SimX meets SCIRun: A Component-based Implementation of a Computational Study System*

Siu-Man Yau, Eitan Grinspun[†], Vijay Karamcheti, and Denis Zorin
Courant Institute of Mathematical Sciences, New York University
[†] Department of Computer Science, Columbia University
{*smyau,vijayk,dzorin*}@*cs.nyu.edu, eitan@cs.columbia.edu*

Abstract

This paper describes the ongoing implementation of the SimX system for multi-experiment computational studies within the SCIRun problem solving environment. The modular, component-based nature of SCIRun enables a natural integration of the SimX runtime modules with the simulation codes that constitute the experiments underlying the study, and provides a rich steering and visualization environment for study interactions. Experience with a computational study involving a SCIRun defibrillator device simulation code (DefibSim) highlights these advantages, and identifies several avenues for future work.

1 Introduction

SimX is a parallel software system for conducting interactive multi-experiment computational studies. Its design was motivated by the recognition that computer simulation has become an integral part of the scientific method, often delivering deeper insights into complex physical processes than possible using only the traditional dyad of theory and experiment. Computer simulation manifests itself in the scientific exploration process in the form of *computational studies* built out of multiple *computational experiments* corresponding to individual runs of simulation software. Examples of such studies range from exploration of design spaces in engineering to molecular simulations for drug design. Driven by the availability of higher-performance computational resources, the number of experiments involved in computational studies has increased dramatically, and their structure has become more complex.

However, what has not fundamentally changed is the pattern using which such studies have been conducted. Typically, the scientist (1) makes some a priori decisions about simulation parameters; (2) runs the simulation, or a batch of simulations; and (3) analyzes the output. These steps are repeated as necessary; the organization of the study is left to the scientist or domain application developers. While this is a reasonable pattern for studies involving small numbers of experiments, for hundreds and thousands of experiments, the need to set up individual experiments manually or using simple predetermined parameter variation patterns becomes a severe limitation. For efficient exploration, the scientist needs high-level tools for manipulating collections of simulations, and the ability to adjust the parameter space traversal pattern continuously, based on partial results aggregated from large numbers of running experiments.

The SimX system aims to support intuitive, high-level management of computational studies by permitting users to interact at the level of aggregate studies (instead of individual experiments), by providing continuous feedback about running simulations, and by dynamically adapting allocation of system resources to reflect changing priorities set by the user. To achieve these goals, SimX relies on a more permeable interface between parallel system software and numerical simulation codes than is assumed by state-of-the-art middleware in grid computing [7, 2, 4, 6] and problem solving environments [8, 5, 3] that target related problems. These interfaces help in two primary ways:

- they enable SimX to gather knowledge about how individual experiments are contributing to progress towards overall study goals; and
- they provide information about the internal state and resource requirements of individual experiments to enable substantially more efficient mapping of these experiments to computational resources.

A prototype standalone implementation of the SimX system was described in an earlier paper [10], along with a simplified bridge design study highlighting the kinds of system-level optimizations SimX is capable of performing and their impact on overall user experience.

This paper describes our ongoing efforts to build a more complete implementation of SimX in the context of

*Supported by a collaborative NSF grant comprising CSR-0615225, CSR-0614770, and CSR-0615194.

the University of Utah's SCIRun problem solving environment [8]. SCIRun is a modular, component-based system for interactive scientific computing applications, which permits the expression of a simulation experiment along with its steering and visualization interfaces as a dataflow net. SCIRun provides the foundation for the CCA-compliant [1] SCIRun2 framework [11]. Implementation of the SimX system within SCIRun offers several advantages:

1. It permits natural component-level integration of the various SimX runtime modules with existing simulation functionality already expressed in component form (a monolithic simulation code would have required extensive rewrites to provide the permeable interface SimX relies upon);
2. It supports easier experimentation with alternate implementations for the SimX modules, necesssary as we gain experience with different applications;
3. It enables us to leverage existing scientific computing software, especially tools supporting user interaction and visualization, both in SCIRun/SCIRun2 proper and in other toolkits to which SCIRun2 provides an interface, including CCA; and finally,
4. It ensures access to our techniques through a well-established framework, which we expect to be increasingly common in scientific computing applications, thus increasing the potential for practical impact.

As an example of the third point above, we have experimented with the SimX implementation using a computational study built around an existing SCIRun-based defibrillator design simulation code (DefibSim) [9]. In addition to testing our implementation, experience with this application has validated the advantages above and exposed several avenues for future development of the SimX system.

The rest of this paper is organized as follows. Section 2 briefly reviews the SimX architecture and its standalone implementation. Sections 3 and 4 describe in turn, the SimX implementation within SCIRun and our experiences with the defibrillator design study. We conclude by identifying several next steps for the SimX system in Section 5.

2 SimX Architecture

A SimX computational study involves the systematic exploration of a *design* or *parameter* space of computational experiments so as to identify a desired *target set*. The latter is usually indirectly specified, in terms of the design space points where the experiment outputs defining the *performance* or *observation* space, satisfy certain constraints.

2.1 An Example Computational Study

Our earlier paper [10] described a computational study looking at the problem of designing an elastically deforming bridge with four supports, two of which are fixed at the

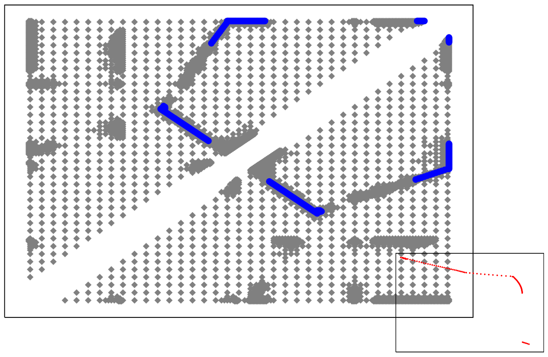

Figure 1. The Pareto optimal points for the bridge design study in the *design space* and the *performance space* (bottom right overlay).

endpoints. This study's design space was defined in terms of two parameters specifying the locations of the two non-fixed supports. The objective was to find a set of bridge designs, which best traded off among two performance measures: the cost of bridge construction and the maximal deformation of the bridge. Formally, this objective translates to identifying the *Pareto optimal* points, the target set of parameter points such that there is no other point that can improve upon all dimensions of the performance metric.

SimX permitted this study to be managed at a high level: the user provided (1) the individual computational simulations, capable of computing deformation of the bridge (modeled as a one-dimensional rod elastically deforming in two dimensions) given specific values for the parameters and a predefined cost function; (2) specifications of the parameter and performance space domains; and (3) requirements on the target set of interest. The underlying system automatically made decisions about which parameter points to simulate, and in what order, so as to provide increasingly refined estimates of the Pareto frontier. Figure 1 shows the Pareto optimal points identified for this study in the design and performance spaces: the former also shows which parameter points were evaluated by SimX. The non-uniformity of these evaluations highlights a major advantage of SimX, namely its ability to dynamically allocate computational resources to make progress towards the (possibly changing) high-level objectives of the study.

2.2 SimX Runtime Modules

The SimX system supports such computational studies using the high-level architecture shown in Figure 2. The core functionality of the system is realized by two modules, the *active sampler* and the *resource allocator*. A *shared object space layer* provides a machine-wide repository of shared state, including both simulation checkpoints and different meta-information about the ongoing study.

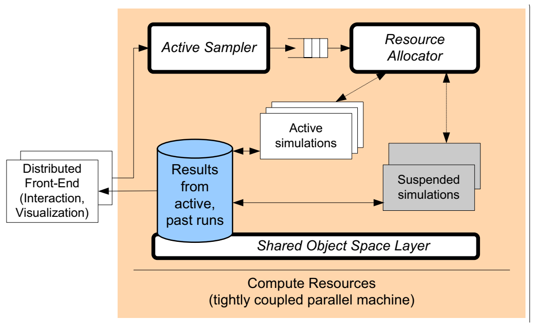
Active Sampler
Resource Allocator
Active simulations
Suspended simulations
Distributed Front-End (Interaction, Visualization)
Results from active, past runs
Shared Object Space Layer
Compute Resources (tightly coupled parallel machine)

Figure 2. Architecture of the SimX platform.

The active sampler converts user specifications of parameter and observation space domains, the target set, and priority/time target/precision functions into a collection of sample points in the parameter domain for which simulations need to be run. Whenever new user input arrives (communicated to the active sampler by the user interface modules), the sample set is adjusted. The active sampler occupies an intermediate position between system and application software. Adding domain knowledge to the sampler is likely to enhance its performance, but narrow the applicability of the system; making the sampler completely application-independent may result in suboptimal sampling strategies in important cases.

The resource allocator manages the pool of simulations of the computational study. It receives its directives from the active sampler module via a task list, and responds by starting new simulations and modifying the parameters of or terminating active ones. The goal of the resource allocator is to optimize completion time for these simulations.

Standalone SimX The standalone implementation of the SimX architecture described in [10] relied on two types of communicating processes: *managers* and *simulation containers*. Simulation containers constitute the worker pool to which the manager farms off individual simulations. These processes communicate via a generic *satellite* interface; in the standalone implementation, the processes are just standard UNIX processes, and the satellite interface is implemented using TCP socket calls.

In addition to these explicit interactions, the processes also implicitly communicate using the shared object space layer. Simulations running within the simulation containers write checkpoints and results to this layer, while manager processes optionally write meta-information about the study; this information is read by other simulations and the manager. The dominant usage pattern indexes the information using the parameter space coordinates, so our implementation of the layer, SISOL, provides a spatially-indexed interface: objects are associated with spatial coordinates, and can be retrieved using neighborhood queries.

3 SimX/SCIRun Integration

3.1 SCIRun Architecture

SCIRun applications are composed out of *modules* connected together via datalinks from the output *data port* of a module to the input port of another [8]. The resulting directed acyclic graph is called a *dataflow net*, a reference to SCIRun's dataflow model of computation.

The SCIRun runtime associates each module with its own thread, which *executes* the module's code. The code blocks until the data in the module's input ports are available, does some calculation, and then sends data to the module's output port. The SCIRun environment flows this data to the input ports of other modules, and triggers their execution. Thus, executing a module can cause all the modules reachable from to it to execute in the DAG's partial order, as the data is sent down the DAG.

Each module can also optionally be associated with a *user interface* (UI) module, which is executed whenever the user clicks on a module. The UIs are the way a user can interactively steer a SCIRun application, by altering the parameters used in the code associated with the application module and viewing any information the latter provides.

3.2 SimX Modules and System Architecture

Integration of SimX within SCIRun requires both additional modules (reflecting the various aspects of SimX functionality described in Section 2) and a system architecture more suitable for a computational study environment.

Instead of the traditional single SCIRun process associated with a steerable application, SimX/SCIRun-based computational studies rely upon a front-end SCIRun process and several instances of back-end SCIRun processes. The front-end process runs on the user's terminal and permits interaction with the ongoing study like a normal SCIRun session. The back-end processes, optionally run on remote computers, correspond to the *simulation container* processes, and perform the actual computation of the study.

The front-end process includes two modules: **SimX-Manager** and **SelectExperiment**. The first module combines the functions performed by the SimX manager and active sampler components, and is responsible for selecting a subset of experiments from the design space to explore, issuing these experiments to simulation container SCIRun processes, collecting the results, and sending the results to the SelectExperiment modules. We are currently separating out the sampler functionality into its own module to allow users to plug-in different sampler policies. **SelectExperiment** modules provide UI functionality to allow users to examine the completed experiments and pick an experiment to send downstream for visualization. We are extending this module to allow the user to visualize progress on the entire study instead of the results of just one experiment.

The back-end processes augment the existing SCIRun application net with an additional SimX module. **Arbiter modules** are responsible for communicating with the SimXManager modules on the manager process, retrieving experiment parameters, and executing the net that performs the simulation. Arbiter modules require the use of a downstream helper module, which receives the performance metrics of the initiated experiment and passes them back onto the Arbiter module; the latter passes these results to the manager process and receives the next piece of work.

Both the front- and back-end processes also utilize additional **Shared Object Layer Reader/Writer** modules to read or write data (currently, checkpoints) into the Shared Object Layer. The current implementation supports SCIRun's primitive Fields datatype, and is being extended to include other datatypes. As the SimX implementation matures, additional modules to handle transformations to and from checkpoints will also be developed.

The component- and dataflow nature of SimX/SCIRun permits reuse of existing application nets as the basis for larger computational studies with minor (if any) code modifications. As an example, consider the defibrillator device simulation described in Section 4, whose original net includes four main subnets that (1) allow users to input experiment parameters (thereby steering the experiment); (2) run the experiment; (3) visualize the experiment results; and (4) calculate the experiment's performance metrics. Figure 3 shows the SCIRun nets for the simulation container (top) and manager processes (bottom) built out of the original nets primarily by manipulating module linkages. The simulation container net retains the original simulation-execution and performance metric-extraction subnets, and replaces the user input modules with SimX Arbiter modules (top right). An extra connection is provided between the performance metric-extracting subnet and the Arbiter helper module (bottom left). Finally, the Shared Object Layer Writer module is added to the net (right), so that the result of the simulation can be stored.

The manager net retains the original visualization subnet, while replacing the user-input and the simulation execution nets with the SimXManager and SelectExperiment modules. Notice that the two subnets are unconnected. This is because the SimXManager subnet is executed whenever a new experiment result arrives, but the SelectExperiment modules get executed only as a result of user intervention. Further downstream, a Shared Object Layer Reader module (left) reads the experiment results of the user-selected experiment, and sends it to the visualization subnet (bottom).

4 Defibrillator Design Computational Study

To evaluate the performance of our integrated SimX/SCIRun system, we experimented with a computational study looking at defibrillator device design.

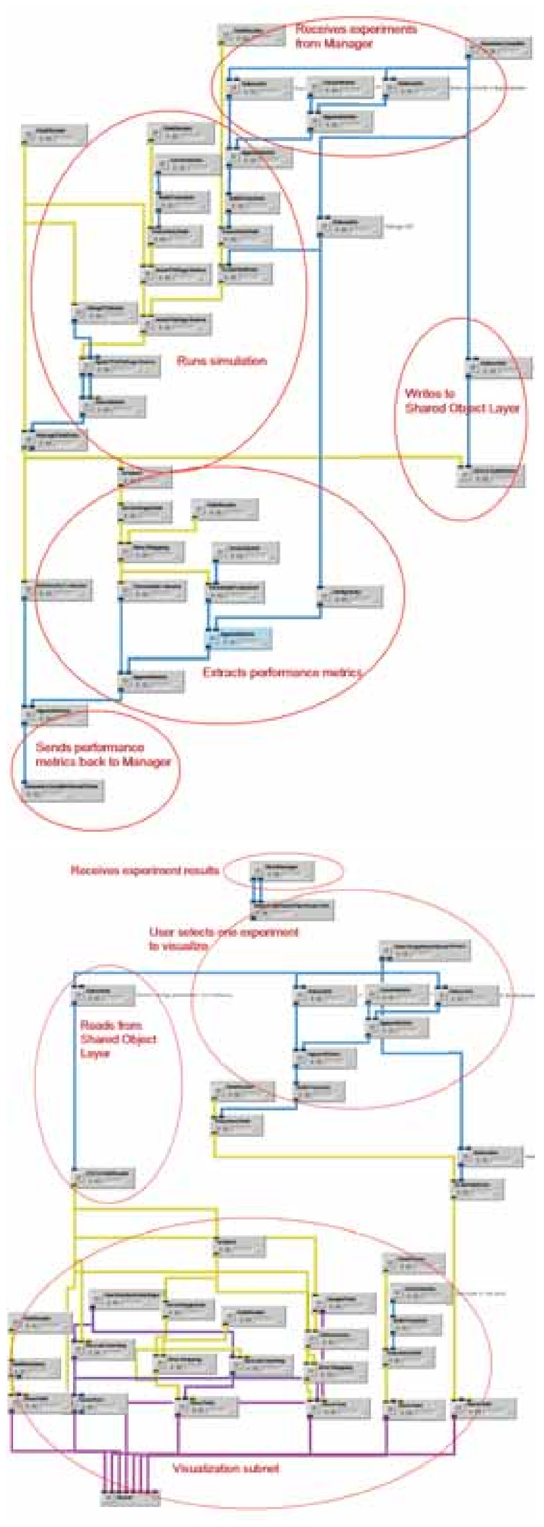
Receives experiments from Manager
Runs simulation
Writes to Shared Object Layer
Extracts performance metrics
Sends performance metrics back to Manager
Receives experiment results
User selects one experiment to visualize
Reads from Shared Object Layer
Visualization subnet

Figure 3. SCIRun nets for the simulator container (top) and manager (bottom) processes.

The underlying simulation code is an existing SCIRun application net, DefibSim [9], which takes as its inputs a mesh representing the conductivity of the human torso, the positions of two electrodes, and the potential difference between the electrodes, and calculates as its output the electric potential inside the torso mesh.

The problem is governed by the Poisson equation relating the local conductivity tensor, the voltage over the domain, and the current source. The discrete form of the equation approximates the divergence of the electric field with the stiffness matrix A and the voltages at the nodes with the vector Φ. They yield a zero current source, the system: $A\Phi = 0$. The electrodes at the front and back are modelled as Dirichlet Boundary conditions, and are incorporated into the equation by eliminating from Φ the nodes with known potentials. The simulation code solves the system $A'\Phi' = b$, where Φ' is the set of unknown electric potentials in the torso and A' and b represent the adjusted stiffness matrix and RHS respectively.

The SCIRun net loads the torso mesh into memory at initialization, and reads the electrode meshes from transformation subnets, which contain UIs to move the electrodes on the torso surface and set the magnitude of the potential difference between the electrodes. Every time the user alters the position or strength of the electrodes, the new stiffness matrix A' and RHS b are re-calculated, and the new system is solved. The gradient, $\nabla\Phi'$, is then taken and passed on to the visualization subnet for rendering (Figure 4).

The study's design space consists of parameters defining the electrode position and strength, and the performance metric involves 4 dimensions: uniformity (variations in the electric potential in the heart volume), effectiveness (percentage of heart volume whose potential gradient is above the activation threshold), damage (percentage of heart volume whose potential gradient is above the damage threshold), and the electric potential between the electrodes. The study's objective was to find the points in the design space where the performance metrics yielded the best uniformity and effectiveness while keeping damage and electric potential within specified bounds.

4.1 Performance Evaluation

The application as described above has 5 input parameters that define the design space: coordinates of the front electrode (2 parameters), coordinates of the back electrode (2 parameters), and the potential difference between the electrodes. To simplify our analysis, we reduced the problem into 2D and 3D design problems by fixing the coordinate of the back electrode and fixing one of the coordinates of the front electrode (for 2D problem).

We measured study runtimes on a homogeneous IBM eServer cluster comprising 256 nodes, each with two 64-bit 2.2 GHz PowerPC 970 processors and 2 GB RAM, interconnected using a GbE network. The configurations involved a single SCIRun manager process, four SISOL processes serving the Shared Object Layer, and varying numbers of simulation container SCIRun processes. A typical experiment runs for 2.35 seconds, of which 0.25 seconds is used to calculate A' and b from A, 0.5 seconds is used to solve for Φ', and 1.5 seconds is used to calculate $\nabla\Phi'$; various other tasks account for the rest.

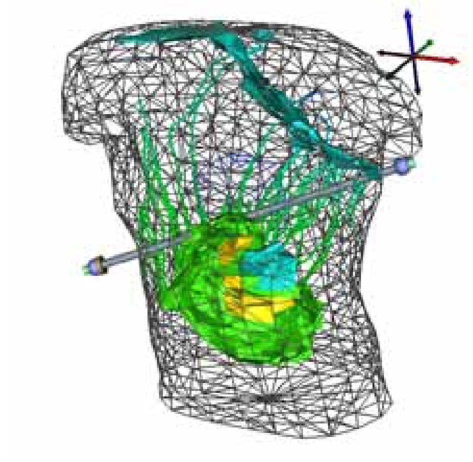
Figure 4. Visualizing one experiment from the defibrillator study.

Table 1 shows the study runtimes seen using different sampler strategies on different numbers of simulation processes for the 2D and 3D design problems. The grid sampler issues experiments for all points at a given refinement level; the active sampler identifies points where performance metric values are higher than their neighbors, and evaluates higher resolution points only in their vicinity [10]. Our results show that the SimX/SCIRun system scales well up to 64 processors; at 128 processors, all configurations face a load imbalance problem because the simulation containers co-located with the SISOL servers are able to complete their experiments faster, and thus receive a higher load. We also find the active sampling strategy working better for the 2D design problem, where it cuts down the number of experiments by 80%, as compared to the 3D case, where the improvement is only 40%.

5 Next Steps

We have conducted additional analyses using the defibrillator study. While space limitations prevent us from including the details, these analyses have identified several avenues for further development of the SimX/SCIRun system that we summarize below.

Steering/visualization of entire studies As shown in Figure 4, by leveraging existing SCIRun visualisation nets, SimX/SCIRun permits the user to interactively select and

	2D Grid Sampler		2D Active Sampler			3D Grid Sampler		3D Active Sampler		
No. of simulation processes	Time (in sec)	Load of busiest worker	Time (in sec)	Load of busiest worker	Experi-ments issued	Time (in sec)	Load of busiest worker	Time (in sec)	Load of busiest worker	Experi-ments issued
1	9100	4096	1790	760	760	9160	4096	5217	2329	2329
2	5673	2049	1074	385	769	5702	2051	3313	1183	2365
4	2842	1025	555.2	199	788	2856	1025	1371	608	2416
8	1427	514	315.6	98	771	1278	567	771.4	340	2418
16	818.6	257	146.4	51	787	717.0	257	421.9	148	2330
32	359.3	129	83.80	28	795	362.5	156	218.3	88	2348
64	181.4	65	42.66	17	881	182.2	65	123.7	43	2390
128	134.0	44	39.31	12	1035	95.62	36	95.35	25	2511

Table 1. Scalability of the SimX/SCIRun system on the defibrillator study. The design space is refined 6 times for the 2D samplers, 4 times for the 3D ones, before the study is complete. Checkpoints are written to the shared object layer for visualization purposes.

view the results of individual completed simulations in a study. However, what is missing are good mechanisms to allow users to visualize, in the aggregate, intermediate results of the study as a whole and steering mechanisms to permit dynamic updating of study objectives.

Advanced checkpoint reuse The run-time breakdown of the defibrillator simulation shows limited overall benefit from reuse of result checkpoints to improve the iterated solver performance, unlike what we saw in the bridge design study [10]. What is likely to be more useful are schemes that employ similar reuse ideas to improve the gradient computation step: this in turn requires checkpointing and reusing intermediate execution state in the simulation.

Active sampling schemes The 3D version of the study shows that active sampling is less effective in higher dimensions. Our analysis shows that this happens because the sampler resolves all parameter dimensions equally, even when the performance values exhibit different sensitivity. Incorporating this knowledge in the sampling policy, e.g., in our case the fact that the electrode voltage difference dimension can tolerate lower refinement, can yield similar or better study results with substantially fewer experiments.

Shared object layer improvements Our experiments show the need for two improvements to our SISOL implementation. First, with larger numbers of simulation processes, the overhead of checkpoint transfer to/from the layer becomes a significant contributor to the overall runtime. Utilizing a higher bandwidth interconnect is one way of alleviating this issue, as is developing a caching implementation of the server; the latter also has the positive side-effect of improving load balance across the system.

The second improvement stems from the observation that in the 3D version of the defibrillator study, more checkpoints were generated than the object layer is able to hold in memory. Incorporating application-level knowledge about which checkpoints are most likely to be valuable in terms of providing the most savings in subsequent reuse, can guide SISOL policies for deciding which objects to retain in memory and which to drop (or flush to disk).

References

[1] R. Armstrong et al. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proc. Intl. Symp. High Performance Distributed Computing (HPDC)*, 1999.

[2] J. Blythe et al. The role of planning in grid computing. In *Proc. Intl. Conf. Planning and Scheduling (ICAPS)*, 2003.

[3] I. Geist, G.A., J. Kohl, and P. Papadopoulos. CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications. *Intl. J. Supercomputer Applications and High Performance Computing*, 11(3):224–35, 1997.

[4] A. Grimshaw et al. Legion: The next logical step toward a nationwide virtual computer. TR CS-94-21, U. Virginia, 1994.

[5] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: Online monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[6] M. Litzkow, M. Livny, and M. Mutka. Condor - A hunter of idle workstations. In *Proc. Intl. Conf. Distributed Computing Systems*, June 1988.

[7] J. Nabrzyski, J. Schopf, and J. Weglarz, editors. *Grid Resource Management: State of the Art and Future Trends*. Kluwer, 2003.

[8] S. G. Parker, M. Miller, C. D. Hansen, and C. R. Johnson. An integrated problem solving environment: the SCIRun computational steering system. In *Hawaii Intl. Conf. on System Sciences (HICSS)*, 1998.

[9] J. Schmidt and C. Johnson. DefibSim: An interactive defibrillation device design tool. In *Proc. IEEE Engg. in Medicine and Biology Soc. Conf.*, 1995.

[10] S. Yau, E. Grinspun, V. Karamcheti, and D. Zorin. Sim-X: Parallel system software for interactive multi-experiment computational studies. In *Parallel & Distributed Processing Symp. (IPDPS)*, 2006.

[11] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. Parker. SCIRun2: A CCA framework for high performance computing. In *Proc. HIPS Workshop*, 2004.