteristics. Distributing SCIRun has been largely motivated by large-scale computational needs and usage of remote computing facilities necessitated by the Utah DOE ASCI Alliance Center and the NCSA PACI Alliance.

Interactive rates for final solution, visualization, and steering interaction must be maintained. Tying together very different machines to work together on pieces of the same computation at interactive rates is challenging. By building an underlying base, or infrastructure, to support fast, efficient communication of application-specific data and control within the PSE, we can maintain acceptable interactive rates.

This work demonstrates the feasibility of combining a high-end, graphics-capable shared memory machine, in this case an SGI Origin 2000, with a distributed memory machine, in this case an IBM SP-2, for the use of an interactive computational steering and visualization system, namely SCIRun. Other hardware configurations, such as a workstation/supercomputer configuration, could be used to run this version of SCIRun. We have extended the SCIRun problem solving environment without drastic effects on existing code. We achieve this by extending the existing communications layer and converting multi-threading capabilities into directives recognizable by large MPPs. We have designed a layered architecture for SCIRun that abstracts away the machine-dependent details, although we would like the user to be able to steer the performance aspects of each execution. Figure 2 shows an example of such a layered architecture.

Requirements of a distributed version of SCIRun include limited performance degradation, and minimal source code changes in the domain-specific modules (application code). In addition, supporting large parallel machines could allow the problem size to scale upwards, permitting solutions of larger problems than currently possible using a shared memory machine.

2.1 Methodology

The model for running SCIRun on heterogeneous machines calls for the user 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 (*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. The compute-intensive dataflow subnetwork executes on the remote machine, potentially a distributed memory architecture. Data locality for the remote modules is maintained and remote inter-module communication is optimized through the abstract layers.

We achieve the abstraction 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 networked, distributed memory environment or a shared memory environment.

2.1.1 Architectural Software Changes

The software architecture, shown in Figure 3, depicts this solution for a computation distributed across two machines. This architecture is based on a master/slave model. Steering and visualization will take place on the local machine, the user's computational workbench, which becomes the **Master** SCIRun instance. The remote instance of SCIRun will be the **Slave**. The application writer configures a program by connecting together existing modules, some of which will be run remotely. In Figure 3, the application program (network) consists of reading in data using a `Reader`

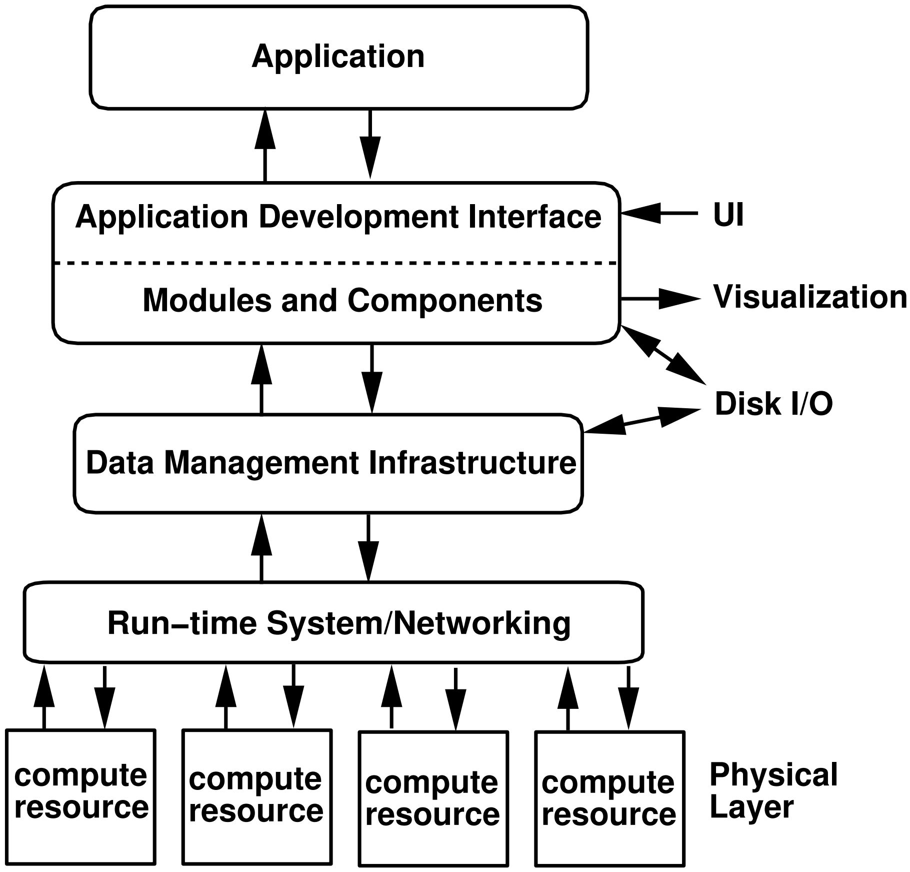

Figure 2: A layered architecture that hides the details of the number and type of machines being used while providing mechanisms for user interaction with a running computation.

module, then executing `RMod1`, `RMod2`, and finally `Salmon` to render the visualization, in that order. Modules to be run remotely have a skeleton module (copy) remain on the Master side to keep the state and run information needed for the scheduler to perform its task, while the real module is instantiated on the Slave. The architecture of the Slave will consist of a daemon, the *Slave Controller*, which listens on a communication channel for directives from the Master, keeping track of 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.

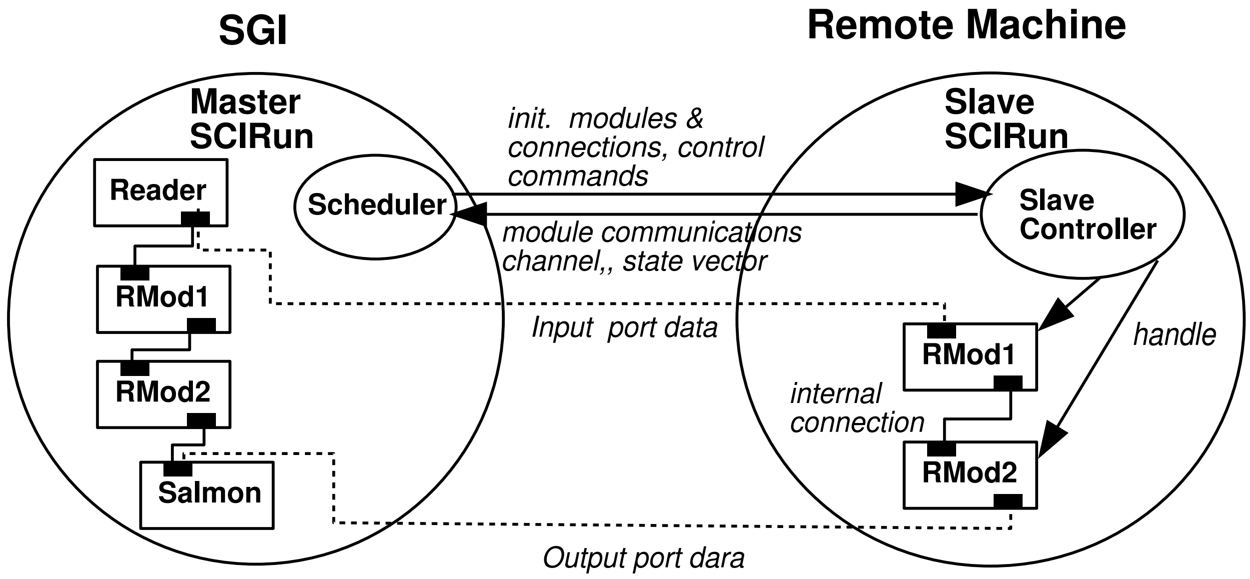

Figure 3: Software architecture for partitioning a SCIRun dataflow network across two machines. Communications paths are shown. The Slave Controller acts as a daemon listening on a communications channel for commands from the Master SCIRun scheduler. It creates modules, connections between modules, and 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.

This set of directives was based on control commands in the current shared-memory version of SCIRun, extending existing messages and functionality of the scheduling code to support 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 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, Figure 3 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.

Designing and implementing support for utilizing distributed memory environments into 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 communications that span machines.

3. Parallelize modules to run on multiple nodes of a distributed memory machine. Build a central module controller that decides how many nodes and which ones should run the code,

and handles the data communication issues.

2.1.2 Distributed Control Infrastructure

To permit modules to run on two different machines, the SCIRun scheduling mechanism was extended. The scheduler keeps track of how modules are arranged in a dataflow network (*i.e.,* execution order), and which modules need to execute. The shared memory version of SCIRun collects all those modules which need to execute in a first pass through the dataflow network, then it calls the `execute()` method of those modules in the second pass. The distributed version of SCIRun must signal remote modules to execute in the second pass. It also permits user restart of parts of the computation (steering) by signaling an output port to resend its data downstream after a user has modified parameters associated with the module below the output port, thus re-calculating. It is important to maintain just one locus of control, the scheduler, but a software daemon is also needed on the remote side to control modules executing there.

2.1.3 Remote Module-to-Module Communication

In SCIRun, modules are connected by *data pipes* through which data flows as the computation proceeds and control moves from an upstream module to a downstream one. The data pipes are connected to modules at an input or output port, consistent with the direction of data flowing from the top to bottom of the network. The data are bundled into a message and sent from an output port for a given module to an input port for the following module(s). These messages are sent by an output port writing directly into a public mailbox for the input port. In the current shared-memory implementation, data are passed using pointers to globally-known data. In the cross-platform model, this mechanism is replaced by streaming a copy of the data across a machine-to-machine communication channel, requiring marshaling and unmarshaling of the data on either side. We have chosen TCP/IP sockets as the communications channel to simplify our prototyping work.

2.1.4 Parallelize Modules

A subset of computationally-intensive modules have been targeted to run on a distributed memory machine. The parallel shared memory version of these modules partitions data in a data parallel fashion, spawning the desired number of threads to operate on each data partition, typically one thread per processor. Threads communicate via mailboxes. To leverage this parallelizing mechanism to work with distributed memory, we have built a central module controller that acquires the number of nodes needed to achieve the specified level of parallelism, segments the data accordingly, and sends code and data to the nodes, using typical scatter/gather techniques. This work utilizes MPI for communication.

2.2 Results

To evaluate the effect of combining machines, in the final paper we will benchmark two different SCIRun scientific application codes using both architectures, the original SCIRun architecture using shared memory and the new heterogeneous, distributed architecture. We will have achieved low-latency, high-bandwidth communications.

3 Conclusions

We have demonstrated an effective architecture for steering computations with SCIRun using a combination of a graphics-capable workstation, or shared-memory multiprocessor, and a distributed memory parallel multiprocessor. While this work can be considered preliminary, we have shown that by modifying the underlying control mechanisms and thread-to-thread communications (*i.e.,* mailboxes) of SCIRun, the user need not be concerned with the details of inter-machine communication.

References

[1] 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.

[2] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[3] S.G. Parker, M. Miller, C.D. Hansen, and C.R. Johnson. An integrated problem solving environment: The SCIRun computational steering system. In *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*. IEEE Computer Society Press, Jan. 1998.

[4] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.