Sim-X: Parallel System Software for Interactive Multi-Experiment Computational Studies

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

Advances in high-performance computing have led to the broad use of computational studies *in everyday engineering and scientific applications. A single study may require thousands of* computational experiments*, each corresponding to individual runs of simulation software with different parameter settings; in complex studies, the pattern of parameter changes is complex and may have to be adjusted by the user based on partial simulation results. Unfortunately, existing tools have limited high-level support for managing large ensembles of simultaneous computational experiments.*

In this paper, we present a system architecture for interactive computational studies targeting two goals. The first is to provide a framework for high-level user interaction with computational studies, rather than individual experiments; the second is to maximize the size of the studies that can be performed at close to interactive rates.

We describe a prototype implementation of the system and demonstrate performance improvements obtained using our approach for a simple model problem.

1 Introduction

Computer simulation has become widely accepted as an integral part of the scientific method. Advances in high-performance computing hardware and software have enabled simulations of ever increasing scale and accuracy, and have led to the broad use of computational studies in everyday engineering and scientific applications. A typical *computational study* is built out of multiple *computational experiments* corresponding to individual runs of simulation software. Simple computational studies may involve either few experiments or a larger number of experiments with simple predefined parameter variation patterns. In more complex studies, the pattern of parameter changes is more complex and may have to be adjusted by the user continuously, based on partial results. Examples of the latter kind range from exploration of design spaces in engineering to molecular simulations for drug design.

With few exceptions, existing parallel system software tools are inadequate for managing complex computational studies. Such tools either offer limited interactivity, or require the scientist to explicitly control individual experiments, an approach that does not scale for studies that involve thousands of experiments.

This paper describes the architecture of the Sim-X system, which attempts to address these shortcomings, and our initial experiments with a prototype of the system. Sim-X builds on several research efforts over the past decade on computational steering tools [20, 12, 11, 23, 4], domain-specific problem solving environments [14], and parameter-sweep tools introduced in the context of grid computing [6, 2, 1, 22, 19, 3, 5, 10] to enable interactive computational studies. Sim-X uses the following key principles:

Dynamic resource allocation. Sim-X relies on a more permeable interface between parallel system software and numerical simulation codes than is usually assumed by traditional environments. By exposing more of the internal simulation structures to the system and vice-versa, our approach enables application-specific prioritization of activities (reflected in how resources are assigned), faster adaptation to changing objectives, and the ability to flexibly trade-off simulation result accuracy and precision versus resource needs so as to meet user requirements.

Reuse. Sim-X relies on the expanded system-application interface to aggressively reuse available data in carrying forward new computations. Sim-X enables a study to return to a previously abandoned experiment, or perform a new experiment faster using the results of an experiment for nearby parameter values as a starting point.

High-level user control. Sim-X frees the user from micromanaging every individual computational task, complementing existing computational steering tools in targeting *interactive manipulation of entire computational studies*, rather than individual experiments.

1-4244-0054-6/06/$20.00 ©2006 IEEE

Reuse and high-level control are crucial features for bringing computational studies closer to interactive performance rates. In our experiments, we observe a 40-fold improvement in performance compared to simple parameter sweep-like approaches.

The rest of this paper is organized as follows. Section 2 draws a clearer distinction between previous efforts and the goals of the Sim-X system, and introduces a bridge design study where the designer's interest is in identifying a Pareto frontier that defines optimal tradeoffs among several design parameters. This study serves as a running example through the rest of the paper. Section 3 presents the overall architecture of the Sim-X system, and we discuss our implementation of key components of the architecture in Section 4. The benefits of the architecture and this expanded interface are evaluated in Section 5. We conclude in Section 6.

2 Background

2.1 Related Work

Sim-X builds on a considerable body of prior work in computational steering toolkits and scheduling of parameter sweep applications.

Computational steering infrastructures. Since Bob Haber and David McNabb first demonstrated the concept in 1989 [13], several systems have been developed to provide an infrastructure for visualizing and steering large-scale parallel scientific experiments. A brief chronological history includes Falcon [12, 21], SCIRun [20, 18], CUMULVS [11], UINTAH [9], DISCOVER [16], CSE [23], RealityGrid [4], gViz [24], and SCIRun2 [25].

Sim-X shares with these systems the ideas of *standardized component architectures*, enabling application developers to mix-and-match compute, steering, and visualization modules and reuse generic modules across multiple application domains, and supporting a *low latency visualization/steering interaction cycle* [12, 21].

However, in marked contrast to most of these systems, which have focused on steering individual (or a small number of) experiments, Sim-X aims to steer computational studies involving a multitude of short-running experiments, requiring introduction of higher level user interface tools and a framework for dynamic resource management that emphasizes reuse and adaptation rather than fault-tolerance and heterogeneous task migration.

Scheduling of parameter sweep applications. Several grid computing infrastructures (APST [6], Nimrod [2, 1], Condor [22], Globus [19, 3], Netsolve [5], Virtual Instruments [10]) provide support for the scheduling of parameter sweep applications, where the same application is run with a change in parameter values across distributed resources. The underlying tools simplify the running of multiple simulation experiments, possibly running into the tens of thousands. Such support differs from our focus on an interactive computational study; with the exceptions below, these schedulers assume that the simulations are independent and hence can be run in batch mode.

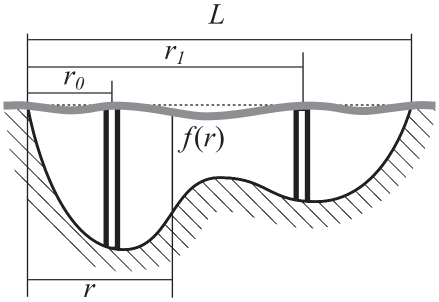

Figure 1. The bridge design problem; positions of supports r_0 and r_1 are the parameters, and the maximal displacement and the construction cost define the performance measures. $f(r)$ yields the river depth as a function of the distance from the left bridge endpoint.

A small number of parameter sweep tools do support user interaction. Condor DAGman permits specification of a workflow involving several related parameter sweep jobs; however, each job is still executed in batch fashion. Nimrod/O [1] supports applications where the user's interest is in identifying the point in parameter space that maximizes a particular objective function. Sim-X generalizes this support to support more interactive exploration of fuzzier and dynamically changing optimization criteria. Closer to our goals is the scheduling module of the Virtual Instruments project [10], which implements priority-based resource allocation for the application's tasks. Sim-X extends this support to factor in reuse of results from previous tasks and control of the internal behavior of tasks.

2.2 Example Computational Study

In this paper, we use as an example, a sample computational study from the domain of engineering design. Our model problem is to design an elastically deforming bridge with four supports, two of which are fixed at the endpoints (see Figure 1). This specific problem is relatively straightforward, and the individual simulations are relatively fast. At the same time, it captures many essential features present in a real-life engineering design situation, e.g., one involving simulations of a car body or a more complex structure.

Engineering design is a process of *design space exploration* where one trades off amongst multiple *performance measures*, subject to a set of constraints.[1] In our bridge de-

[1] Design space and performance measures correspond to Section 3's parameter space and observation measures, respectively. In this section,

sign example, the two parameters defining the design space are the locations of the two non-fixed supports. We assume that the cost of supports changes as we move away from the endpoints (e.g., due to the variable profile of the riverbed and the difficulty of constructing a support further away). A desirable design would thus trade off among two performance measures: the cost of bridge construction and the maximal deformation of the bridge.

A typical strategy for design space exploration is *Pareto optimization* (e.g. [17]). Pareto optimization seeks to find a *set* of optimal points, with each *Pareto-optimal* point corresponding to a different trade-off of design preferences, e.g., two bridge designs may be equally desirable, one for its lower deformation and the other for its cost. A Pareto-optimal point has the property that improving one measure can only be achieved at the expense of another, e.g., a bridge design is Pareto-optimal if cost cannot be improved while holding deformation fixed, and vice-versa. This set of optimal points is the *Pareto frontier*.

In general the number of simulations to fully resolve the Pareto frontier is prohibitive for all but the simplest design problems, however using aggregation methods engineers can steer the exploration toward regions of the Pareto frontier that are particularly promising. Furthermore, engineers are able to choose a good design point in the presence of an incomplete or fuzzy localization of the frontier, further reducing computational cost. Both aggregation and fuzzy localization make the underlying computational study interactive: the engineer looks at the frontier as it is evolving and makes decisions about how to steer the study to refine specific regions of the frontier to the desired accuracy.

3 System Architecture

Figure 2 shows the overall structure of the Sim-X architecture. We briefly discuss its main components from a user, application and system points of view.

User view. The platform is able to run a large number of computational experiments (*simulations*) organized into a study. Simulations differ in their input parameters, which control properties of the underlying physical/mathematical model and the simulation technique and define the *parameter space* for the study.

Typically, the result of a simulation is not examined by the user directly; rather, a collection of *measures* is observed. In our example, the measures are maximal deflection of the bridge and the cost of the construction. The ranges of all measures forms the *observation space*.

The system permits a user to select domains of interest in the parameter and observation spaces, by identifying admissible ranges of parameter values and measures. In addition, inside the observation domain, the user may implicitly specify a *target set* of interest: the goal of the study is to identify this set. The most common example of such a set is the Pareto frontier.

The user may additionally define functions on both the parameter and observation domains, which inform the system about the priorities of the user. The crucial aspect of the system is that most if not all of the user selections above can be modified as the study is underway.

Application developer view. The application developer is expected to provide three kinds of modules: (1) *simulation* modules whose execution corresponds to the actual experiment; (2) *visualization and interaction* modules that make up the user interface, and (optionally) (3) *transformation* modules, which make it possible to transform a state of a simulation with one parameter set into that compatible with another (e.g., interpolation of an intermediate solution from a finer to a coarser grid).

Simulation modules need to adhere to a standard interface to permit their active manipulation by the system. Specifically, they need to support:

- **Checkpointing:** With some sufficiently fine granularity, the module should be able to create a dataset from which the current state of the simulation can be reconstructed with a minimal amount of computation.
- **Time estimates:** To allocate resources to active experiments, the modules must be able to provide estimates for their *time to completion*, assuming simulation parameters do not change; other cost estimates, such as the cost of adding/releasing resources, can be added.
- **Snapshots of observation measures:** In simulations using iterative and hierarchical solvers, intermediate results are useful approximations to the final solution; observation measures can be evaluated against these results to provide quick feedback to the user.

System view. As 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.

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

we use terminology more typical of the engineering design community.

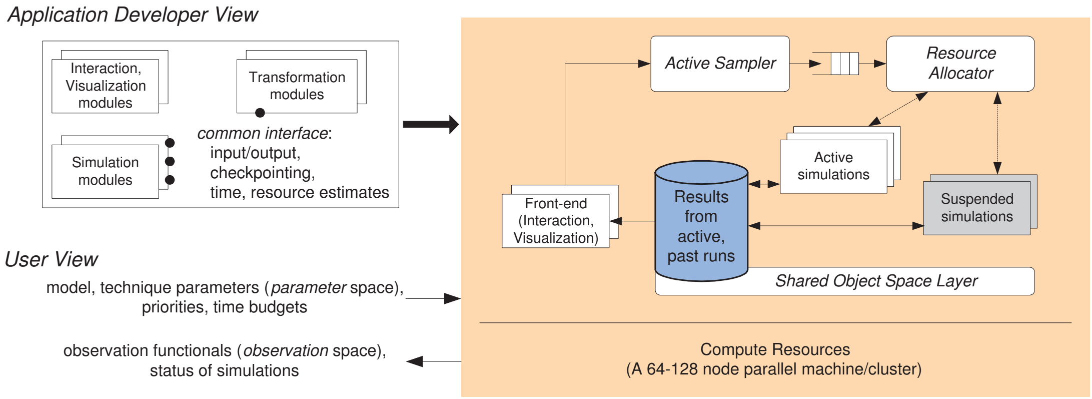
Application Developer View
Interaction, Visualization modules
Transformation modules
Simulation modules
common interface:
input/output,
checkpointing,
time, resource estimates
User View
model, technique parameters (parameter space),
priorities, time budgets
observation functionals (observation space),
status of simulations
Active Sampler
Resource Allocator
Active simulations
Suspended simulations
Results from active, past runs
Front-end (Interaction, Visualization)
Shared Object Space Layer
Compute Resources
(A 64-128 node parallel machine/cluster)

Figure 2. Overall architecture of the Sim-X platform for multi-experiment computational studies.

strategies in important cases. Our prototype implementation provides an active sampler for computing the Pareto frontier (Section 4.3).

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.

4 Current Implementation

We report on a prototype implementation of the Sim-X architecture, which is being refined as we continue to gain experience with supporting different kinds of studies.

4.1 Execution substrate

The overall functionality of the Sim-X architecture is realized by two types of communicating processes: *managers* and *simulation containers*. These processes communicate via a generic *satellite* interface. In our current implementation, the processes are just standard UNIX processes, and the satellite interface is implemented using TCP socket calls. The code is modular, so replacing the satellite implementation with one that uses a different transport mechanism (e.g., MPI) is relatively simple.

The interaction between these processes is shown in Figure 3. One or more simulation containers are started off first; they constitute the worker pool to which the manager farms off individual simulations. After initializing themselves, the simulation containers wait to be contacted by a manager process, which sends it simulations corresponding to regions in the parameter space identified by the active sampler and resource allocator modules.

The simulation container code is structured to overlap manager interactions with computation (of a previously received piece of work). Similarly, the manager code is structured to permit asynchronous receipt of messages from any of its simulation containers; message arrival triggers a preregistered callback function. The *satellite* interface provides a generic interface for any process to retrieve and send information about the ongoing studies to the simulation containers. It allows different kinds of processes to be distinguished, and supports both pull-based and push-based interactions (e.g., for viewers to update their display).

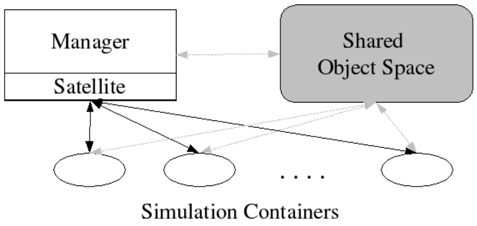

Figure 3. Sim-X manager and simulation container processes.

In addition to the explicit interactions described above, 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.

4.2 Test simulation problem

Our prototype implementation works with the bridge design study introduced earlier (see Figure 1). We present its details here to provide context for the discussion of the active sampler, resource allocator, and shared object space layer modules below. The design problem is to construct a simple bridge with four supports, two of which are fixed and clamped at the end points, with the remaining two modelled as columns made of a homogenous elastic material.

The design parameters of the problem are positions of the two columns. The performance parameters are the combined costs of the column construction as a function of the distance from the endpoints, and the maximum vertical deflection of the bridge.

We consider a *nonlinear model* for the problem, as linear models typically allow considerable simplifications of the design process, which do not apply in more general cases. To obtain a realistic understanding of typical solver behavior, our implementation uses a widely used scientific computing library (PETSc) to solve the discretized systems.

Problem formulation and the parameter space. The bridge is modeled as a one-dimensional rod, elastically deforming in two dimensions. The continuous energy of the bridge in a static configuration is given by

$$\int_{t=t_0}^{t_1} (k_b\kappa(t)^2 + k_m \left|\frac{ds(t)}{dt} - 1\right|^2 + \rho g p^y(t) \\ +K_c(p^y(r_0))^2 + K_c(p^y(r_1))^2)dt$$

where the five terms correspond to the bending energy, the membrane energy, the potential energy due to gravity, and the potential energies due to the two elastic supports.

The parameter t is the distance from a material point to a reference point (t_0 and t_1 are the endpoints), $s(t)$ is the arclength distance from a material point to the reference point in the deformed configuration, $\kappa(t)$ is the curvature of the deformed configuration at t, and $p(t) = (p^x(t), p^y(t))$ is the displacement of a point. r_0 and r_1 are the locations of intermediate elastic supports, and K_c is the effective elastic stiffness of the supporting columns.

The boundary conditions for the problem are fixed locations for supports: $p(t_0) = (t_0, 0),\ p(t_1) = (t_1, 0)$. The parameters r_0 and r_1 define the parameter space for the model.

Performance space. We use two performance measures: one is the maximal displacement of the bridge; for support positions r_0 and r_1 and the corresponding deformed configuration of the bridge $p(t)$

$$F_1(r_0, r_1) = \max_{t_0 \leq t \leq t_1} |p(t) - \bar{p}(t)|,$$

where $\bar{p}(t) = (t, 0)$ is the undeformed configuration.

The second performance measure is computed using a user-defined cost function $f(r - t_0)$ for support placement:

$$F_2(r_0, r_1) = f(r_0 - t_0) + f(r_1 - t_0)$$

Intuitively, the function $f(r)$ is dictated by the shape of the riverbed as shown in Figure 1.

Discretization. We use a simple nonlinear discretization of the problem, where $t_i = hi$, $i = 0 \ldots N$ and $h = (t_1 - t_0)/N$. The discretized energy is given by:

$$\sum_{i=1}^{N-1} (\frac{2\theta_i^2}{l_i + l_{i+1}} + \rho g p_i^y) + \sum_{i=0}^{N-1} (l_i/h - 1)^2$$

where $l_i = |p_{i+1} - p_i|$, and θ_i is the angle between segments (p_{i-1}, p_i) and (p_i, p_{i+1}). One can show that for small displacements, this is equivalent to using piecewise linear basis functions to discretize the membrane energy and piecewise quadratic functions to discretize curvature. In addition, we assume supports of the bridge to be elastically deformable, modeling them as simple elastic springs of high stiffness. The forces and force derivatives are obtained by differentiating the energy with respect to the degrees of freedom.

Numerical methods. To solve the nonlinear problem, we use the Newton method with cubic line search; the only numerical parameter that we choose is the residual tolerance. We solve the linear system in the interior loop of the linear solver using a direct LU solver.

Instrumentation. To interface the code with the system, we added two components; in each case, the resulting modification was minor.

First, after each Newton iteration, a checkpoint is saved into the shared object space layer (SISOL) decribed below. Each checkpoint contains the information about the parameters of the system and the positions of all nodes. In the current implementation, the checkpoint is a PETSc vector containing t_0 ... t_N.

Second, the simulation can be started using checkpoint information from a different simulation run. A simple transformation module is included, which maps the solution obtained for a particular configuration of supports to a new configuration using linear rescaling and cubic interpolation of the solution.

4.3 Active sampler

Our current implementation of the Sim-X architecture features an active sampler that resolves the Pareto frontier in a breadth-first, coarse-to-fine manner. Consequently, (a) the user perceives a low response time as a rough depiction of the frontier is displayed within seconds, (b) as the display resolution progressively increases, so does the required number of simulations, but so does the sampling density, enabling better reuse of simulation results in the resource allocator as described below.

A variety of techniques have been developed for discovering Pareto frontiers (e.g. [8, 17, 7]). We have implemented a general hierarchical approach which requires no specific information about cost functions other than an evaluation procedure, which is consistent with our goals of providing a framework for interactive experimentation: coarse scale results can be obtained first and gradually refined, and arbitrary simulations can be used as a part of the cost functional computation.

Approach. Assume that the parameter space has been sampled at a finite number of points, yielding a discrete finite set, V, of evaluated designs. The discrete approximation of the Pareto frontier is the subset, $R \subseteq V$, containing only the *undominated* points.[2]

We start by discretizing the parameter space into a coarse lattice and proceed as follows. Seed the computation by evaluating one or more (arbitrary) lattice points, always adding evaluated points to V. Every time a change is made to V, incrementally update R. If a point is added to R, then the sampler requests evaluation of all lattice-neighbors of the new point. This effectively walks along the Pareto frontier: lattice-neighbors that are off the frontier will be dominated, hence will not propagate further; neighbors on the frontier will be added to R, thus advancing the walk. At all times, R represents the best approximation to the Pareto-frontier given current data. At the end of this computation, R is the best approximation of the Pareto frontier at the current lattice resolution. When we reach this point, we refine the Pareto approximation: resolution is increased and points on a finer lattice adjacent to points in R are evaluated.

Parallelization The active sampler must efficiently and correctly resolve the Pareto frontier in the setting of multiple simulation containers. This is an inherently difficult task consisting of conflicting goals: *optimal sampling* dictates using all information at hand in choosing the next evaluation point; *full loading* requires that every simulation container have queued jobs.

Full loading of simulation containers requires the sampler to issue evaluation points in the absence of complete knowledge. Therefore in comparing a parallel to a sequential implementation we expect (and observe) that some evaluations, issued with incomplete knowledge, would not have been issued under complete knowledge. These *excess evaluations* are inevitable in the parallel setting. Even so, the active sampler satisfies the goal of optimal sampling by issuing simulations based only on known completed simulations (in effect, treating pending simulations as if they would be dominated). When new information may indicate that a previously issued evaluation is (in retrospect) in *excess*, this may lead to a truncation of search stemming from the excess evaluation.

4.4 Resource allocator

Our current implementation of the resource allocator allocates work units to simulation containers using a simple FIFO queue. This decision was largely motivated by the fact that individual simulation runs for our test problem execute on the order of a few tens of microseconds each, so a more sophisticated scheme would have yielded few additional benefits.

The one decision that the resource allocator does make is to choose a nearby checkpoint from which to jumpstart the requested simulation. This choice is made by querying the shared object space layer for the nearest evaluated parameter space point and then supplying this as an additional modifier to the work unit.

4.5 Shared object space layer (SISOL)

A central component of the Sim-X implementation is the SISOL module, which implements a *spatially-indexed shared object layer*.

Interface. The primary abstraction provided by SISOL is one of a spatially-indexed *object set*, elements within which can be inserted and retrieved using spatial coordinates. In addition to looking up specific objects by their coordinates, SISOL also supports neighborhood queries to retrieve an arbitrary number of objects that are within a certain distance from a specified coordinate.

Table 1 shows the high-level interface functions provided by SISOL. The interface supports typed objects (e.g., PETSc vectors) which can be either sequential or parallel.

Implementation. The choice to explicitly distinguish between object reads and writes, and to delimit these accesses with start and stop calls enables an efficient distributed caching implementation of the SISOL layer [15]. Multiple read copies of an object can be simultaneously cached at multiple nodes; however, only one of these copies can be valid for a write access. The SISOL interface supports application-specific custom coherence protocols to reduce coherence traffic in commonly encountered access patterns.

For the results that we report in Section 5, we work with a simpler implementation where the object storage is partitioned among multiple SISOL server processes; each SISOL client communicates with a statically chosen server process over TCP connections to implement the interface functionality but does not itself cache retrieved objects. Note that this implementation can be scaled easily by growing the number of SISOL server processes as the number of clients increase. Such scaling comes at the cost of not all updates being visible to all clients, but this tradeoff is likely acceptable in most situations including in our model problem.

The test simulation code and active sampler work with two object sets: a 2-dimensional set that stores the simulation checkpoints corresponding to different points in the parameter space, and a 2-dimensional set that stores the simulation results for retrieval by the active sampler and viewer modules. Both the active sampler and the simulations themselves make heavy use of neighborhood queries.

Our implementation internally supports two variants for how one maintains the objects within a set: a simple hashtable, and a more involved R-tree based implementation.

[2] A parameter space point, $x = \{x_1, \ldots, x_n\}$, with associated performance measure, $p(x) = \{p_1(x), \ldots, p_m(x)\}$, *dominates* another point x_2 if and only if $\forall 1 \leq i \leq m : p_i(x_1) \leq p_i(x_2)$. Note that a lower value of a performance measure is preferred.

Operation	Signature	Function
Initialization	`int CreateSet(int setID, int typeID, int arity, double *weights, int capacity)`	Create object set of `arity` dimensions to store objects of type `typeID`. The `weights` array specifies a weighted Euclidean distance metric.

Registration	`int RegisterSet(int setID, void** objSet)`
	`void UnregisterSet(void* objSet)`
		Registers client as participant; retrieves object set metadata in `objSet`.
		Unregisters client.

Access	`void Insert(void* objSet, double* coords, void* obj)`
	`void Remove(void* objSet, double* coords)`
	`void* StartRead(void* objSet, double* coords)`
	`void* StartWrite(void* objSet, double* coords)`
	`void EndRead(void* objSet, double* coords, void* obj)`
	`void EndWrite(void* objSet, double* coords, void* obj)`
		Insert/remove an object into/from the set.
		Start/end a read/write operation on an object.

Query	`void QueryClosest(void* objSet, double* coords, int numToLookup, int* numRetrieved, double** retrCoords)`	Query for up to `numToLookup` closest neighbors

Table 1. Interface of the spatially-indexed shared object space layer (SISOL).

The results in the next section use the hash table implementation, which for the particular object sets of interest yielded lower operation times.

5 Evaluation

To understand the costs and benefits of the Sim-X architecture, we evaluated the performance characteristics and outputs of the bridge design computational study for multiple user input and execution environment scenarios. We discuss some of these scenarios below.

Experiment setting. For all of the experiments in this section, the study was run on a homogenous IBM eServer cluster comprising 256 nodes, each with two 64-bit 2.2 GHz PowerPC 970 processors and 2 GB RAM, interconnected via a Myrinet network. The study involved a single manager process, from one to four SISOL server processes serving the checkpoint object set, and varying numbers of simulation container processes, each hosted on a separate physical processor.

To quantify the impact of the different features of the Sim-X architecture, we defined the following four system configurations:

1. GS: a grid-based sampling of the parameter space, which progressively refines the Pareto frontier by evaluating simulations at grid points of increasing resolution.
2. GS+C: grid-based sampling, with reuse of checkpoints from prior runs.
3. AS: active sampling of the parameter space.
4. AS+C: active sampling, with reuse of checkpoints from prior runs.

In each case, the objective of the study was to identify a Pareto frontier corresponding to minimum bridge deflection and a bridge cost function. To ensure that the study results were not affected by human interaction delays, the manager read the study specifications from a file. Both samplers are seeded with a 40x40 grid over the parameter space, and the studies terminate after the samplers resolve the Pareto frontier to the fourth refinement level of the original grid. The results are written into the SISOL server processes, and collected after each refinement of the partial Pareto frontier.

High-level control of the study. Figure 4 shows the evolution of the Pareto frontier over time for the AS+C configuration run with 128 simulation containers. For comparison, we are showing the evolution of the Pareto frontier for the GS configuration. Table 2 shows in detail the wall-clock time needed by each of the GS, GS+C, AS, and AS+C configurations to reach each stage of the Pareto frontier's evolution.

Two points can be observed from the plots. First, even relatively early on in the study, the overall shape of the Pareto frontier is visible, which permits a user to interact and optionally change the study parameters without waiting for completion of a set number of simulation runs.

Second, the Sim-X architecture managed to provision resources to explore only those portions of the parameter space that were most productive from the point of view of developing the Pareto frontier. This is seen by the sparsity

	Time (in secs) to produce level			
Configurations	1	2	3	4
GS	97.48	360.91	1407.76	5678.22
GS + C	6.73	15.52	50.39	429.06
AS	98.64	199.26	315.31	362.83
AS + C	6.62	8.90	12.26	13.63

Table 2. Times required on 128 processors to produce the frontiers shown in Figure 4.

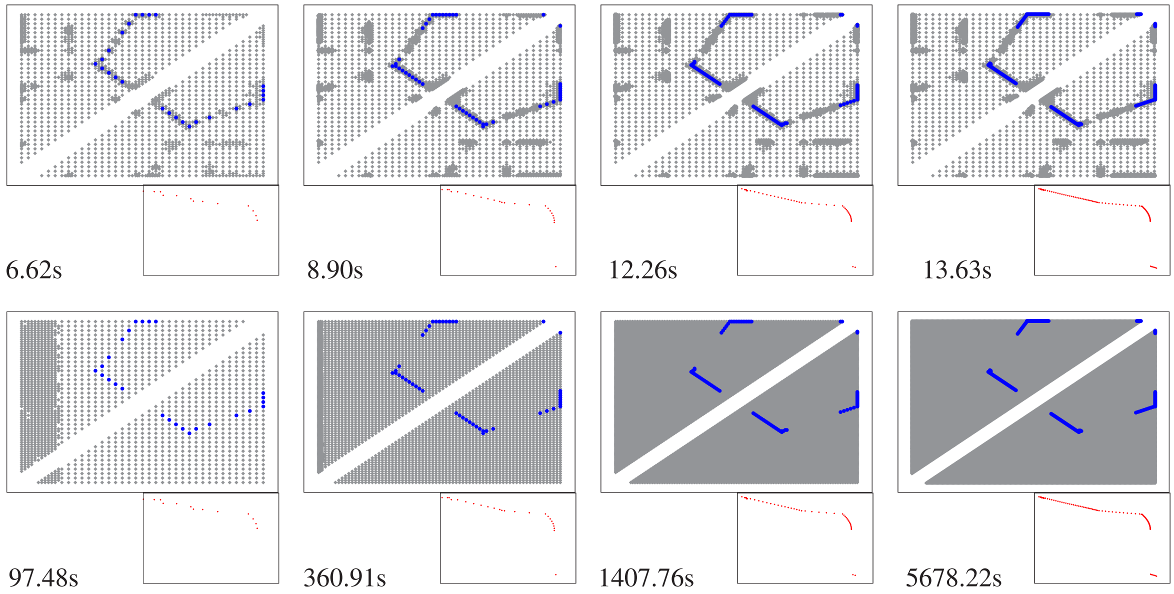

Figure 4. Time evolution of the Pareto frontier as observed for the AS+C (rows 1-2) and the GS (rows 3-4) configurations, after refining to (left to right) 1st, 2nd, 3rd, and 4th levels, with runtime listed for each snapshot. Rows 1 and 3 depict evaluated designs (light gray) and Pareto points (dark blue). In the GS configuration the sampling in design space is too dense for individual points to be visible. Rows 2 and 4 depict Pareto points in observation space. The number of simulation experiments required to generate these frontiers are 1727, 2684, 4243, and 4526 for the AS+C configuration, and 1735, 4950, 18632, and 75351 for the GS configuration.

of the evaluations in the AS+C configuration, which is very unlike what is seen for the GS configuration.

Figure 5 shows the quality of the Partial frontier available at particular times for the AS+C and GS+C configurations respectively, as measured by the Hausdroff distance metric in the observation space . It can be seen as early as ten seconds into the study that the AS+C configuration already yields a good approximation of the final answer. The GS+C configuration is clearly non-competitive.

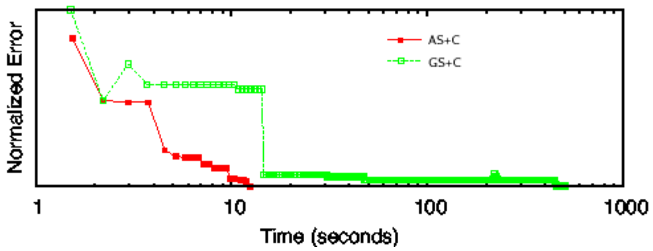

Figure 5. Relative error in Pareto frontier approximation for the AS+C and GS+C configurations as measured using the Hausdorff distance metric, shown for observation space.

Performance benefits of Sim-X. Table 2 shows the performance benefits from Sim-X's use of the active sampler and reuse of application checkpoints. Checkpoint reuse has the effect of reducing average per-simulation runtimes by 60% to more than 90%, depending on the pool of checkpoints that the manager can draw from. These savings result from a reduction in the number of iterations taken by the solver to converge, which goes down from 40-50 for the initial simulations to as low as 3-4 for later ones.[3] Independent of this, the active sampler cuts down total runtime by up to an additional factor of 35×, depending on the target refinement level. These savings result from a reduction in the number of simulations needed to generate a frontier.

Scaling behavior. Table 3 shows the breakdown of execution times on the manager, simulation container, and SISOL server processes into various components for different numbers of processors. The runs used four SISOL server processes, each handling requests from an equal-sized partition of the simulation container processes. (See the caption for an explanation of the different components).

The high level conclusion that one can draw from the table is that all of the components of the Sim-X architecture

[3] In fact the reduction in iterations is so dramatic that we found relatively little additional performance gains from controlling error tolerances achieved by the simulations.

		Components					
Procs	# Simul.	Manager	Simulator	SISOL-1	SISOL-2	SISOL-3	SISOL-4
1	4125	1203.14 (1.30)	1191.82 (11.65)	(1.01, 12.51)			
2	4224	625.71 (1.35)	620.26 (5.81)	(0.52, 4.00)	(0.51, 4.04)		
4	4284	334.35 (1.45)	329.93 (4.84)	(0.26, 1.56)	(0.26, 1.49)	(0.20, 1.39)	(0.20, 1.39)
8	4353	175.59 (2.04)	170.33 (5.72)	(0.26, 5.74)	(0.26, 5.83)	(0.22, 1.59),	(0.22, 1.44)
16	4380	89.39 (2.29)	86.54 (9.35)	(0.26, 5.69)	(0.26, 1.37)	(0.24,1.31)	(0.23, 1.18)
32	4394	45.33 (1.79)	44.98 (3.10)	(0.28, 1.39)	(0.28, 1.41)	(0.24, 1.23)	(0.24, 1.10)
64	4392	24.12 (1.90)	24.05 (5.30)	(0.30, 1.48)	(0.31, 1.47)	(0.27, 1.25)	(0.28, 1.29)
128	4490	13.13 (1.95)	12.60 (2.10)	(0.39, 1.86)	(0.38, 1.8)	(0.32, 1.35)	(0.33, 1.43)

Table 3. Number of simulations and breakdown of process execution times (all times are in seconds) for runs with different numbers of processors. The manager components correspond to the total runtime and the time spent on the active sampler. The simulation container times correspond to the time spent on executing the simulations and the time interacting with the manager respectively. The SISOL components are the time spent for each of the four server processes, on communication with client processes, and the time spent processing the read, write, and neighborhood search queries.

Simul. #	Avg. time	Simul. #	Avg. time
0-200	7.069	800-1000	2.652
200-400	3.714	1000-1200	2.467
400-600	3.231	1200-1400	2.461
600-800	2.681	1400-1600	2.372

Table 4. Average per-simulation run times (in milliseconds) of the first 1600 simulations, in 200 simulation increments, based on a 128-processor run.

scale relatively well for our target numbers of processors, and spend relatively little time on overhead activities, e.g., dispatching new work to the simulation containers or interacting with the SISOL service. While we did have to increase the the number of SISOL servers as the number of simulation containers increased, no additional implementation effort was needed. Our detailed measurements also indicate that the tradeoff of not having all checkpoints accessible to all containers, while having some effect, did not increase overall run times in any noticeable fashion.

Benefits in other environments. More detailed analysis of the data in Table 3 and the reasons behind them is instructive to understand how the Sim-X architecture would perform on different workloads or on larger numbers of processors.

The first observation not directly seen in the data is that simulation sizes are non-uniform. At the beginning of the study, few checkpoints have been created. The manager has a relatively small pool of checkpoints to draw from, and thus is less likely to find a checkpoint that benefits the simulation. Therefore, early simulations take longer to run. This phenomenon is illustrated in Table 4. Such non-uniformity ultimately affects scalability because the active sampler sets up dependencies between the simulations it issues. On the other hand, these results suggest that manager resources can perhaps be used to do additional (exploratory) work during the early stage of the study to make more informed choices about which simulations to run.

Beyond this non-uniformity, the primary source of parallel overhead in Sim-X appears to be algorithmic, resulting from the active sampler basing its decisions on incomplete information. As discussed in section 4.3, when the active sampler chooses a parameter point to issue, it assumes all the parameter points that are still running to be not on the Pareto frontier. If those points are found to be on the frontier when their results arrive, then the active sampler will have to issue extra simulations to fix the frontier. As the number of parallel processes increases, the number of simulations being run at any one time increases, therefore the chance of the sampler making a mistake also increases. As a result, the number of evaluations needed to construct a Pareto frontier also increases. Table 3 illustrates this overhead, which indicates the need for improving the active sampler algorithm to better track inter-simulation dependencies.

Table 3 also shows that on larger numbers of processors and for studies involving lighter-weight simulations, the interaction overheads of the manager and SISOL server processes can potentially become bottlenecks. Manager scalability can be addressed by setting up a hierarchy of managers each of which takes responsibility for exploring a portion of the design space; the challenge here is in orchestrating active sampler decisions across these portions.

SISOL server scalability is easier to address: the dominant cause for the increased runtimes seen for SISOL server operations appears to be the linear search of the hash table entries to resolve neighborhood queries. A smarter search strategy, e.g., one that combines an R-tree implementation with more approximate responses to neighborhood queries would alleviate this problem, as would further partitioning the SISOL server activity. Ultimately however, the tradeoff inherent in partitioning, where not all updates are visible everywhere, may become a problem. We expect that the distributed caching implementation of the SISOL server to provide a longer-term solution to this issue, effectively approximating a SISOL server per simulation container, yet still permitting propagation of updates.

6 Summary

The results we have obtained with our initial prototype of Sim-X suggests that the Sim-X system does go a long way towards meeting the two goals we started off with: providing a framework for high-level user interaction with computational studies, and increasing the sizes of computational studies that can be performed at interactive rates. For our sample problem, significant improvements were observed both from intelligent parameter space navigation strategy implemented in the active sampler, and reuse of previously computed solutions.

There are several next steps we intend to pursue: support for parallel simultion codes that run on multiple simulation containers, support for heterogeneous multiscale models often used in engineering, and a user interface supporting computational study steering. As our implementation matures, we hope that Sim-X will substantially improve the ease with which scientists and engineers can perform sophisticated computational studies on small and medium-size parallel platforms.

Acknowledgments

This study is funded in part by NSF grants CCR-0312956 and DMS-05-28402. The authors would also like to thank Michael J. Scott for his valuable discussions on engineering design, SimX protoype's application domain.

References

[1] D. Abramson, A. Lewis, T. Peachey, and C. Fletcher. An automatic design optimization tool and its application to computational fluid dynamics. In *Proc. SC'01*, 2001.

[2] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proc. HPDC*, 1995.

[3] J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *Proc. ICAPS*, 2003.

[4] J. Brooke, P. Coveney, J. Harting, S. Jha, S. Pickles, R. Pinning, and A. R. Porter. Computational steering in RealityGrid. In *Proc. UK e-Science All Hands Meeting*, 2003.

[5] H. Casanova and J. Dongarra. Netsolve: A network server for computational science problems. *Intl. J. Supercomp. Appl. and High Perf. Comp.*, 11(3):212–223, 1997.

[6] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *Proc. SC'00*, 2000.

[7] C. Coello, D. Veldhuizen, G. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer, 2002.

[8] I. Das and J. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM J. on Optimization*, 8(3):631–657, 1998.

[9] J. Davison de St. Germain, J. McCorquodale, S. Parker, and C. Johnson. UINTAH: a massively parallel problem solving environment. In *Proc. HPDC*, 2000.

[10] M. Faerman, A. Birnbaum, H. Casanova, and F. Berman. Resource allocation for steerable parallel parameter searches. In *Proc. Grid'02*, 2002.

[11] G. Geist, J. Kohl, and P. Papadopoulos. CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications. *Intl. J. Supercomp. Appl. and High Perf. Comp.*, 11(3):224–35, 1997.

[12] 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.

[13] R. Haber and D. McNabb. Eliminating distance in scientific computing: An experiment in televisualization. *Intl. J. Supercomp. Appl.*, 4(4):71–89, 1990.

[14] C. Johnson, R. MacLeod, S. Parker, and D. Weinstein. Biomedical computing and visualization software environments. *Comm. ACM*, 47(11):64–71, 2004.

[15] V. Karamcheti and A. Chien. Architectural Support and Mechanisms for Object Caching in Dynamic Multithreaded Computations. *J. Parall. and Distrib. Comp.*, 58(2), 1999.

[16] V. Mann, V. Matossian, R. Muralidhar, and M. Parashar. DISCOVER: An environment for Web-based interaction and steering of high-performance scientific applications. *Concurrency: Practice and Experience*, 13(8–9):737–754, 2001.

[17] A. Messac. Physical programming: Effective optimization for computational design. *AIAA J.*, 31(4):149–158, 1996.

[18] M. Miller, C. Hansen, S. Parker, and C. Johnson. Simulation steering with SCIRun in a distributed environment. In *Proc. HPDC*, 1998.

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

[20] S. Parker and C. Johnson. SCIRun: A scientific programming environment for computational steering. In *Proc. SC'95*, 1995.

[21] B. Plale, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin, and J. Vetter. From interactive applications to distributed laboratories. *IEEE Concurrency*, 6(2):78–90, 1998.

[22] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[23] R. van Liere, J. Mulder, and J. van Wijk. Computational steering. In *Proc. HPCN*, 1996.

[24] J. Walton, J. Wood, and K. Brodlie. Visualization aids steering of complex simulations on the grid. *Scientific Computing and Instrumentation*, 21(4):17–19, 2004.

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