Using Application-Domain Knowledge in the Runtime Support of Multi-Experiment Computational Studies

by

Siu M. Yau

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January, 2009

Approved: _____

Research Advisor: Vijay Karamcheti

Approved: _____

Research Advisor: Denis Zorin

Acknowledgements

First, I would like to thank my advisors Vijay Karamcheti and Denis Zorin. Their advice, ideas, support and directions have been invaluable to the research efforts leading to this thesis. I would also like to thank my collaborators Eitan Grinspun, Steven Parker, Kosta Damveski, Akash Garg, and Jeremy Thornock for their continuing interest, support, and effort in this project. Thanks also to my thesis defense committee members Allan Gottlieb, Marsha Burger, and Laskhmi Subramanian. I'm also grateful for my first mentor, Kathy Yelick, who got me started in high-performance parallel computing research.

I would also like to thank my friends who have stood by me throughout my graduate career: Andrea Smith, who "[yelled at me] every now and then to stop being such a bonehead", Donna Lavins and Rhoda Ross, my adoptive New York moms, and Vivien Lee, aka Auntie USA, my adoptive California mom.

Finally, I would like to extend my deep and heart-felt gratitude to Jill Burger, who has been a dear and faithful friend for over a decade. This thesis wouldn't exist — and I wouldn't be who I am — without Jill.

The research presented in this thesis was supported by NSF grants CSR-0615255, CSR-0614770, CSR-0615194, CCR-0312956, DMS-05-28402, and AFOSR grant F49620.

Abstract

Multi-Experiment Studies (MESs) is a type of computational study in which the same simulation software is executed multiple times, and the result of all executions need to be aggregated to obtain useful insight. As computational simulation experiments become increasingly accepted as part of the scientific process, the use of MESs is becoming more wide-spread among scientists and engineers.

MESs present several challenging requirements on the computing system. First, many MESs need constant user monitoring and feedback, requiring simultaneous steering of multiple executions of the simulation code. Second, MESs can comprise of many executions of long-running simulations; the sheer volume of computation can make them prohibitively long to run.

Parallel architecture offer an attractive computing platform for MESs. Low-cost, small-scale desktops employing multi-core chips allow wide-spread dedicated local access to parallel computation power, offering more research groups an opportunity to achieve interactive MESs. Massively-parallel, high-performance computing clusters can afford a level of parallelism never seen before, and present an opportunity to address the problem of computationally intensive MESs.

However, in order to fully leverage the benefits of parallel architectures, the traditional parallel systems' view has to be augmented. Existing parallel computing systems often treat each execution of the software as a black box, and are prevented from viewing an entire computational study as a single entity that must be optimized for.

This dissertation investigates how a parallel system can view MESs as an end-to-end system and leverage the application-specific properties of MESs to address its requirements. In particular, the system can 1) adapt its scheduling decisions to the overall goal of an MES to reduce the needed computation, 2) simultaneously aggregate results from, and disseminate user actions to, multiple executions of the software to enable simultaneous steering, 3) store reusable information across executions of the simulation software to reduce individual run-time, and 4) adapt its resource allocation policies to the MES's properties to improve resource utilization.

Using a test bed system called *SimX* and four example MESs across different disciplines, this dissertation shows that the application-aware MES-level approach can achieve multi-fold to multiple orders-of-magnitude improvements over the traditional simulation-level approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software applications are rarely run in isolation. Across the fields of science, engineering, and art, the computer user often needs to execute the same software multiple times and only obtains scientific, engineering, or artistic insight by aggregating the results over all of the executions. This type of software usage pattern, where the same software is executed many times and only the aggregate result is useful, is referred to in this dissertation as a Multi-Experiment Study (MES).

As computer simulations become increasingly accepted as part of the scientific process, supplementing the traditional dyad of theory and experiments, more and more research groups turn to MESs made up of multiple computational experiments to obtain scientific and engineering insights. For example, a civil engineer may need to execute a bridge deformation simulation multiple times, each time using a different bridge structure, in order to find the optimal bridge design. Similarly, a medical device research group may need to execute multiple software runs to simulate defibrillation shocks, each time using a dif-

ferent configuration, in order to find the optimal defibrillator configuration for a given patient. Or, a group of chemical engineers may need to run a helium-air mixing simulation multiple times, each time using different model parameters, in order to discover the best set of model parameters that match real-life experiment results. MESs are so named because each software run often represent a computational experiment — e.g., the deformation of a bridge design, the activation of a heart due to a defirillation shock, or the formation of a helium plume shape — but each individual experiment does not convey the whole information. It is the aggregate information — the best support structures, the optimal electrode placement, or the most accurate model parameter — that is of ultimate interest to the user.

This usage pattern is not limited to scientific applications. An animator, for example, could run a physical simulation software of an animation scene many times, each time using a different setup, to discover the most artistically-satisfying setup for a physically-realistic scene. MESs can be found in diverse fields such as medicine ([5, 71, 85, 16]), engineering ([73, 28, 44, 6, 54]), finance ([68, 67, 31, 84]), and computer graphics ([43, 81]).

The widespread utility of MESs makes them an interesting area of investigation, to understand what type of computational system infrastructures are best suited to run MESs. In this regard, MESs present several challenges. First of all, many MESs require constant user monitor and feedback in order to steer the direction of a study as it is being conducted. Most traditional interactive systems, however, are focused on the steering of one individual experiment at

a time, rather than the steering of an entire MES that is made up of those experiments. For example, the defibrillator designer could specify the cardiac characteristics of different patients while the defibrillator MES runs, and the MES would provide the optimal electrode placements for each patient interactively. An interactive MES will need to gather partial information from multiple executions — often hundreds — of the software, aggregate those information, present the aggregated information to the user, receive user feedback, and disseminate that information to currently-running or yet-to-run executions. The second problem MESs pose is the sheer amount of computation needed. On a production system, often times a single simulation could take days on a multi-hundred way parallel cluster. For MESs built on such simulation codes, even a modestly-sized MES consisting of dozens of such executions could take up weeks of computational resources. The sheer volume of computation needed can make the time to completion for the MES prohibitively long.

Through the 80s to the early part of 2000s, the Moore's Law rate of growth in computational power has largely been realized by the continual improvements in processor clock speed ([63]). However, as transistors are packed ever more densely onto processor chips, transistor density approaches the physical density limit, and processor clock speeds have stalled. As a result, the growth in computational power is now increasingly being provided through parallelism, both at the chip-level and the cluster level. The shift from clock speed to parallelism manifests itself in different ways, each with its implications on computer system design.

Since Multi-Experiment Studies are made up of multiple independent executions of the same application code, they can be made embarrassingly parallel, making them an attractive candidate for massively-parallel systems. Recent trends in parallel architecture design appear to be a good match for addressing the challenges of MESs. On one end, widely-available, universal, low-cost computing resources like commodity processors have shifted their emphasis away from pipeline parallelism to instruction-level parallelism, as evidenced by the advent of multi-core processors, such as the Intel Cell, GPUs with support for general-purpose computing, and heterogeneous multi-core chips. This trend enables small groups of researchers or even individual researchers to have dedicated access to large amounts of parallel computational power previously only available to users of super computing centers. This availability of small-scale parallel computing resource opens up new usage models for parallel computing, such as interactive MESs. On the other hand, massively parallel computer architecture has shifted its focus from connecting a modest number of high-performance chips, such as multi-hundred way parallel cluster of SMPs (e.g., [86]), to connecting tens of thousands of inexpensive slower processor nodes using a dedicated high-performance interconnect (e.g., [78]). This new type of configuration allows for a level of parallelism never seen before. MESs made up of long-running simulation code are example of workloads that can take advantage of the computing power in this type of architecture.

However, in order to enable interactive parallel MES, and to effectively utilize the computational resources at hand, one needs to reconsider how paral-

lel software systems are structured and how resources are managed. Specifically, traditional parallel computing job scheduling software, most of which are based on the batch execution mode, treat each execution of the software as a black box. As such, they are prevented from viewing an entire computational study as a single entity to be optimized for. Thus, when running MESs on such schedulers, the execution sequence of the simulation software is either pre-determined (e.g., in sweeping over a parameter space), or is prescribed by the user serially, where the system runs one execution at a time, and the user manually determines which experiment to execute next based on the results from the previous simulation. MESs on traditional systems are thus unable to effectively interact with the user on the study-level, adapt to changing goals of the study, and take full advantage of the available computational resources.

To aid the execution of MESs effectively, the parallel system needs to be able to do the following. For interactive MESs, the system must be able to: 1) collect and aggregate execution results and display them to the user, and 2) receive user inputs while a MES is ongoing, interpret those inputs, and distribute the user inputs to subsequent executions. To effectively utilize computational resources, the system must be able to: 1) schedule all the executions within an MES as a single unit, based on the (potentially dynamically changing) goals of the MES, and 2) re-distribute computational resources from one execution to another as needed. To avoid redundant calculation across multiple executions of the simulation software, the system needs to: 1) re-use partial or completed results from an early execution in order to speed up subsequent executions, and

2) store and share data across multiple executions of the software.

In order to address the needs of MESs on parallel systems, a new system structure is needed. This thesis investigates a new approach to conducting MESs on parallel architectures. In this approach, the parallel runtime systems take the responsibility to conduct an entire computational study (i.e., collection of experiments), instead of just executing simulation experiments individually. The system can thus provide scheduling decisions (which, and in what order, are computational experiments run), resource allocation decisions (how many processing elements to assign to each experiment), storage layer support (for storing information that can be reused across different experiments), and a study-level interface for user interaction, in a way that optimizes the execution of the computational study as a single unit.

1.2 Approaches

This thesis focuses on MESs that use multiple executions of the simulation code to explore the space of input parameters to the simulation code. Most MESs fall into this category, referred to in this disseration as Design Space Exploration (DSE). In DSE, the user executes the simulation code multiple times, each time using a different set of input parameters. Then, based on the results of those executions, the user identifies a "region of interest" within the input parameter space, i.e., a region of the input parameter space that has some desirable qualities. For example, in a defibrillator study, the user would look for a set of input parameters to the defibrillation simulation code (placement of

defibrillator electrodes) that results in optimal activation, while in an animation design study, the user wants to find the inputs to the physical simulation engine (placements of objects in an animated scene) that results in an aesthetically-pleasing and physically-realistic animated sequence. This "region of interest" can only be identified if all of the executions' results are known, and thus reflects an aggregation of the results of individual simulations.

Some DSEs' goal is to discover a single region of interest, but in other DSEs, the region of interest changes dynamically as the user interacts with the study (e.g., the defibrillator study user can interactively define the patient's characteristics and find a new set of optimal defibrillator designs). Moreover, some DSE's region of interest is defined by the user's subjective taste (e.g., the animator's region of interest is defined by his aesthetic tastes of what constitutes an interesting animation sequence). These DSEs require constant user feedback and interaction.

Since a DSE consists of multiple executions of the same code, often times the same or similar computations are replicated in different executions (e.g., the bridge simulation code may end up solving slightly different non-linear systems for slightly different bridge designs). If some partial results can be stored, they can be reused in later simulations (e.g., a solution to a Newton solve for one execution could be used as an initial guess to another execution). This way, the latter simulations will only be computing the "delta" between experiments, which are often less computationally intensive than computing from scratch.

Thus, a DSE system should be able to dynamically receive user input, and,

based on those inputs, dynamically generate a collection of simulation experiments to be conducted, manage to run the experiments on parallel computing resources, receive the results, aggregate them, identify the region of interest, and display the aggregate result back to the user. In addition, a system should ideally have some sort of storage facility that allows sharing of information across different executions of the simulation code. Ideally, the system should be able to bring all these considerations together — user interaction, scheduling and resource allocation, and data reuse — to make optimal progress toward the DSE's goals.

The optimal way to conduct a particular DSE, unfortunately, often depends on the properties of the specific DSE being conducted. For example, the defibrillator study, where the region of interest is clearly defined, has very different requirements from an interactive animation scene design study, where the region of interest is more subjective. As another example, an DSE comprised of a small number of long-running simulation experiments have different resource requirements from an interactive DSE comprised of thousands of short-running experiments.

This thesis shows that, in order to conduct DSE-like Multi-Experiment Studies efficiently on parallel architectures, the parallel run time system needs to make policy decisions that are informed by application-domain knowledge. The system must treat an MES as a single entity in order to exploit optimization opportunities that become available from the study context. Only then can MESs be conducted efficiently on parallel architectures, enabling the next

level of scientific discovery. Current parallel systems, however, are inadequately positioned to meet such challenge.

The parallel runtime system described in this thesis leverages domain knowledge in multiple ways. The domain knowledge can be used to optimize the execution time of a single execution of the application code, to reduce the number of executions required, to enable user-steering, or to improve resource management. Specifically, with a study-level view, the runtime system is able to take advantage of the following opportunities that are not used in traditional systems:

- **Aggressive reuse of results from one execution of the application code to another.** In many applications, knowing the result from one execution can reduce the computational cost of a similar execution, because only the "delta" between those two runs needs to be calculated. For example, an application that uses an iterative solver can use the result of a previous solve to reduce the number of iterations needed for a different solve: if the two systems are similar, the solution should be similar as well, so, by using the solution of one as the initial guess to the iterative solver of another, one should be able to reduce the number of iterations needed. In some applications, the same intermediate states may be calculated during different runs. If these intermediate states can be stored from one execution, they can be retrieved and reused in subsequent executions, eliminating the need to recalculate them.

- **Aggregation of information from earlier executions to inform**

later scheduling decisions. In some MESs, the aggregated results from early runs are used to determine which of the later runs are required to complete the study. For example, some Design Space Exploration algorithm may use early runs to identify important parts of the design space associated with the region of interest, so that later runs are only needed if they explore those parts. This technique can reduce the actual number of subsequent executions needed.

- **Display of the aggregate information to the user.** In interactive MESs, the system needs to display the aggregate study-level result to the user while the study is on-going (as opposed to displaying the results of individual executions of the application). That way the user can monitor the progress of the MES as it is being conducted. Therefore it is crucial that the system can gather the results of completed executions while the study is still on-going, and translate the results into useful information to be displayed to the user.

- **Collection of user input while the MES is being run, and dissemination of the input to subsequent executions.** Analogous to collecting and displaying aggregate results, an interactive MES system must be able to collect the user's study-level directives from the user interface, translate them into inputs of the individual executions of the application, and relay the inputs to the individual executions of the application code.

- **Scheduling of executions to maximize result reuse potential.** As discussed above, the results from an execution can be beneficial to subsequent executions. However, in some MES, not all executions' results are equally beneficial. For example, some application code can only reuse the results from a run whose input is similar enough. A system that schedules the runs early whose results have a higher result potential can maximize the benefits it can give to later runs.

- **Intelligent scheduling based on the scaling behavior of the application code.** For MESs built out of application code that can run on multiple processors, the system also needs to determine the level of parallelism to allocate to each execution of the application code. As shown above, not all executions have equal priority: it is desirable, for example, to complete runs with high result potentials early, and it is also desirable to complete executions early that can produce the most useful aggregate information for the user in interactive MESs. It is therefore desirable to schedule more computational resource to these "high-priority" runs. However, because of scaling overhead, it is not clear whether the advantages gained from putting more resources in those high-priority runs are offset by parallelism overheads. The optimal way to schedule the executions thus depends also on the scaling characteristics of the application code.

- **Redistribution of computational resources.** On MESs whose application code has checkpoint mechanisms, the system should be able

to make a decision to reallocate computational resources that are freed up by executions: should the freed resource be used to start new executions? Or should it be reassigned to ongoing executions, in which case the ongoing execution has to be stopped and then restarted from the checkpoint using more resources? The system needs application-specific knowledge, such as the scaling behavior of the application code, and the expected work between checkpoints, in order to make such a decision.

To test this thesis, a parallel runtime system framework containing the above-mentioned strategies was built for conducting MESs. The system is called System Software for Interactive Multi-Experiment Computational Studies (SIMECS, or SimX for short). The central feature of SimX is that it contains a number of APIs which allows application-specific knowledge to be passed into the system, enabling SimX to make informed policy choices that are specific to the MES being conducted.

SimX is designed to be run on clusters of hundreds of nodes, and uses standard TCP/IP and MPI APIs for communication. For easy integration with existing frameworks, it is packaged in three forms: as a stand-alone library for MESs with stand-alone application code, as a collection of components in the SCIRun Problem-Solving Environment ([50, 49]) for interactive MESs, or as part of the Uintah component-based framework ([27, 60]) for MESs with component-based application code.

To evaluate the approach and techniques used in SimX, we take the four MESs across different disciplines introduced above — one in mechanical en-

gineering (the bridge design study), one in chemical engineering research (the helium gas mixing study), one in bio-medical device design (the defibrillator study), and one in computer animation (the animation scene design study) — and investigate how SimX's use of domain-specific knowledge can improve the performance of these MESs. These studies represent a range of system requirements: they vary in their user interactivity requirements (two are interactive, two are not), simulation code properties (three are serial code each taking seconds of run time, one is parallel code taking hours), and study objectives (three have well-defined regions of interest, one has a subjective region of interest).

1.3 Contributions

This thesis shows that, by applying application-specific domain knowledge to guide the runtime system design and decision-making process, a parallel system can efficiently manage the collection of experiments in a Multi-Experiment Study to achieve its goals: enabling user-interactivity at a study-level and reducing the overall run time.

In particular, this thesis proposes an API that can be used to import the application-specific knowledge into a parallel run time system. Through the API, the user can specify the goals of the MES to guide the system's scheduling descisions, reducing the amount of computation required. The user can also use the API to specify how to aggregate execution results from, and disseminate user inputs to, multiple executions of software, thus enabling steering of the entire computational study. Additionally, the user can use the API to store

information across executions of the software, enabling the reuse of computation results across the different executions. Finally, the user can use this API to specify the simulation code's scaling behavior, and reuse behavior, so that the system can adapt its resource allocation policies to improve resource utilization.

Finally, this thesis quantifies the benefits shown by each type of domain-specific knowledge on the performance of four example Multi-Experiment Studies. Using SimX as a test bed, and the four MESs as test cases, this thesis shows that the approach it describes can achieve multi-fold to multiple-orders-of-magnitude improvements over the traditional approach.

1.4 Thesis Organization

Chapter 2 describes related work in the area of parallel runtime systems, on many of which this thesis is based. Chapter 3 presents a formal description of our motivating problem, Design Space Exploration, and describes in detail the four example applications, which serve as running examples in this thesis. Chapter 4 describes the system that was built to test this thesis, SimX. Chapter 5 describes in detail how application-specific knowledge can be used by SimX in each of the four example MESs. Chapter 6 presents the experiment results of using SimX to conduct the example studies. Chapter 7 summarizes and generalizes what is learned in this thesis, and presents ideas for future work.

Chapter 2

Related Work

The problem addressed in this thesis is related to efforts undertaken in a number of research areas related to parallel runtime systems, including parameter sweep applications on computational grids, steerable and interactive parallel applications, as well as component-based problem solving environments. This chapter presents an overview of the most important works in each of these areas. However, as discussed below, these prior efforts do not adequately address key properties of the Multi-Experiment Study problem, namely, study-level interactivity, data reuse across experiments, resource allocation, and the interaction between these concerns.

2.1 Grid Parameter Sweep Applications

A large body of work has been devoted to exploring the idea of using computational grids to conduct parameter sweep-based Multi-Experiment Studies. Examples include **Nimrod** [2], **Nimrod/O** [1], **Condor** [80], **Globus** [10], **NetSolve** [18], **VI-Steering** [33], and **Condor DAGMan** [79].

In such systems, a computational grid — a group of loosely-coupled, likely

heterogeneous computers whose membership can be dynamically changed ([36]) — is used as the execution platform for the MES. The grid scheduler system steps through the input space of the simulation experiment, effectively sweeping through all the possible input parameters of the code.

These systems deal with a problem that is similar to, but different from, the problem addressed in this thesis. Both use parallel computers to explore a parameter space. However, typically, grid parameter sweep systems deal with the loosely-coupled nature of the grid, and pay little attention to how the properties of a particular MES can be leveraged to conduct it more efficiently.

There are exceptions. VI-Steering, Nimrod/O, and Condor DAGMan, for example, leverage knowledge of specific MESs and apply them to the grid architecture. In the subsections below, we first discuss grid schedulers in general, and then discuss some of the MES-specific Grid-based parameter sweep systems.

2.1.1 Grid Schedulers

The basic grid scheduler system are concerned with coping with problems arising from the loosely-coupled nature of the grid resources. This subsection looks at the architecture of two such systems, NetSolve and Globus, to illustrate the ideas behind their designs.

NetSolve [18] is an early example of a grid system scheduler. The grid in NetSolve consists of a loosely-coupled set of machines, each running a NetSolve *computational service* or *communication service*, or both. The machines communicate through TCP/IP, and use the External Data Representation Standard

protocol ([45]) for exchanging data. The machines running the communication service are called NetSolve *agents*, and are responsible for much of the communication, book-keeping, and system-level decisions. The NetSolve agents provide a directory service that locates machines providing computational services. The user, using a NetSolve *client*, can dynamically add or remove computational servers to the computational grid through agents. Then, when the client needs to submit a job, it can ask the agent for a list of computational servers that the client should submit to. The NetSolve agents keeps track of the computational power, workload, and network link characteristics of each computational servers it knows about, and decides which computational server to return to the client with the objctive of balancing the load on the grid. It uses a rudimentary model to estimate the performance of a given job on a given computational service, taking into account the computational power of the server, the complexity of the algorithm used in the job, the size of the problem in the job, the workload of the server, and the latency/bandwidth of the network link to the server. Apart from directory service and load balancing, the agents are also responsible for detecting failures of computational servers, and removing them from the list of available servers when a failure happens. NetSolve thus addresses the underlying problems of grid computing: load-balancing, directory service, and fault tolerance.

In the **Globus** architecture [37], there are no separate agent processes. Instead, each Globus server can provide both computational and communication services. Globus uses Web Services ([11]) to define its interface and structure

its components. Here, the grid is composed of machines providing a number of services. Some of these services are application-specific, implemented by the application developer on Globus service *container components* to perform useful application-specific functions. These services are analogous to the services provided by NetSolve computational servers. Other services are provided by pre-packaged Globus *components* to solve infrastructure-level problems. These services are analogous to the services provided by the NetSolve Agents, and they cover the areas of execution management, data management, monitoring and discovery, and security. Execution management services allow a client to dispatch tasks, control the tasks' resource consumption, restart a task in the event of failure, and even run MPI processes on grid resources. Data management services provide tools for the client and grid machines to transfer files, replicate and manage data, locate data replica, and provide access to data. Monitoring and Discovery services dynamically collect and aggregates information about which machine provide which services. Security components implement multiple credential formats and protocols to provide message protection, authorization, authentication, delegation, and auditing functions. Globus users can access all these services using a set of Globus *client libraries*. Application developers using the Globus infrastructure can utilize these services to suit the needs of their applications.

As is exemplified by NetSolve and Globus, the goal of basic grid systems is to manage a loosely-coupled, dynamically changing set of computational resources via a set of service provided by these resources. Their main concern is to

provide fault-tolerance, load-balancing, discovery and directory services. More sophisticated systems provide security and data replication features. Other grid systems may have more specific goals. For example, in Condor, a grid machine may not be dedicated exclusively to running jobs from the grid, and only wishes to make itself available to the grid when it is idle. Condor can be configured to opportunistically detect idle CPUs on those machines, and special functionality is added in so a machine can add itself to the grid when it is idle, and removes itself from the grid when it detects non-Condor activities.

2.1.2 Parameter Exploration on Grid Systems

In the Grid systems described above, the policies governing some of the system services, such as data replication and load balancing, should be influenced by requirements of the applications being run. In order to search a parameter space using only these tools, one needs to determine *a priori* which parameter points to explore (i.e., which computational experiments to run), submit those points to the system, and rely on the system to load-balance and distribute the jobs to computational elements on the grid.

More specific tools such as VI-Steering and Nimrod/O show that, by introducing more application-knowledge into the system policies, it is possible for the grid system to make better choices. Specifically, the decision of which tasks to execute can be determined dynamically by the system. This flexibility enables the system to more efficiently explore the parameter space and react to user actions. These application-specific grid-based parameter search systems

are built by adding a layer on top of the grid systems described in the previous subsection.

Nimrod/O [1] is a tool based on Nimrod, and is used for one specific type of MES, namely, MESs that perform a single-objective optimization on a parameter space. Here, each computational experiment corresponds to a *function evaluation*, and Nimrod/O tries to find a point on the parameter space that minimizes that function. The user only needs to use a declarative language to indicate the parameter space he wishes to explore, and Nimrod/O's job control module will determine, based on the search algorithm chosen by the user, which tasks need to be executed. These tasks are sent to Nimrod for execution on the grid, and the execution results (function values) are stored in a data cache as well as sent back to the job controller. Nimrod/O's job controller supports four search algorithms: parallel BFGS (Broyden-Fletcher-Goldfarb-Shanno) method, the Simplex method, Simulated Annealing, and a Divide-and-Conquer heuristic. In essence, Nimrod/O makes use of the domain knowledge of its target application to enable efficient scheduling decisions. Without Nimrod/O, the user would need to perform a parameter sweep using Nimrod, which is much less efficient.

VI-Steering [33, 17] is a framework for conducting interactive steerable parameter sweep studies on distributed machines. It tries to solve the same problem as Nimrod/O: optimization of a function within a parameter space, where the function is evaluated by executing a simulation code. Instead of using a pre-determined search algorithm, however, VI-Steering takes the approach of

using the user's interaction to guide its search of the parameter space. In VI-Steering architecture, there are three main components: the VI Daemon, the VI Database, and the VI Interface. The VI Interface is responsible for receiving the user's inputs, which indicates the parameter space he wishes to explore. The VI Interface then communicate this space to the VI Daemon, which translates the parameter space specification into a list of tasks by sweeping the selected parameter space, and off-loads the tasks on a grid service. The grid service executes the tasks, and returns the function evaluations to the VI Daemon. The VI Daemon updates these function evaluations into the VI Database, which the VI Interface can access and analyze. The structure of VI-Steering is designed to enable to user to look at partial result of the MES — the function evaluations of early experiments — while some tasks are still being executed on the grid. This way, the user can interactively specify to the system sub-regions of the parameter space that seems to him worthy of more exploration. The user can view the partial study-level result, dynamically select a new region of parameter space to explore, and the VI-steering system will immediately start sweeping the new parameter space region. VI-steering thus represents some elements of study-level steering.

Both VI-steering and Nimrod/O make use of application-level knowledge to provide system-level support for scheduling and user interaction. Thus, they point a way to solving the problem presented in this thesis. Unfortunately, both of these systems are geared toward one specific type of problem (single-objective optimization), whereas this thesis aims to find a more general way of expressing

application-specific knowledge in MESs. In addition, these systems are designed to run on grids, which have very different communication/computation characteristics from the systems investigated in this thesis: tightly-coupled massively parallel clusters and chip-level parallel systems. In particular, the interaction between scheduling, data reuse, resource allocation, and user interaction are vastly different when it is easy to move data between computational elements, and easy to dynamically form and reform process groups from computational elements.

2.2 Steering of Parallel Computations

As VI-steering shows, it is important for some MESs to have the ability for the user to actively engage in the parameter space exploration while the study is still going on — viewing the partial result of the study, and changing the specification of the study interactively. Effectively, the user can dynamically control which experiments are issued, and by doing so, steer all the machines on the grid simultaneously.

One of the goals in this thesis is to investigate an API that can be used to enable study-level user steering of MESs. While VI-steering demonstrates one way to conduct a specific kind of study-level steering, this section looks at the topic of enabling the user to simultaneously steer multiple machines running a single computation. This topic has received a fair amount of attention in prior work, in systems including **Falcon** [46], **CUMULVUS** [41] [55], **Computational Steering Environment (CSE)** [56], **Mirror Object Steering Sys-**

tem [30], **Distributed Laboratories** [69], **DISCOVER** [59], **WEDS** [22], **GViz** [48], and **RealityGrid** [13].

2.2.1 Parallel Steering Systems

A class of parallel steering systems focuses on the technique of steering computations on tightly-coupled parallel machines — how to gather data from processing elements and present it to the user, and how to disseminate user interaction to the processing elements. These systems, including Falcon, Mirror Object Steering System, CUMULVUS, and CSE, are concerned mainly with controlling the latency with which messages can travel between the user and the parallel machines, and devising an appropriate API with which the user can specify how the steering can be done.

As an example, **Falcon** uses a steering server in its run time system to facilitate the communication between the user and computation nodes. To run his application on Falcon, the application developer must have access to the source code of the application program. The developer first declares which variables in his application program can be monitored and/or steered. Then he uses Falcon's instrumentation tool to modify his program, inserting probes and sensor codes, after which he recompiles his program. He then starts the Falcon run time environment. The instrumented application program runs on the compute nodes of the parallel machine, while a Falcon steering server is run on a front-end. The instrumented application code can, at run time, communicate with the Falcon Steering server to relay the variable's status back to the user,

or to set those variable's value based on the user's actions. At the front end, Falcon presents a graphical user interface to the user to display the values of the variables, as well as receive users' inputs.

Other similar systems are variations on Falcon's idea: first the user instruments his code to insert logic to communicate with the user's front end through an agent. Then, he deploys the instrumented application code on the processing elements. Then he launches the user interface and the steering agent (which can itself be a separate process or a component within the user's front end process) to remotely steer the collection of machines. CUMULVUS uses the same idea, and takes a data-centric view (where variables are declared in advance and the user can monitor and set them interactively), whereas CSE uses the same idea, but takes a function-centric view (where handler functions are declared in advance and the user can cause them to be executed interactively).

The difference between these systems and the problem dealt with in this thesis is that in these systems, all the variable changes are global: when a user changes the value of a variable, the change is visible to all processing elements. As a consequence, Falcon, and other basic parallel steering systems, are good for steering single executions of parallel software, rather than MESs, which can contain many executions of potentially parallel codes.

2.2.2 Grid-based Steering Systems

While CSE, CUMULVUS, Mirror Object Steering, and Falcon are targeted toward steering of parallel applications on tightly-coupled parallel ma-

chines, other parallel steering systems target steering on computational grids. VI-steering is one such example, which was discussed earlier. RealityGrid, Distributed Laboratories, DISCOVER, WEDS, and GViz are other middleware layers designed to solve the problem of steering a grid-based computation. They differ from VI-steering in that they are geared toward more general applications, rather than just one specific type of application (single-objective optimization MES).

In these systems, the grid-based applications are composed of multiple computing resources, each providing a different piece of service. The steering system needs to coordinate between these computing resource to ensure a consistent view. These steering systems are built on top of existing grid infrastructures. For example, RealityGrid is built using Open Grid Service Infrastructure (OGSI) [32], while WEDS is built on the Web Services Resource Framework (WSRF) [35].

In **WEDS** (WSRF-based Environment for Distributed Simulation), special web services are deployed on the grid to facilitate such coordination. When a user launches an application, a *broker service* is selected on the grid, which connects with the various component services of the application to coordinate its execution. These component services could include *machine services*, which provide the actual computation, *wrapper services*, which act as a communication gateway between the machine services and the outside world, *file services*, which runs on the user's machine and is responsible for moving data stored on files, and *visualization service*, which interacts with the user. Each of these

services may run on different machines, and the WSRF framework is used to coordinate the communication between them. These grid-based steering framework can also be used for monitoring and checkpointing grid-based applications. They take a data-centric approach: only certain parameters in an application are monitored and steerable, and they are to be declared in advance.

These grid parallel steering problems share many characteristics as the problem investigated in this thesis: the need to coordinate data coming from multiple machines, aggregating the data and displaying to the user, receiving user action, and disseminate the user's action to the machines. However, the computation resources considered in this thesis are more tightly-coupled, and typically more homogeneous. So, the service-oriented architecture in a grid parallel steering system is forgone in favor of the instrumentation approach used in more traditional parallel steering systems. Also, the grid scheduler is more constrained in where it can schedule a task: for example, the broker service on a grid can only choose to schedule a particular task on a machine that provides the service, whereas in a more homogeneous environment running an MES, the tasks are (mostly) identical, and the system has more flexibility on how to schedule them. As a result, in this thesis, performance concerns such as load balancing and resource utilisation can be used to guide this decision. Moreover, the problem in this thesis is more focused, namely, we are only looking at MESs. So, it is possible to use application-specific knowledge to facilitate the scheduling and resource management. Finally, in this thesis, the user has more control over the computational resources: unlike the grid, he is guaranteed

access to the parallel computation resources, because he either has an allocation on a data center, or he owns the parallel desktop machine. As a consequence, he has more flexibility on how the computational elements are grouped together and used.

2.3 Scheduling and Resource Management Techniques in Parallel Runtime Systems

In addition to parameter sweep applications and parallel steering systems, many scheduling and resource management techniques in traditional system research are applicable to this thesis's problem.

2.3.1 Scheduling of sequential tasks on parallel machines

One class of research focuses on scheduling inter-dependent sequential tasks on parallel machines. Many of these works utilize heuristics to map these tasks onto processing elements in order to minimize the total makespan of the group of dependent sequential tasks.

In one example, a simple model to describe **fine-grain scheduling** on multi-threaded multi-processors was proposed in [23]. Here, scheduling is done in units of "quanta", representing a fixed amount of work. Using this quanta model, the scheduler is able to take a dataflow-based computation graph and construct a schedule — the mapping of tasks onto distributed computing resources. This way of scheduling minimizes the overall run time, taking into account the time it takes for data to travel, the time of each task's execution,

and the dependency between tasks.

In another example, **ICC++** [52] offers a hierarchical approach to load balancing threads on a distributed platform. It groups threads into subsets and load balances each subset independently. This hierarchical approach simplifies the scheduling problem on a distributed platform, at the cost of requiring user intervention to identify subsets and give the scheduling policy for each subset.

The **Multiprocessor Scheduling Problem** [40] has also been studied extensively as a theoretical topic. In MPS, a system tries to allocate processing elements to a group of inter-dependent tasks while minimizing the total makespan of the entire group of tasks. The tasks are arranged into a graph: each task is represented by a vertex, and data dependency between the tasks are represented by directed edges. The communication costs are represented by the weight of the edges. Many heuristics are surveyed in [40]. Critical path-seeking heuristics such as Highest Level First with Estimated Time and Coffmann-Graham algorithms (both greedy algorithms) works well when there is no communication costs, while the Extended List Scheduling algorithm can accommodate communication costs by starting from assuming there is no communication costs, and inserting them into the resultant schedule later. Clustering algorithms seek to simplify the graph into *clusters* and schedule entire clusters on processors. Linear Clustering decomposes the task list into linear clusters (sub-graphs which involve only vertices that have one incident edge and one out-going edge), and schedule entire clusters onto the same processors. Internalization algorithms create clusters by merging adjacent vertices with heavy

edges in an attempt to reduce communication cost. The optimal heuristics depends on the situation — the connectivity of the task graph, the distribution of edge weights, and the number of processing elements available.

The work in this thesis can be thought of as extensions to the work mentioned above. Even though MESs consist of collections of independent tasks, some of the optimization techniques used in this thesis introduce dependencies between them, creating a group of inter-dependent tasks. Hence, our system then needs to solve the problem of minimizing makespan of a group of inter-dependent tasks. One difference between the work in this thesis and the above-mentioned systems, however, is that the tasks in the MESs discussed in this thesis are dynamically formed. In fine-grain scheduling, for example, static analysis is performed, i.e., all the pieces of the application has to be known before hand, whereas in MES, the dataflow and resource requirements are dynamically-determined. Another difference is that the tasks in this thesis can potentially be run in parallel. So, there is an extra dimension of deciding how much parallelism each task should receive.

2.3.2 Scheduling of parallel tasks on parallel machines

Another class of research focuses on scheduling parallel tasks on parallel machines. Here, the system needs to decide not only what order, and on which processing element the tasks ought to be scheduled, but also how many processing elements should be assigned to each particular task.

Affinity Scheduling [38, 19] is a language extension and runtime system that allows programmers to specify affinity of functions and data within a single distributed application consisting of many inter-dependent parallel tasks. The programmer can specify, in his program, that particular functions (which the runtime system translates into tasks) ought to be executed on a particular set of machines, or that a particular distributed data structure ought to be placed on a particular set of machines. The runtime system takes hints from these affinity specifications, and performs the scheduling decisions that would satisfy the users' constraints. This provides a mechanism for the user to gain fine-level control of each execution of his application code, but leaves the runtime system with a reduced flexibility to balance the load. The system balances the load by assigning processors to tasks and data whose affinity hints are are not specified by the user, and by job-stealing, where idle processors steal jobs from busy processors.

Research on the scheduling of **moldable jobs** [76, 77] has been conducted on large scale parallel systems. These works come from the research of traditional batch scheduling systems, where the goal is to improve individual jobs' turn-around times. Moldable jobs are jobs that can be run on a variable number of processors, so a moldable job batch scheduler needs to decide the amount of parallelism that is allowed for each job, such that the turn-around time — queue time plus execution time — of the average job in the system is minimized. These works introduce the Fair Share policy, where a job is assigned the number of processors such that the fraction of processing resources it is assigned ver-

sus the total amount of resource in the system is comparable to the fraction of job-weight (total number of processor-seconds requested) of the job compared to the total job-weight of the rest of the jobs in the system. This fraction is called the weight-fraction. Depending on the system's workload distribution, and the parallelization overheads of the tasks in the system, variations of the Fair Share policy have been shown to be beneficial. Submit-time fair share and schedule-time fair share make the assignment of computational resources at the time when the job is submitted and when the job is scheduled, respectively. Some Fair Share variants modify the definition of weight fraction, calculating it using the square root of the job's job-weights. Some variants set aside a number of processors to handle small and quick-to-finish jobs, so as to reduce those jobs' queue time. The idea of Fair Share policy has been found to be beneficial in the work in this thesis also, however, it needs to be adapted to accommodate a different goal. In traditional research on moldable jobs, the number of jobs are fixed, and its objective is to optimize for average turn-around time of individual jobs, while in MES, the number of jobs can dynamically change at runtime, and the goal is to optimize the total execution time.

2.4 Component-Based Problem-Solving Environments

A prototype system, SimX, was built to test the ideas presented in this thesis. For easy deployment, SimX is interfaced with two component-based Problem Solving Environments (PSEs). This section describes the two PSEs in detail, in order to facilitate the discussion of SimX in a later chapter.

2.4.1 SCIRun

SCIRun [65, 66, 21]) is an interactive graphical Problem Solving Environment (PSE), where the user can create interactive scientific applications out of component modules. It is based on a dataflow model, and designed for ease of use and interactivity, but not for large-scale parallel programs or MESs. With the exception of one variant ([62]), the applications composed in the SCIRun PSE are designed to run in the SCIRun graphical run time environment on a single computer. A typical application is described in [58], where an interactive application for bio-electric field calculation is built out of SCIRun components. This application forms the basis of the defibrillator design example MES described later in this thesis.

In SCIRun, common functionalities for scientific computing, visualization, and steering are encapsulated into *components*. The components' interfaces are specified as *ports*. Each component may have *input ports*, which specify inputs to the component, and *output ports*, which specify the results computed by the component. For example, Figure 2.1 shows the solver component, which solves a system of linear equations $Ax = b$. It has four input ports, shown on the top of the component. The four ports correspond to the inputs: the stiffness matrix A, the RHS vector b, an optional preconditioner matrix, and an optional initial guess to its iterative solver. There are two output ports, corresponding to the solution vector x, and the preconditioner matrix computed for A.

When using SCIRun, the user mixes and matches these components through a graphical user interface. He connects the output ports of some components

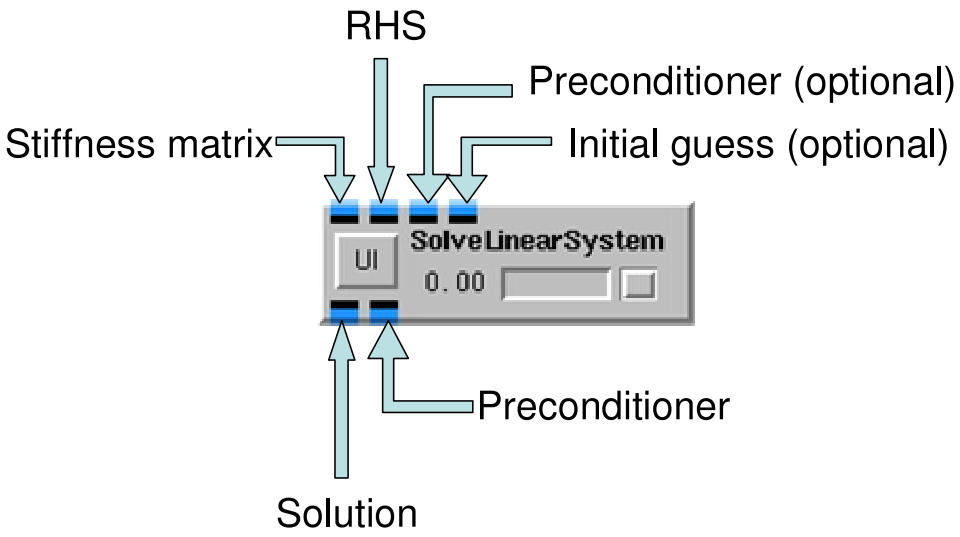


Figure 2.1: SCIRun Solver component

to the input ports of other components in order to construct a *workflow* that, when executed, computes the functionality he needs. Figure 2.2 shows a SCIRun workflow net, annotated with labels of functions of each part of the workflow. Once the workflow is constructed, the user can then specify inputs to this workflow using the components' UI. The UI is a way (in addition to input ports) for components to receive inputs. However, unlike ports, which receives inputs from another components' outputs, the UI receive inputs from the user directly. Figure 2.3 shows the UI of the solver component. In the UI, the user can specify information to the component directly, such as the residual tolerance, iterative solver, and iteration limits of the solver. Each user action in the UI of a component causes the workflow net to which the component belongs to execute. So, if the user uses the UI of a component upstream to change the value of the RHS of the solver component, the solver component will be executed as a result.

To display simulation results to the user, SCIRun provides visualization and rendering components. When these components are connected to a workflow net, that can transform the outputs of the workflow net into an image of an object in 3D space. Figure 2.4 is an example of a visualization component which shows the bio-electric field inside a human torso. When the user changes the inputs to the workflow net via UIs in the net's components, he causes the workflow net to be re-executed. As a result, he can observe the new output from the visualization components as SCIRun executes the workflow. Each time he specifies a new input, a new workflow execution is triggered, and the visualization component shows the updated image. In this fashion, SCIRun enables the

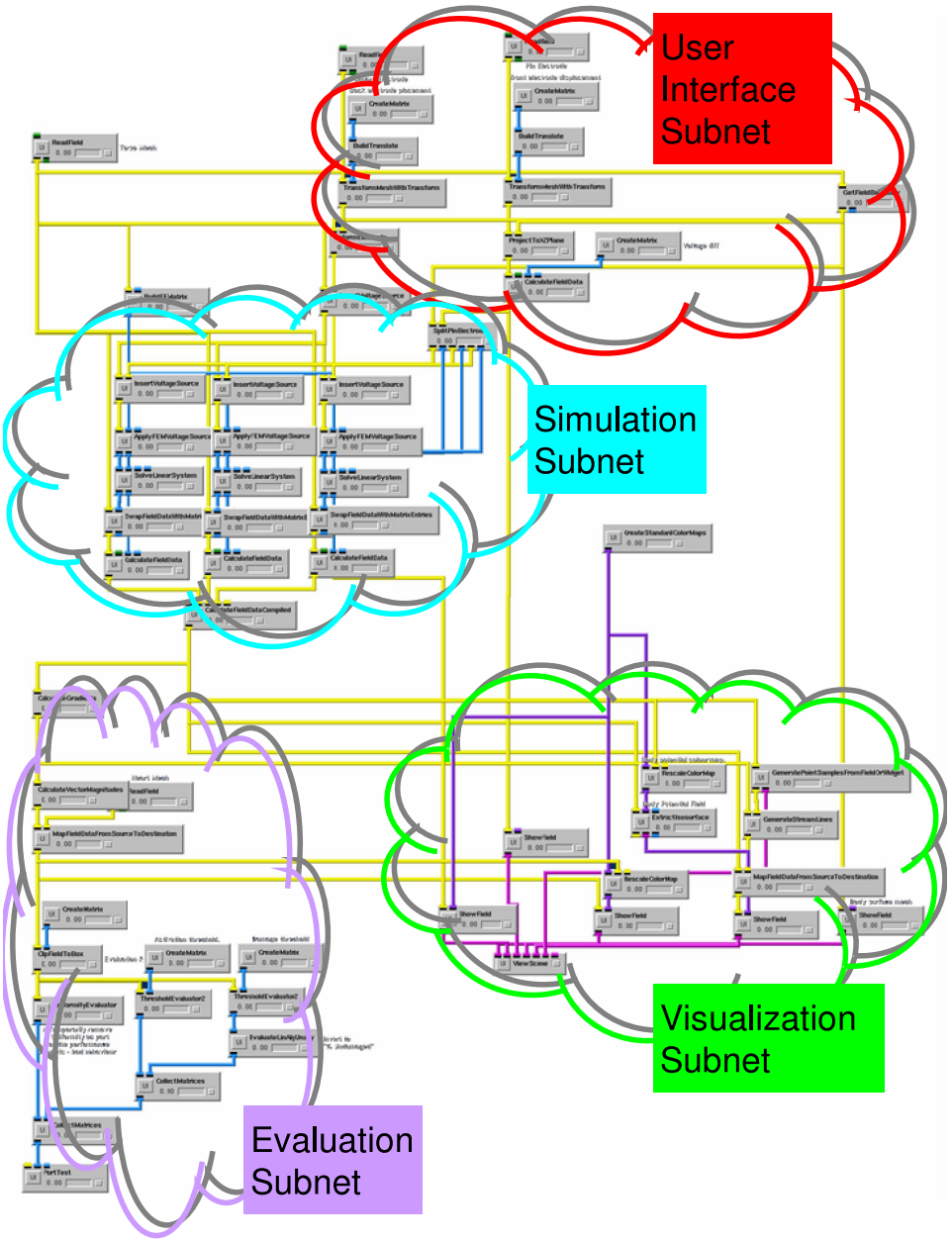


Figure 2.2: SCIRun workflow net.

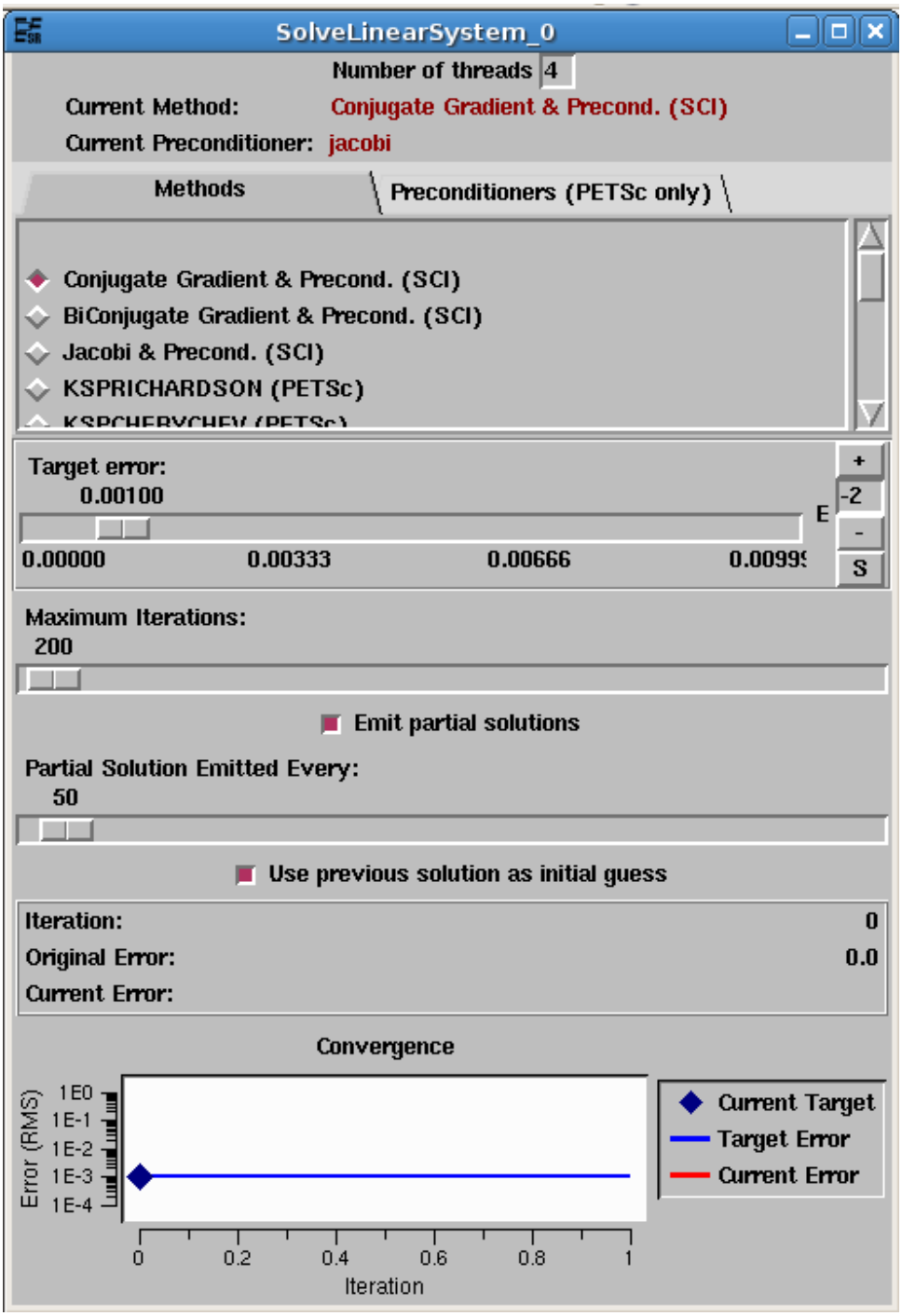


Figure 2.3: User Interface of the SCIRun Solver component

user to interactively control the application.

Using SCIRun, a scientist or engineer can interactively steer a single application on a serial computer. The work in this thesis uses SimX's API to extend SCIRun's capability to allow it to steer and monitor entire computational studies. In addition, a SimX/SCIRun interface is also developed so that any SCIRun workflow applications can easily be incorporated as the simulation code in an MES. Section 4.2 shows how SimX extends SCIRun to enable the steering of entire computational studies on a parallel cluster.

2.4.2 Uintah

Like SCIRun, **Uintah** [26, 42] is a component-based, workflow-based Problem Solving Environment. Unlike SCIRun, however, it is not designed for real-time user interaction, and thus does not have a graphical user interface for the purpose of steering an execution. Rather, it is designed for composing large-scale parallel simulations out of reusable components.

There is no graphical user interface and no component UIs in Uintah. Instead, the Uintah user connects output ports and input ports using Uintah components' C++ attachPort() interface. By calling this interface on various components in his driver code, the user can implicitly set up a workflow. The component's interfaces (input ports and output ports) are specified in the form of component XML files. When the user starts a Uintah run, he supplies a problem specification XML file as an argument to the driver executable. The problem XML file contains the input data required by all the components in the

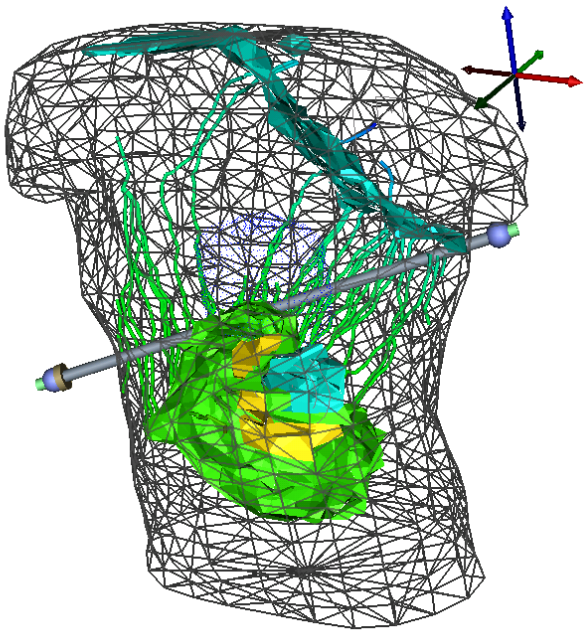


Figure 2.4: SCIRun visualization component shows the electric potential in the torso during a defibrillation shock.

workflow, and is parsed by the Uintah run time at the start of a Uintah run. Each of the components within the workflow net interpret their own blocks in the XML files. The Uintah runtime then takes care of the execution of components in the Uintah workflow net, moving data from output ports to connected input ports.

Uintah is designed to run on parallel clusters. Uintah components have access to an MPI communicator, so each component can potentially be executed in parallel. In addition, some components have access to common high performance libraries like PETSc ([7, 8]) or Hypre ([34]).

There are no visualization or rendering components in Uintah. In their places, a data archiver component is used. All components that have some information to return to the user would write their outputs to the data archiver. The archiver then writes this information in the form of several XML files. Each data archive dump is put into a separate sub-directory, and the XML files are written into the sub-directory. Visualization is performed off-line.

Uintah applications proceed in timesteps. At each timestep, the Uintah run time determines the timestep size, advances the simulation domain by executing the workflow, and then determines whether the timestep has to be retaken (if, for example, the timestep size was too big). Finally, it determines whether it needs to write the data archive out to the file system. Uintah is intelligent in that it would skip executing a component if its inputs are unchanged from the last timestep.

Uintah is designed for long-running workflows on massively parallel clus-

ters. The data archives can be re-used: if the user specifies a data archive to his driver code instead of an input XML file, Uintah can pick up from the data archive and continue the simulation. I.e., data archives can be used as checkpoints. As will be seen, this ability to restart a simulation from a checkpoint is especially useful for reusing simulation states across different experiments.

Using Uintah, a scientist or engineer can easily compose parallel checkpointable applications by connecting existing software components. In this thesis, an SimX/Uintah interface is constructed so that Uintah as be used as a substrate by which we can investigate resource allocation and scheduling issues in MESs that are composed from parallel simulations. In particular, the driver code in Uintah is modified so that it acts as a "shell" that treats the Uintah workflow as if it were a simulation experiment that makes up an MES. Section 4.2 shows how SimX extends Uintah to enable the execution of an entire computational study.

2.5 Summary

This chapter has provided an overview of research efforts related to the area of interactive parallel Multi-Experiment Studies. Various approaches contain promising ideas, and some are found to be beneficial. Parameter sweep applications on computational grids provide insights into how to conduct MESs on distributed resources, but the different network characteristics of computational grids and parallel desktops and clusters means that not all of the techniques are tranferrable to this thesis. Interactive and steerable parallel application frame-

works provide a model using which an MES system can monitor and steer an MES on distributed resources, but the fact that they are designed for steering single parallel application limits their applicability to the MES problem, which requires a more flexible steering mechanism to deal with multiple experiments. Other works in traditional run time system research also deal with scheduling and resource management of inter-dependent serial jobs on parallel computers, but the static nature of these analyses limit their applicability to the dynamic nature of interactive and adaptive MESs. Most importantly, with few exceptions, none of the above systems takes the approach of using application-specific information in aiding their system decisions.

Chapter 3

Problem Statement

To demonstrate the challenges and requirements faced by Multi-Experiment Studies, this thesis looks closely at the four MESs as running examples. Each of these MESs have a different set of requirements — some consist of a large number of short-running simulation experiments, some consist of a small number of long-running simulation experiments, some require constant user input, and some have ill-defined goals. These motivating examples are intended to show, in concrete terms, the challenges posed by MESs, and also to serve as a test bed for the system techniques proposed in this thesis.

In each of the following sections, a detailed description of each of the four example MESs is given. A description for the simulation code that makes up the study is given first, then the studies' objectives are explained, followed by a discussion on the requirements and challenges posed by the study on the system used for conducting them.

In the last section, a formal formulation of MESs is given. This formulation puts all MESs into a framework around which a parallel system can be developed specifically to address their requirements.

3.1 2D Bridge Design

The first computational study is from the domain of engineering design. MESs are frequently used in the engineering design domain ([73, 28, 44, 6, 54]). Often times engineering design MESs aim to look for trade-offs in design parameters of particular structures.

This example is a model civil engineering design problem. Suppose a strut bridge is being designed to be built across a river. There are many design choices the engineer has to make: the placement of the struts, the width of the span, the material used, etc... To make these decisions, many considerations have to be taken: the traffic requirements on the bridge, and load requirements of the bridge, the climate condition, the cost of construction (which itself depends on the cost of material and shape of the riverbed), etc... These considerations often can be translated into design objectives: minimize the deformation of the structure under normal traffic load, minimize the cost of construction, etc.

To make a given design choice, the engineer may need to employ repeated runs of a simulation software. He would need the software to find out, for example, the stresses and deformations of various designs under various loads. The results of this software can help him make the design choices that will best meet the design objectives.

Often, however, the design objectives conflict with each other, e.g., a thicker span provides more support but is more expensive to construct, so there is not one obvious design choice: rather, there is a number of equally good design choices that represent the trade-off between the objectives. It is the goal

of the MES to discover this set of design choices among all the possible design choices the designer could make.

In this first MES, we simplified the bridge design problem on several fronts. First, we use a simplified version of the bridge simulation code. Instead of using a full-scale strut simulation, we use a simplified 2D version. This allows the study to be performed in a reasonable amount of time. Second, we limit our design choices to the placement of two struts only. Third, we limit our design objectives to two dimensions: the deformation of the bridge and the cost of construction. This version contains many of the same properties as the full version — a simulation code involving a non-linear solver, conflicting design objectives, and multiple design choices. Thus, a full version is expected to be a scaled versions of the one described.

Figure 3.1 illustrates the simplified design problem. In this simplified version, the user tries to design an elastically deforming bridge with four supports, two of which are fixed at the endpoints, and the placement of the other two is to be decided. The MES tries to find a set of placements of the two non-fixed supports r_0 and r_1 in order to minimize *both* the cost of construction $f(r)$ as well as the maximum deformation of the bridge under uniform load. The specific problem is relatively straightforward, and the individual simulations are relatively fast — typically taking 7 seconds each. At the same time, it captures many essential features present in a real-life engineering design situations, e.g., one involving simulations of a car body or a more complex structure.

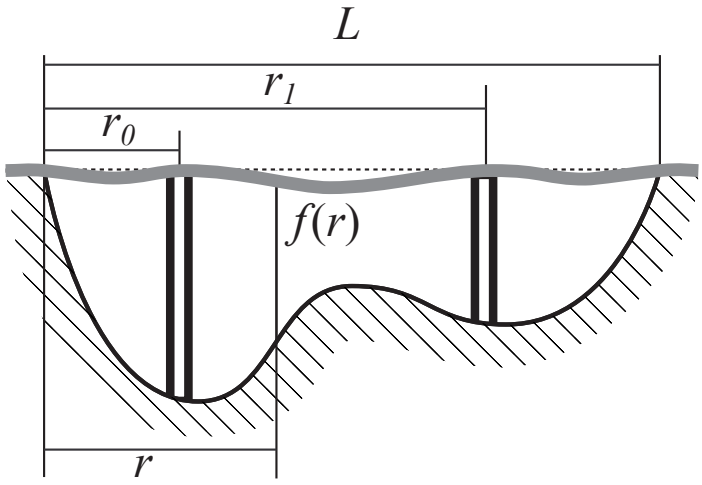


Figure 3.1: The bridge design problem

3.1.1 Simulation software

The *simulation code* used in this study simulates the bridge's deformation under uniform load. The supports at the endpoints are fixed and clamped, but the remaining two supports are modelled as columns made of a homogeneous elastic material. The bridge itself is modeled as a one-dimensional finite element rod elastically deforming in two dimensions. The four support points are modelled as boundary conditions to the finite element problem. A nonlinear model is used to model the bridge.

Mathematically, the problem is described by the potential energy for the bridge, the boundary conditions, and a predefined cost function for the construction of the non-fixed supports.

The bridge is discretized into a number of degrees of freedom that control the actual material points. The reference (initial) position is first defined as a straight horizontal rod. The potential energy is defined as the sum of five terms on each material point: the bending energy, which depends on the distance between two material points, the membrane energy, which depends to the curvature at a material point, the potential energy due to gravity, which is proportional to the displacement of the material point from its reference position, and the potential energies due to the two elastic supports, which are modelled as simple elastic springs of high stiffness, but only applied to material points near to where the elastic supports are placed. A Dirichlet boundary condition is set at the fixed support at both end points.

The simulation starts from the undeformed position, then a Newton search

is conducted to find the deformed configuration that minimizes the potential energy of the rod. To solve the nonlinear problem, we use the Newton method with cubic line search. We solve the linear system in the interior loop of the linear solver using a direct LU solver.

We use a simple nonlinear discretization of the problem. The parameter t is the distance from a material point to a reference point; t_0 and t_1 are the endpoints. The rod's displacement $p(t)$ is discretized at points $p_i = p(t_0 + hi)$, where $i = 0 \dots N$ and $h = (t_1 - t_0)/N$. The discretized energy of the bridge is given by:

$$\sum_{i=1}^{N-1}(\frac{2\theta_i^2}{l_i + l_{i+1}} + \rho g p_i^y) + \sum_{i=0}^{N-1}(l_i/h - 1)^2 + K_c(p_{r_0}^y)^2 + K_c(p_{r_1}^y)^2$$

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. K_c is the effective elastic stiffness of the supporting columns, and r_0 and r_1 are the indices corresponding to the location of the intermediate elastic supports. The forces and force derivatives are obtained by differentiating the energy with respect to the degrees of freedom.

The boundary conditions for the problem are fixed at the end points of the rod: $p(t_0) = (t_0, 0),\ p(t_1) = (t_1, 0)$.

Once the deformation is determined, we run the *evaluation code* to determine its maximum deformation F_1, as well as the cost of construction F_2. The

maximum deformation of the bridge is simply the displacement of the material point furthest from its reference position. The cost of construction is the sum of constructing each of the two elastic supports, where the cost of constructing each elastic support is defined by a user-defined cost function. This function can be thought of as being dictated by the shape of the riverbed as shown in Figure 3.2, which dictates the difficulty of constructing a support. The deeper the support has to reach the bottom of the river, the higher the cost.

For support positions r_0 and r_1 and the corresponding deformed configuration of the bridge $p(t)$, F_1 and F_2 are given by:

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

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

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

The simulation code is implemented serially in C++, and uses a widely used scientific computing library (PETSc) ([8, 7]) to solve the discretized systems. There are 2000 degrees of freedom, so the data set for each simulation result is about 50KB. Each simulation takes about 7 seconds to run.

3.1.2 Study objectives

As discussed in the previous subsection, in this simplified form of the bridge study, the user tries to find the placement of the two non-fixed supports (r_0 and r_1) in order to simultaneously minimize the cost of construction (F_1) and the maximum vertical deformation of the bridge (F_2). In general, an MES

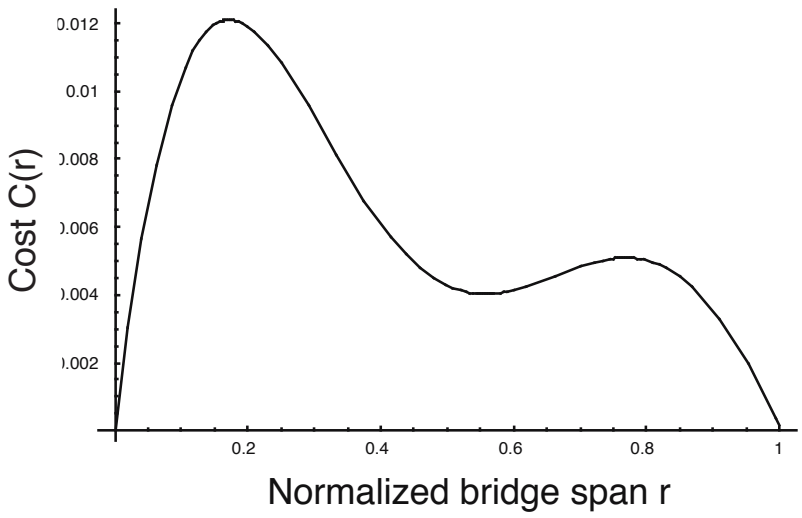


Figure 3.2: Support cost as the function of the distance to the left endpoint.

could have a larger number of design choices, as well as a larger number of design objectives.

As is often the case in multi-objective design, the two objectives in our bridge study work against each other. For example, placing the supports near the riverbanks can minimize the cost of construction, but the large unsupported span in the middle of the bridge will yield a larger deflection. A desirable placement of the support would thus trade off among the two objectives.

The goal of this bridge MES is thus to run as many configurations of as needed in order to find a *set* of configurations that meet the objective. In particular, this MES employs the multi-objective optimization technique called *Pareto Optimization* to achieve this goal.

3.1.3 Pareto Optimization

The general form of this problem, which explores a *design space* spanned by multiple *design parameters* (in this case, the placement of the two elastic supports) in order to optimize for multiple *performance metrics* (in this case, the cost of construction and maximum bridge deformation) that span a *performance space* is commonly found in many disciplines. Two of the other example MESs in this thesis take the same form of multi-objective optimization (see Sections 3.2 and 3.4).

A typical strategy for multi-objective optimization is *Pareto optimization* (e.g. [61, 85, 12, 47, 57]). Pareto optimization seeks to find a *set* of optimal designs, with each *Pareto-optimal* design 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 lower 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 the cost of construction cannot be improved while holding the maximum deformation fixed, and vice-versa. This set of optimal points is the *Pareto Frontier*.

Formally, the Pareto Optimization problem can be defined as follows. Consider design and performance spaces with n and m dimensions respectively. A design space point, $x = \{x_1, \ldots, x_n\}$, has an associated performance measure, $p(x) = \{p_1(x), \ldots, p_m(x)\}$. A point x_1 *dominates* another point x_2 (we write $x_1 \succ x_2$) if and only if

$$\forall_{1 \leq i \leq m} \left[p_i(x_2) \leq p_i(x_1) \right]$$

Note that the domination relation is a partial ordering, i.e., it is possible (and common) that neither $x_1 \succ x_2$ nor $x_2 \succ x_1$ holds.

The *Pareto frontier* is the set of all undominated design space points. Assume that the design 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.

$$R = \{x_i \in V | (\nexists x_j \in V)[x_j \succ x_i]\} \ .$$

Figure 3.3 illustrates the Pareto Frontier concept in the context of the bridge design study. At the top, the set of possible designs is sampled evenly on the design space, which, in this study, is two-dimensional. Each point on

the design space corresponds to a design, i.e., a placement of the two supports. Notice that the diagonal part of the design space is unexplored: we do not permit the two supports to be placed too close to each other. The corresponding performance metrics are plotted at the bottom on the performance space, which, in this study, is also two-dimensional. The simulation code can thus be thought of as a function that maps a point on the design space to a point on the performance space. The Pareto Frontier is highlighted in red circles on both the design space and performance space. Notice that, on the performance space, the Pareto Frontier is characterized as the North-East boundary of the cloud of points. This is a characteristic of Pareto-optimal point: if a design is on the Pareto Frontier, there is no other design that can perform better in both performance metrics, i.e., mapped to the point's North-East region.

3.1.4 Requirements and Challenges

Even though this bridge MES is a simplified version, the user requirements pose challenges to the parallel system design. These challenges are representative of the larger real-world MESs.

The first challenge is the sheer volume of computation. In order to achieve the target resolution of the design space (320x320) using brute force parameter sweep, over 100K simulations are required. At 7 seconds per simulation, close to 200 hours of CPU time will be needed. For a research group running a small 128 processor cluster, the time-to-result is over 90 minutes, which would too long if the group wishes to conduct MESs at an interactive rate. So, an exhaustive

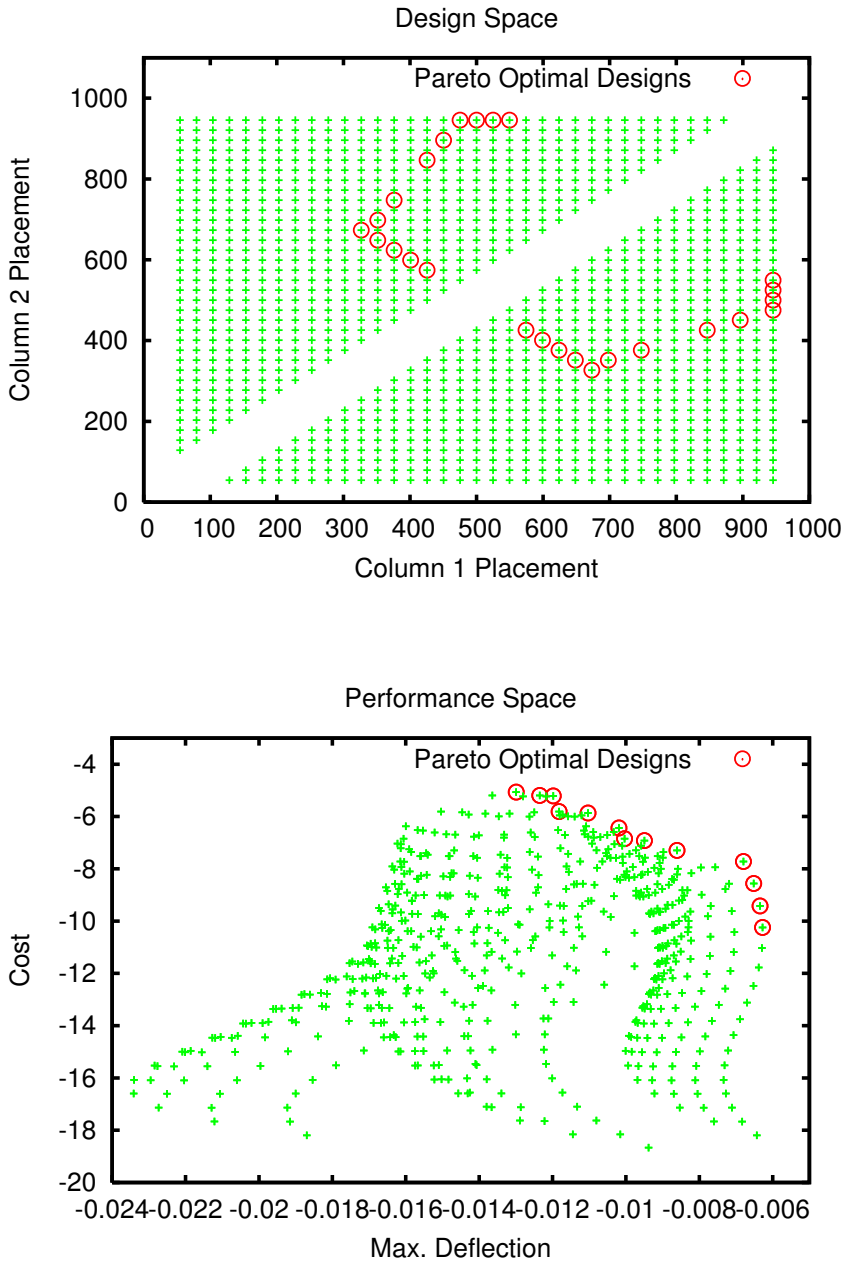


Figure 3.3: Pareto optimization in the 2D bridge design study. Pareto-optimal points are circled in red.

search of the design space is no longer an option. The challenge for the runtime system is how to schedule and run the simulation experiments so as to minimize the required computation.

The second problem is providing the user with study-level feedback. Even though the MES takes 100K simulations to complete, at any point in time during the study, by examining only the results of the simulations that are already completed, it should be possible for the system to provide the user with an incomplete or imprecise, but still accurate snapshot of the approximation of Pareto Frontier, so the user has the option to make some initial decisions before the entire MES is completed. In order to produce this incomplete snapshot, the system needs to be able to aggregate the result of simulation experiments dynamically, rather than of doing it all at once after all simulations have completed. To ensure this snapshot provides the most useful information to the user, the system needs to order the simulation experiments so that at any point in time, looking at the completed experiments can yield maximum information.

3.2 Defibrillator Design

The second example study is from the domain of medical device design. In the field of medicine, while deciding a course of treatment for a patient, there are many variables in the treatment plan that can be controlled, each affecting the outcome of the treatment. The treatments may have many conflicting goals as well, so much work needs to be done to ensure the treatment plan can meet these conflicting goals. For example, in using radiotherapy to treat a cancer

patient, the dosage, angle of radio rays, time of exposure, etc., can all affect the outcome of the treatment. The ideal treatment plan should maximize energy delivered to the cancerous tissues but minimize damage to healthy tissues, even though the two goals are often conflicting. Similarly, an implantable defibrillator device has many design variables — placement of the electrodes, shape of the electrodes, voltage of the defibrillation shock, duration of the shock, the shape of voltage curve, etc. It also has conflicting goals: activating as much of the heart tissue as possible, while damaging as little tissue as possible. There can also be additional goals such as the ease of implantation and cost of manufacturing the device.

This second MES is a simplified version of the defibrillator device design problem. In this simplified version, we limit the design choices to only decide the placement of electrodes and strength of voltage, and we limit the study's goals to three dimensions: to maximize activation, minimize damage, and maximize voltage uniformity. Due to the complex design space, new techniques are required in order to allow the user to *interactively* explore the design space. This study represents an interactive Pareto optimization problem.

3.2.1 Simulation software

This study uses a bio-electrical simulation software called DefibSim ([72, 71]) from the Scientific Imaging Institute at University of Utah to simulate the effect of a defibrillation shock on the human torso. DefibSim takes, as its inputs, 1) a 3D mesh representing the conductivity of the human torso ([53]), 2) the

positions of two electrodes, one placed in the front and one at the back of the torso, and 3) the voltage potential difference between the electrodes. It can then calculate the electric potential generated by the defibrillation shock inside the torso mesh. Figure 2.4 from Chapter 2 was generated from the DefibSim application.

Mathematically, the simplified version of the DefibSim problem is governed by the Poisson equation relating the local conductivity tensor, the voltage over the domain, and the current source:

$$\nabla.\sigma\nabla\Phi = -I_v$$

where σ is the local conductivity tensor, Φ is the voltage over the domain, and $-I_v$ is the current source inside the domain. For the defibrillator simulation, the current source is set to zero.

The discrete form of the equation approximates the divergence of the electric field with the stiffness matrix A and the voltages at the torso mesh nodes with the vector Φ. They yield a zero current source on the RHS:

$$A\Phi = 0$$

The electrode at the back and front are modeled as Dirichlet Boundary conditions. To add the boundary conditions, the stiffness matrix and a RHS are re-computed to eliminate the nodes with known potentials from Φ.

$$A'\Phi' = b$$

Here A' is the new stiffness matrix, b is the new RHS, and Φ' is the set of unknow electric potentials in the torso.

For each electrode placement, DefibSim solves three systems of the form $A'\Phi' = b$. The three systems correspond to setting the boundary condition of the front electrode at each of the three surface mesh nodes closest to it. The code then solves the three systems, and takes the distance-weighted combination of the three Φ's to produce the weighted average torso voltage potential $\hat{\Phi}$ for that simulation. Every time the user alters the position or strength of the electrodes, the new set of stiffness matrices A' and RHS b are re-calculated, and the new systems are solved and the new $\hat{\Phi}$ re-calculated.

Once a $\hat{\Phi}$ is re-calculated, the DefibSim can determine the "goodness" of the configuration by extracting the following information:

- Damage level, defined as the percentage of heart tissue that experiences a current flow above a pre-specified threshold (called the damage threshold),
- Activation level, defined as the percentage of heart tissue that experiences a current flow above a pre-specified threshold (called the activation threshold), and
- Uniformity, defined as the ratio of the maximum voltage gradient found in the heart to the average voltage gradient in the heart.

These performance metrics are used by the study to evaluate how desirable a given configuration is. Note that these performance metrics depend not only

on $\hat{\Phi}$; they also depend on the damage threshold and activation threshold.

An evaluation code is used to calculate these three performance metrics. Every time a new $\hat{\Phi}$ is calculated, or a new damage threshold is specified, or a new activation threshold is specified, the evaluation code calculates the differential of $\hat{\Phi}$, i.e., the electric potential gradient of the torso from the voltage shock. From $D(\hat{\Phi})$, the current flow through the heart is then extrapolated, and the uniformity, activation level, and damage level can be calculated.

DefibSim is implemented on the interactive Problem Solving Environment SCIRun. The SCIRun workflow net shown in the previous chapter, Figure 2.2, is an annotated SCIRun workflow net of the DefibSim application. It is broken into four parts: the user interface subnet, the simulation subnet, the visualization subnet, and the evaluation subnet. In the user interface subnet, the user has the ability to dynamically change any of the design parameters: placements of the front and back electrode, as well as the voltage strength. A change made by the user results in new data being sent down the data links. The simulation subnet responds to new data by re-calculating the torso's potential based on the user's input. The new torso potential is sent to both the visualization subnet and the evaluation subnet. The visualization subnet uses SCIRun's built-in visualization macros to show the torso potential to the user (Figure 2.4). This torso potential is also fed into the evaluation subnet. In the evaluation subnet, the user specifies the activation threshold and damage threshold. Every time the user changes them, or when a new torso potential has been calculated, the evaluation subnet will re-calculate the the current flowing through the heart.

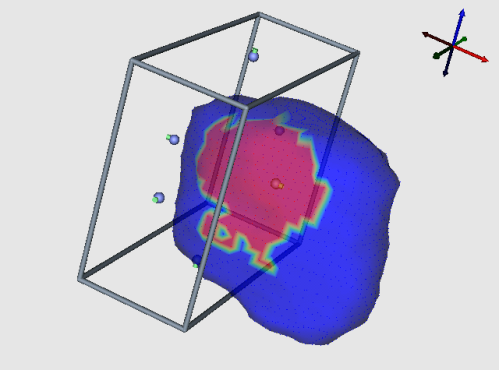


Figure 3.4: Visualizing individual simulations: this screen shows the region of the heart (colored in red) damaged by a particular defibrillator design.

Regions of the heart that are damaged or activated are then displayed to the user (Figure 3.4). DefibSim runs relatively fast — it typically takes less than two seconds to respond to a user action. There are 9000 nodes in the torso model, so the simulation result (the torso's potential) is about 60KB.

3.2.2 Study objectives

The objective of the study is to find the optimal placements of electrodes and voltage strength that will satisfy multiple conflicting criteria: maximize activation level, minimize damage level, and maximize uniformity. A high voltage shock close to highly conductive organs can increase activation, but also increase damage, and vice versa.

Thus, like the bridge design study, the defibrillator MES is a multi-objective optimization study. However, there are two characteristics that make the defibrillator design study different from the bridge design study. First, the

design space is five-dimensional (two dimensions for each electrode placement, plus one dimension for the voltage strength), which is more difficult to explore in a completely automatic fashion. Second, the presence of the activation threshold and damage threshold means that the behavior of the evaluation code is dependent on these *performance metric parameters* — input parameters not to the simulation code, but to the evaluation code. That means that every combination of these performance metric parameters defines a new evaluation criteria, and thus essentially defines an entirely new study.

In order to cope with those added complexity, user *interactivity* is required.

In an interactive setting, the user would fix the performance metric parameter (the activation and damage thresholds), fix some of the study's design space dimensions (e.g., the voltage strength, the placement of the back electrode), and identify a region for exploration for the remaining design space dimensions (for example, an area in the front of the chest for the placement of the front electrodes). The system will show him the Pareto frontier at the region of exploration in the given fixed setting (e.g., optimal placement of the front electrode). The user can then *interactively explore* the design space by changing either the performance metric parameters, the value of the fixed design space parameters, or the region of the remaining design space parameters to examine (e.g., he may increase the activation threshold to model a patient with a more difficult-to-activate heart, or increase the defibrillation shock voltage, or examine the region near the diaphragm instead of the front of the chest). The system should then respond to the action by showing him the new Pareto

frontier interactively.

As discussed above, each time the user fixes the value of the performance metric parameters, he is essentially defining a new study. Also, each time he fixes some of the design space parameters, he is actually selecting a subspace, or slice, of the design space, resulting in a Pareto optimization problem with a reduced design space dimension. Thus, each user action essentially defines a non-interactive, mini-study, whose Pareto Frontier needs to be calculated and displayed to the user. Each mini-MES corresponds to a *unit of interaction* — each user action defines a mini-MES, and causes a new Pareto Frontier to be calculated and displayed back to the user.

Figure 3.5 shows a series of Pareto Frontiers as a result of the user increasing the activation threshold. As the user increases the activation threshold voltage (bottom zoomed-in view, from left to right), the system responds by discovering and displaying the Pareto optimal from electrode placements interactively. Figure 3.6 shows two Pareto Frontiers as a result of the user changing the placement of the back electrode. Note that each Pareto Frontier is caused by a *single* user action, but each Pareto Frontier is the aggregated results of *multiple* executions of DefibSim. This is what defines an Interactive Multi-Experiment Study: a single user action (e.g., increasing activation threshold, moving back electrode) causes multiple executions of the simulation code, and it is the aggregated result of those executions (Pareto Frontier) that is presented to the user in an interactive fashion.

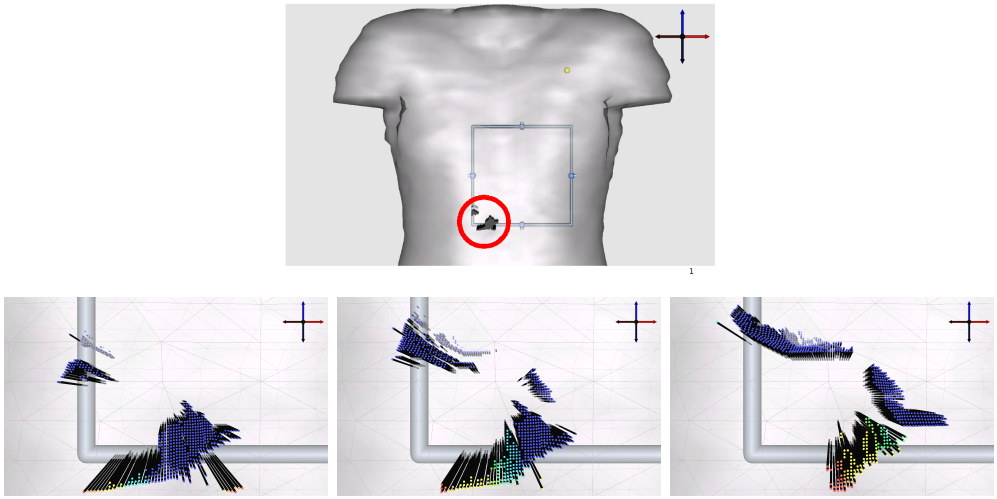


Figure 3.5: Interactive Design Space Exploration

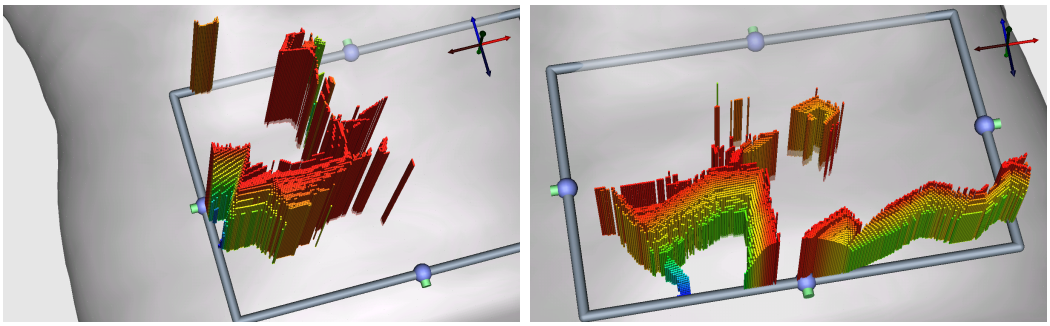


Figure 3.6: Pareto optimal points before (left) and after (right) the back electrode is moved.

3.2.3 Requirements and Challenges

The interactive nature of the defibillator MES poses additional requirements and challenges to the system. Firstly, there is again the sheer amount of computation. To resolve the Pareto Frontier for each study at the desired resolution (256x256) using brute force, 65K simulations are needed. At 2 seconds per simulation, it would take more than 36 hours on one CPU to complete a study. Even with 128 processors, that would take 17 minutes. So, every time the user performs a user action, he would need to wait 17 minutes before the system provides him with feedback, which does not meet the interactive requirement.

Secondly, the system needs to present a user interface with which the user can steer the MES at the study level. In other words, at any point in time during a study, the system must be able to aggregate the results from completed simulations and present a coherent representation of the study up to that point to the user. In addition, the system must be able to receive user's action from the UI, interpret those actions as study-level steering actions, decide how that action affects the subsequent conduction of the study, and disseminate the changes to individual simulations.

Finally, even though the user is interested in study-level results, he would still require to look at individual simulation results (e.g., he might want to find out, for one particular defibrillator configuration, which part of the heart it activates). The system is thus required to store the individual simulation results in a way that can be easily called up for visualization, even though individual simulations may be conducted in distributed computational elements.

3.3 Animation design

The third computational study comes from animation design. To create more physically-realistic animated scenes, animators often use physics simulation engines to help them design the scenes (e.g., [9]). The resulting problem often takes the form of an MES.

For example, in order to animate a scene where a piece of cloth being carried by the wind is caught by a tree branch (an unlikely occurrence), the animator may run a fluid dynamics simulation many times, and each time, the animator would change the settings of the simulation — the viscosity of air, wind speed, shape of the cloth, etc., — in order to produce a scenario that achieves the unlikely result of the cloth getting caught. The resulting scene would be physically realistic (because the simulation is based on physics), but also achieves an unlikely occurrence.

In this type of MES, the user is exploring the design space not to optimize for some metrics, but rather to search for a region in the space that meets certain requirements. In addition, the animator could discover multiple versions of the scene where the physical objective is achieved. In those cases, in order to choose from among the multiple version of the scene, the animator would need to rely on his aesthetic tastes and judgements. Therefore, this type of MES has a secondary goal: to help the user explore the design space, guided by his aesthetic tastes.

In this thesis, we focus on an example animation study. In this study, the animator uses a rigid-body simulation to help design a physically-realistic

scene where a single buggy and some pieces of debris are allowed to roll down an unevenly-shaped terrain. The terrain is shown in Figure 3.7. At the start of the simulation, the buggy is placed at the top of the slope, and it rolls down the slope in a straight line as the simulation progresses. If the number of debris pieces is large, the buggy is unlikely to make it to the bottom of the slope without being knocked off its course by one of the debris pieces. The goal of the animator in this study is to find the initial placement and orientation of all of the debris pieces in such a way that would achieve the unlikely outcome of having none of the pieces come into contact with the buggy. As a secondary objective, he would also want these debris pieces to miss the buggy in some visually-interesting manner, e.g., come into close proximity of the buggy, pass by it in with high velocity, etc., but never making contact with it.

3.3.1 Simulation software

The simulation is implemented using the Open Dynamics Engine (ODE) [74]. The buggy and the debris are represented as *rigid bodies* under the influence of gravity and collision. The buggy is represented by a rectangular box with four spherical wheels, and the debris are simple geometric objects: spheres, cylinders, and rectangular columns. At each timestep, data structures that restrict the movement of rigid bodies (called joints) are formed dynamically to account for collisions between rigid bodies. During each timestep, the linear and torsional accelerations of each rigid body is calculated, based on the influence of joints and gravity. Then the timestep advances by updating the objects' velocities

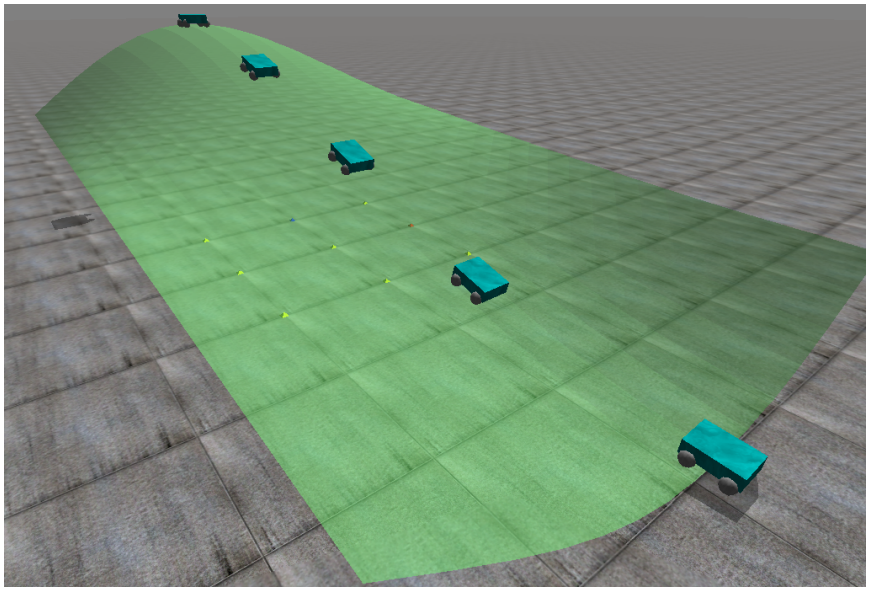


Figure 3.7: A buggy rolls down a terrain in the animation design study.

and positions. The simulation terminates when a piece of debris hits the buggy, or when the buggy reaches the lowest point of the terrain.

In the particular implementation used in this MES, 15 pieces of debris are used: five spheres, five cylinders, and five rectangular columns.

ODE is a real-time physics simulation engine, so the simulation is relatively fast — it takes at most four seconds to complete a simulation with UI enabled, and with UI disabled (as it will be when the code is run on the cluster nodes), it takes on average half a second to run on a sequential processor.

3.3.2 Study objectives

The objectives of this MES are relatively simple. The primary goal is to find initial placements of the debris field such that the buggy can reach the lowest point of the terrain without coming into contact with any of the debris pieces. In order to keep track of whether debris pieces come into contact with the buggy, each debris piece carries a history of its distance to the buggy. At each timestep, the debris' distance to the buggy is calculated and stored. An *admissible* simulation is one in which none of the debris pieces' distance-to-buggy history contains a zero.

A secondary goal of the MES is for the user to choose, among the admissible simulations, the most aesthetically pleasing ones. This depends on a user interface that constantly receives user input to determine which simulations are more aesthetically pleasing.

3.3.3 Requirements and Challenges

The animation MES is different from either of the Pareto Optimization MESs described earlier, and presents unique challenges. Firstly, this study has a large number of design space dimensions. The user needs to decide, for each debris piece, its initial position (3 dimensions), orientation (3 dimensions), and time of release (1 dimension), so there are seven design dimensions for each debris piece. With fifteen debris pieces total in the study, there are 105 design space dimensions.

Secondly, the performance metrics are ill-defined — they depend on the animator's subjective taste in what constitutes an interesting animated scene. One animator may consider debris pieces missing the buggy by small margins to be visually appealing, while another animator may enjoy debris missing the buggy by larger margins, but travelling at faster velocities. Another animator might tolerate larger missing margins and lower travelling velocities, but like that the debris pieces fall more evenly around the buggy to create a sense of the buggy being surrounded. However, there are common preferences adopted by all animators, i.e., that all debris must come within a certain distance of the buggy at some point during the scene, and that the debris must not touch the buggy.

As a result, this kind of MES requires an animator to drive this study. The system needs to provide the user with constant updates on the progress of the study, and the user needs to provide the system constant steering action so as to guide the study in the direction he wishes to go.

One possible way of framing a user-driven MES is by treating the MES as an exploration of a search tree. In this study, the fifteen debris are grouped into five groups of three, with one sphere, one cylinder, and one rectangular block in each group. The study is conducted in stages, and in each stage, the user decides the placement of one debris group. In the first stage, the placement of the first group of debris is decided, and in the next stage, the placement of the next group is decided, and so on. At each stage, the user is shown a number of possible choices of the placements of the debris group to be decided, and he is allowed to look at the animation of the partially-completed scene of each of the possible choice. Out of these choices, he can select one, and that will complete the stage, and begin the next one. Figure 3.8 shows the state of the animated scene at various stages of a study.

This way of framing presents the MES as an exploration of a search tree, where each level represents a stage of the study, and each node represents a partially-completed scene. For a node on level l, l debris groups' placements are fixed. Its first $l-1$ debris groups have the same placement as its parent. The root node (level zero) has no debris placed, each leaf node (level 5) has all 15 debris placed, representing a completed scene.

The advantages of framing the MES as a tree-exploration is that there is a clearly-defined notion of a partial result of the study in the form of internal nodes of the search tree. In our case, these partial results correspond to partially-complete animation scenes, which can be visualized. So the user's guidance can come in the form of selecting a child-node for exploration. Essentially, the

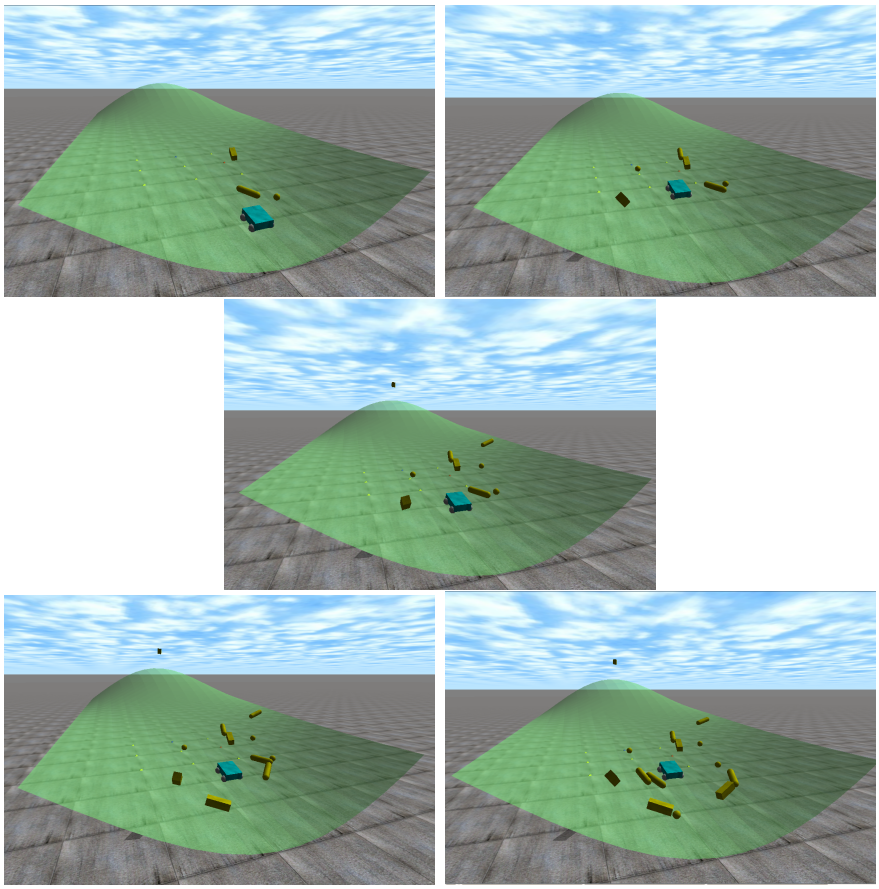


Figure 3.8: As the animation study progresses (left to right, top to bottom), more debris groups are added into the animated scene.

user is steering the system to explore the search tree. At any point during our MES, the animator is exploring a non-leaf node of the tree (the *current* node). The system would continuously provide him with new child nodes to the current node that correspond to admissible simulations. The animator can then animate the partially-completed scene corresponding to these child nodes, and choose to: 1) pick one of the child nodes and move the exploration to it, thus making it the current node, 2) wait for the system to discover more possible child nodes, or 3) move the exploration to an entirely different node altogether (back-tracking). When the animator conducts the study, he starts on the root node. The study terminates when the animator finds one or more satisfactory leave nodes.

From the system's point of view, these user-driven tree-exploration type MESs present additional challenges. Firstly, the user could, at any point in time of in the study, examine any of the explored nodes on the tree. So, the system must be ready to present the user with visualization of any of the partial results found thus far. Secondly, the users' heavy involvement means that the system must be responsive: it must constantly provide the user with new information as the exploration continues, and be ready to accept user inputs and change the direction of its subsequent explorations by disseminating the user's action to individual simulations. Finally, in addition to being able to provide the user with a visualization of partial results, the system must be able to constantly present the user with an updated version of the high-level view of the exploration, so that the user can make global as well as local decisions.

3.4 Helium Model Validation

The fourth computational study used as an example in this thesis is from the discipline of chemical engineering. Chemical engineers often rely on complex computational models to describe complex chemical and physical phenomenons. In cases where these phenomenon involve multiple models, the resulting coupled model need to be validated and verified as a whole.

On such example is pool fires, which involve combustion, turbulent air-mixing, heat transfer by radiation, and conventional fluid flow. A complex computational model was developed at the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) to simulate pool fires. The model is implemented in an application known as Arches ([75]), which is built using the Uintah Problem Solving Environment ([25]). Arches's components include a Navier Stokes solver to determine the gases' pressure and velocity, a chemical reaction module to model changes in chemical species of the gaseous mixture, a heat transfer module to model the distribution of temperature, and a turbulence solver to model turbulent gas flow.

Software like Arches contain many "knobs" that the user can control to produce a different outcome. These include model parameters, such as the Prandtl number, the turbulent mixing model, or the Smagorinsky coefficient; simulation parameters, such as the resolution of the simulated domain, the solver used in timestepping, and if the solver is an iterative one, its residual tolerance; and experiment parameters, such as the gaseous fuel's gas velocity, or the size of the inlet. The type of numerical models used, the numerical method used,

the model parameters can all affect the accuracy of the simulation.

In order to ensure that a model can accurately describe the physical phenomenon, the scientists are often required to run the software multiple times, each time using a different model/numerical method/model parameter combination, and compare the simulated results against the results obtained in real-life experiments. Their goal is to find, if possible, the set of inputs that matches most closely with real-life experiment results. This process is known as model Verification and Validation ([64]).

3.4.1 Simulation software

For this thesis, we look at one piece of the validation problem on Arches. As part of pool fire simulation, Arches needs to simulate the turbulent mixing of gases of different densities (e.g., helium and air). To validate the turbulent mixing model, Arches can be configured to perform simulation experiments where it only simulates helium gas mixing with air in a controlled environment. To validate the gas flow model, the experiment is replicated in real-life, and the real-life experiment results are then compared to the simulated experiment results ([28]).

In the experiment, helium is pumped into a cube-shaped container from an inlet at the bottom of the container. Due to the buoyancy of helium and the velocity with which the helium gas is pumped into the container, the simulated domain undergoes a "puffing" motion (Figure 3.9). Due to the "puffing" motion, the gas at the central part of the simulation domain is pushed upwards, resulting

in a velocity profile like Figure 3.10. The centerline gas velocity profiles are then measured at two heights. These velocity profiles are used to gauge the accuracy of Arches simulations.

Arches simulates this phenomenon in a three-dimensional domain using a numerical technique called Large Eddy Simulation (LES). A brief explanation is given here; the detailed explanation, as well as the discretization scheme, is presented in [28] .

Mathematically, LES separates out the large eddies in the fluid from smaller-scale motions. Large-scale motion is modeled by the Navier Stokes equations, and small-scale motion is modeled by a turbulence model. A filtering operation is used to separate out the motions. A filtered variable, denoted by the overbar operator, is defined as

$$\overline{f} = \int_D f(x')G(x-x')dx'$$

where D is the entire domain and G is the filter function. Arches uses the sharp Fourier cut off filter:

$$G(x-x') = G(k) = \begin{cases} 1 & if k < \pi/\triangle \\ 0 & otherwise \end{cases}$$

where $\triangle$ is the grid size. This size determines the size and structure of small-scale motions that are eliminated from f and hence require separate modeling.

The filtered Navier-Stokes equations become:

$$\frac{\partial \overline{u}_i}{\partial x_i} = 0$$

$$\frac{\partial \overline{u}_i}{\partial t} + \frac{\partial(\overline{u}_i\overline{u}_j)}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_i} - \frac{\partial \tau_{ij}}{\partial x_j} + v\frac{\partial^2 \overline{u}_i}{\partial x_i \partial x_j}$$

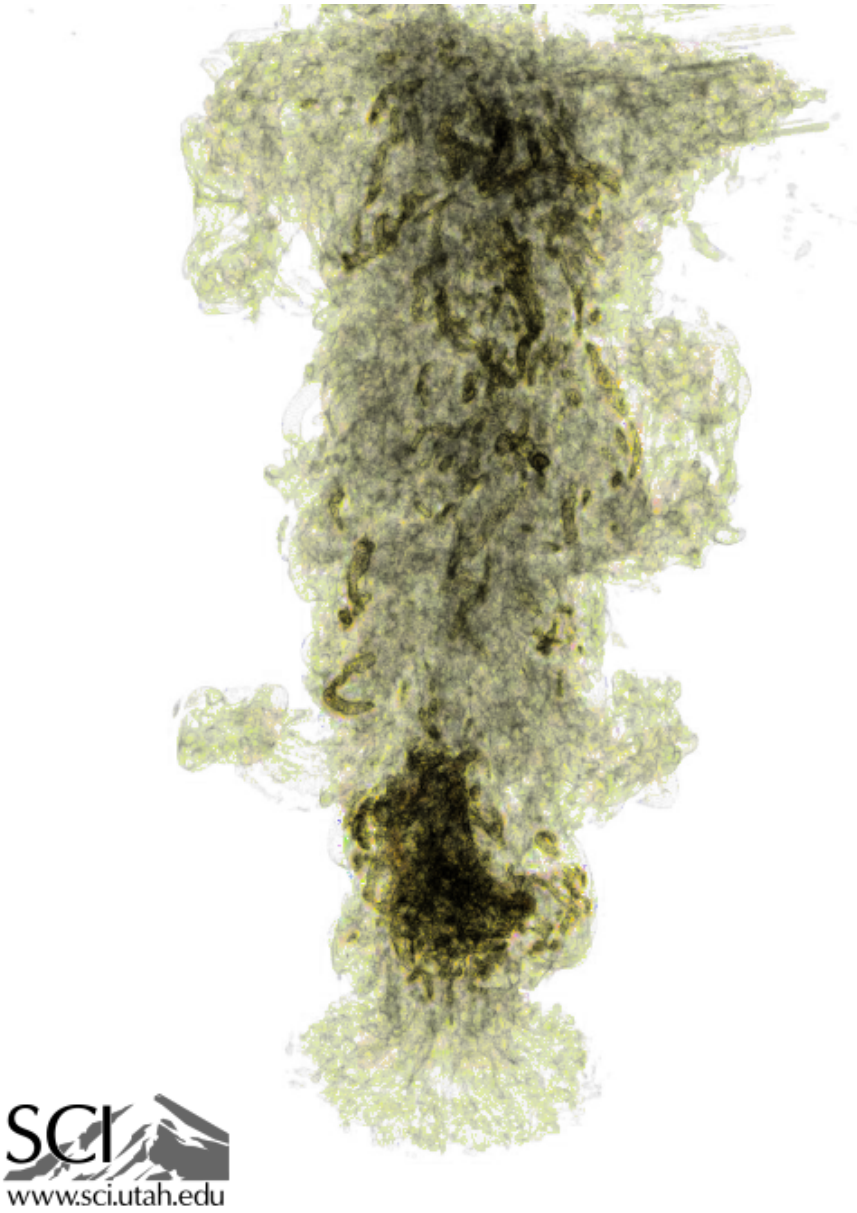


Figure 3.9: Visualizing the helium plume. Image courtesy of Chemical Reaction Simulation group, University of Utah

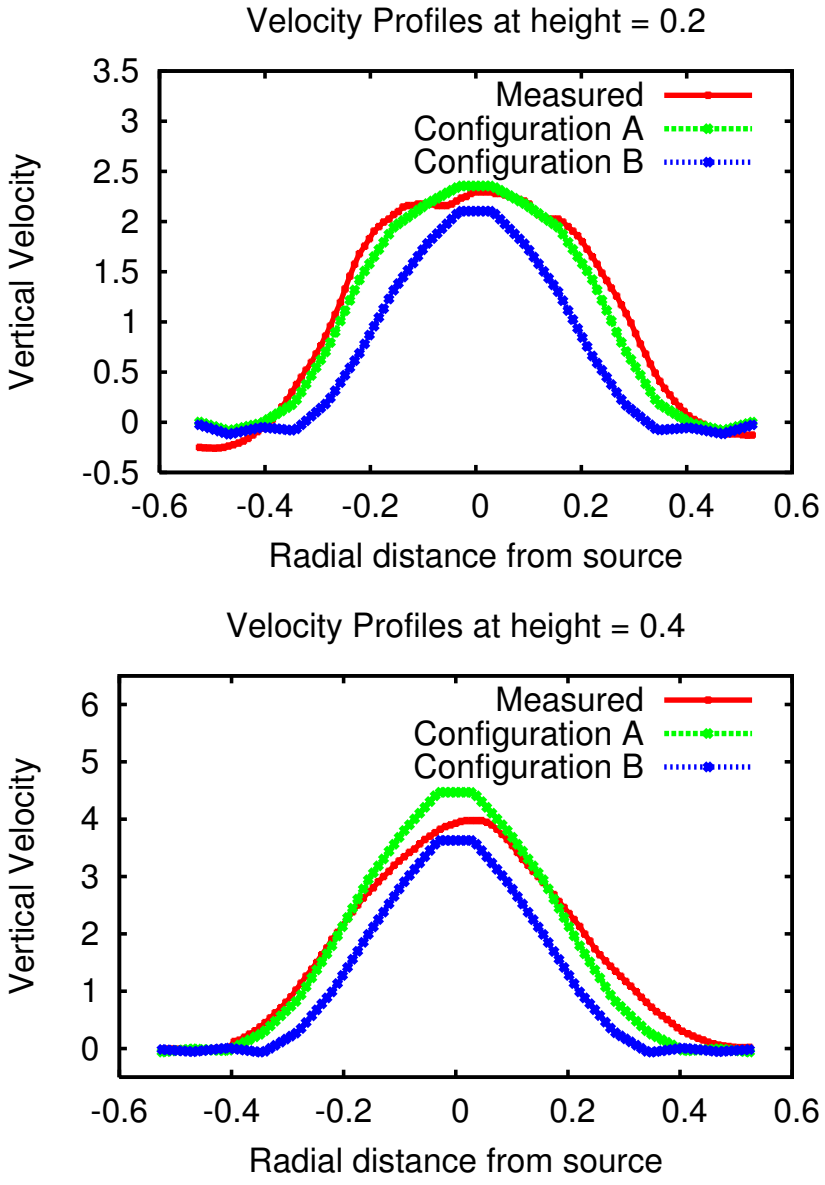


Figure 3.10: Comparing the velocity profiles of two input configurations at two heights.

The term τ_{ij} is the subgrid scale stress (SGS) tensor that represents the small-scale motions that are filtered out by the filter operator. In this thesis, the Smagorinsky model is used to approximate the SGS. In the Smagorinsky model, τ_{ij} is given by:

$$\tau_{ij} = -2v_T\overline{S}_{ij} + \frac{\delta_{ij}}{3}\tau_{kk}$$

where $\tau_{kk} = \overline{u'_k u'_k}$, $\overline{S}_{ij}$ is

$$\overline{S}_{ij} = \frac{1}{2}\left(\frac{\partial \overline{u}_i}{\partial x_j} + \frac{\partial \overline{u}_j}{\partial x_i}\right)$$

and v_T is

$$v_T = (C_s\triangle)^2\left(2\overline{S}_{ij}\overline{S}_{ij}\right)^{\frac{1}{2}}$$

C_s is the Smagorinsky coefficient, and is one of the inputs to the Arches simulation code.

The initial condition has the domain filled with air at zero velocity. As the simulation progresses, the helium inlet acts as a boundary condition with helium gas flowing in at constant speed, and the Navier-Stokes equations are solved to find the pressure and velocity in the rest of the domain. The turbulence model is then applied to the domain to adjust for the SGS. Finally, the gaseous mixture at the domain is updated, so the buoyancy forces can be applied to the next timestep's Navier Stokes solve. The timestepping is then repeated. Arches writes checkpoints (the pressure, velocity, and gaseous mixture) to disk periodically. The total kinetic energy of the system is monitored at each timestep, and the timestepping stops when the KE of the system has settled into a steady state.

Arches is a Uintah application, and like other Uintah applications, is built up out of components. Uintah components perform common operations, such as mathematical operations like Navier-Stokes solves, chemical reactions, heat transfers, turbulence models; and system operations such as writing checkpoints and load balancing. By connecting the outputs of components to inputs to other components, a user of Uintah can compose workflows that perform his application. Uintah components are designed to run on parallel clusters using MPI, and use a shared file system to store the simulation result and checkpoints. Uintah applications proceed in timesteps. At each timestep, the application advances its simulation domain according to a workflow determined its components.

The time to execute an Arches run depends on the per-timestep run time and the number of timestep required to reach the steady state. More specifically, it depends on the input parameters of the particular run — simulation parameters like grid resolution and residual tolerance affect the per-timestep run time, and model and experiment parameters such as the Prandtl number or Smagorinsky coefficient affect the time it takes for the system to settle. A full version of Arches (200x200x200 resolution) could take tens of hours to run on a 512-processor cluster. However, in this thesis, a lower-resolution run (48x48x48) is used, which takes about half an hour to complete on a 32-processor cluster.

3.4.2 Study objectives

In the real-life experiment, the gas velocities along the centerline at two heights are measured at approximately 100 evenly-spaced points. To evaluate

how "good" an Arches run is, the simulated gas velocities at these heights in the simulated domain are interpolated to these points. Then, for each height, a vector difference between the measured and simulated velocities is taken. The goal of this study is for the user to find the model parameter of Arches so that the simulated results matches most closely with real-life results, i.e., to minimize the velocity profile difference at both heights. Figure 3.10 shows the centerline velocity profiles produced by running Arches twice, each time using a different set of input parameters (or configuration). The velocity profiles of the two configurations are overlaid on top of the measured velocity profile. Configuration A is clearly superior to Configuration B at height=0.2m, as its velocity profile is a better match for the measured profile, but at height=0.4m, the two configurations are more evenly-matched.

In the simplified form of the study, we only consider two input parameters: the Prandtl number and inlet velocity. This simplified version preserves the properties of the full version; the only difference is that there are fewer design parameters. The principles learned from designing the runtime system to support this simplified version are applicable to the full system.

Since the goal of the helium validation is to minimize the velocity profile difference on multiple heights simultaneously, it is a multi-objective optimization. It is also possible for input parameters that match the profile from one height well to match the profile on another height poorly, i.e., the two objectives can be conflicting. So, like the bridge study, it can be framed as a Pareto optimization study. Here, performance metrics are the velocity profile differences

at each height, and the design space dimensions consist of the Prandtl number and the inlet velocity.

Figure 3.11 shows the Pareto Frontier of the simplified validation study. The design space (top) is sampled on a regular grid, and the performance metric is plotted on the performance space (bottom). The Pareto optimal designs are circled in red in both plots.

3.4.3 Requirements and Challenges

Unlike the other MESs discussed thus far, the Arches validation study is special in that the simulation code is long-running (multiple-hours rather than seconds), but a relatively small number of simulations are needed (dozens, instead of thousands). Therefore, the goal is no longer interactivity, but rather the reduction of study completion time.

The simulations in Arches can be run in parallel. Therefore, in addition to the usual concerns in the MESs — how to schedule and order the experiments — there is an additional challenge of needing to decide the level of parallelism assigned to (i.e., the amount of resources allocated for) each experiment.

The system needs to use the scheduling and resource allocation decisions to reduce the overall run time of the MES. It needs to increase the utilization rate of computing resources — by not letting processor nodes sit idle waiting for an experiment to be assigned to it, and also by reducing the parallelism overhead within a single simulation. However, as will be seen, the parallelism requirement often intersects with other considerations like enabling early user

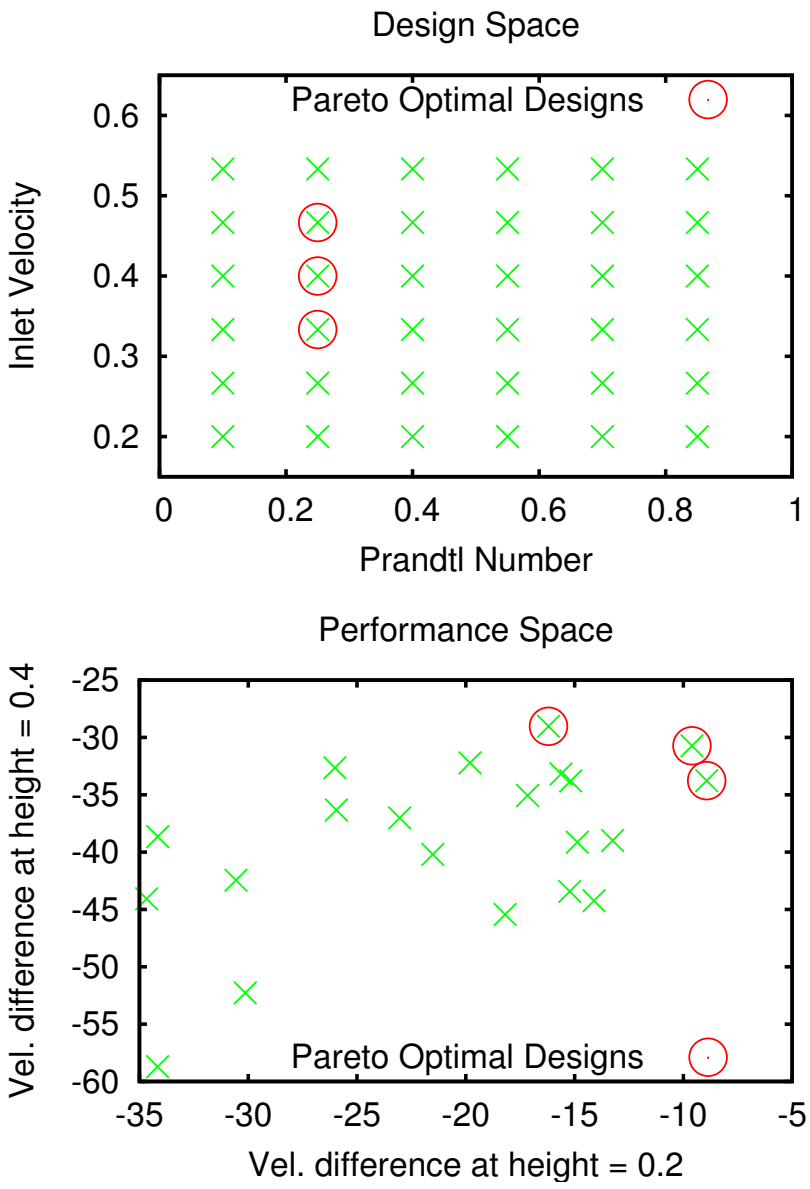


Figure 3.11: Helium model validation as Pareto Optimization.

feedback, therefore, the parallelism requirement is an additional dimension to the MES system.

3.5 Problem Formulation

All four examples used in this thesis fall into a category of Multi-Experiment Study known as Design Space Exploration (DSE). As should be clear from the descriptions above, DSEs from different disciplines share many common traits. In this section, we factor out the common traits from our four DSEs in order to formulate an abstract description of DSEs. With an abstract formulation of the DSE, it is then possible to design a generic parallel system to target conducting DSEs. This formulation will be adhered to in the rest of this thesis.

At the highest level, a DSE is an application that runs a simulation software many times, each time using a different input, in order to find, by measuring the output of the software, the subset of inputs that are somehow desirable. Formally, we can think of a DSE as consisting of the following:

Simulation Code: This is the code which takes a set of simulation parameters, runs a simulation, and returns a simulation result. The DSE runs this code multiple times, each time with different input parameters, in order to discover the "best" input parameters. In the context of the bridge study, the simulation code corresponds to the non-linear solver that calculates the bridge deformation. Its input parameters are the positions of the two columns. In the context of the defibrillator study, the simulation code corresponds to the portion of DefibSim that computes the torso potential. Its input parameters are

the coordinates of the front and back electrodes, as well as the voltage difference between them. In the context of the animation design study, it corresponds to the rigid-body simulation code that animates the scene. Its input parameters are the placements of the debris pieces. In the context of the helium model validation study, it corresponds to Arches. Its input parameters are the Prandtl number and inlet velocity.

Evaluation Code: This code takes the output of the simulation code, and optionally a set of performance metric parameters, and evaluates the performance metric for the simulation. The DSE uses this code to evaluate the "goodness" of the designs. In the context of the bridge study, the evaluation code corresponds to the code that evaluates the cost function and extracts the maximal displacement of the bridge. In the context of the defibrillator study, the performance metric evaluation code corresponds to the portion of DefibSim that computes the percentage of heart tissues that are above the activation threshold (activation level), the percentage of heart tissues that are above damage threshold (damage level), and calculates the uniformity of the potential gradient distribution. In the context of the animation design, the evaluation code is the code that extracts the information about the closest distance each piece of debris has come to the buggy. In the helium validation study, the evaluation code is a script that reads the Arches output and the real-life experiment data, and calculates the gas velocity vector difference at each height.

Design Space: This is the space of possible inputs to the simulation code. For the bridge design study, the design space is two dimensional, each

dimension corresponding to the placement of one support. In the context of the defibrillator study, the design space is a 5 dimensional space: back electrode placement (2 dimensions), voltage (1 dimension), and front electrode placement (2 dimensions). In the animation study, the design space corresponds to the 105 dimensions that determine the placement of all 15 pieces of debris. In the context of the helium mode validation study, the design space consists of the space spanned by the Prandtl number and inlet velocity.

Performance Space: This is the space of all possible outputs of the performance evaluation code. In the bridge design study, the performance space is spanned by the cost of construction and the maximal deformation of the bridge. In the context of the defibrillator study, the performance space is a 3-dimensional space, spanned by uniformity, damaged level, and activation level. In the animation study, the performance space is a 15-dimensional space, each dimension corresponding to the closest distance each debris piece has come to the buggy. In the helium model validation study, the performance space is a two dimensional space, each dimension corresponding to the centerline velocity vector difference on each height.

The simulation code/evaluation code pair can be thought of as a function that maps points from the design space to performance space.

Regions of interest: The ultimate goal of design space exploration is to identify points on the design space that are of interest, representing optimal design strategies, for example. In the bridge design, defibrillator design, and helium model validation studies, the Pareto Frontier represents the region of

interest. In addition to the Pareto Frontier, the user can specify admissible regions. Admissible regions are regions in the performance space which the user specifies *a priori* and that any design points must satisfy in order to be of interest. In the animation design study, the admissible region is used to filter out animation scenes where the buggy comes into contact with a debris (the debris's closest distance to the buggy is 0), and where there is a debris piece that does not come closely enough to the buggy for the scene to be interesting.

In the most abstract terms, the goal of a DSE is to discover points from the design space that will get "mapped" by the simulation and evaluation codes to the regions of interest.

Note that this DSE formulation is applicable not just to the four examples motivating this thesis. In fact, DSEs adhering to this formulation, as a type of computational study, are used in many contexts, including, but not limited to, medical treatment planning ([5]), medical device design ([71, 85]), automotive design ([73]), chemical engineering ([28]), processor design ([44]), finance ([68, 67, 31]), drug design ([16]), computer graphics ([43]) and animation ([81]).

3.6 Summary

Table 3.1 summarizes the properties of the four computational studies used in this thesis. Each of these characteristics pose its own set of challenges to the system.

Two of the studies are interactive, i.e., provides constant user feedback at interactive speed and can respond to user input while the study is on-going.

	Bridge Design	Defibrillator Design	Animation Design	Helium Model Validation
Interactive?	No	Yes	Yes	No
Design Space Dimension	2	5	105	2
Performance Space Dimension	2	3	15	2
Parallel Simulation?	No	No	No	Yes
Time of 1 simulation on 1 proc	7 sec	2 sec	< 1 sec	6.5 hours
Region of Interest	Pareto Frontier	Pareto Frontier	Subjective measure	Pareto Frontier

Table 3.1: Summary of multi-experiment computational studies discussed in this thesis.

Studies that are interactive require constant user input and feedback, so the system must have a way to aggregate the simulation results as well as disseminate user actions to individual simulations.

Three of the studies are Pareto optimizations. As was shown above, it is impractical for the system to resolve the Pareto Frontier by brute force parameter sweep. Instead, the system must schedule the simulation runs intelligently to discover the Pareto frontier with fewer experiments. Additionally, the system should order the runs so as to maximize the amount of information available to the user when he looks at the results of a partially-completed study.

One of the studies has an ill-defined, subjective region of interest. These studies require the system be able to constantly update the user on the progress of the study, and to provide the user with a readily available visualization of any of the partial results of the study. It also must respond quickly to user's actions — the user's actions needs to produce an immediate effect on the study's direction.

One of the studies is made up of simulations that can itself be run on parallel. Studies like this pose an additional concerns to the system, namely, that the system needs to decide not only the scheduling, but also the resources allocated to each simulation experiment. The system needs to minimize parallelization overhead within each experiment, but at the same time improve processor nodes' utilization rate.

On the most abstract level, for all of the studies, the main goal of the system is to intelligently make scheduling, resource allocation, and user interaction

decisions that reduce the users' wait time from user action to useful information.

Chapter 4

SimX Testbed

System Software for Interactive Multi-Experiment Computational Studies (SIMECS, or SimX for short) ([89, 90, 87, 88]) is a system framework designed as a test bed to explore the techniques a parallel system can use to conduct Multi-Experiment Studies on parallel clusters. This chapter provides the architectural details as well as the implementation details of the SimX system.

As discussed in Chapter 3, an MES can be described in the abstract as a collection of simulation and evaluation code executions that search for regions of interests within the design and performance spaces that meet the study's objectives. Since these executions are independent of each other, they can be conducted in parallel, so parallel platforms are a natural choice for running MESs. However, as discussed in Chapter 2, parallel systems that manage collections of simulation runs on a parallel machine so as to make progress toward overall study goals have not been extensively studied yet, and with the growing availability of parallel machines, the need for such a system is apparent.

SimX is such a system. SimX targets the MESs that fit the formulation described in Section 3.5. SimX is structured in a way that allows easy imple-

mentation of the techniques described in later chapters aimed at addressing the requirements of MESs, as represented by the four examples discussed in this thesis. In particular, SimX's functionalities are accomplished by a set of different modules running on different processes. Each of these modules is designed to achieve a different functionality: interacting with the user, choosing the experiments from a design space, scheduling the chosen experiments, assigning computational resources to the experiments, running the experiments, storing and retrieving reusable information across experiments, communicating between processes, and aggregating experiment results. As will be seen in later chapters, each module can then be independently adapted to serve the particular type of MESs being conducted.

4.1 Architecture

An architectural overview of SimX is presented in Figure 4.1. It shows three types of processes. The manager process runs on the front-end of the cluster and interacts with the user. The worker processes run on the compute nodes of the cluster and run the actual simulation code. The Spatially-Indexed Shared Object Layer (SISOL) servers run on the compute nodes and facilitate the sharing of data between worker processes.

In a typical SimX run, after the user receives an allocation of a partition of the cluster, one or more allocated nodes are selected to run the SISOL servers. The rest of the nodes run the worker processes. The user then starts the manager process on the front-end machine.

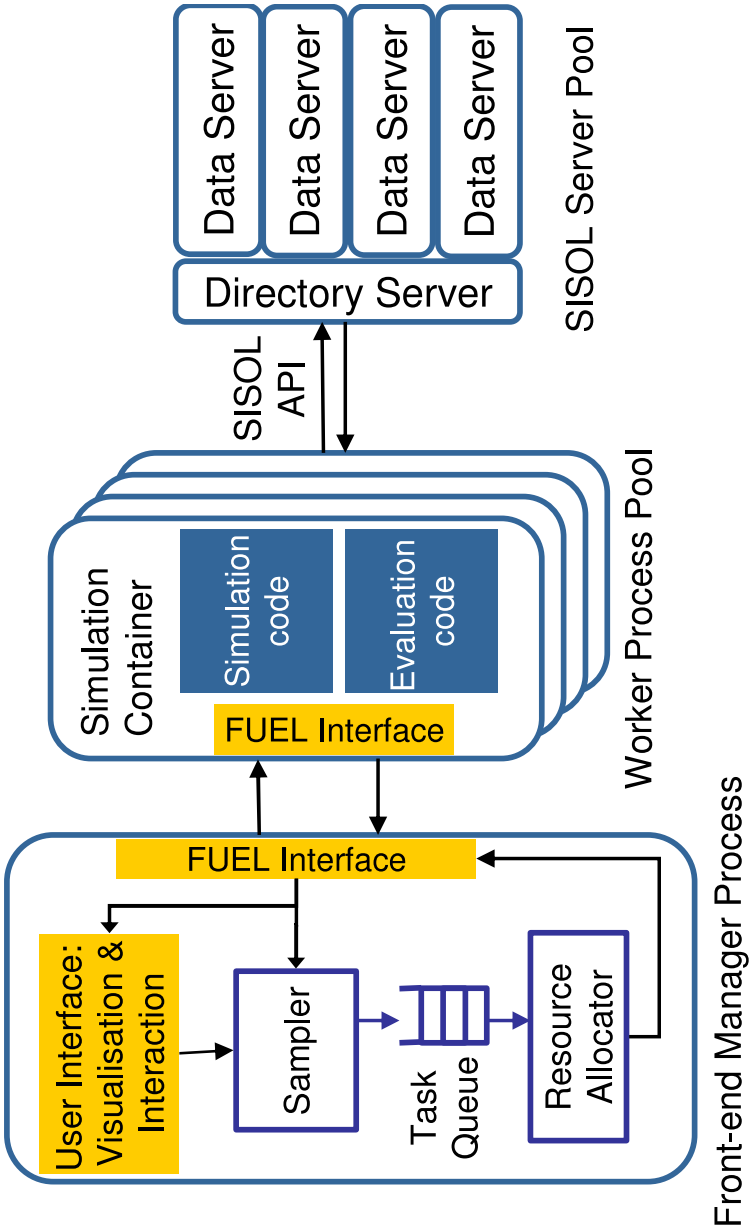


Figure 4.1: SimX architecture

The user employs the **UI module** on the manager to specify the computational study he wants to perform: how many design space dimensions there are, where is the admissible region, what is the range of exploration, and what is the region of interest. In addition, if the study is interactive, he would also use the UI to update the slice of design space (for interactive Pareto explorations) or current search-tree node (for tree-exploration MESs) he wishes to explore while the study is still going on.

Once the UI module receives information about the computational study the user wishes to perform, SimX needs to decide which simulations need to be run in order to complete the study. The **Sampler** module translates the UI module's study specification into a list of simulations, or computational experiments, that needs to be executed in order to explore the design space.

The Sampler module puts the list of computational experiments it issues onto the **Task Queue**. The order of execution of the issued experiments is decided there. Depending on the study, the task queue could be a FIFO queue or a priority queue.

The **Resource Allocator** module takes the contents of the task queue and assigns the tasks to available worker processes. It decides which worker process runs which experiment. If the simulation code is parallel, it also decides how many worker processes should run a given experiment, and coordinates the selected worker processes so that they can form into a worker process group.

Communication between the manager process and worker processes is handled by the **Frame/Update Exchange Layer** (FUEL). FUEL is an ex-

tensible communication protocol written on top of TCP/IP. Marshalling and unmarshalling in FUEL is done by handlers — pieces of associated code that gets executed when a piece of information needs to be sent or received. The user can adapt FUEL to his computational study by implementing the handlers.

The **Simulation Container** module running on the worker process is responsible for receiving the experiment sent from the Resource Allocator via FUEL. Once it receives the experiment, it acts as the substrate on which the simulation and evaluation code is run. It coordinates the running of the simulation and evaluation code, and optionally performs optimization routines based on the computational study. It is also responsible for sending the performance metric back to the manager process via FUEL when the simulation is done.

As the manager process receives simulation results, they are sent to the UI to update the user of the progress of the study, as well as to the Sampler module so it may adjust the contents of the Task Queue.

As will be seen in later chapters, capturing and sharing information from past experiments between worker processes is a key technique essential to meet the study requirements. The **Spatially Indexed Shared Object Layer** (SISOL) provides implicit communication between simulation workers and the manager process. Objects are stored in SISOL as *object sets*, with objects identified as an <object set ID, spatial index> tuple. Almost all the objects stored in SISOL are indexed by their design space coordinates, and often a nearest-neighbor query is needed. Therefore a spatially-indexed database is the interface of choice.

	Manager Process	Worker Processes
Bridge Design	Standalone	Standalone
Defibrillator Design	SimX/SCIRun	SimX/SCIRun
Animation Design	SimX/SCIRun	Standalone
Helium Model Validation	SimX/SCIRun	SimX/Uintah

Table 4.1: Versions of SimX in the example Multi-Experiment Studies.

4.2 Implementation

As part of this thesis research, three implementations of SimX have been developed: a standalone version, where SimX is packaged as a set of shared libraries, a SimX/SCIRun version, where SimX is packaged as a set of SCIRun modules, and a SimX/Uintah version, where SimX is packaged as modified Uintah components. Some modules are implemented the same way across different versions, but others have version-specific interfaces. In all versions, SimX is implemented in C and C++.

The inter-process communication protocols (FUEL and SISOL) are the same across different versions, therefore it is possible to use one SimX version's manager process and another version SimX version's worker process, and both can talk to each other and to the same SISOL servers. The SimX/Uintah version, for example, does not have a manager process, which means the user will use a Standalone or SimX/SCIRun version's front-end when he uses the SimX/Uintah worker process. Table 4.1 shows which version of SimX each of the four example computational studies use.

The rest of this section provides the implementation and interface details to each of the modules and versions described above.

4.2.1 User Interface

The UI interface is the means by which the user informs the manager process the locations of the worker SISOL processes. In addition, it is also the means by which the user controls an ongoing computational study. The implementation and interface are different in the Standalone version and the SimX/SCIRun version. The SimX/Uintah version does not have a UI module.

In the standalone version, the UI module reads the user preferences from several config files. There are three major pieces of information required by the UI: SISOL-related information such as the SISOL discovery server's address and object IDs used in the SISOL server, FUEL-related information such as the worker processes' addresses and TCP ports, and problem-related information such as the design space dimension, and ranges of the region of interest. These information are passed to the UI module via three config files. A typical set of config files for the bridge design problem is shown in Figure 4.2.

Since the Standalone version does not support interactivity, there is no mechanism for the user to steer the study once it starts.

SCIRun provides a means for SimX to steer the computational study interactively. In the SimX/SCIRun version, this interactive UI is realized by the SCIRun module called the SimXManager (Figure 4.3). The user can specify the address of worker processes in the SimXManager's UI. SimXManager also

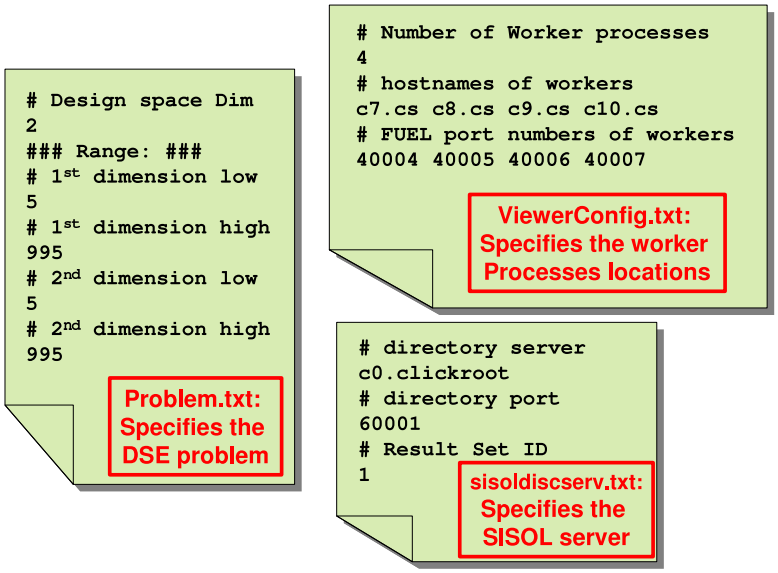


Figure 4.2: Standalone SimX UI: three configuration files

has an input port, which receives information about the current computational study — the design space dimension, the slice to be explored, the region of the slice to be explored, the values of the performance metric parameters, and the ranges of the admissible region. This input port continuously receives updates via SCIRun's interactive steering mechanism, so, by sending new data through this port, the user can specify a new slice of design space or tree node to explore as the computational study is going on. The application developer can thus use SCIRun's existing modules to allow intuitive specification of the design space slice. For example, Figure 4.4 shows how the slice of the design space is controlled by the user. He uses the SCIRun-provided frame widget to specify the range of front electrode placement he is interested in, the point widget to specify the placement of the back electrode, and a text UI to specify the voltage strength. The SCIRun net gathers all this information and translates them into a SCIRun Matrix object which can be sent to the SimXManager via its input port.

An example of that matrix is shown in Figure 4.5. This particular example specifies a slice of the design space in the defibrillator study. The matrices sent to the SimXManager input port have seven columns. The first column specifies the study's setup: the total number of design space and performance space dimensions in the computational study, and which design space dimensions are fixed for the particular slice the user wishes to explore. Our example has seven design space dimensions: four for electrode placements, one for shock voltage, and two for the performance metric parameters (here, the performance metric

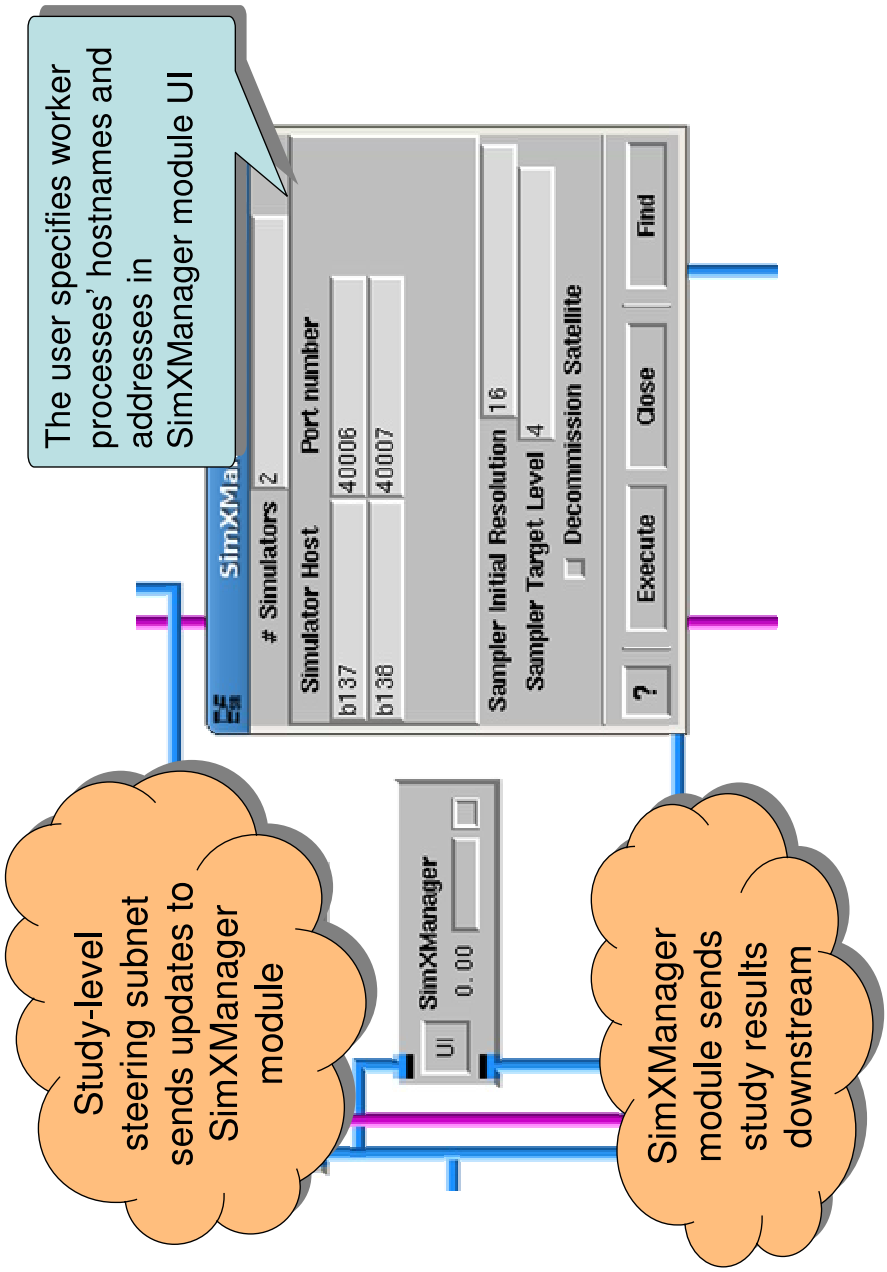


Figure 4.3: SimXManager module and UI

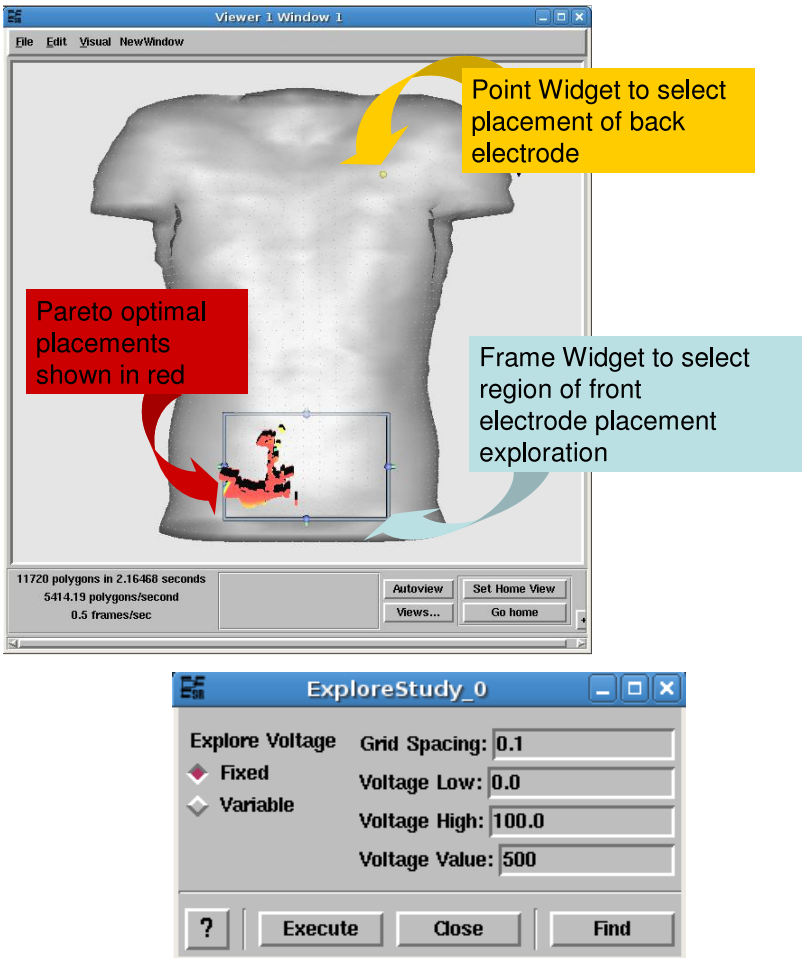


Figure 4.4: Using SimXManager for interactive computational study-level steering

parameters are counted as design space dimensions). The second and third columns specify the indices and values of the fixed design space dimensions. In our example, dimensions 2 to 6 are fixed: dimensions 2 and 3 are the placement of the back electrode, set here to 200 x 100mm off the torso's center; dimension 4 is the voltage strength, set to 150V; dimensions 5 and 6 are the activation and damage threshold respectively, set to 0.5 and 5 Vcm^{-1}. The fourth and fifth columns specify the range of exploration for the non-fixed design space dimensions. In our example, the unfixed design space dimensions are dimensions 0 and 1, which represent the placement of the front electrode, and the user wishes to explore the rectangle bounded by (-100mm,0mm) and (100mm,200mm) off of the torso's center. Finally, columns five and six represent the admissible region. In our example, the user only admits simulations whose performance metric 0, the activation level, is above 80%, the performance metric 1, the damage level, is below 10%, and the performance metric 2, the uniformity, is any value.

SimXManager also contains a set of APIs with which the user can retrieve study-level information. They are listed in Table 4.2. These APIs gives the user an easy way to retrieve aggregated information of a study, and present that aggregated information to the user.

4.2.2 Sampler

The Sampler module has the same implementation across all versions of SimX. The goal of the Sampler is to translate the specifications of a study (be it the entire computational study, a slice of a design space, or a node in the

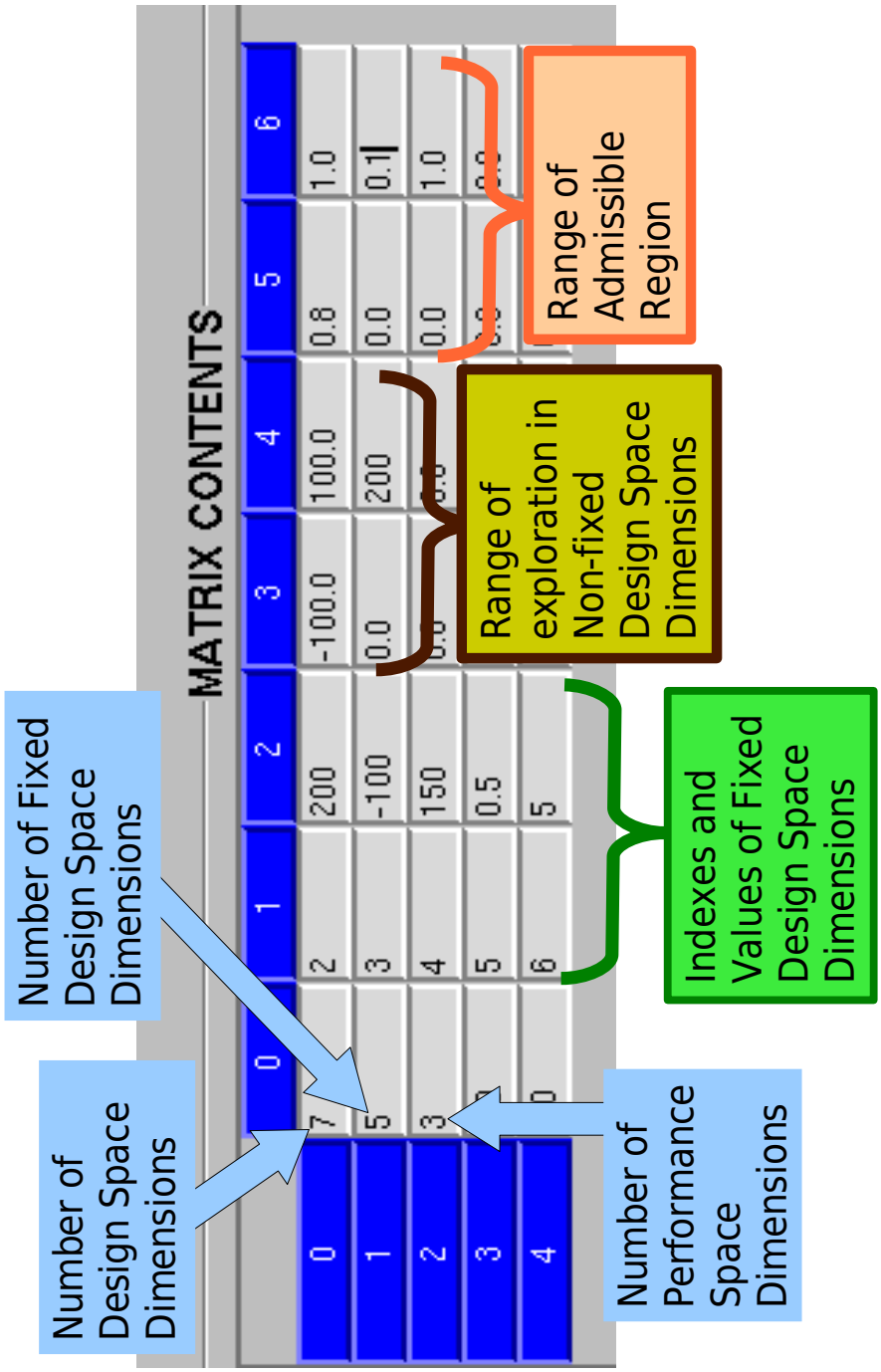


Figure 4.5: Matrix format to SimXManager's input port

Signature	Function
bool getAllExpr (MatrixHandle study, vector <pair<Values, Values> >& result, string& msg)	Fills the result vector with <experiment,result> pairs conducted for the given slice (or study). Returns true on success, otherwise msg contains the error message.
bool getPareto (MatrixHandle study, vector <pair<Values, Values> >& result, string& msg)	Fills the result vector with only experiments that are currently in the region of interest
bool getIsDone (MatrixHandle study, bool& result, string& msg)	Save to result whether the the sampler has finished exploring the given slice.
bool getNumOfExpr (MatrixHandle study, int& numExprs, string& msg)	Save to numExprs the number of experiments finished for the given slice

Table 4.2: API of SimXManager module in SimX/SCIRun

search tree) into a list of experiments to be run. The sampler interface is given in Table 4.3.

As is discussed in later chapters, in some types of computational studies, e.g., Pareto optimization, the Sampler module can take advantage of the underlying logic of the study by issuing the simulations in stages. Therefore, the interface is structured in a way where only one experiment is issued at a time (getNextPointToRun()), and there is a mechanism where the performance metric of the simulations are passed back to the Sampler (registerResults()).

SimX provides three built-in implementations of the Sampler, and the user can specify which one to use in a computational study (using the SamplerType option in the API) to tailor its behavior to the type of computational study he wishes to perform. In addition, the user can build their own custom Sampler module by implementing the Sampler API. The three built-in sampler types are as follows: 1) the Grid sampler, which samples the design space on a regular grid; 2) the Random sampler, which samples the design space randomly; and 3) the Active sampler, which is used for Pareto optimization, and samples the design space in stages of increasingly refined grids. The use of different types of sampler is discussed in Section 5.1.

4.2.3 Task Queue

The Task Queue is the temporary area that stores experiments that are issued by the Sampler but not yet executed by a worker process. When the Resource Allocator tries to find a simulation to issue to its worker processes, it

Interface	Explanation
SimXExploration(int N, int perfN)	Initializes an exploration object, to be run in a design space with N dimensions, and a performance metric parameter space with perfN dimensions.
struct StudyID {vector<int> fixedDimensionIndices, vector<double> fixedDimensionValues, vector<int> nonFixedDimensionIndices, vector<range> nonFixedDimensionRanges, vector<double> performanceMetricParameterValues, SamplerType, InitialSamplerOptions}	A data structure to identify a study. The first five arguments defines the region/slice of exploration. The last two arguments defines the type of sampler used to conduct this study.
SetCurrentStudy(StudyID)	Sets the current study. If an existing study is not found, add create a new sampler for it.
GetEvaluated(StudyID)	Gets a list of results (<parameter metrics, design space points> tuples) for this study.
SetSamplerOptions (StudyID, SamplerOptions)	Sets the study's sampler options. The options, (e.g., initial grid resolution, target refinement, etc.) depends on the sampler used in the study.
IsCurrentStudyDone()	Tests whether the current study is finished.
registerResults(Experiment, Result)	Registers the result (performance metric) of an experiment.
getNextPointToRun (bool* hasPointReaady)	Gets the next experiment to issue from the current study.

Table 4.3: SimX interface to the sampler code.

has the freedom to choose among the tasks on the Task Queue.

There are two implementation of the Task Queue: as a simple FIFO queue, or as a priority queue.

In the FIFO version, the task queue is implemented as a simple STL list, and the tasks are issued from the Task Queue in the same order they are issued by the Sampler.

In the priority queue version, a subset of the tasks on the Task Queue is selected into a *batch*. The batch consists of experiments that are determined by application-domain knowledge to be more important, and thus need to finish early. The Task Queue will then issue the tasks that are batched first. When all tasks in a batch are executed, a new batch is selected. If no task is available then (i.e., the Task Queue is empty), then the Task Queue will ask the Sampler to issued more experiments. The application developer who creates the MES can implement the Task Queue's API (Table 4.4) to tell SimX how the experiments on the Task Queue ought to be batched, as well as the computational resources that should be assigned to each experiment in the batch. When the user conducts the MES on SimX, the experiments will receive the correct priority.

4.2.4 Resource Allocator

The Resource Allocator is responsible for assigning the batched tasks from the Task Queue to the worker processes. If the simulation code is serial, the Resource Allocator merely waits until the next worker process is available, and

Interface	Explanation
AddTask (Experiment)	Called by the Sampler. Adds an experiment to the Task Queue.
CreateBatch (set<Experiment>*)	A customizable function that selects a subset of the tasks on the Task Queue and mark them as being in the current batch.
GetIdealGroupSize()	The user uses this function to tell the Task Queue the ideal number of processors to be assigned to each task on the current batch. All task in the batch get the same number of processors.
AssignNextTask (groupID)	Called by the Resource Allocator. Removes a task from the current batch and assigns it to the worker process group identified by groupID)

Table 4.4: SimX interface to the Task Queue.

sends the next task on the task queue to it.

However, if the simulation code can be run in parallel, i.e., the user is using the SimX/Uintah version of worker processes, the Resource Allocator is also responsible for configuring the worker processes so that they can be formed into process groups to run individual simulations in parallel. The Resource Allocator tries to adjust the sizes of process groups so they will be as close to the requirements of the task (corresponding to the return value of GetIdealGroupSize()) as possible. SimX normally manages this module without the user's intervention. However, the user can override the default SimX policy and create their own worker process groupings by calling the Resource Allocator API. This API consists of the following function:

```
reconfigure(const int* assign)
```

which creates one or more worker process groups and destroy old ones. The assign array is an array of length N, where N is the total number of worker processes involved in the computational study. The worker processes' new group assignments are stored in this array, indexed by their global processor ranks. Worker processes with the same assignments will form themselves into the same process group. Section 5.5 discusses how application-specific knowledge is used in the reconfigurations of worker process groups.

4.2.5 FUEL

The inter-process communication between the manager process and the worker processes is performed by FUEL. FUEL has the same implementation

across different versions of SimX (hence the interoperability between processes of different SimX versions).

As discussed before, FUEL is handler-based. When a user starts up SimX, he provides several handler functions to specify how to marshal and unmarshal objects (Table 4.5). The information sent by the worker processes to the manager process is called a *frame*; the information sent by the manager process to the worker processes is called an *update*. Instead of bytes, however, in FUEL, all information travelling across the wire are vectors of double precision numbers, so there is no need for the user to worry about endian conversion. The rationale is that, since information exchanged between worker processes and the manager process are design parameters or performance metrics, they can always be represented by vectors of doubles.

The execution model of FUEL handlers is different on manager process and worker processes. On the manager process, the FUEL handlers are asynchronous, so multiple handlers may be called at the same time, and the user must make their handlers thread-safe. The handler calls are triggered by arrival of a frame from a worker process. On the worker process, the FUEL handlers are synchronous, so the worker processes must call a special function (`arbitrate()`) explicitly in order to trigger the execution of the worker-side FUEL handlers.

FUEL tries to hide the communication cost by overlapping the communication of an experiment with the execution of the last. This way, the worker processes can be kept busy at all times and not have to wait for the manager process to process its last experiment's performance metric before receiving the

Operation	Signature	Function
Typedefs of handler methods on workers	typedef void* (*ARBITER_CNXTINITMETHOD)();	Init method. Called during initialization. Should return an opaque pointer representing an application-specific context.
	typedef void (*ARBITER_UPDATEMETHOD) // input update vector (const double *, int, // output frame vector const double **, int*, /*context*/ void *);	Update method. Gets an update vector and applies it to the simulation, and extracts next frame vector from the simulation.
	typedef bool (*ARBITER_TRIGGERMETHOD) // input frame vector (const double*, int, /*context*/ void *);	Trigger method. Gets a frame vector, and return true if the frame should be sent to the manager.
	typedef void (*ARBITER_CNXTDEINITMETHOD) (void*);	DeInit method. Reclaim the memory pointed to by the opaque pointer.
Usage on worker processes	void registerSatellite (int satelliteType, ARBITER_CNXTINITMETHOD pCnxtInit, ARBITER_CNXTDEINITMETHOD pfCnxtDeinit, ARBITER_UPDATEMETHOD pfUpdate, ARBITER_TRIGGERMETHOD pfTrigger);	Register a collection of handlers function (known as a "Satellite").
	void arbitrate();	Called by the worker process to send back the last frame and receive the next update.
Typedef of handler methods on the manager	typedef void (*SATELLITE_CALLBACKMETHOD) // input frame vector (const double *, const int, // output update vector const double **, int*, /*context*/ void*)	Asynchronous call back method on the manager. The function should process the frame vector sent by the worker process and initialize the update vector.
Usage on manager process	Satellite(int satelliteType, SATELLITE_CALLBACKMETHOD callback);	Creates a Satellite object, which is used to connect to the worker processes
	void Satellite::addArbiter (const char* hostname, unsigned int port, void* cntxt);	Spawns a handler thread to talk to the arbiter on the given host.

Table 4.5: FUEL interface.

next simulation to run. Figure 4.6 shows two of the "FUEL cycles" where two experiments are performed on the worker process while two are issued by the Manager process. Boxes in yellow indicate activities on the worker process; boxes in white indicate activities on the manager process; boxes in blue indicate activities on the network.

4.2.6 Simulation Container

The Simulation Container is the wrapper code that runs on the worker process.

In the Standalone version, it provides a substrate on which the simulation and evaluation codes are run. It is provided as a shared library and only does a few things. It calls FUEL's arbitrate() to send the performance metric of the last simulation to the manager and receive the next simulation from it; it calls the simulation code to run the simulation; it calls the evaluation code to extract the performance metric. Optionally, it may also write and read intermediate state of the simulation code to and from SISOL for optimization opportunities.

In the SimX/SCIRun version, the Simulation Container is packaged into two modules: the BoundarySatellite module, and the BoundarySatelliteResultStore module. When the worker process (which, in the case of SimX/SCIRun, is just a SCIRun workflow net with the BoundarySatellite and BoundarySatelliteResult-Store module in it) receives a simulation, the FUEL handler function provided in SimX/SCIRun would automatically unmarshall the experiment specification into the form of a SCIRun matrix. This Matrix is fired off from the output port

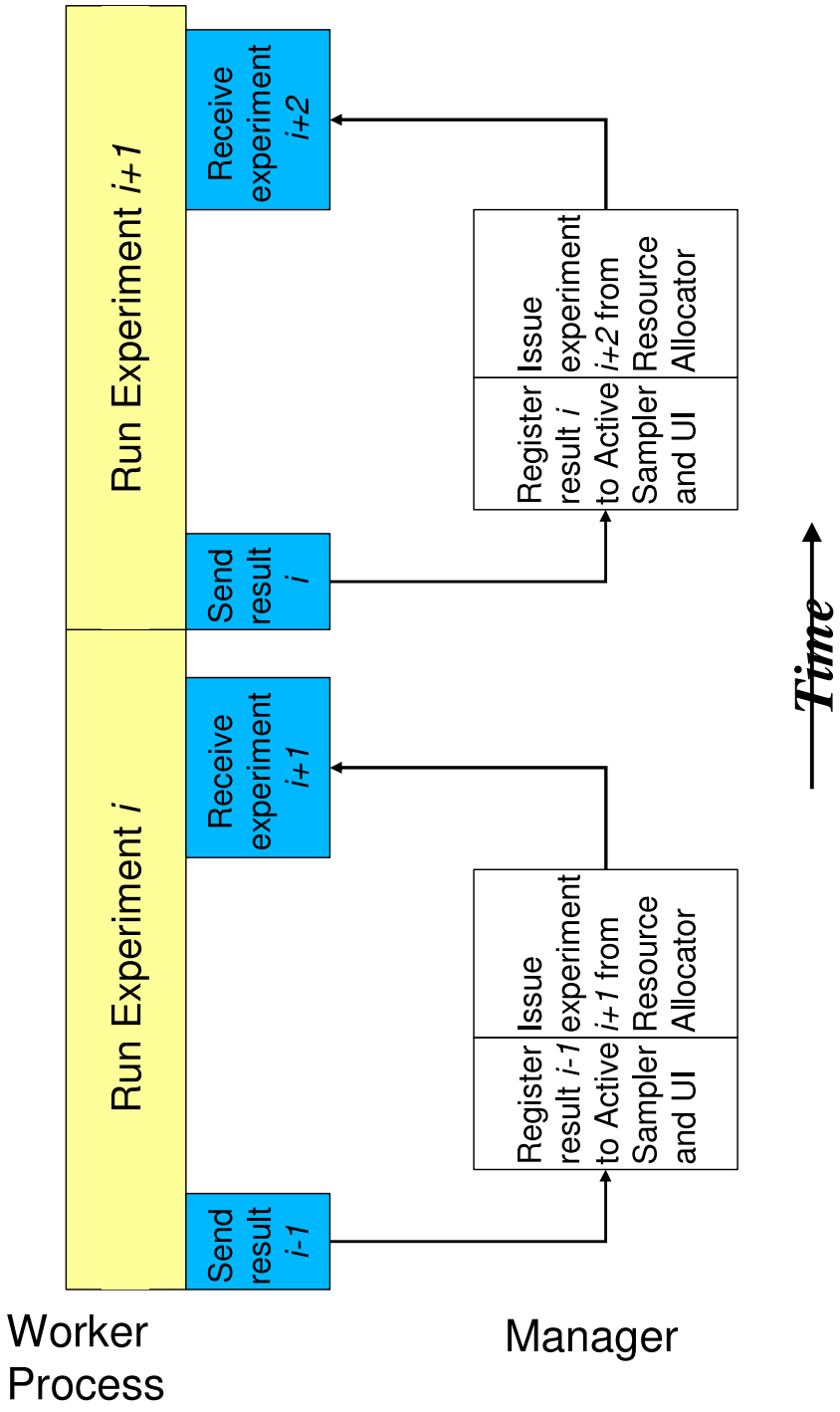


Figure 4.6: Communication pattern in FUEL

of the BoundarySatellite module, causing the workflow net downstream from it to execute. This execution corresponds to the execution of the simulation and evaluation codes. When the execution and evaluation subnet completes, the simulation's performance metrics are sent to the input port of the BoundarySatelliteResultStore module. This module uses shared memory and pthreads to communicate with the BoundarySatellite module, which causes the BoundarySatellite to call arbitrate() to send the experiment's performance metric back to the manager (using, again, SimX/SCIRun-provided FUEL handler to marshall the SCIRun Matrix), as well as receive the next experiment, which causes the simulation and evaluation net to execute again (Figure 4.7).

In the SimX/Uintah version, the Simulation Container has an additional responsibility. All Uintah applications are MPI-based and thus can run in parallel, thus the Resource Allocator can potentially instruct the worker processes to form themselves into process groups to run a particular experiment in parallel. The Simulation Container is the module that handles these reconfiguration directives.

In SimX/Uintah, each worker process is part of two MPI communicators: the global communicator (i.e., MPI_COMM_WORLD), and a sub-communicator actually used to run the simulation. Initially, the sub-communicator is a duplicate of MPI_COMM_WORLD. When a Simulation Container receives an instruction from the Resource Allocator to form a group, it destroys its worker process's old sub-communicator, and rebuilds a new MPI sub-communicator with the other processes making up the same group. The Simulation Container

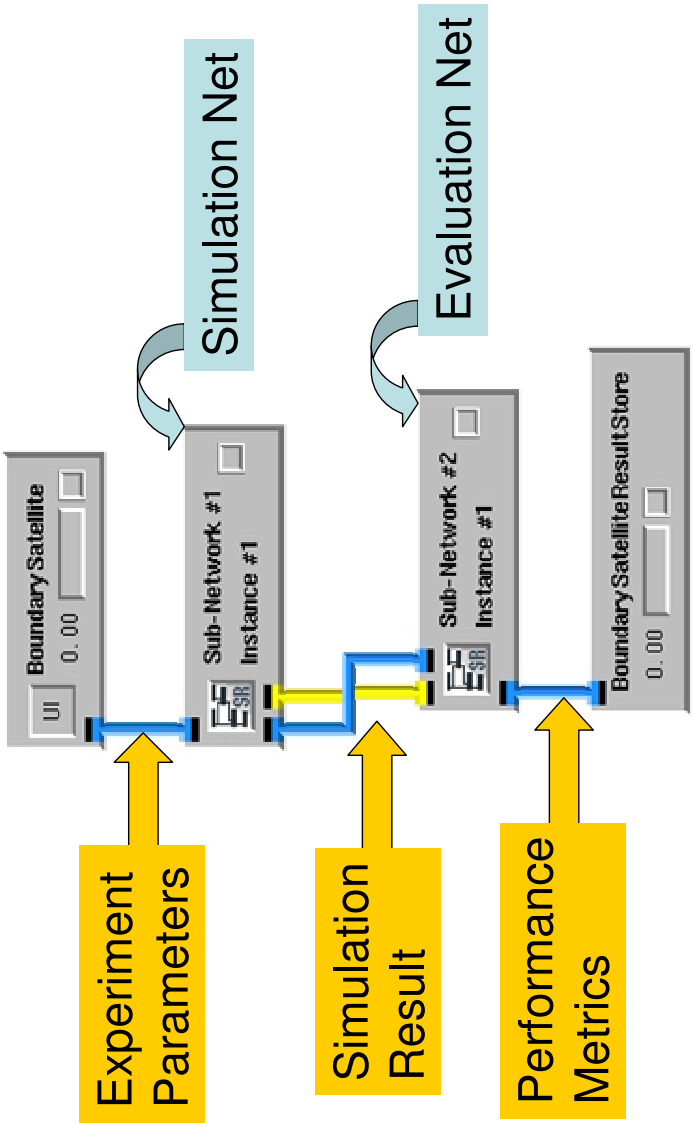


Figure 4.7: Simulation Container modules in SimX/SCIRun

does this in a multi-step process. It starts by duplicating MPI_COMM_SELF. It then uses the duplicate to create an MPI intercommunicator with another process in the same group (identified via the reconfiguration directive sent from the Resource Allocator, and reached through the gloabl communicator), and merges the MPI intercommunicator to form a new MPI intracommunicator. It repeats the communicator-merging process until the MPI_COMM_SELF's of all the processes in the same group are merged into the same communicator. This communicator then becomes the sub-communicator for the process group. The Resource Allocator thus acts as a global coordinator: by sending reconfiguration directives only to the worker processes involved in a reconfiguration, it can form and destroy groups without involving the other worker processes. Once the process group is formed, the Simulation Container hands over the thread of execution over to the simulation code. In SimX/Uintah, the simulation code may periodically return control to the Simulation Container to check if there is a need to terminate the simulation early.

When the simulation is finished (or terminated early), the Simulation Container optionally writes the simulation result into SISOL to enable their use in optimizing future simulation runs. It also executes the evaluation module to calculate the performance metric for the experiment just conducted. It then sends the performance metric back to the manager process by calling arbitrate().

In SimX/Uintah, the Simulation Container is implemented as a modified Uintah component. This modified component sets the correct MPI communicators and data archives, so that the rest of the Uintah workflow can proceed

with no changes required.

4.2.7 SISOL

SISOL is used for implicit communication between worker processes, as well as between worker processes and the manager process. While FUEL handles the sending and receiving of design parameters and performance metrics, SISOL handles the storing of the simulations' intermediate internal states and final results, and, in case of SimX/Uintah, a database of idle worker processes. They facilitate reusing of simulation's results, for visualization of simulation results, and for checking for the existence of idle worker processes.

SISOL is implemented as a collection of standalone servers communicating with a client library via TCP/IP.

There are two types of SISOL severs: the directory server and the data server. Only one directory server is run during a computational study, but multiple data servers may be run. The data servers are where the actual data is stored. Typically they are spawned on the cluster nodes, so the bandwidth between them and the worker processes is high. The data servers store the objects in volatile memory, and each data server can handle read/modify/write consistency — conflicting read/writes are serialized, and reading or writing non-existent objects returns an error.

The single SISOL directory server maps objects to the data server where they are stored. It also decides how the index space is partitioned between data servers. The directory server acts as the single point of access to guarantee

read/modify/write data consistency across the whole SISOL layer.

An optimistic protocol is used to guarantee the directory server's consistency. When a worker process wishes to access an object, it asks the directory server where this object is located. If the object already exists, the directory server points the worker process to the data server hosting it. Otherwise, the directory server decides which data server the object "should" be located at, and points the worker process to that. The worker process then contacts the data server to access the object, and the data server would respond as required by the nature of the operation. If the operation is a insert operation, the data server would additionally notify the directory server for permission to create the object. The directory server would either grant permission and update its objects-to-server mapping, in case such an object doesn't already exist, or, in case such an object already exists on another server (as is the case when another worker process wishes to insert a conflicting version of the same object concurrently, and the directory server had pointed it to a different data server), deny the permission. If the insert operation is denied, the data server will return the denial to the worker process, which will retry the operation. During the second try, the directory server will be able to point the worker process to the correct data server.

In the Standalone and SimX/Uintah versions, the SISOL client is provided as a library. Table 4.6 shows the SISOL interface. Split-phase read and write operations guarantee data consistency on the same object. The marshalling and unmarshalling of data is performed by user-provided functions declared

using the interface in Figure 4.8, and referenced by the typeID argument in the CreateSet function.

The SimX/SCIRun version provides two SISOL client wrappers, one to store SCIRun::Field objects to SISOL, the other to store SCIRun::Matrrix objects. Both Field and Matrix objects datatypes included in the SCIRun core library. A Matrix object represents a dense matrix, while a Field object consists of a geometric mesh and a set of data values. The SISOL interface in SimX/SCIRun is realized in four modules: SISOLFieldWriter, SISOLFieldReader, SISOLMatrixReader and SISOLMatrixWriter (Figure 4.9). All SISOL modules require the user to enter the SISOL discovery server address and port number using the UI, as well as the object ID. They also take a Matrix input port, which specifies the spatial coordinate of the object stored. The reader modules have an output port, where the stored object is injected into the workflow downstream; the writer modules have an input port where the object is read from the workflow upstream and stored in the SISOL. In addition, a QueryClosestObject module allows SCIRun to lookup the coordinates of the object in an object set that is closest to a coordinate provided.

Figure 4.9 shows the SISOL modules being used for in the context of simulation result reuse (see Section 5.3). This is an expanded version of the net shown in Figure 4.7. The Simulation Container modules are shown on the left. The simulation net is shown here as two separate subnets. One subnet is connected to the Simulation Container modules, and consists of a SISOLFieldReader and a ReuseFieldConditionalExecute module. The other subnet, on the

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)	Registers client as participant; retrieves object set metadata in objSet.
	void UnregisterSet(void* objSet)	Unregisters client.
Access	void Insert(void* objSet, double* coords, void* obj) void Remove(void* objSet, double* coords)	Insert/remove an object into/from an object set.
	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)	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 4.6: Interface of the spatially-indexed shared object space layer (SISOL).

```
// Functions to describe a SISOL object
typedef struct {
  // extracts the descriptor context, in serial form, from an object
  int(*m_descriptorSizeFromObj)(void* obj);
  int(*m_getDescriptorFromObj)(void* obj, char* cntxt);

  // remaps the descriptor context, in serial form, from another
  // descriptor, to be used for the number of processors on
  // this process.
  int(*m_descriptorSizeRemap)(int remoteDescCntxtSize,
                              const char* remoteDescCntxt);
  int(*m_getDescriptorRemap)(int remoteDescCntxtSize,
                             const char* remoteDescCntxt,
                             char* cntxt);

  // returns the number of processors described by the context
  int(*m_numProcs)(int descSize, const char* descCntxt);
  // returns the number of patches described by the context
  int(*m_numPatches)(int descSize, const char* descCntxt);
  // returns the size of the patch (in bytes) described by the conte:
  int (*m_getSizeOfPatch)(int descSize,
                          const char* descCntxt,
                          int patchNum);
  // returns the owner of the patch described by the context
  int (*m_getOwner)(int descSize,
                    const char* descCntxt,
                    int patchNum);

} ISISOLDescriptor;

// Functions to allocates and deallocates sisol objects.
typedef struct {
  // frees the memory of object
  void(*m_freeObj)(void* obj,
                   ISISOLDescriptor* descriptor,
                   const char* descriptorCntxt);
  // allocate memory for the object described by the descriptor
  void(*m_allocObj)(void** obj,
                    ISISOLDescriptor* descriptor,
                    const char* descriptorCntxt);
  // copies the patch from the object to the buffer.
  // (assumes buffer has m_getSizeOfPatch bytes)
  // (assumes this processor owns the patch)
  int(*m_readFromPatch)(void* obj,
                        int patchNum,
                        char* buffer);
  // copies the patch from the buffer to the object.
  // (assumes buffer has m_getSizeOfPatch bytes)
  // (assumes this processor owns the patch)
  int(*m_writeToPatch)(void* obj,
                       int patchNum,
                       char* buffer);
} ISISOLFactory;

/* Information needed to describe an object type */
typedef struct {
  int m_objTypeID;
  int m_isDistributed;
 // if the object type is distributed, its factory
 // and descriptor functions must be collectives.
  ISISOLFactory* m_factory;
  ISISOLDescriptor* m_descriptor;
} SISOLObjectType;
```

Figure 4.8: Object Type declaration interface in SISOL

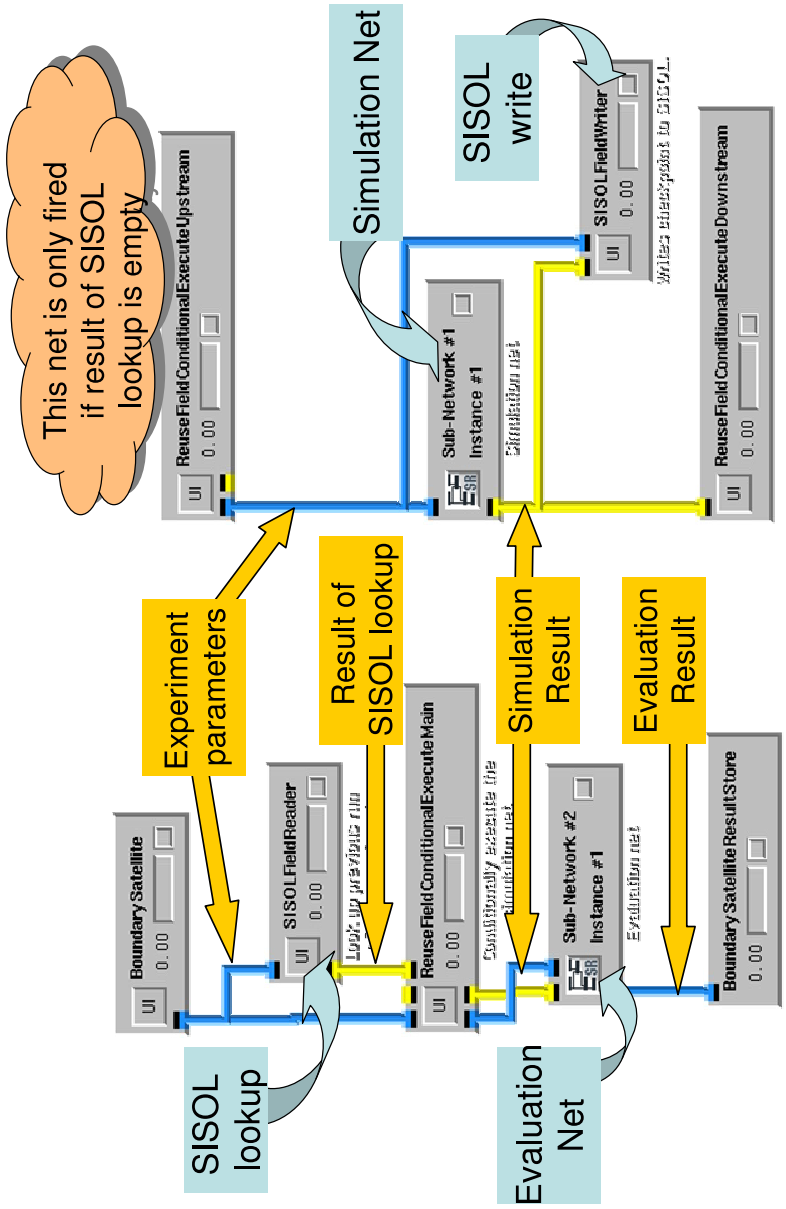


Figure 4.9: SISOL modules in use in SimX/SCIRun

right, is executed only if the ReuseFieldConditionalExecute module does not receive a SCIRun Field object from the SISOLFieldReader module. The subnet on the right also writes the result of the simulation net into the SISOL layer via the SISOLFieldWriter module. Note that the simulation's parameters here (the SCIRun Matrix links) are used as the spatial coordinates for the SISOL modules.

4.3 Summary

This section presents the SimX system, a suitable platform to conduct Multi-Experiment Studies on parallel platforms. Its modular structure separates out key functionalities in an MES — user interaction, sampling, scheduling, resource allocation, sharing data, coordinating computational elements, and inter-process communication — such that optimization techniques can be adapted onto each module independently, as dictated by the needs of the particular MES being conducted. This adaptability, as will be seen in later chapters, is key to achieving high-performance MESs.

In total, the three versions of SimX — Standalone, SimX/SCIRun, and SimX/Uintah — contain over 10,000 lines of C++ code and represent three years of engineering work. SimX interfaces with networking protocols and communication APIs such as TCP/IP and MPI, as well as scientific libraries such as PETSc. Table 4.7 gives a sense of the complexity of the SimX system by showing a breakdown of the number of lines of codes written for each module in each version of SimX.

	SimX Version		
	Standalone	SimX/SCIRun	SimX/Uintah
UI	N/A	832	N/A
Sampler	2634	974	N/A
Task Queue	709	628	N/A
Resource Allocator	N/A	485	N/A
FUEL	1093		
Simulation Container	406	585	883
SISOL	3746		

Table 4.7: Number of lines of code in SimX, broken down by modules

Chapter 5

Using Application-Level Knowledge

As discussed in previous chapters, Multi-Experiment Studies present a special set of challenges to parallel system software. The tradition system approach of running a collection of simulation experiments on parallel machines, which treats each experiment as a separate black box, is too inefficient to cope with the requirements of MESs: a realistic response time under heavy computational load, and study-level user interactivity. As part of this research, a number of techniques have been developed to cope with these challenges.

All of these techniques benefit from the system leveraging some knowledge about the particular MES being conducted to make policy choices to optimize for its goals. In the following sections, each technique is described in detail. The general underlying principle which enable the technique is discussed first, followed by a detailed description using the formulation used in Section 3.5. Finally its implementation on the SimX system described in Section 4.2 is given.

These techniques can be generalized into three broad categories. The first group (Sections 5.1 and 5.2) aims to minimize the number of experiments required in a study, either automatically (Section 5.1) or in a user-driven fashion

(Section 5.2). The second group (Sections 5.3 and 5.4) focuses on reusing results from computation done in an earlier part of a study to speed up the later parts, either at the individual experiment level (Section 5.3) or the study level (Section 5.4). The final group of techniques (Sections 5.5 and 5.6) uses resource allocation and scheduling decisions customized to the characterisitcs of the MES at hand, to ensure efficient progress towards study goals.

5.1 Using Early Aggregate Results to Schedule Pareto Optimizations

The first technique is designed for MESs that are optimization studies. The main idea is to incorporate search algorithms in the system's sampling logic to cut down the number of experiments required.

5.1.1 Principle

Many MESs are conducted to solve optimization problems, where each execution of an experiment corresponds to a function evaluation, and the goal of the study is to optimize that function. Using the terminology from Section 3.5, the simulation code/evaluation code pair corresponds to the function evaluation, the design space corresponds to the search space, the performance metrics correspond to the optimization metrics, and the region of interest corresponds to the optimal points in the design space where the function is optimized. In optimization MESs, experiments can be selected to sample the performance space, and the function evaluations at these sample points can then be used

to identify the region where the functions achieves the desired optimal values. Optimization MESs are a common occurrence; some examples can be found in [85, 47, 57, 16, 5].

Three of our four example MESs are optimization studies. As discussed in Chapter 3, the bridge design, defibrillator design, and helium model validation studies are examples of Pareto optimization. The particulars of Pareto Optimization was discussed in Section 3.1. By scheduling the simulations intelligently as described in this section, the Sampler module can decrease the total number of simulations required to discover the Pareto frontier.

In general, if the system has the knowledge that the particular MES it is conducting is an optimization study, it can be free to utilize more intelligent sampling techniques. Instead of application-unaware grid scheduling like traditional systems ([2, 18]), the parallel system can incorporate parallel search algorithms — such as parallel genetic algorithms ([15, 14]), simulated annealing ([70, 29]), parallel line searches, etc. — in its scheduling decisions, and tailor the algorithm to suit the MES being conducted.

In effect, the search algorithms help the system reduce the number of experiments required to conduct the study. Compared to the naïve approach, which covers the entire search space in an evenly-spaced grid, search algorithms can cut down the number of function evaluations needed for an optimization problem. In MESs, this translates into a reduced number of experiments needed, and thus improved study response time.

5.1.2 Description

For the three Pareto Optimization MESs in this paper, we use an Active Sampling algorithm that is designed for multi-objective optimization problems.

A variety of techniques have been developed for discovering Pareto frontiers (e.g. [24, 61, 20]). We use a hierarchical approach which requires no specific information about cost functions other than an evaluation procedure.

Recall the definition of the Pareto Optimization problem in Subsection 3.1.3. The Pareto frontier is a subset of points in the design space that contains only undominated points. A point x_1 dominates another point x_2 (denoted by $x_1 \succ x_2$) if, for all the performance metrics, x_1's metric is better than x_2's. In a discrete setting, assume that the design space has been sampled at a finite number of points, yielding a finite subset, V, of the design space points. The discrete approximation of the Pareto Frontier is the subset $R \subseteq V$ that contains only the *undominated* points.

The Active Sampler's goal is to choose samples V such that R rapidly converges to the continuous Pareto frontier. In general, the Pareto frontier is a low-dimensional subset of the design space, therefore uniformly sampling the design space would be inefficient. Instead, our algorithms tries to walk along the frontier once an undominated point is discovered.

We start by discretizing the design space into a coarse lattice, and proceed as follows. Seed the computation by evaluating the coarse 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. This procedure is repeated until the desirable refinement is achieved.

In effect, the Active Sampler uses the experiments on the coarse-level lattice to identify promising regions of the design space, and explore the promising region in higher detail using the finer-level experiments. In other words, the coarse-level experiments are used to prune uninteresting regions of the design space early in the study.

The Active Sampler strategy is illustrated in Figure 5.1. A Pareto Frontier on the two-dimensional design space is shown in the figure, and the Active Sampler tries to approximate the frontier, which is shown as a curve through the design space. Initially, the Active Sampler issues nine experiments on a 3x3 grid (top left; issued experiments shown in black dots). After the results have returned, a first-level Pareto frontier can be identified (top right; Pareto optimal points shown in red crosses). The Active Sampler then refines the grid to a 6x6 grid, but only issues experiments that are neighbors of Pareto optimal points on the first-level (middle right). After the results of the second level have returned

(middle left), it repeats the refinement process on a 12x12 grid (bottom left and bottom right). The result is a progressively-refined approximation of the Pareto Frontier after each level.

5.1.3 Implementation

The Active Sampler is implemented as a SimX sampler module. It can be plugged into the SimX architecture by implementing the APIs listed in Table 4.3.

The Active Sampler module keeps an internal representation of V, R, as well as a list of points to be added to V (the to-do list). At initialization (the SetCurrentStudy() call), the Active Sampler module populates the to-do list with points on the initial coarse lattice. Every time the sampler is asked for the next experiment (the getNextPointToRun() call), it returns the next point on the to-do list. When experiment results arrive (the registerResults() call), the Active Sampler examines the performance metric of the newly-evaluated design space point d. It adds d to V, and, if d is not dominated by any point in R, updates R by removing from it the points that are dominated by d, and adding d to R. During refinement (which can either occur when the to-do list is depleted, or when all outstanding experiments are completed), the Active Sampler refines the lattice, examines the points in R, and adds their nearest neighbors to the to-do list.

The Active Sampler signals the completion of a study when a sufficient number of refinement steps have occurred. At that point, R contains the final Pareto Frontier approximation at the desired level of detail.

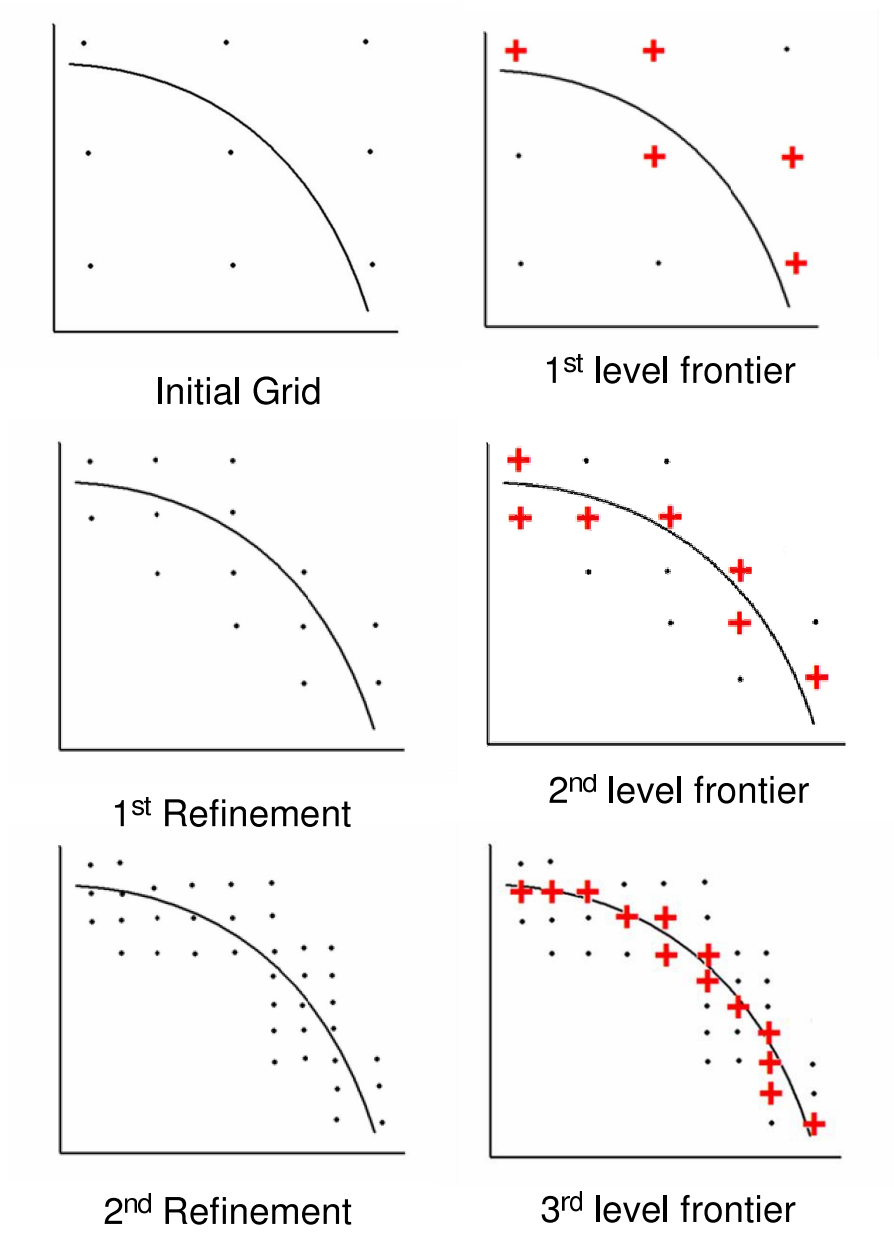


Figure 5.1: Illustration of the Active Sampling technique

5.1.4 Challenges

The Active Sampler must efficiently and correctly resolve the Pareto frontier in the setting of multiple worker processes. 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 worker process have queued jobs.

To ensure full loading of worker processes, the sampler is required to provide a worker process with an evaluation point whenever it is available to receive one. Sometimes this means issuing evaluation points in the absence of complete knowledge. In those cases, the active sampler must issue 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. 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 to ensure full loading. As the number of worker processes increases, the number of pending simulations in the system also increases, and so more experiments are issued with incomplete knowledge. Therefore, the amount of excess evaluations increases with the level of parallelism.

To ensure optimal sampling, the sampler is required to have complete knowledge before it issues the next evaluation point. In particular, the Active

Sampler cannot issue an evaluation point unless all information that could affect the issuing of that point has arrived. If a worker process is ready to receive an evaluation point, but the Active Sampler is unable to ascertain whether it can issue the next point (because simulations are still being run on other worker processes), the worker process will be forced to stay idle. This *synchronization overhead* among worker processes is inevitable to ensure optimal sampling. As the numer of worker processes increases, more worker processes are likely to stay idle, thus the synchronization overhead also increases.

The trade-off between full loading and optimal sampling depends on the MES being conducted. Full loading is preferable in MESs where the penalty for sampling on incomplete knowledge is small — i.e., where individual simulation experiments are fast-running relative to the total amount of computation in the MES, because the wasted experiments incur a relatively small performance penalty. Conversely, optimal sampling is preferable when the individual simulation experiments are long-running. There are thus two versions of Active Sampler implementation in SimX. One version of the Active Sampler ensures full loading, and the other ensures optimal sampling. The full loading version is used in the bridge and defibrillator studies, where the MESs are made up of a large number of short-running experiments, while the optimal sampling version is used in the helium model validation study, where the MES is made up of a small number of long-running experiments.

In contrast to the active sampling technique discussed above, a naïve parameter sweep sampler does not use gathered information in choosing evaluation

points. While this allows relatively trivial parallelization, the sweep sampler's disregarding of available information results in one or more orders of magnitude of wasted simulation experiments. As we describe in Chapter 6, compared to the Sweep Sampler, the Active Sampler reduces the number of experiments that needs to be run to resolve the Pareto frontier at the finest level by anywhere from 33% (in the helium model validation study) to 94% (in the bridge design study).

5.2 User-directed Sampling

The technique described in Section 5.1, i.e., incorporating search algorithms in the system's sampling logic, is one way for the system to automatically prune the design space during its exploration. This works well when the MES's region of interest is clearly defined and when the design space has a manageable dimensionality. However, in MESs where the design space is too large, or where the region of interest does not have an objective definition, automatic pruning is not sufficiently effective, or possible. Instead, the system needs to rely on the user to direct the pruning of the design space.

5.2.1 Principle

There are two main categories for interactive user involvement in selecting experiments as part of an MES. The first category is for MESs with a large number of design spaces dimensions. In such MESs, it is often practical for users to explore and visualize only *slices* of the design space — a lower dimension

subspace — at a time. For example, a two-dimensional Pareto Frontier is easier to visualize and interpret than a five-dimensional one. However, the selection of slices is most efficiently directed by the user, because the user can base the slice-selection decisions on the study-level results of previous slices, and he can also react to incomplete results of a slice, e.g., he could choose a new slice before the exploration of the current slice is completed. By giving the user control over the slice selection, but retaining automatic exploration of a slice once it has been selected, the system enables interactive and efficient exploration of a design space that is otherwise too large for either the system or the user alone.

The second category of interactive user involvement is when the MES's region of interest is not clearly-defined. This happens often in MESs where a user's subjective sense is needed to define the region of interest, such as in our example animation study. In that setting, user's input is essential to the forward progress of the MES. These MESs would execute a number of scaled-down versions of the simulation, and present a group of candidate scaled-down results for the user to choose from. The user would review these results, and select one of the candidates as being worthy of further exploration. Based on the users' choice, the system would continue the study in that particular direction, adding more details to the chosen candidate in order to generate the next set of candidates. This way, the user has direct control over the direction in which to take the MES. Note that the system cannot make forward progress without the user: its job is simply to collect the candidates based on the user's preference.

In both categories of user involvement, the system is required to aggregate

the partial results (of the slice or of the collection of candidate results) and present the study-level result to the user, receive the user's decision, and change the direction of the MES by disseminating that information to worker processes.

5.2.2 Description

As described in Subsection 3.2.2, the defibrillator design study requires user interaction because of its high number of design space dimensions. So, it fits the first category.

In the defibrillator design study, the user continually selects slices of the design space for the system to explore, and SimX has to explore the slice and present the slice's Pareto frontier to the user at an interactive rate. SimX uses aggregation methods to display the Pareto frontiers to the users, allowing them to steer the exploration toward slices that are particularly promising. Furthermore, because the Active Sampler can provide a coarse-level approximation of the Pareto frontier early, users are given the choice to move to an alternative slice of the design space even though only an incomplete or fuzzy frontier is discovered for a given slice, further reducing computational cost. Both aggregation and fuzzy localization make the underlying computational study interactive: the user looks at the frontier as it is evolving and makes decisions about how to steer the study to refine specific slices of the design space to the desired accuracy.

The animation design study, described in Subsection 3.3.3, fits the second category. In the animation design study, SimX relies on the user to make forward progress. At all time during the study, SimX displays the current search tree to

the user, representing all the nodes it has explored, and the one it is currently exploring. The user is given the choice to *animate* any of the nodes in the search tree in order to see what each of the partially-completed scenes corresponding to the nodes look like. While the user is viewing the scene, SimX continuously explores the current node using the RandomSampler (4.2.2). At any point, the user can choose to animate another node, or tell SimX to explore another node, or just wait until more child nodes are available to view. Figure 5.2 shows one instant during the animation study. The search tree is printed in the background in text form, the UI of the buggy scene design SCIRun module is displayed as a dialog window, and the preview window that shows a partially completed scene is shown at the back. The user is currently exploring node 17, and has chosen to preview the partially-completed scene corresponding to node 51. He could now choose to continue exploring node 17, in which case he just waits and refreshes the tree, or animate a different node, in which case he would type in the ID of the node he wishes to preview and click the preview button, or explore a different node altogether, in which case he would type in the Node ID and click the "execute" button, which will update the SimXManager module (not shown) of a change in the current node. In addition, SimX gives the user the ability to backtrack in the search tree. When the user moves on to explore a different node, the SimX would store all intermediate information about the current node in a database, so that when the user wants to resume exploring the current node, SimX can retrieve the information and continue.

```
Exploring node: 17
Tree:
0
 1
 2
 3
  11
    25
    24
      33
      34
    23
  12
    13
    14
      15
      17
        51
        52
        53
|
```

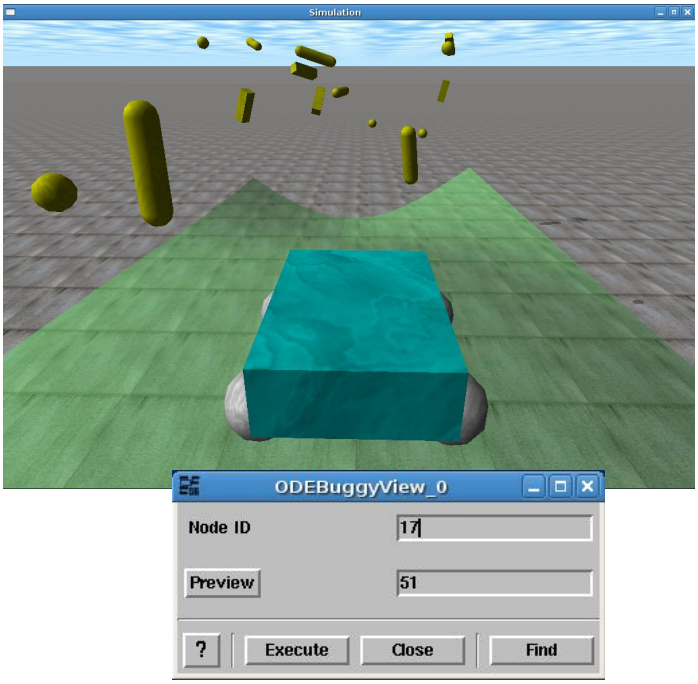


Figure 5.2: Animation designer steering the animation design study

5.2.3 Implementation

In between each user interaction in an interactive MES, the system is in effect conducting a mini-, non-interactive MES with a more confined design space. Each slice of the design space, or stage in a user-driven study, can be thought of as a mini-MES in itself — the system is still issuing experiments to explore a design space, but that design space is a constrained version of the design space of the overall study. The design space of the mini-MES is a subspace of the overall design space, constrained either by the currently selected slice (in the first category), or by the partially-completed candidate (in the second category).

Interactive MESs are supported only on the SimX/SCIRun version. In order to accommodate an interactive MES, the SimXManager module includes a meta data structure that stores *multiple* samplers. Each sampler is created as a result of the users' action. When the user performs an action, the manager first decides the specification of the non-interactive mini-MES that corresponds to that action. Then it looks up the sampler store to see if the same mini-MES had already been conducted before. If not, the manager will create a new sampler, store it in the sampler store, and start conducting that mini-MES. If the same mini-MES had been conducted before, the manager can look it up from the store, and continue that mini-MES from where it had left off.

As shown in Table 4.3, the StudyID data structure uniquely identifies a mini-Study's specification. Thus it is used by the manager to lookup and store the sampler objects. A new user action results in the manager creating a new

StudyID object, and invoking the SetCurrentStudy() call. The sampler for the mini-MES will be either created or looked up, and that sampler will be regarded as the current sampler — i.e., where the getNextPointToRun() call is made — until the next user action occurs.

In the defibrillator study, each slice is still being explored by the Active Samplers, so the sampler store contains a set of Active Samplers module instances. In the animation design study, each stage is explored by the Random Sampler, so the sampler store contains a set of Random Sampler module instances.

5.2.4 Challenges

The StudyID structure is designed to encapsulate all the information in the specification of a mini-MES, i.e., it captures all the information specified by a user's action. However, the remaining challenge is to translate the user's action translate into a StudyID object.

Study-level user interaction is only supported on the SimX/SCIRun version. It leverages SCIRun's user interface and component model to enable user interaction with SimX (see Section 2.4.1). SimX/SCIRun allows the user to steer the progression of an entire computational study by using an application-specific User Interface module. The UI module can 1) retrieve, aggregate, and display simulation results, and 2) collect user input and translate them into a study specification.

More specifically, the UI modules, implemented as SCIRun modules, use

the API in Table 4.2 (e.g, getPareto()) to retrieve aggregate results from SimX. The retrieved results are then displayed to the user via SCIRun's visualization components. The UI modules also use SCIRun's GUI mechanisms such as widgets and Tk/Tcl scripts to allow the user to specify, on the visualization screen, which slice he wishes to explore. Once a user makes a selection, the UI modules then package those preferences by creating a SCIRun matrix as described in Section 4.2.1, and send this matrix downstream to the SimXManager module's input port. The SimXManager then translates the matrix into a StudyID object, and uses SimX's SetCurrentStudy() to change the MES's direction.

5.3 Reusing Experiment Results

In Multi-Experiment Studies, many similar experiments are issued. Because the calculations are so similar, the result of one experiment — be it the final result, or an intermediate internal state of the simulation code — can often be used to speed up the execution of experiments whose input parameters are similar. If these results can be stored during one experiment, they can be retrieved later to be re-used during another experiment.

5.3.1 Principle

There are many sources of result reuse in MESs. The ability to reuse the results of earlier experiments stem from the numerical properties of the simulation code and the code structure of the simulation experiment. Thus, depending on whether an MES's simulation code has certain numerical properties

Type	Result Reused	Applicability
Checkpoint reuse	Simulation code output	Time-stepping simulation code, Iterative solver
Preconditioner reuse	Preconditioner	Iterative solver
Intermediate result reuse	Internal states of simulation code	Simulation code with shared intermediate states
Simulation result reuse	Simulation code output	Interactive MESs
Performance metric reuse	Evaluation code output	Interactive MESs

Table 5.1: Summary of result reuse types

or structure, it can take advantage of one or more forms of experiment result reuse. Table 5.1 summarizes the types of result reuse discussed in this section.

MESs made up of simulation codes that use an iterative solver to solve a non-linear system can often benefit from result reuse. Assuming that experiments that are close by each other on the design space are expected to have similar systems, the solutions to the two systems should be similar as well. Therefore, using the solution to the same system from an earlier experiment as the first guess to the iterative solver is expected to reduce the number of iterations needed to reach the solution. This results in a reduced number of iterations required by the solver, which reduces the run time of the later experiment, which in turn reduces the overall run time of the study. This form of result reuse takes advantage of the checkpointed result from a previous experiment, so it is called *checkpoint reuse*.

In addition, MESs made up of simulation codes that use an iterative solver

to solve a linear system can benefit from reusing the preconditioner. Assuming that the two systems of two close-by experiments are similar — in particular, if the stiffness matrices are similar — the preconditioner calculated for one stiffness matrix could be useful for the other system. The preconditioner from an earlier experiment, even though it is not the one calculated for the present experiment, can still reduce the number of iterations compared to the unpreconditioned system, but not as much as the "correct" preconditioner. However, as long as the cost of calculating the "correct" preconditioner outweighs the penalty of using the readily available but "incorrect" preconditioner, it will be beneficial for the MES to reuse the preconditioner from an earlier experiment. Since this technique reuses the preconditioner in a different experiment, it is referred to as *preconditioner reuse*.

Some other MESs are made up of time-stepping simulation codes. Usually, these MES require that the simulation codes to timestep until a certain condition is satisfied — e.g., when the system enters a steady state, or when the energy in the system dissipates. These MESs present another possibility of result reuse. Assuming that the solutions of two close-by experiments are similar, it is sometimes possible to use the final state of one experiment as the initial state of another. Depending on the numerical method, the timestepping simulation code can reach the same final state regardless of what the initial state is. If the simulation code has this property and the MES uses the final state of a close-by experiment (which has already reached the termination condition) as the initial state of another, the number of timesteps required by the later

experiment to reach the termination condition will be reduced. This speeds up the later experiment, and thus speeds up the overall study. This type of reuse also takes advantage of the checkpoint result from a previous experiment, and is a variant of *checkpoint reuse*.

Some MESs are made up of simulation codes with multiple intermediate states. These intermediate states often have specific physical meaning. Moreover, simulation codes close by to each other often share the same intermediate states. If these intermediate states from one experiment can be stored, they can be retrieved and re-used in a close-by experiment that is executed subsequently, thus saving the later experiment the need to re-calculate the intermediate state. We refer to this type of result reuse as *intermediate result reuse*.

One particular example of intermediate result reuse is applicable to all MESs that fits the formulation specified in Section 3.5. There, the output of the simulation code itself can be considered an intermediate state, because the simulation code's output needs to be fed into the evaluation code to produce the "real" result that the MES cares about: the performance metrics. In the presence of interactivity, it is possible that a user could create a new mini-MES by changing only the performance metric parameters. In those cases, the inputs to the simulation code remain the same, but only the evaluation code need to be re-executed. If the outputs of the simulation code are stored for the first mini-MES, some of them can be reused in the second mini-MES, saving the time it takes to re-execute the simulation code. This type of result reuse is referred to as *simulation result reuse*.

Also due to user interactivity, sometimes the system needs to execute the same experiments for different mini-MESs. This could happen if the user merely shifts the region of exploration, but keeps the same slice. Then some experiments in the overlapped region could be issued by the new mini-MES, but may have already been completed by the first mini-MES. A MES system that stores all experiment results generated by any mini-MES can simply look up those results instead of issuing them to the worker processes. We refer to this special type of result reuse as *performance metric reuse*.

As may be apparent, the levels of reuse range from generic to specific, with performance metric and simulation result reuse being most generic: any interactive multi-experiment study fitting into the formulation described in Section 3.5 can take advantage of them. Checkpoint reuse, intermediate result reuse, and preconditioner reuse require the simulator code to store and retrieve information specific to the application.

We note that although the specifics may differ somewhat, the overall notion of result reuse is broadly applicable beyond our context of computational studies. Any parallel workload which involves the repeated execution of similar kinds of computation can benefit from a similar approach, where one focuses on the "delta" of computation that needs to be performed beyond what already exists as opposed to computing the result from scratch.

5.3.2 Description

Examples of result reuse can be found in the example bridge design study, defibrillator design study, and animation design study.

The bridge design study is an example of an MES made up of simulation codes that use an iterative solver to solve a non-linear system. Here, the simulation code spends the bulk of the time in a Newton iterative solver. Normally, the first guess to the Newton solver is a zero vector. However, if we consider two simulations that are close by on the design space — i.e., the placements of the columns are close by — the stiffness matrix and RHS to the non-linear system should look similar, with only a few rows that are different. Therefore, if we used the solution of one system as the initial guess of another, we can reduce the number of Newton iteration required to solve the second system. Our experiments have shown an order of magnitude decrease in the iterations needed when checkpoint reuse is enabled.

The defibrillator design study presents an example with intermediate result reuse. As discussed in Subsection 3.2.1, the DefibSim simulation code solves three linear systems, each corresponding to setting the boundary condition of the front electrode to the three surface mesh nodes nearest to it. In Figure 5.3, for example, two front electrode placements, one represented in red and one in green, are explored on the torso's surface mesh. To solve the body potential caused by the green placement, DefibSim needs to solve the three system surrounding it, namely, $A_a x = b_a$, $A_c x = b_c$, and $A_d x = b_d$. Each of the stiffness matrices A and RHSs b corresponds to placing the boundary condition on the

corresponding nodes. Once the three solutions x are calculated, a weighted average is taken, based on the distance between the mesh node and the placement. As is apparent, when the user wishes to explore the placement represented in red, DefibSim has to solve two of the same systems as in the green placement. By storing the solutions of these systems in SISOL, and the simulation code can look up the solutions and skip solving those systems that had previously been solved.

As an interactive MES, the defibrillator design study also provides an example of simulation result reuse. A new slice of design space could be selected by the user changing only the performance metric parameters. E.g., a user may fix the back electrode placement and voltage shock, select an area for the placement of the front electrode, and discover the Pareto frontier of the front electrode placement. Then he may raise the activation level to try to find the Pareto frontier for a patient whose heart is harder to activate. In these cases, many of the simulations are already done when the user was exploring the first slice because the input to the simulation code — placements of electrodes and shock voltage — remain the same across the two slices. The only difference between the two slices are the performance metric parameters, which only affects the behavior of the evaluation code. The Simulation Container can detect when such cases occur and by-pass executing the simulation code.

Besides multiple intermediate states, each solve of the system $Ax = b$ in the defibrillator design study uses an iterative solver. Thus, it is possible for the defibrillator code to reuse the preconditioner from one system for another.

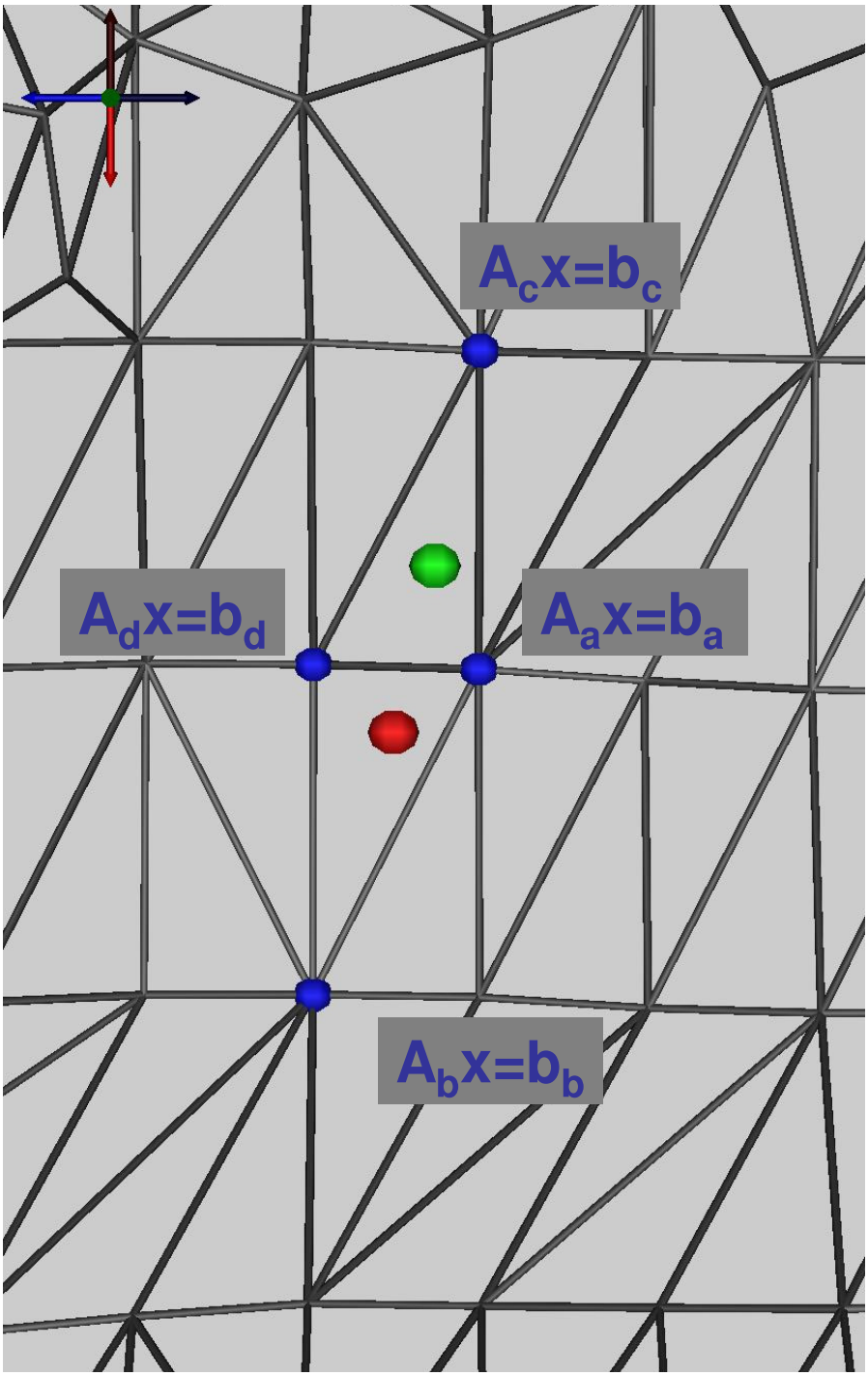


Figure 5.3: Intermediate result reuse in DefibSim

The helium model validation study provides an example of checkpoint reuse based on timestepping code. As discussed in Section 3.4, Arches's execution time depends on the per-timestep run time and the number of timesteps required to reach a steady state. An inherent characteristic of Arches's timestepping method is that, if two simulations with the same inlet velocity are run, it is possible to use one simulation's final state as the initial condition of another. By using the first simulation's final state instead of the default initial condition, the second simulation can reach its steady state with fewer number of timesteps. As we describe in Chapter 6, on average, without reuse, Arches takes 2900 timesteps to reach the stable state; with reuse, it takes 1641 timesteps. However, as pointed out above, only simulations with the same inlet velocity can reuse each other's results. We refer to experiments that can reuse each other's final states as belonging to the same *reuse class*.

5.3.3 Implementation

Result reuse is mainly supported using the SISOL component in SimX. In all the reuse scenarios described above, the system needs to store intermediate results across different executions of the computational experiments. Since the SISOL implementation is supported on all three versions of SimX, result reuse is a technique that can be exploited in all versions.

To take advantage of result reuse across the MESs, the reusable objects are grouped into SISOL object sets. E.g., all preconditioner objects created in an MES belong to one object set, all iterative solver's solution belong to another

object set, and all outputs of the simulation code belong to a third object set, etc. Object sets are assigned, *a priori*, an object set ID.

SISOL's spatial index naming system is well-suited to distinguish objects within an object set because the objects are created by the execution of simulation code using inputs from a point on the design space, so the spatial coordinate of the design space point can uniquely identify an object within an object set. Therefore, the <object set ID, spatial index> tuple can uniquely identify an object.

In some types of reuse the exact coordinate of the object to be reused is unknown. E.g., to take advantage of preconditioner reuse, the Simulation Container needs to find the preconditioner calculated for the closest numerical system, but doesn't know the exact design space coordinate of that system. The spatial index system supports neighborhood queries, which allows for nearest-neighbor lookups, becomes useful for such reuse types.

In the bridge design study, when the Simulation Container finishes executing the bridge deformation code in the bridge design study, the simulation result, which is a PETSc vector, is written into a `ResultSet` in SISOL (with ObjectSet ID of 0), using the design space parameter of the simulation as the spatial index. When the Simulation Container executes the next experiment, it looks through the previously-stored `ResultSet` in SISOL using NearestNeighbour query to find the closest solution.

In the defibrillator design study, intermediate state reuse is realized using the SISOL SimX/SCIRun modules. The relevant simulation net is shown in

Figure 5.4. The coordinates of the surface node, along with the position of the back electrode and the strength of the voltage shock, is calculated by a collection of modules upstream (top right) and used as the spatial index of the SISOL-MatrixReader module. The module uses the spatial index and attempts to read an object associated with that spatial index from the IntermediateResultSet (it has the Object ID of 1) from the SISOL layer (top). The result is passed down to the ConditionalExecute module (middle left). The ConditionalExecute module will cause the solver sub-net on the right to fire only if the input matrix port is empty (i.e., the intermediate result isn't already stored in SISOL). Otherwise, it passes the result of the SISOL lookup directly to the module downstream (bottom right). On the solver sub-net, a module constructs the stiffness matrix and RHS, and the two are passed into SCIRun's SolveLinearSystem module (right). The SolveLinearSystem solves the linear system, and fires the solution on its output port. The solution is written to SISOL for future reuse (bottom right), and also passed back to the main workflow net (bottom right).

In the helium model validation study, since Arches already writes the checkpointed result onto disk, the Simulation Container only needs to write the disk location of the checkpoint into the `ResultSet`, using the experiment's design space point coordinates as the spatial index. Later on, when the Simulation Container is about to executes another simulation, it uses SISOL's nearest neighbor query to extract the simulation result closest to the simulation it is about to run. During result reuse, the Simulation Container needs to additionally check that the retrieved result belongs to the same reuse class. The "nearest

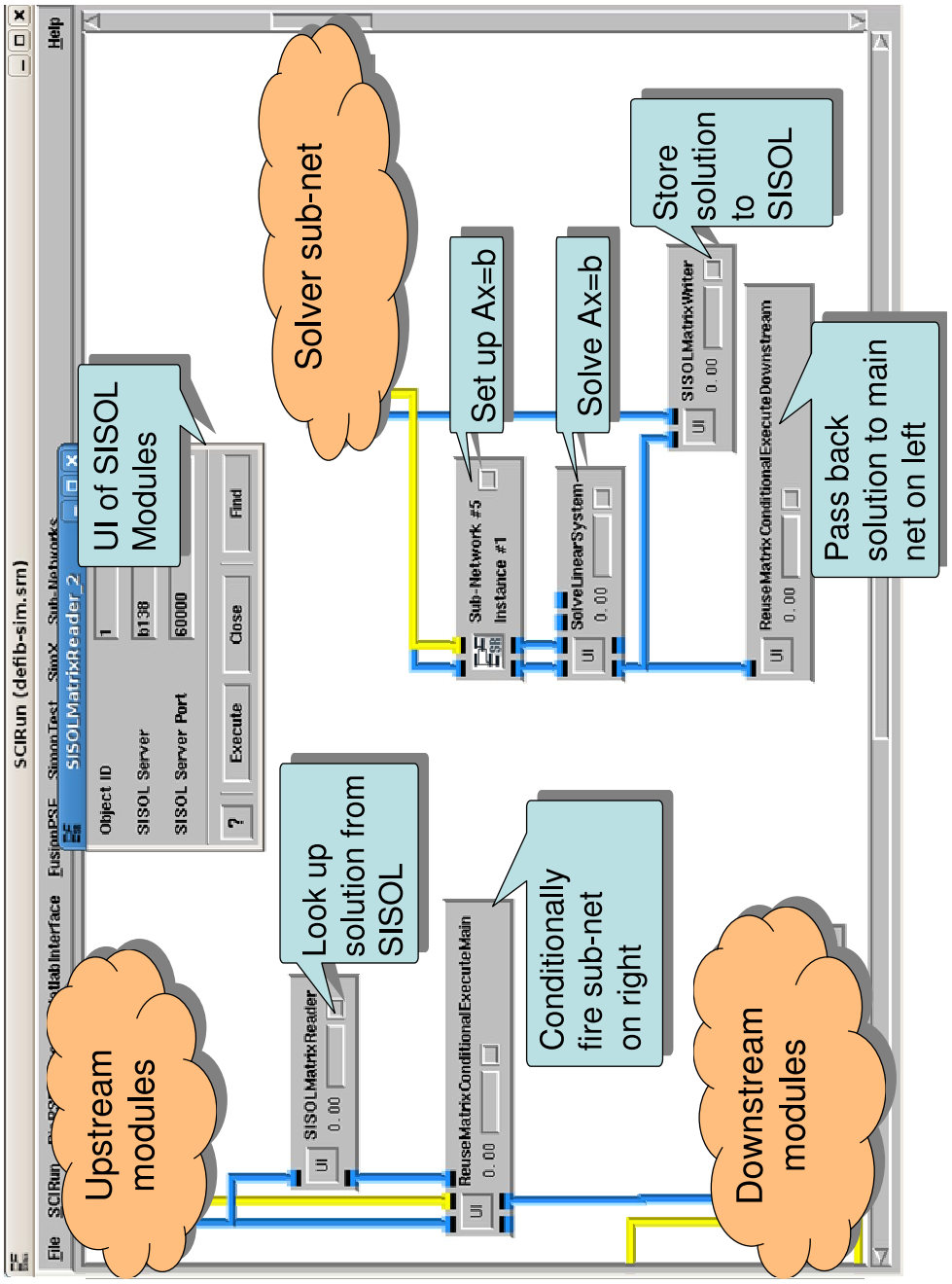


Figure 5.4: DefibSim net conditionally executes the solver subnet

neighbor" query in helium model validation is therefore heavily weighted for the Inlet Velocity dimension in order to guarantee that, if a result with the same inlet velocity already exists, the nearest neighbor query will return it first.

5.3.4 Challenges

With result reuse, during each execution of the simulation code, multiple objects can potentially be written into SISOL: all the intermediate results, the output of the simulation code, preconditioners and solutions of every system solved. The network requirements can therefore become arbitrarily large.

The latency problem can be mitigated by overlapping the writing of SISOL objects with the execution of the *next* experiment. Since the correctness of the MES is not affected by delaying the writing of SISOL objects, it is possible for the Simulation Containers to receive the next experiment without waiting till the SISOL objects associated with the previous experiment are all written. The penalty for delaying the writing of SISOL objects is that the objects may not be available for reuse as early. Therefore, there is a trade-off between making reusable objects available as soon as possible and taking a performance delay of waiting for the objects to be written to the SISOL layer. The writing of SISOL objects should be delayed to overlap with the execution of the next experiments if the performance penalty of waiting for the SISOL layer is higher than the performance penalty of not having the benefits of the reuse of all of the objects. It is highly dependent on the MES: the wait time of SISOL writes is a function of the object sizes, but the performance gain of having reuse is a function of the

characteristics of the problem.

Even though the latency problem can be mitigated on the worker processes by hiding it with the next experiment, the SISOL layer can still become a performance bottleneck on the server. In runs consisting of a large number of worker processes, all processes would simultaneously try to read and write multiple objects into SISOL. The SISOL servers would have to service more simultaneous requests for checkpoints, and the SISOL data servers could be overwhelmed by requests and become a performance bottleneck. At an inflection point, the time that worker processes spend waiting for the SISOL data server could be large enough that the reduction in per-experiment run time is negated. SimX deals with this challenge by using multiple SISOL data servers, and uses the single directory service to direct traffic. Worker processes need to lookup from the directory service which data server an object resides on, and perform the actual transferring of objects by directly communicating with the data servers. This way, the bandwidth requirement to any one data server is reduced to a level that does not impede the worker processes' progress, and the bandwidth requirement of the single directory server is minimal, because it only serves object IDs and server IDs. As the number of worker process increases, it is possible to simply add more data servers to retain the per-data server load.

As discussed in Subsection 4.2.7, SISOL data servers store objects in volatile memory. If the size of objects stored on a data server exceeds its memory capacity, the server will experience a severe performance degradation due to thrashing. To prevent data servers from exceeding their capacity limit, SISOL

enforces a per-object-set per-data-server cap on the number of objects stored. The cap is specified by the user, and is dependent on the data server's memory capacity. The user can start as many data servers as needed in order to obtain the overall storage capacity required by the MES.

5.4 Reusing Study-level Results

As discussed in Section 5.3, an individual experiment's simulation result may be reused to speed up another experiment if the two experiments are close to each other. In the same vein, in an interactive MES, if two design space slices are close to each other, the aggregate results of one mini-MES may be used to speed up the execution of another mini-MES.

5.4.1 Principle

In an interactive MES, when the user elects to explore two design space slices that are close to each other, their regions of interest are expected to be similar. The two mini-MESs may have identical specifications but a slightly different performance metric parameters, or their fixed design space dimensions maybe slightly different, or their specifications are the same, but the regions of exploration overlap with each other.

In such cases, it may be beneficial to use the region of interest in the first mini-MES as the starting point for the exploration of the second mini-MES. Recall that, in the Active Sampling strategy (Subsection 5.1.2), the aggregate result of early experiments, which are issued on design space points correspond-

ing to a coarse lattice, is used to guide the exploration of later experiments, which correspond to a finer lattice on the design space. If there is an existing mini-MES that has completed, one could use the region of interest of this other mini-MES, instead of experiments on the coarse lattice, to guide the exploration of the finer lattice. This way, the system can skip having to re-issue the experiments corresponding to a coarse grid, and immediately work on exploring the fine-level grid. This can potentially enable the system to focus on the "interesting" region of the second mini-MES sooner.

By using the first mini-MES as a guide, this strategy allows the system to by-pass the coarse-level exploration of the second mini-MES, and potentially resolve the region of interest with fewer number of experiments.

5.4.2 Description

In the defibrillator design study, if a user performs an action that causes the system to switch from exploring one slice to another slice (e.g., changes a performance measure parameter such as the activation threshold), the system can take the approximation of the Pareto frontier obtained in the first mini-MES to act as the initial Pareto frontier approximation for the second, reducing the total number of experiments needed.

This type of reuse effectively uses the Pareto frontier of the first slice as the initial approximation to the Pareto frontier of the second, thus removing the need to run the coarse-level exploration in the second mini-MES. This type of reuse is particularly beneficial in situations where the second mini-MES is a

refinement of the first (as is typical for design space exploration).

5.4.3 Implementation

Since study-level reuse is only applicable to interactive MESs, study-level reuse is only supported on the SimX/SCIRun version. As discussed above, the SimXManager keeps a meta data structure to store multiple samplers. To enable study-level reuse, these samplers are indexed in a spatial database, using the specification of the mini-MESs as the spatial index. (This database is not SISOL, because it is not accessible from worker processes).

When the user performs an action that creates a new mini-MES, SimX translates its specification into a spatial index, and looks up the closest mini-MES from the database. If it determines that the two mini-MESs are "close enough", it can query the existing mini-MES's sampler for the region of interest it has discovered. When it creates the sampler for the new mini-MES, it uses the old sampler's result to "seed" the new sampler. This "seeding" of the second mini-MES's sampler is accomplished by the SetSamplerOptions() call in the SimX sampler interface. This way, the new sampler will by-pass the coarse-level exploration and immediately begin exploring its design space at the finest level.

5.4.4 Challenges

The difficulty with this approach is that there is no easy way to know, *a priori*, whether two mini-MESs are close enough to each other to make study-

level reuse worthwhile. To deal with this problem, in the current implementation, SimX uses user-defined heuristics to decide whether to enable study-level reuse. The user can specify the distance between two mini-MESs (as defined by the spatial indexes of their specifications) that the mini-MESs must fall within in order for SimX to enable study-level reuse.

5.5 Parallelism-driven Resource Allocation

In computational studies where the simulation code can be run in parallel, the system has the ability to decide how many, and which, processing elements are assigned to each simulation. This is especially important in scenarios where the number of processing elements is greater than the number of experiments.

5.5.1 Principle

In general, most parallel codes do not scale perfectly because of parallelization overhead. The non-linear scaling behavior has several implications.

Firstly, in order to minimize parallelization overhead within each process group, the system should aim to schedule "long-and-thin" scheduling graphs — many executions running concurrently on relatively small process groups. However, when the number of processing elements is not a multiple of the number of outstanding experiments, the processing elements cannot be divided evenly among the experiments, so the question of how many and which experiments get higher priority becomes an issue.

In this thesis, experiments are grouped into *batches*, and only the exper-

iments inside the same batch are allowed to, but not required to, execute concurrently. The algorithm to select a batch from a pool of experiments to be run is called the *batching policy*. By controlling which experiments can be grouped into the same batch, the system can adapt its parallel scheduling algorithm to suit the requirements of the MES it executes.

The batching policy is influenced by a variety of factors, all of which are specific to the MESs being conducted. These factors include sampling policy, scaling characteristics of the simulation code, and result reuse. This section discusses the influence of sampling policy and scaling characteristics; the influence of result reuse is discussed in Section 5.6.

A second implication is that, when multiple experiments are issued concurrently, some experiments might finish earlier than others. What to do with the freed-up processing elements becomes an issue. Does the system issue new experiments onto them? Or does the system expand existing experiments to fill them out?

It is not clear at the outset whether *preempting* an ongoing simulation and expanding it onto the idle processes will be beneficial — this depends on the preemption's overhead and the parallelization overhead of the simulation code. In SimX, the user is given control over whether preemption happens. In the default implementation, SimX provides a simple heuristic explained below.

No. of CPUs	4	8	16	32	64
Run time (s)	1.95	1.14	0.80	0.60	1.82

Table 5.2: Arches scaling behavior: run time per timestep

5.5.2 Description

The most important factor influencing batching policy is the scaling behavior of the simulation code. To explain how the batching policy works, consider the helium validation study. The time it takes Arches to execute a single timestep on different numbers of processors is shown in Table 5.5.2.

The parallelization overhead of the simulation code is used to decide the optimal batching policy. For example, consider a case when 32 processors are assigned to conduct a study, and 6 simulations with the scaling behavior shown in Table 5.5.2 need to be run. Assume for the moment that they all take the same exact number of timesteps, n. The optimal batching strategy would be to issue two simulations concurrently first, and then issue the remaining four. That way the total time will be $0.8n$ seconds to execute the first batch (each simulation getting 16 workers), and $1.14n$ seconds to execute the second batch (each simulation getting 8 workers), totaling $1.94n$ seconds. However, if there are 7 simulations to be run, it's better to issues all 7 in a single batch, and finish in $1.95n$ seconds (each simulation getting 4 workers, with 4 workers left idle).

The optimal way of grouping experiments in batches can be found by dynamic programming. Let $T_{m,n}$ be the time to complete n jobs with m batches or fewer. $T_{1,j}$ can be looked up from Table 5.5.2.

Then, for $2 \leq m$:

$T_{m,n} = min_{1 \leq i \leq n}(T_{m-1,i} + T_{m-1,n-i})$

Here, i represents the "dividing point" of the batch: the n jobs are divided into two sub-batches of i jobs and $n - i$ jobs, which themselves could be further sub-divided. The optimal way to schedule the n jobs can then be looked up from the dividing points used to calculate the $T_{m,n}$ table.

Unfortunately, the "ideal" batch sizes assumes that every simulation executes the same number of timesteps. Since simulations do require different timesteps, particularly when SimX employs optimizations such as checkpoint reuse, there will always be "holes" left on the scheduling graph.

This is where preemption becomes important. Like most checkpointing-capable MPI applications, Arches is not only moldable (can run on different number of processes) but also malleable (can adjust to changing allocations at run time). Arches leaves checkpoints periodically, so when idle worker processes are detected, it is possible to preempt an ongoing simulation, assign the idle processes to the process group that has been working on the simulation, and restart the simulation from the last checkpoint using the resulting process group, now with a larger number of worker processes. This preemption policy fills up the "holes" on the scheduling graph by "spreading out" existing simulations, at a cost of having to re-do the work that the original process group has done from the time it left the last checkpoint up to the time it was preempted.

In the default implementation of SimX/Uintah, the scaling behavior is used to decide when it is beneficial to preempt an existing simulation. Suppose

a simulation has completed a fraction σ of work between the last checkpoint and the next ($0 < \sigma < 1$), and some worker processes become available at that time. Let α be the speedup ratio between the original process group and the expanded process group (i.e., for perfect scaling, α = original group size / expanded group size). Then, in order for the preemption to be beneficial, the time it takes for the new process group to advance to the next checkpoint from the last checkpoint must be smaller than the time it takes for the original group to advance to the next checkpoint from where the simulation currently is (otherwise, the preemption should performed at the next checkpoint). Therefore, for preemption to be beneficial, the following inequality must hold:

$\alpha < 1 - \sigma$.

To estimate σ, we assume that the number of timesteps per simulated second is the same, so σ is just the fraction of completed simulated time and expected simulated time between checkpoints.

5.5.3 Implementation

Support of parallelism-driven resource allocation requires co-operation between SimX/SCIRun's manager process and SimX/Uintah's worker processes.

To enable batching, the priority queue version of the Task Queue module must be used. During initialization, the priority queue version Task Queue loads the scaling behavior of the simulation code, and calculates the ideal batch sizes using the dynamic programming algorithm mentioned above.

During the study, the SimXManager "looks ahead" at the experiments

that the sampler wants to issue. It calls getNextPointToRun() repeatedly until the sampler cannot issue more simulations without knowing more evaluation results. It calls the Task Queue's AddTask() interface to store the return values of the sampler. Once the experiments are stored in the Task Queue, SimXManager calls CreateBatch() to gather the next group of experiments to be issued concurrently. The Task Queue then tries to batch the tasks according to the ideal batch sizes.

This CreateBatch() call is customizable, so the batch creation algorithm can be adapted to the particular requirements of the MES. In the default version, the call simply picks a number of random experiments from the queue to match the number of ideal batch size.

Once the SimXManager determines the batch membership, it calls Task Queue's GetIdealGroupSize() interface to determine the ideal number of processors to assigned to each group. Normally the Task Queue would divide the processing elements evenly among the processing elements, *a la* the Fair Share policy used in traditional parallel systems with moldable jobs ([76, 77]). Then SimX creates an assignment vector and calls the Resource Allocator's reconfigure() function to ensure that the processing elements arrange themselves into processor groups with the ideal group size. From there on, the rest of the system would assign the experiments to the worker process groups concurrently.

To help carry out the preemption strategy, a database is kept in SISOL to keep track of idle process groups. Every time a process group finishes its simulation and is not assigned a new one, the SimX manager process writes the

process group's information about its processors into the database.

As discussed previously, while Arches runs, it may periodically return control to the Simulation Container. When that happens, the Simulation Container would look for idle worker processes on the SISOL database and decides whether to claim them according to the rule described in the last section. If it decides to claims the processors, it sets a flag in the database, terminates Arches early, and sends back a "claim ticket" to the manager process. On the manager process, when the Resource Allocator recognizes a claim ticket, it sends a reconfiguration directive to the claimed processes. The claimed processes and the original worker processes then form themselves into a new process group, which continues the original simulation from the last checkpoint left on disk.

5.5.4 Challenges

To enable the default optimal batching policy, SimX needs to know the scaling behavior of the simulation code. In the current implementation, it requires the user to explicitly specify the scaling behavior of the simulation code, with the expectation that offline profiling is performed with the simulation code to prepare this table. A more sophisticated implementation of SimX, however, should be able to infer the scaling behavior automatically by performing test runs of Arches during the study's initialization.

5.6 Scheduling to Maximize Reuse Potential

As discussed in Section 5.5, the optimal batching policy is determined by the scaling overhead of the simulation code. However, it may also be influenced by concerns of result reuse. As discussed above, not all simulations can reuse all other simulations' results. For example, in the helium model validation MES, only experiments in the same reuse class can reuse each other's result. In order to maximize the reuse potential of experiments (and thus reduce the overall study run time), a parallel system conducting an MES needs to schedule experiments in a way that enables the most result reuse to take place. The batching policy is the most direct way to accomplish this.

5.6.1 Principle

Because an experiment can only reuse a result from another experiment that has already been executed, the system should ideally avoid executing groups of experiments concurrently with the following characteristic: the experiments within the group can potentially reuse each other's results, but there are no previously-completed results in the system that the experiments can reuse. If the system executes those experiments concurrently, all of the experiments would be unable to reuse any result, because at the time of issuing, none of the experiments have been executed yet. Yet, if the system schedules one experiment from that group first, and then issues the rest concurrently, all of the rest of the experiments would be able to reuse the first experiment's results.

5.6.2 Description

In the helium model validation study, the knowledge of reuse classes is incorporated into scheduling decisions.

At any one moment, there is a set of experiments that are determined by the sampling algorithm to require execution. Of those experiments, some will be the first of their reuse class seen by the system, so, there are no checkpoints stored in the system whose reuse can benefit the experiments. In order to ensure that reuse potential is maximized, the MES needs to ensure that the result store is populated by checkpoints from all reuse classes as early as possible. To do so, it would select one "representative" out of each first-time reuse classes, and give them the highest priority. This ensures that these experiments are executed ahead of the remaining experiment in the reuse class, and possibly concurrently with the high-priority experiments from other reuse classes. This way, the system can ensure that all subsequently-issued experiments have a result to reuse.

Another benefit to this policy is that it ensures that all of the experiment issued concurrently either all have a checkpoint to reuse, or all do not have a checkpoint to reuse. Since experiments with checkpoints are faster, the experiments issued together will have more comparable run times, limiting the sizes of "holes" on the scheduling graph.

5.6.3 Implementation

In order to ensure that the maximum reuse potential is realized, the user can customize the implementation of the CreateBatch() call in the Task Queue module to modify the batching policy.

In the helium study, the Task Queue keeps track of all completed experiments. During the CreateBatch() call, it scans the list of unissued tasks, groups the tasks into reuse classes (i.e., those with the same inlet velocity), and finds out which reuse classes do not already have a representative in the completed experiment set. If there is at least one such resuse class, the Task Queue selects one experiment from each previously-unseen reuse class and adds the experiment to the batch. Otherwise, it adds all experiments on the queue to the batch. In effect, it searches for first-time reuse classes, and executes one representative from each concurrently with highest priority. This way, subsequent batches are guaranteed to find completed experiments from their reuse classes.

This strategy ensures that the system has the most complete set of results for reuse, but at the same time maximizes the number of experiments concurrently executed.

5.7 Summary

In this chapter, a number of system technique are presented that would help MES reach its goal of user interactivity and reduced run time. They leverage application-specific knowledge to help the system make better decisions to decrease the number of experiments required, reduce the per-experiment run

time, and improve resource utilization rate.

Table 5.3 summarizes these techniques, the application-level knowledge each requires, the source of benefits, and the studies that each technique is applicable to.

In Chapter 6 we investigate the performance benefits of these techniques obtained in our four example MESs in this thesis.

Techniques	Knowledge Required	Benefits	Applicable to:
Sampling	MES type	Reduce number of experiments	Bridge Design Study, Defibrillator Design Study, Helium Model Validation Study
User Direction	MES type	Reduce number of experiments, Enable Subjective Steering	Defibrillator Design Study, Animation Design Study
Result Reuse	Intermediate results	Reduce per-experiment run time	Bridge Design Study, Defibrillator Design Study, Helium Model Validation Study
Study-level Result Reuse	MES type	Reduce number of experiments	Defibrillator Design Study
Parallelism-Aware Scheduling	Scaling behavior	Improve resource utilization	Helium Model Validation Study
Reuse Maximization	Reuse classes	Reduce per-experiment run time	Helium model Validation Study

Table 5.3: Summary of application-aware system techniques

Chapter 6

Evaluation

In order to evaluate the approaches and techniques described in Chapter 5, we performed the four example Multi-Experiment Studies using SimX on two available computing clusters. This chapter presents and discusses the experiments and results of each study. Specifically, it discusses what part of application-level knowledge in these MESs helped improve their run-time and interactivity performances, and how.

In each of the following four sections, a recap of an example MES is given, then we describe the experiments conducted using the MES to evaluate the techniques presented in this thesis, and the results of those experiments are next presented and discussed.

6.1 Bridge Design Study

The bridge design study is a Pareto Optimization MES, with a two-dimensional design space and two-dimensional performance space. It is non-interactive, and its simulation codes are run serially. It demonstrates how an MES can benefit from the use of Active Sampling (Section 5.1) to reduce the

overall number of executions needed to complete the study, as well as the reuse of the final checkpoint of one simulation to speed up the execution of another (Section 5.3).

The experiments for this MES aims to 1) measure the overall benefits of the techniques used in SimX, 2) quantify the benefits of the active sampler technique for Pareto optimization MESs, 3) quantify the benefits of result reuse, and 4) measure the scalability of the resulting system.

6.1.1 Methodology

The bridge design study was conducted 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 via a Myrinet network. The standalone version of SimX is used both on the manager process and the worker processes. A single manager process is run on the front end machine, a varying number of SISOL data server processes (from 1 to 4) are run on the cluster nodes, and a varying number of worker processes (from 1 to 128) are run on the cluster nodes. All processes are assigned their own physical nodes, so there is no one physical node that is shared among SISOL data processes and worker processes.

To measure the benefits of the techniques described in Chapter 5, we conduct the bridge design study on the above system using four configurations.

The **GS** configuration (standing for Grid Sampler) represents the baseline comparison. In this configuration, the system issues 102400 experiments, covering a 320x320 grid on the design space.

The **AS** configuration (standing for Active Sampler) is used to measure the benefits of the active sampling technique. In this configuration, the system uses the Active Sampler module, beginning with a 40x40 grid and refining it three times. This results in a Pareto Frontier equivalent to the one resolved by the grid sampler on a 320x320 grid.

The **GS+C** (standing for Grid Sampler with Checkpoint reuse) configuration uses the same grid sampler as the GS configuration, but also enables the reuse of previous results following the techniques described in Section 5.3.

The **AS+C** (standing for Active Sampler with Checkpoint reuse) configuration uses the Active Sampler as the AS configuration, but also enables result reuse.

The overall benefits of these techniques can be seen by comparing the GS and AS+C configurations. We compare the Pareto Frontiers obtained by these two configurations to ensure the application-specific techniques are correct. We also measure the improvement in run time.

We then look closer at the source of these benefits. We quantify the benefits of active sampling in two ways. To quantify the benefits of active sampling in terms of reducing needed experiments, we compare the number of experiments required by the GS and AS configurations. To quantify the benefit of active sampling in terms of providing early feedback to the user, we measure how quickly the partial Pareto Frontier approximation of the GS+C and AS+C converge onto the final answer over time. To quantify the benefits of reusing checkpoints, we compare the run times — overall runtime, per-level runtime,

and per-simulation run time — of the AS and AS+C configurations. Finally, to measure the scalability of the system, for each of the GS, AS, and AS+C configurations, the study is conducted using 1 to 128 worker processes, and the response time of the system is measured.

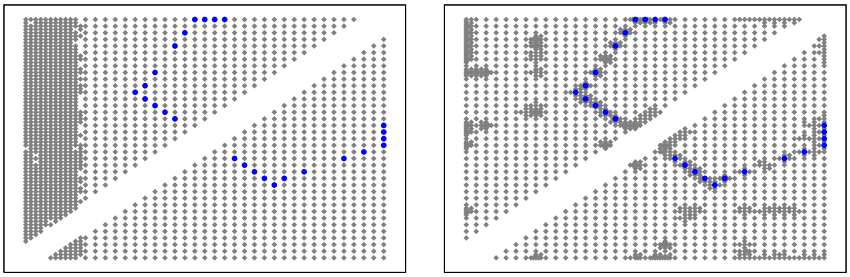
6.1.2 Results and Analysis

6.1.2.1 Overall Benefits of Application-Level Knowledge

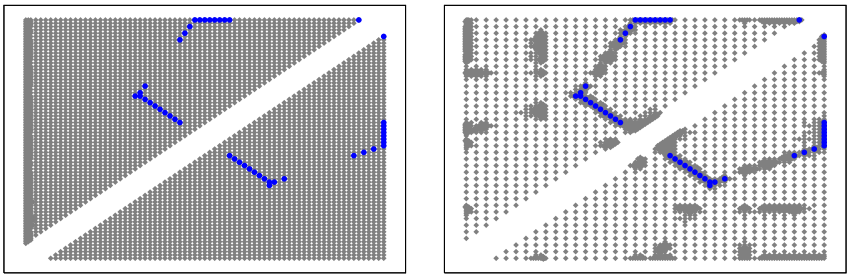
We compare the GS and AS+C configurations to demonstrate the overall benefits of application-specific knowledge in the bridge design MES.

First, to demonstrate the correctness of AS+C, we examine the result provided by GS and AS+C configurations. They are presented in Figure 6.1. Figure 6.1 shows two series of plots on the design space, representing the evolution of the Pareto Frontier in the two runs. The evaluated points are colored in grey, and the Pareto Frontier is colored in blue. On the left is the frontier discovered by the GS configuration, and on the right is that discovered by the AS+C configuration. The Pareto Frontier (blue dots) are the same on both sides, which means the AS+C configuration yields the same results as the GS configuration, so the correctness of the system is preserved.

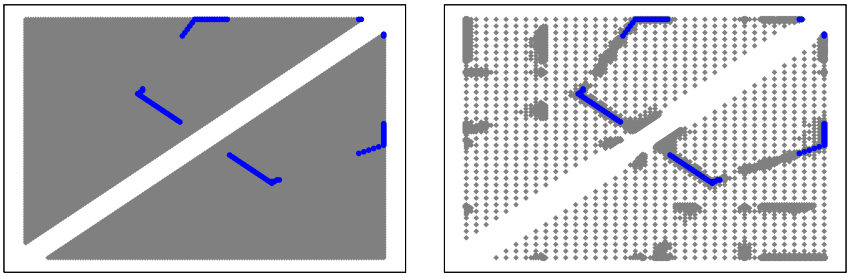
Table 6.1 shows the time required to achieve the frontiers shown in Figure 6.1 using 128 worker processes. For the GS configuration, the time it takes to generate the Pareto frontier grows as the square of the level of refinement. At the finest level, it takes 5678 seconds to resolve the frontier. For the AS+C version, it only takes 13.6 seconds to resolve the same frontier. One can conclude



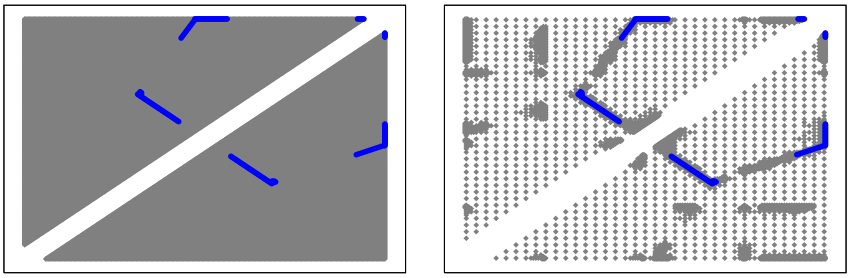
First Level Pareto Frontier Approximation



Second Level Pareto Frontier Approximation



Third Level Pareto Frontier Approximation



Final Level Pareto Frontier Approximation

Figure 6.1: Time evolution of the Pareto frontier in Bridge Design study. Naïve run shown on left, application-level knowledge enabled run shown on right.

Configuration	Time (in seconds) to reach level			
	1	2	3	4
GS	97.48	360.91	1407.76	5678.22
AS+C	6.62	8.9	12.26	13.63

Table 6.1: Bridge design study runtime on 128 worker processes

that, by combining active sampling and checkpoint reuse, SimX can reduce the run time of the bridge study by more than 400-fold.

The following subsections look closely at the source of this performance gain, by studying separately the benefits of active sampling, result reuse, early feedback, and scaling.

6.1.2.2 Reduced number of experiments issued

In Figure 6.1, it can be seen, qualitatively, that the AS+C configuration issues fewer experiments — the sparsity of grey points on the right shows that active sampling successfully prunes uninteresting regions of the design space early on in the study. This subsection looks quantitatively at the reduction in experiments issued due to active sampling.

As discussed in Subsection 5.1.4, the AS and AS+C configuration of the bridge design study uses a full-loading variant of the Active Sampler. Depending on the amount of parallelism, the Active Sampler only requires 4000–5000 experiments. As the number of worker processes increases, the overhead due to wasted experiments also increases.

Table 6.2 shows the total number of experiments issued by the AS+C

Num. of Worker Processes	1	2	4	8	16	32	64	128
Num. of Experiments	4125	4224	4284	4353	4380	4394	4392	4490

Table 6.2: The number of experiments required by the AS+C configuration

configuration. Compared to the GS configuration, which takes a naïve approach of issuing all experiment on the finest grid (i.e., $320^2 = 102400$ experiments), the Active Sampler issues two orders of magnitude fewer experiments.

We have also experimented with the original non-optimistic variant of the Active Sampler, which kept the number of experiments issued at 4125, but due to dependency between experiments of different levels, gave a slightly worse overall run time performance.

6.1.2.3 Improved per-simulation run time by result reuse

As explained in Section 5.3, the bridge simulation code can take advantage of the result of a previous run to reduce the number of Newton iterations required. The solution of the Newton solve from one experiment can be used as the initial guess of another experiment. Moreover, we expect that, the closer the two experiments are on the design space, the closer the two solutions will be, and thus the greater the benefits of result reuse.

This benefits of result reuse is demonstrated in two ways. We first compare the time it takes to discover the first-level Pareto Frontier with (AS+C configuration) or without (AS configuration) checkpoint reuse turned on. Without

checkpoint reuse, to resolve the first level Pareto Frontier, it takes 97.5 seconds and 1735 experiments on 128 worker processes, i.e., 7.2 seconds per simulation. With reuse, it takes 6.62 seconds and 1725 experiments on 128 worker processes, i.e., 0.49 seconds per simulation. Using the checkpoint of a close-by experiment to jump-start another one yields an order of magnitude improvement in per-simulation run time.

The second way to demonstrate the benefits of checkpoint reuse is through the reduction of the per-simulation run time as the study proceeds in the AS+C configuration. As the study progresses, more checkpoints are being accumulated in SISOL, so that any given experiment is more likely to find a nearby completed experiment whose checkpoint it can reuse. Table 6.3 shows the average time it takes for the worker process to execute an experiment as a function of when the experiment is issued. It shows that as the study goes on, the time it takes to execute one single simulation decreases. The largest improvement is shown in the second group (3.8 ms), because the first 128 experiments do not have any result to reuse, bringing up the average per-simulation time of the first group (7 ms). This mean that while a closer result is a better candidate for reuse than a further result, *any* result is better than having no result to reuse.

6.1.2.4 Early user feedback based on incomplete results

Besides reducing the number of experiments required to complete a study, the Active Sampling technique also has the advantage of being able to present the user early feedback of a fuzzy, low-resolution Pareto Frontier approximation,

Simulation #	Average time	Simulation #	Average 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 6.3: Average per-simulation run times (in milliseconds) of the first 1600 simulations, in 200 simulation increments, based on a 128-processor run.

before the Pareto Frontier is fully resolved. This enables the user to make decisions based on the low-resolution frontier early.

We demonstrate this benefit by measuring the Hausdorff distance between the fully-resolved Pareto Frontier and the Pareto Frontier approximations obtained by the study as it is being run. The Hausdorff distance is defined as the number d such that, for every point p on the approximated Pareto Frontier, there is at least one point on the fully-resolved Pareto Frontier that is d units away from p. This distance measures the similarity of two shapes.

The Hausdorff distance is measured at regular time intervals, thus we obtain a profile of how the Pareto Frontier approximation approaches the "real" frontier over time. We measure the same metric twice, once during a AS+C configuration run, and once during a GS+C configuration run. We compare the two time-evolutions in Figure 6.2. Notice the time axis is in log scale. The plots show that, in order to achieve the same Hausdorff distance as the AS+C configuration, it takes the GS+C configuration approximately an order of magnitude more time.

6.1.2.5 Scaling

The use of application-specific knowledge in the AS+C configuration can cause potential scaling problems. As discussed in Subsection 5.1.4, the excess simulation or synchronization overhead of the Active Sampler introduce parallelization overhead in an otherwise trivially parallelizable problem. Also, as discussed in Subsection 5.3.4, in supporting result reuse, the SISOL layer could become a performance bottleneck if it is overwhelmed by object requests.

In contrast, the GS configuration is trivially parallelizable, and it is expected to speed up linearly with the number of worker processes. Therefore, as the number of worker processes increase, the system could reach an inflection point where the benefits of application knowledge is negated by the parallelism overhead. At the inflection point, the performance of the GS configuration could overtake the performance of the AS+C configuration.

However, our scaling experiments show that the AS+C configuration is scalable up to 128 worker processes, and we do not appear to have hit the inflection point in our experiments. Figure 6.3 shows that the AS+C configuration scales almost linearly up to 128 worker processes. Moreover, it shows that SISOL and the manager process pose negligible overhead, since the bulk of the time devoted to the study is spent by the worker processes running the simulation code, and not on communication with the manager process or interacting with SISOL.

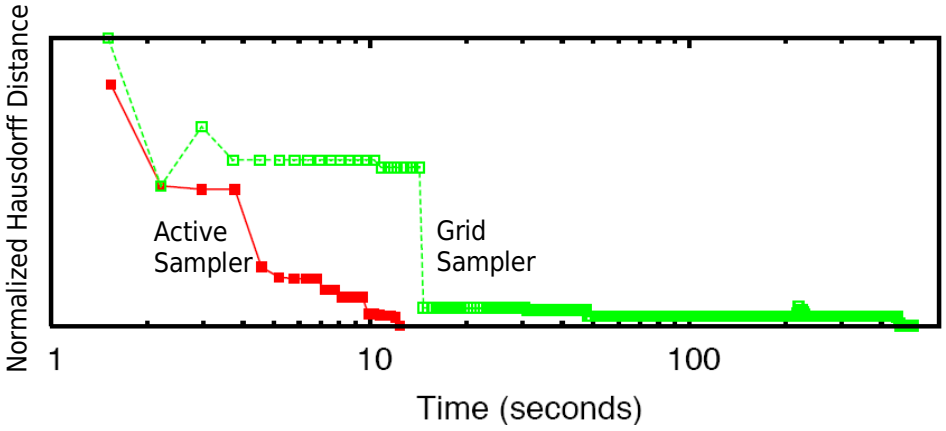


Figure 6.2: Relative error in Pareto frontier approximation as measured by the Hausdorff distance metric, in performance space

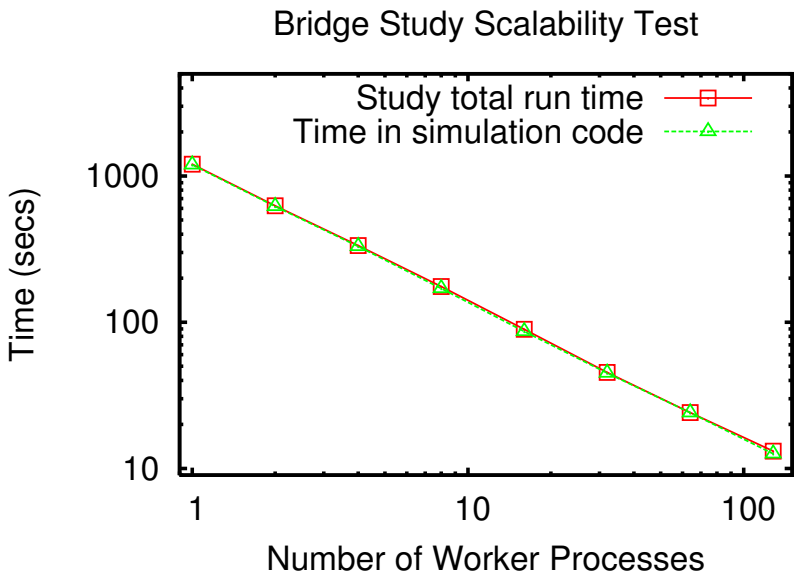


Figure 6.3: Scaling behavior of Bridge Study

6.1.3 Conclusion

This section has demonstrated that, in the bridge design study, by incorporating application-specific knowledge in system decisions can result in an over 400-fold improvement in terms of run time. In particular, active sampling results in a two-orders-of-magnitude improvement, and the reuse of previous results adds another multi-fold improvement. Despite of the potential parallelization overhead and performance bottleneck, the system can scale linearly up to 128 worker processes.

6.2 Defibrillator Design Study

The defibrillator study is an interactive Pareto Optimization MES, with a five-dimensional design space and three-dimensional performance space. The goal of the MES is to allow the user to interactively select and explore two-dimensional slices within the design space, and to resolve the Pareto Frontier on each slice at a 256x256 grid resolution.

As described in Subsection 3.2.2, in this study, the user would select a region in the front of the human torso to place the front electrode, a position in the back of the torso to put the back electrode, and a voltage strength. In addition, he would choose a the activation threshold and damage threshold beyond which the heart tissue is considered activated and damaged. The system would run the DefibSim application multiple times, and then inform the user about the Pareto optimal placement of the front electrode within the region he has chosen.

In order to achieve interactive rates, many of the system techniques discussed in Chapter 5 are applied to this MES. In particular, active sampling (Section 5.1), various types of experiment result reuse (Sections 5.3) and study-level result reuse (Section 5.4) are heavily employed. In this section, we evaluate and discuss the benefit of each of the system techniques.

The SimX/SCIRun version of SimX is used to conduct this study.

6.2.1 Methodology

All performance numbers reported in this section are collected from the same cluster that conducted the bridge design study (Section 6.1). In these experiments, SISOL is always served by four data servers and one directory server, and the manager process is represented by a SCIRun process running a performance evaluation workflow net which simulates the user's action.

To test how application-level knowledge can permit the defibrillator study to be conducted at an interactive rate, we created four usage scenarios. In each scenario, the user discovers the Pareto Frontier of a slice of the design space, and initiates an action to define another slice of the design space. The time it takes for the system to discover the Pareto Frontier of the new slice is measured as the response time of the system to this user action. The four scenarios are listed as follows:

Scenario A: Setting the activation threshold voltage. In this scenario, the user fixes the position of the back electrode, the voltage, and defines the area where the front electrode can be placed. As his action, the user changes

the activation threshold (in order to model, for example, a patient with heart tissues that are difficult-to-activate). One can see that the Pareto optimal points move closer to the center (Figure 3.5). The amount of reuse possible (i.e., the distance between the two slices) thus depends on the amount of change in the activation threshold.

Scenario B: Moving the explored region. In this scenario, the user fixes the position of the back electrode and the voltage, and defines the area where the front electrode can be placed. As his user action, the user slides the area of exploration downwards to a new (but overlapping) area. The amount of reuse depends on the amount of overlapping area between the two areas of exploration.

Scenario C: Moving the back electrode. In this scenario, the user fixes the voltage and the placement area for the front electrode. As the user action, the user moves the back electrode toward the right shoulder (Figure 3.6). The amount of reuse depends on the distance with which the back electrode is moved.

Scenario D: Increasing the shock voltage. In this scenario, the user wishes to get a coarse idea of how the Pareto-optimal placement of the front electrode is affected by the shock voltage. He fixes the back electrode's coordinates, selects a voltage setting, and performs a parameter sweep of the front electrode placement across the entire area in front of the torso. The user changes the shock voltage and performs the sweep again.

Each of the above scenarios are designed to model typical user actions

in this MES, and highlight the advantages of one or more of the application-aware system techniques discussed in Chapter 5. A brief recap of each technique evaluated is given below:

Active sampling: Discussed in Section 5.1, this technique is applicable to all Pareto Frontier optimizations, including the defibrillator study. In particular, the Active Sampler is used to explore the mini-MES defined by each slice of the design space. This technique is applicable to all four scenarios. Each sampler is instructed to start with a 16x16 lattice, and refine its Pareto frontier approximation four times to achieve the equivalent of 256x256 resolution.

Inter-study reuse: This is the reuse technique discussed in Section 5.4. In this type of reuse, the system uses the Pareto frontier of a design space slice as the initial Pareto frontier approximation of another slice. This technique is applicable to all four scenarios.

The following techniques are various forms of inter-study result reuse discussed in Section 5.3:

Performance metric reuse: Recall from Section 3.5 that the user inputs during design space exploration can consist either of changes in performance metric parameters, establishing new admissible performance space, or requesting exploration of a different region of the same design space slice. In the latter two scenarios, it is possible that the *same* simulation can be re-issued with the *same* performance metric parameters for a different slice. This technique is applicable in Scenario B, where the user explores the same design space slice but changes the region of exploration. For the area overlapped by the the

two regions, the same experiments with the same simulation parameters and the same performance metric parameters would have been performed. When that happens, the manager module can simply look up the performance metrics from the list of previously-completed experiments, and thus cut down the number of experiments that need to be issued to the worker processes.

Simulation result reuse: In Scenario A, the user defines a new slice by only changing the performance metric parameters. It is possible that the *same* simulation can be re-issued with *different* performance metric parameters for a different slice. In such cases the worker process can look up the simulation result from a list of previously-completed simulations instead of re-running the simulation code, and re-compute only the performance metrics. One expects the total number of experiments issued to stay the same, but the total number of simulations run by the worker processes to decrease.

Intermediate result reuse: As discussed in Subsection 5.3, some experiments may share the same intermediate results, specifically, the solutions to the three non-linear systems. The system can store these intermediate results into an object store and retrieve them later for a different experiment. This technique is applicable to all four scenarios.

Preconditioner reuse: Even if the solution of the exact linear system is *not* available, reuse is still possible. Each system in the defibrillator MES is solved using an iterative solver with a preconditioner, a compact approximation to the matrix inverse. The result of a simulation from a nearby point in design space can be used to initialize the iterative solver, decreasing the number of

iterations required. If, for different points in design space, the system matrices are similar (e.g. nearby electrode placement on torso) or identical (same placement, but different voltage distributions), we can use a previously computed preconditioner, assuming it was saved along with the results.

In order to quantify the benefits of those techniques, we compare the performance of the configuration where all the techniques are enabled to the performance of configurations where one or more of the system techniques are turned off. The goal in this evaluation is to measure 1) the sensitivity of each type of reuse to the richness of the performance history in each scenario — how much performance history is needed in order for the various types of reuse to become useful? — and 2) the benefits of each reuse technique, in terms of response time, as the number of worker processes is scaled from 1 to 128.

In order to understand the sources of performance benefits, we also measure the number of experiments issued, number of times the simulation code is run, number of times the solver net is fired, and the average time taken by an experiment, during the discovery of the second Pareto frontier.

6.2.2 Results and Analysis

6.2.2.1 Performance History

We first investigate the sensitivity of reuse to the richness of performance history. We expect the benefits derived from reusing past computation to depend on the performance history of the past computation, as well as to the similarity between the past slices of exploration to the current exploration. In

these experiments, for each scenario, we vary the amount of change that the user action introduces, and measure the amount of benefits from reuse. For baseline comparison, we also conduct a run where the user explores the second slice without exploring the first slice first (a 'cold' run), and a run where the system turns off all application-aware techniques (a naïve run. All performance results presented in this sub-subsection are obtained from runs conducted with 32 worker processes.

Scenario A: In this scenario, the user changes the activation threshold and discovers the new Pareto Frontier based on the new threshold. Here, inter-study reuse and simulation result reuse are applicable. To determine the sensitivity of the system to performance history, we conducted the scenario six times, each time using a different activation threshold in the first slice in order to control the amount of information that can be reused in the second slice. As expected, the number of times the simulation code needs to be run is reduced as the change of threshold becomes smaller. The results are shown in Table 6.4. With maximal reuse, we see up to an 80x improvement in response time over the sweep sampler.

Scenario B: In this scenario, the explored region is moved with an 81% overlap. The experiments in the overlapped areas can be avoided (performance metric reuse). We vary the percentage completion of the first slice and observe the benefits of the reuse from an increasingly rich performance history in Table 6.5. As expected, due to performance metric reuse, the more complete the exploration of the first slice, the fewer simulations need to be issued for the

Change in Activation Threshold	Number of required simulations	Response Time (sec)
sweep	16385	1633
cold	1665	213.6
57%	1381	141.9
43%	1158	126.1
30%	423	44.69
20%	444	40.89
11%	260	27.32
3%	248	23.30

Table 6.4: Response time of Scenario A, as a function of the amount of change in the performance metric parameter.

% Completion	Number of required simulations	Response Time (sec)
0	4467	238.5
18%	3521	176.7
37%	2725	139.7
56%	2112	113.6
74%	1292	84.20
93%	304	49.20
100%	13	16.80

Table 6.5: Response time in Scenario B, as a function of the percentage completion of the first slice.

second slice.

Scenario C: In this scenario, the back electrode is moved. The amount of change between the two Pareto frontiers depends on the amount of displacement of the back electrode. Table 6.6 shows that the higher the displacement, the less benefit is derived from inter-study reuse. In fact, for displacements that are too large (50mm or more), inter-study reuse actually results in a higher number of experiments being issued. This behavior is due to a core assumption for this type of reuse being violated, namely, that the Pareto frontiers of the two slices are similar. When the two frontiers are sufficiently different, it is more efficient to use a coarse grid to identify the interesting region instead of using the Pareto frontier of another slice.

Scenario D: This scenario is designed to demonstrate the benefits of pre-conditioner reuse. We find an improvement in the time it takes to solve individual systems (Table 6.7). As the change in shock voltage decreases, the

Displacement of back electrode (mm)	Number of required simulations	Response Time (sec)
sweep	16385	646.855
cold	7284	129.775
60	7666	154.108
50	7381	142.283
40	6762	130.37
30	6884	132.08
20	6868	127.14
10	6867	122.981

Table 6.6: Response time in Scenario C, as a function of the displacement of the back electrode.

Increase in shock voltage (V)	Avg. run time (secs)	Avg. solve time (secs)
cold	5.53	4.30
30	5.37	4.125
20	5.15	3.854
10	4.89	3.79

Table 6.7: Average per-simulation and per-solver run time in Scenario D, as a function of the change in electrode voltage.

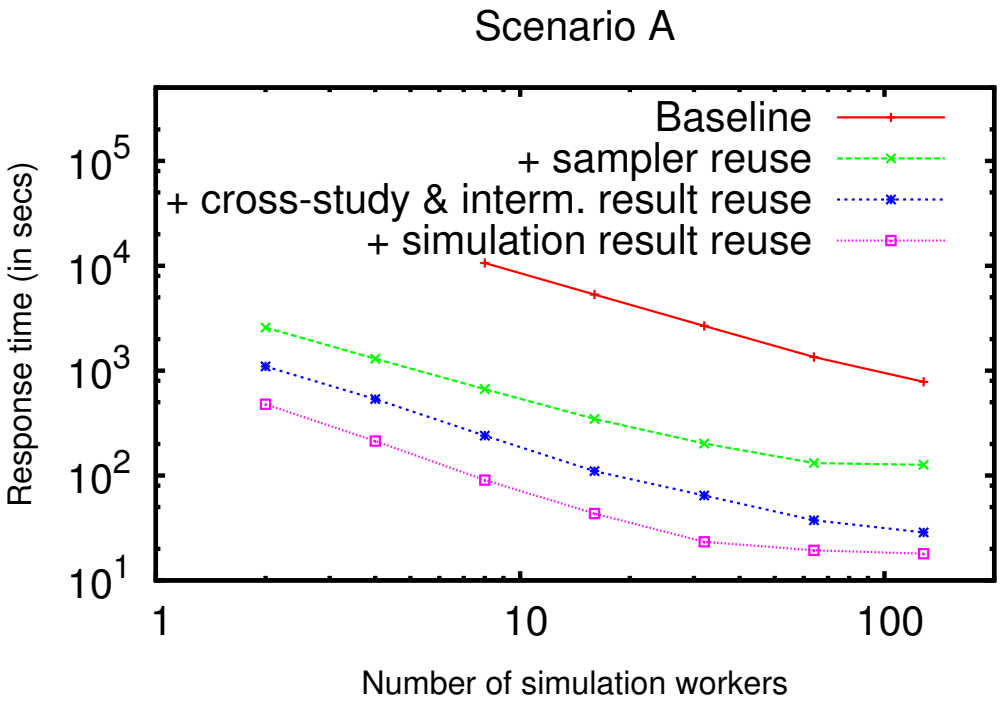
linear systems solved by the two slices become more similar, and hence the preconditioner from the first slice become more "useful" for the systems in the second slice. However, the amount of time spent on solving individual systems is only a fraction of time spent by the simulation code. Additionally, when coupled with the other types of reuse, which aim to minimize the time spent on the simulation code, the time spent on the simulation code is only a fraction of time spent in the overall study. Therefore, the time spent in solving individual systems is not a significant part of the overall response time, and pre-conditioner reuse, which yielded only a 11% improvement in simulation code in the best case, did not register a significant improvement in the overall response time.

6.2.2.2 Scaling

To see how different types of reuse scales in the defibrillator study, we performed usage scenarios A, B, and C, and measured the response time of the system. For each scenario, we vary the type of system techniques enabled, and also vary the number of worker processes.

Figure 6.4 shows the scaling characteristics in Scenario A on the top plot. The bottom plots shows the number of experiments issued in the run involving 128 workers process. The data from the scaling plot is replicated in Table 6.8. The baseline configuration corresponds to a run using no application level knowledge: brute force sweep sampling, with no result reuse. With 65000 experiments issued, each taking 2 seconds, with perfect scaling, it takes about 1000 seconds on 128 worker precesses to complete the exploration of the slice. With active sampling, the number of experiments is reduced to 1600. However, due to dependency of experiments between levels, the scaling is no longer linear. At 128 worker processes, active sampling only results in a 8-fold improvement in response time, taking 126 seconds. With inter-study reuse (shown as "cross-study" reuse in the figure), the number of experiments required is further reduced to 1200. With intermediate result reuse, which stores and reuses the solutions of linear solves in the simulation code, the per-experiment run time is reduced to 1.5 seconds. Combining intermediate result reuse and inter-study reuse results in a 4.5 times improvement in response time, resulting in a system response time of 28 seconds. Finally, turning on simulation result reuse, which eliminates the running of redundant simulation codes, the per-experiment run time is reduced by a further 5 times, to 0.3 seconds. However, SimX runs into a scaling bottle neck at 32 worker processes, and the response time at 128 worker processes is 18 seconds.

In Scenario A, combining all types of application-level knowledge results in a reduction of response time from 1000 seconds to 18 seconds on 128 worker



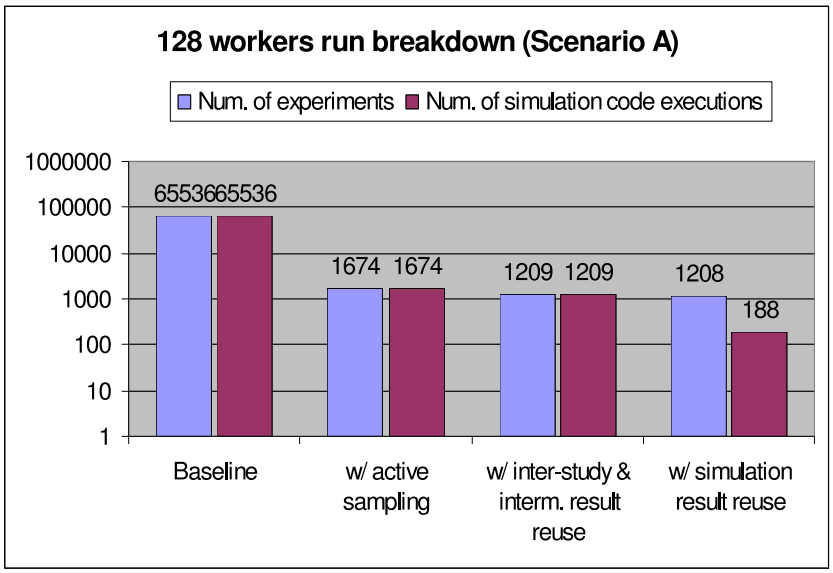


Figure 6.4: Scenario A scaling and breakdown

Number of worker processes	Run time (in seconds) for configuration			
	Baseline	+ Active Sampling	+ Study-level and Intermediate Result Reuse	+ Simulation Result Reuse
2		2580.18	1096.55	478.78
4		1295.76	535.81	213.02
8	10632.9	666.71	239.77	90.52
16	5324.63	346.00	110.12	43.36
32	2673.3	201.60	64.43	23.30
64	1348.12	131.66	37.46	19.33
128	781.52	126.35	28.62	17.98

Table 6.8: Scaling behavior in Scenario A

processes, a two orders of magnitude improvement. Furthermore, at 32 worker processors, before SimX runs into the scaling bottle-neck due to overly-reduced workload, the improvement resulting from application-level knowledge is over 100 times, from 2700 seconds to 23 seconds.

Figure 6.5 shows the scaling characteristics in Scenario B on the top plot, while its bottom plot shows the number of experiments issued in the run involving 128 workers process. The data from the scaling plot is replicated in Table 6.9. One notices that, in the version where all types of reuse are enabled, it does not matter how many worker processes are available: the response time is always about 15 seconds. This is because when performance metric reuse is applicable for studies with large overlapped region, the number of experiments required decreases dramatically: in Scenario B, with performance metric reuse, fewer than 15 experiments are required to explore the second slice. There are simply not enough experiments to divide up between the worker processes, so adding additional worker processes does not change the overall run time. Other than this anomaly, the scaling figures are as expected: the brute-force no-reuse version takes 65000 experiments at 2 seconds each, resulting in a 1000 second response time on 128 worker processes. Active sampling reduces the number of experiments required to 4800, and improves the response time by an order of magnitude at lower number of worker processes, but due to dependency it does not scale linearly and only improves the response time by 60% at 128 worker processes. Inter-study and simulation result reuse further reduce the number of experiments to 3600, and the per-experiment run time to 1.16 seconds, im-

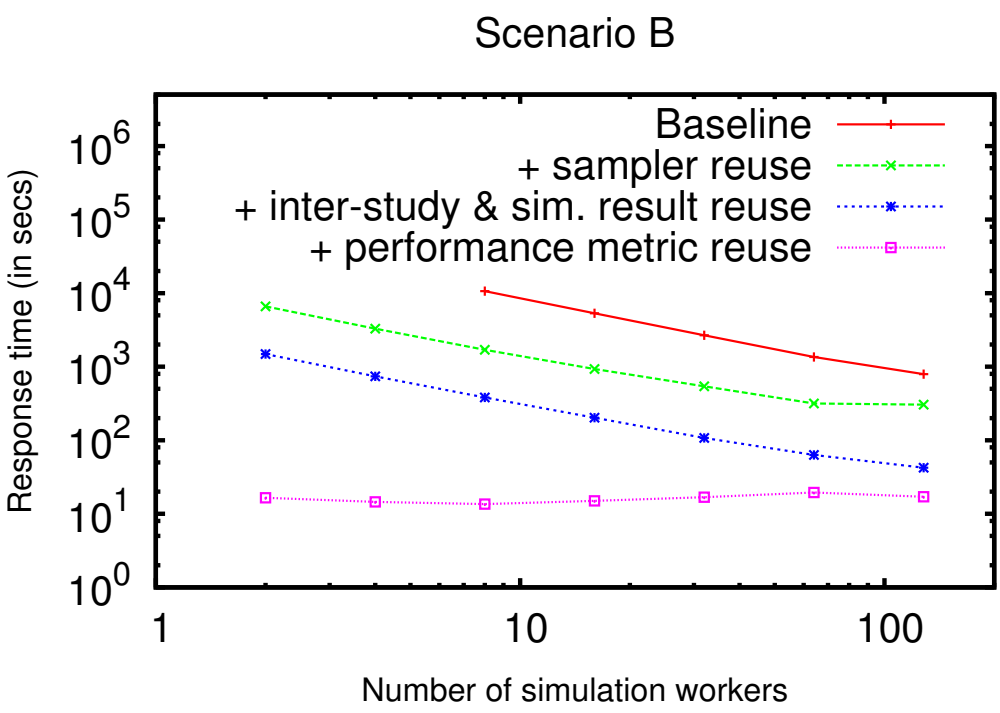
Number of worker processes	Run time (in seconds) for configuration			
	Baseline	+ Active Sampling	+ Study-level and Simulation Result Reuse	+ Performance Metric Reuse
2		6605.2	1483.34	16.49
4		3292.5	738.79	14.56
8	10655.7	1701.0	381.33	13.58
16	5335.79	930.11	203.1	15.01
32	2680.38	540.2	107.71	16.81
64	1353.46	316.47	63.19	19.48
128	792.00	305.12	42.45	17.04

Table 6.9: Scaling behavior in Scenario B

proving the response time by another order of magnitude. Finally, performance metric reuse can reduce the number of experiments to 15 and improve the response time by a further 50% on 128 worker processes.

In Scenario B, combining all types of reuse results in a two orders of magnitude improvement on 128 worker processes, from 1000 seconds to 16.8 seconds.

The scaling characteristics in Scenario C is shown in Figure 6.6's top plot. Its bottom plot shows the number of experiments issued, as well as the total number of linear systems the workers have to solve. The data from the scaling plot is replicated in Table 6.10. Here, the brute-force approach takes 1000 seconds on 128 worker processes to execute 65000 experiments at 2 seconds each. With active sampling, the number of experiments required is decreased by 9 times, to 7300. This results in an improvement in the response time only



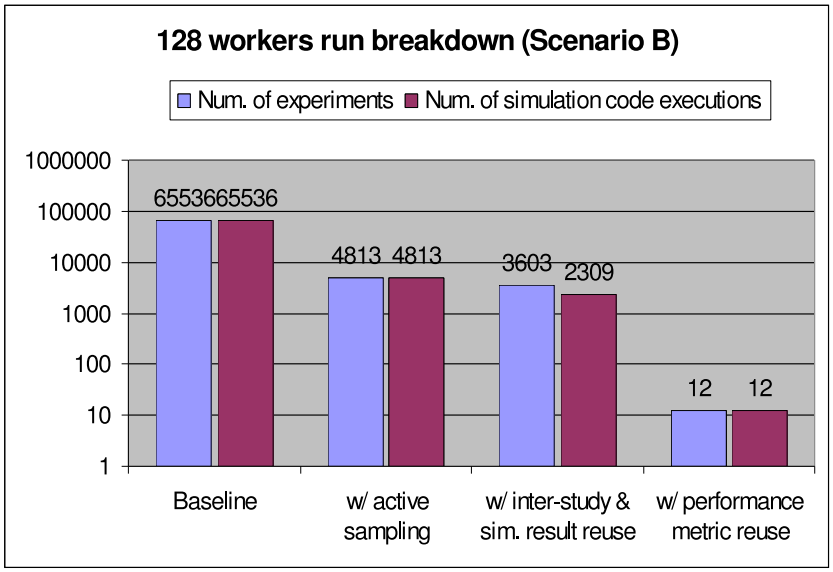


Figure 6.5: Scenario B scaling and breakdown

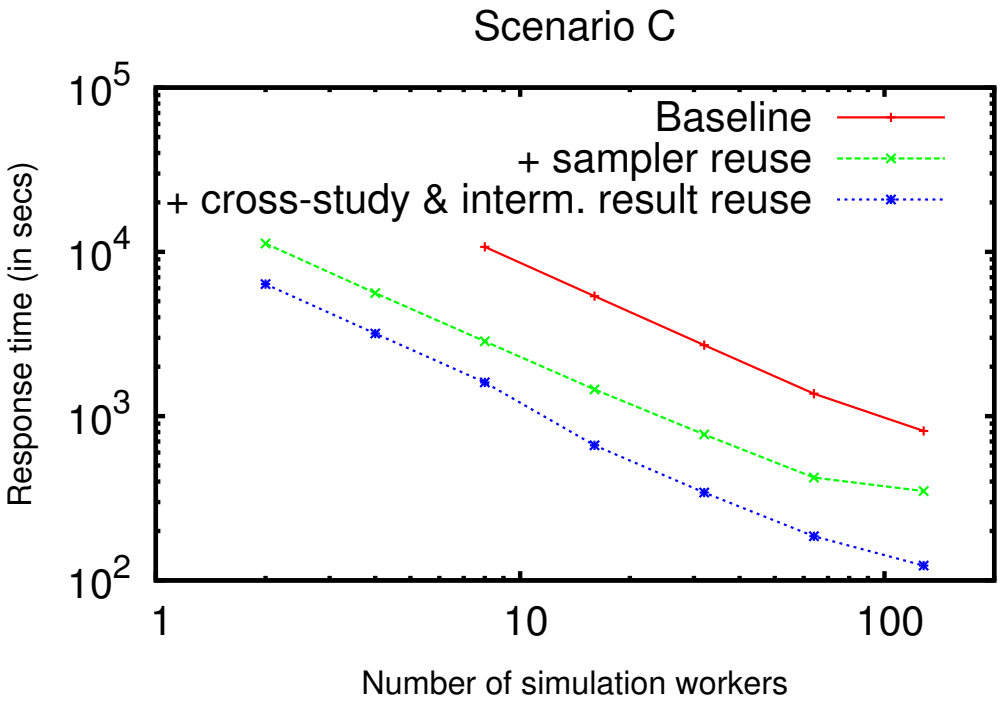
Number of worker processes	Run time (in seconds) for configuration		
	Baseline	+ Active Sampling	+ Study-level and Intermediate Result Reuse
2		11249.9	6365.26
4		5609.91	3184.8
8	10724.9	2854.19	1603.78
16	5369.6	1453.67	664.89
32	2704	773.50	342.91
64	1370.26	422.69	185.83
128	812.16	349.43	122.98

Table 6.10: Scaling behavior in Scenario C

by 60% on 128 worker processes, again due to non-linear scaling in the active sampler. Finally, inter-study reuse reduces the number of experiments needed further to 6500, and intermediate reuse improves the per-experiment execution time by 25%, to 1.5 seconds, resulting in a response time of 120 seconds, which is 8 times faster than the brute-force version.

In Scenario C, combining all types of reuse results in an order of magnitude reduction in the number of experiments issued, and a 25% improvement in per-experiment execution time. However, due to dependencies introduced between experiments, the total improvement in response time is limited to an 8-fold decrease.

As the data shows, while using application-level knowledge in the defibrillator design study can introduce scaling problems, such as introducing dependencies between experiments, and reducing the problem size by such a wide



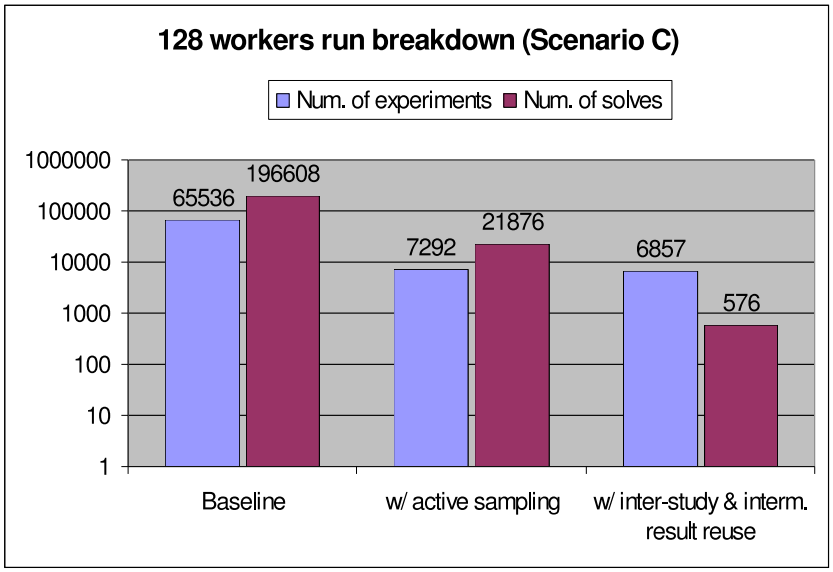


Figure 6.6: Scenario C scaling breakdown

margin to render parallelization useless, the benefits outweigh these drawbacks, as demonstrated by the 8 to 100 times improvement in total response time of the system. In particular, application-level knowledge enabled SimX to conduct Scenarios A and B with a 10- to 20- second response time, which is within the interactive rate. While a response time of 120 seconds in Scenario C is not completely interactive, it is much closer to interactivity than the original 17-minute response time.

6.2.3 Conclusion

In this section, we have demonstrated that, by incorporating active sampling and result reuse techniques in the defibrillator study, the system can achieve interactive rates to enable the user to steer the entire study. In addition, the performance history and scaling studies show that 1) the system should exploit only the types of reuse that are likely to yield benefit in a given situation, 2) more than one type of application knowledge needs to be exploited to yield overall response times that support interactivity, and 3) despite introducing dependencies in a parallel workload that did not have any to start with, the exploitation of application-level knowledge achieves dramatic reductions in response time while still leaving behind enough parallelism to productively employ a moderate-sized parallel machine.

6.3 Animation Scene Design

The animation scene design study is a user-driven study where the user interactively drives the construction of an animated scene. As discussed in Subsection 3.3.3, the study is conducted in stages, with each stage representing a partially completed scene. The system automatically explores possible continuations at each stage, from which the user chooses which extension to pursue. In effect, this formulation frames the MES as a search tree. The system automatically discovers viable children of the tree node chosen by the user, and the user guides the system through the tree by selecting a node from among the available ones, until he finds a leaf node, which represents a completed animated scene.

This MES showcases the importance of user interactivity. In particular, it shows that by combining automatic exploration, random sampling, admissible region filtering, and interactive user guidance, the MES can construct aesthetically interesting scenes in a reasonable amount of time. The goal in this MES is to demonstrate the responsiveness of SimX, and the importance of utilizing user guidance to advance the study.

This study uses the standalone version on worker processes, and the SimX/SCIRun version on the manager process front-end.

6.3.1 Methodology

To evaluate SimX's interactivity, we measure the rate with which SimX can provide the user with timely feedback. In particular, we conduct two sessions of animation scene design, and measure the rate at which the system dis-

covers admissible tree nodes at each level as an indication of the responsiveness of the system.

To measure the impact of user interactivity, we created two baseline configurations of the animation design study that do not require user interaction. The first configuration (Configuration A) explores the entire design space in a single stage. The second configuration (Configuration B) explores the design space in stages, but instead of using the user's guidance to select tree nodes, it chooses tree nodes at random. We use two metrics to measure the aesthetic quality of scenes collected from these configuration, and compare them against the scenes obtained from the two interactive sessions.

All experiments in this section are conducted with a partition of 12 processing elements on the same cluster as the bridge design study (Section 6.1), with 11 running the worker processes and one running the SISOL servers.

6.3.2 Results and Analysis

6.3.2.1 System responsiveness

Figures 6.7 and 6.8 show the number of available tree nodes at each level as the system responds to user actions in two interactive sessions. In both plots, the user actions that direct the study are indicated by call-out boxes.

The detailed description of Session 1 is as follows. (The progress of Session 2 is similar.) When the study begins, the system discovers admissible first-level nodes at a rate of one node per 15 seconds. The user looks at each of these partially-complete scenes, and decides at the 75th second to select one

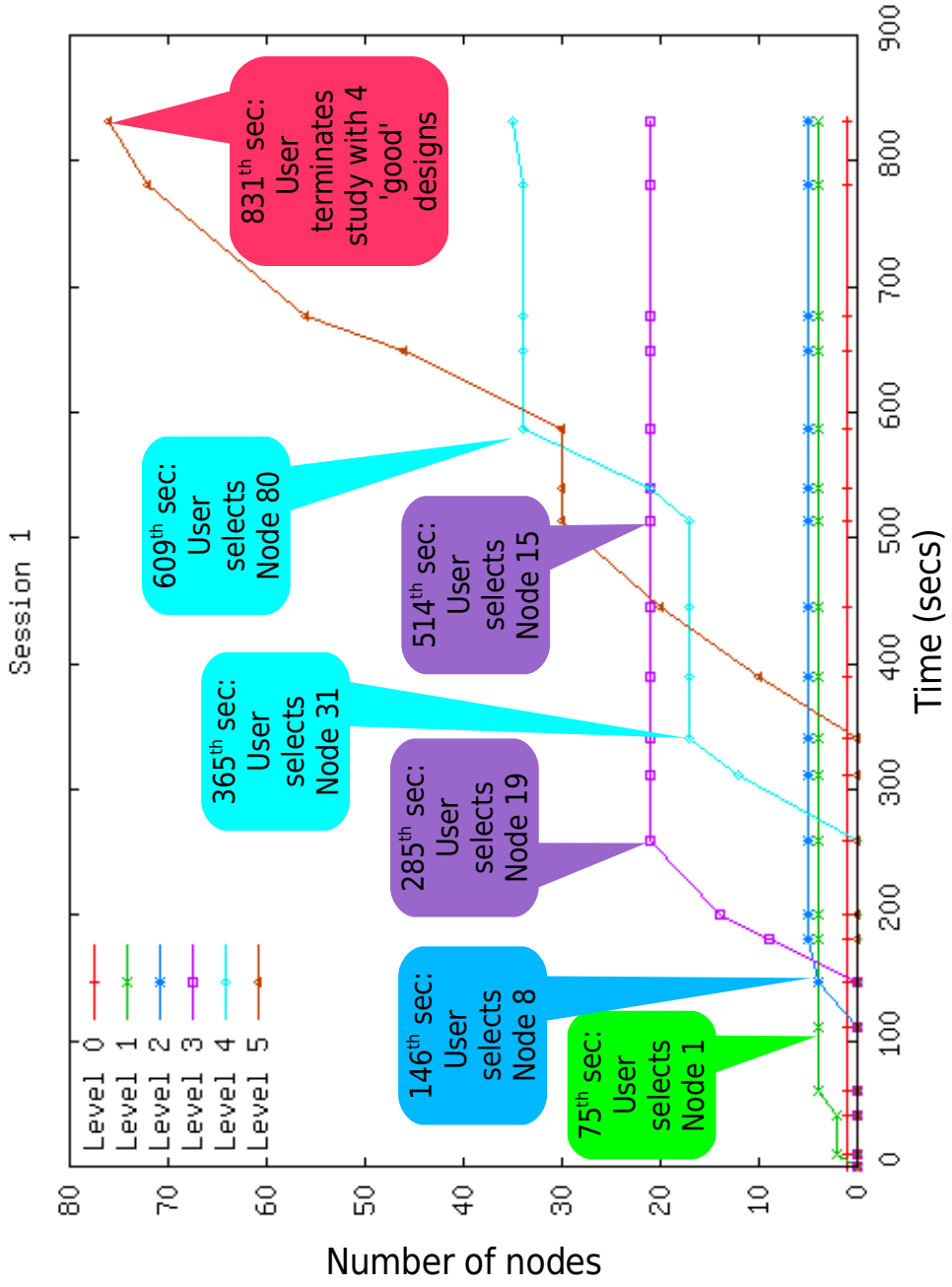


Figure 6.7: SimX response time in Animation Study

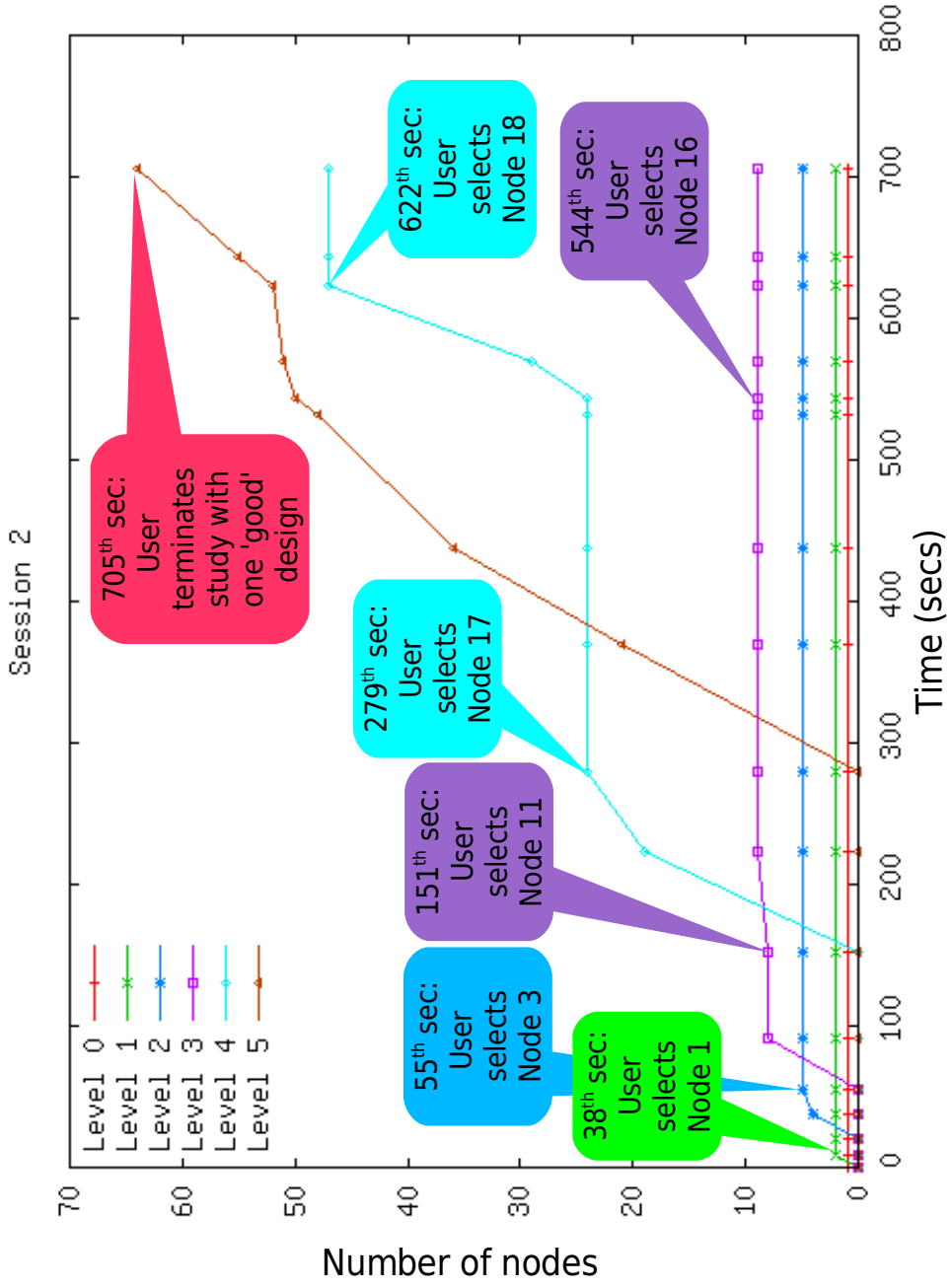


Figure 6.8: SimX response time in Animation Study

of them as being most interesting and worthy of further exploration, thereby completing the first stage of the study. The system then supplies the user with admissible second-level nodes at the same rate. The user takes 60 seconds to decide on the second-level node to explore. The system then supplies him with admissible third-level nodes at the rate of one node per 5.7 seconds. The user takes 120 seconds to decide on a third-level node, and selects the node roughly 270 seconds into the study. The system then supplies him with fourth-level nodes at the rate of one node per 3.2 seconds. The user takes 80 seconds to decide on a fourth-level node to explore. The system then supplies leaf nodes at the rate of 1 node per 5 seconds. However, after waiting and looking at the completed scenes for 150 seconds, he is dissatisfied with the scenes, so he "backtracks" at the 514th second: he selects another third-level node, and continues the exploration from there. This time the system supplies him with admissible fourth-level nodes at 4.2 nodes per second. He takes 90 seconds to decide on a new fourth-level node (at the 600-second mark of the study), after which the system supplies him with a new batch of leaf nodes at the rate of one node every five seconds. Of these he chooses four to be "good" designs and terminates the study at the 840th second. In all, this first session requires 19251 executions of the buggy simulation code and lasted 14 minutes. The second session requires 16000 executions and lasted 11 minutes.

Note that, in both sessions, the user is kept busy at all times: if he is not looking at partially-animated scenes, he is making decisions about which tree node to explore.

The system is able to provide the user with new admissible tree nodes within seconds. This is because the system can automatically explore the node the user has chosen while automatically filtering out inadmissible scenes such as those where the buggy is knocked off course. Without the automatic exploration, the user would have needed to construct and view more than 15000 partially-completed scenes, which is clearly not practical.

6.3.2.2 Impact of Interactivity

As discussed above, the regular sessions took 11 and 14 minutes to discover viable and interesting animated scenes. In contrast, in Configuration A, where the system completely automates the study by exploring the entire design space using one stage, the study ran for an hour without discovering a single admissible scene.

For Configuration B, we configured the system to discover 15 child nodes at each stage, and select one child node from the 15 at random. We run twenty sessions of this configuration, and measure the "aesthetic quality" of the resultant scene from each run. We use two metrics to measure how aesthetically desirable a scene is: the average closest distance that each debris piece is to the buggy (a closer distance is considered more interesting), and the average speed at which the debris piece is travelling when it comes into closest proximity with the buggy (a higher speed is considered more interesting). The results, along with the number of simulations and time required to discover the scene, is listed in Table 6.11.

Configuration	Session	Time (mins)	# sims	Avg. closest distance	Avg. speed at distance
Interactive	1	14	19281	0.292	3.42
	2	12	16117	0.234	1.2
Configuration B	**1**	**16**	**19054**	**0.295**	**3.41**
	2	7	8783	0.349	2.16
	3	15	16843	0.324	0.50
	4	12	13430	0.392	0.94
	5	11	12606	0.250	1.27
	6	15	17419	0.278	1.25
	7	**12**	**14419**	**0.281**	**3.90**
	8	12	14564	0.328	3.33
	9	**9**	**10428**	**0.286**	**7.38**
	10	10	10859	0.366	1.25
	11	19	23033	0.334	1.63
	12	11	13141	0.321	2.63
	13	15	16322	0.311	1.77
	14	12	13723	0.288	2.82
	15	26	20849	1.111	0.69
	16	12	10480	0.309	2.85
	17	**22**	**23450**	**0.198**	**1.90**
	18	12	13450	0.284	0.79
	19	19	21066	0.323	1.78
	20	8	8778	0.314	1.20

Table 6.11: Animated scene quality from animation study

As shown in Table 6.11, by randomly choosing children nodes, it is sometimes possible to obtain an interesting scene, such as Configuration B's sessions 1, 7, 9, and 17 (highlighted in the table in boldface). These sessions achieve completed scenes with aesthetic metrics that are comparable to the scenes constructed from the interactive sessions. In sessions 1, 7, and 9, the debris pieces come to within an average of 0.3 units of the buggy while travelling at above 3 units of speed, which is similar to the scene constructed in interactive session 1. In session 17, the debris pieces come to within 0.2 units of the buggy, and is the only scene with a better distance metric than the scene discovered in interactive session 2. However, the rest of the scenes chosen by the random configuration are inferior to the two chosen by interactive sessions. Therefore, there is no guarantee that a non-supervised exploration can result in a "good" scene. It is only by continuously monitoring the progress of the study that the user can guide the system to discover a scene that satisfies his aesthetic sense and judgement.

6.3.3 Conclusion

In this section, we demonstrated that, by quickly responding to user's actions, the system can enable a user to guide the exploration of a search tree and construct an animated scene within a reasonable time frame. In addition, we also demonstrated that both user guidance and automated exploration are essential to accomplish this goal — without user guidance, there is no way to inject aesthetic qualities into the exploration, and without automated exploration,

the user would need to view an impractically large number of scenes.

6.4 Helium Model Validation

The Helium Model MES is a Pareto Optimization MES, with a two-dimensional design space (Prandtl number and Inlet Velocity) and a two-dimensional performance space (velocity profile difference at two heights). It requires a relatively small number (dozens) of long-running experiments (each taking about a half-hour). The Pareto Frontier for the Helium Model is shown in Figure 3.11.

In this MES, the individual simulations are themselves parallel tasks. Therefore, in addition to deciding which experiments to run, and in what order, the system also needs to decide how many processing elements are assigned to each experiment. This added dimension poses additional challenges and provide additional optimization opportunities for the system. As discussed in Sections 5.6 and 5.5, the system has the conflicting goals of both minimizing the parallelization overhead in any single experiment, and at the same time obtaining information from early experiments soon enough in order to inform the execution of later experiments.

Five of the application-aware techniques discussed in Chapter 5 are applicable to this MES — namely, active sampling (Section 5.1), checkpoint reuse (Section 5.3), scaling behavior-aware batching (Section 5.5), result reuse-aware scheduling (Section 5.6), and preemption (Section 5.5). The goal in this evaluation is to quantify the impact of each of these techniques on the running of the helium validation MES.

All the results reported in this section are based on experiments conducted on a cluster of 2.4GHz Intel Xeon nodes, each with 2GB memory, connected by a 1 Gigabit Ethernet interconnect. The SimX/Uintah version of SimX is used on the worker processes, and the SimX/SCIRun version is used for the manager process front-end.

6.4.1 Methodology

To measure the impact of application-aware techniques listed above, we created five configurations (Configurations A-E below), each one enabling the use of subsets of application-specific knowledge. In addition, two base-line configurations (O1 and O2) are defined and their behaviors are estimated based on experiment run times obtained from the other configurations. The configurations are defined as follows:

Configuration O1: Does not use any application-level knowledge: uses brute force sweeping sampling, without result reuse. Maximizes parallelism by allocating one worker process to each experiment.

Configuration O2: Does not use any application-level knowledge: uses brute force sweeping sampling, turn off result reuse. Minimizes per-experiment run time by always allocating 32 worker processes to each experiment.

Configuration A: Same as Configuration O2, but turn on Active Sampler (Section 5.1). No result reuse. The Task Queue issues one simulation at a time to 32 worker processes. No preemption.

Configuration B: Same as Configuration A, but enable result reuse (Section 5.3). The Task Queue issues one simulation at a time to 32 workers. No preemption.

Configuration C: Same as Configuration B, but the Task Queue issues simulations according to the optimal batch size based on scaling behavior (Section 5.5).

Configuration D: Same as Configuration C, but the Task Queue gives the first member of each reuse class higher priority (Section 5.6).

Configuration E: Same as Configuration D, but enable preemption (Section 5.5).

For each configuration, we conduct the study twice, once using 32 worker processes, and once using 64 worker processes. In both cases, we use one manager process, one SISOL directory server, and one SISOL data server. All worker processes are spawned on independent nodes, so there is no contention for network or memory resources between worker processes. We record the total wall clock time it takes to complete the study, the worker process utilization rate, and the average run time of a single simulation. To achieve the desired resolution of the design space (6x6) using brute force, the study requires 36 experiments. With active sampling enabled (Configurations A to E), the study requires 24 experiments.

Configuration	Total time	Utilization Rate	Avg. simulation run time
O1	12 hr 35 min	56.25%	6 hr 17 min
O2	20 hr 35 min	100%	34.3 min
A	13 hr 44 min	100%	34.3 min
B	9 hr 52 min	100%	24.7 min
C	6 hr 10 min	71.08%	63.4 min
D	5 hr 10 min	71.29%	39.7 min
E	4 hr 27 min	91.81%	33.5 min

Table 6.12: SimX performance on helium model validation study using 32 worker processes

6.4.2 Results and Analysis

The times it take to complete the helium validation study under the various configurations are listed in Tables 6.12 and 6.13. The rest of this section discusses in length the performance impacts of the application-aware techniques by comparing the performance data from different configurations.

6.4.2.1 Parallelization Penalty

Comparing Configurations O1 and O2 shows the parallelization penalty of the Arches code. Configuration O1 out-performs O2, even though in O2 individual simulations finish much faster, and the utilization rate of the worker processes is much higher. This is because, when using 32 workers to execute Arches, the parallelization overhead creates extra work for each worker process to do, resulting in a higher per-experiment-per-worker cost. The scheduling graph for Configuration O1 on 32 processes is shown in Figure 6.9. The X-axis

Configuration	Total time	Utilization Rate	Avg. simulation run time
O1	6 hr 17 min	56.25%	6 hr 17 min
O2	10 hr 53 min	100%	36.3 min
A	7 hr 55 min	92.74%	36.3 min
B	6 hr 0 min	93.33%	26.8 min
C	4 hr 3 min	60.60%	36.1 min
D	3 hr 37 min	70.00%	27.0 min
E	3 hr 41 min	93.67%	25.8 min

Table 6.13: SimX performance on helium model validation study using 64 worker processes

shows the time, and the Y-axis lists the 32 worker processes. The graphs show, for each worker process, which experiment it is working on at any moment. Each experiment is shown in a different color, so that the worker processes working on the same experiment show up as a same-colored block. On 32 worker processes, Configuration O1 performs the study in two stages: in the first stage, all 32 workers are busy, each performing one experiment, and in the second stage, only 4 workers are busy performing the remaining four experiments, while the other 28 are idle. Therefore, the worker processes utilization rate is only slightly higher than 50%.

6.4.2.2 Active Sampling

Comparing Configurations O2 and A shows the benefits of Active Sampling. Even though Configuration O2 performs its experiments serially, and thus incurs high parallelization overhead within experiments, there are potential benefits. One of the benefits is that we can now enable active sampling to cut down the number of experiments required. By withholding experiments until some earlier experiments' results have returned, one can decide, using the active sampling technique, whether the later experiments need to be executed. Active Sampling cuts down the total run time by 33% on 32 workers and 27% on 64 workers. Since Configuration A requires 24 instead of 36 experiments, on 32 workers, Configuration A is estimated to be 33% faster. On 64 workers, however, since Configuration A always uses 32 workers per experiment, when there is only one experiment left in the system, half of the workers are idle,

hence the non-100% utilization rate and lower-than-33% speedup.

6.4.2.3 Checkpoint reuse

Comparing Configurations A and B shows the benefits of checkpoint reuse. While Configuration A shows improvement over Configuration O2, it is still slower than Configuration O1. However, by performing the experiments serially, more application-specific optimizations are made possible. Namely, it enables the ability to reuse previously-computed results in later experiments. This benefit is derived from the reduced average per-simulation run time: down to 24.7 mins for 32 worker runs and 26.8 mins for 64 worker runs. That translates to a 28% (32 workers) and 26% (64 workers) reduction in the study's run time. Configuration B out-performs Configuration O1, so, with active sampling and result reuse, SimX already can recover the performance penalty incurred due to parallelization overhead.

6.4.2.4 Scaling behavior-aware resource allocation

Comparing Configurations B and C shows the benefits of ideally batching experiments based on their scaling behavior. Configuration C uses application-specific knowledge to decide the amount of parallelization each experiment should receive. More simulations are being issued concurrently, each on fewer processors. This increases individual simulation run times, but reduces the overall amount of work because less time is spent in the communication overhead in individual simulations. The overall run time of the study is reduced by a

further 38% on a 32-worker run, and 33% on a 64-worker run.

6.4.2.5 Resource allocation to maximize reuse potential

Comparing Configurations C and D shows the benefits of using the knowledge of reuse classes in batching decisions. Figure 6.10 shows the scheduling graph of Configuration C on 32 workers, where the experiment numbers are annotated on the graph. It shows that sometimes worker processes are kept idle while waiting for some long-running experiments to finish, such as experiment no. 13. While waiting for it to finish, the system leaves 28 worker processes idle for almost 50 minutes, reducing the utilization rate. This is because no. 13 is an experiment that cannot reuse a previously-computed checkpoint, while experiments 11 and 14 through 18 all reuse a checkpoint, and thus all finish much sooner than experiment 13. This unevenness of run time can be addressed by separating out start-from-scratch or start-from-checkpoint experiments. Separating reuse classes increases the utilization rate by ensuring that concurrent simulations are all start-from-scratch or all start-from-checkpoint. The scheduling graph for Configuration D on 32 workers is shown in Figure 6.11. Experiments 0, 1, 2, 9, 10 and 11 all start from scratch, and are shown to require more worker-processor-minutes than the other experiments. By separating them from the rest of the experiments and scheduling them concurrently, the system can ensure that the experiments scheduled to run at the same time have comparable run times, thus reducing the size of "holes" on the scheduling graphs. As a result, the utilization rate is improved by 10% on 64 workers. Separating

out reuse classes also reduces the average per-simulation run time by maximizing reuse potential. In Configuration C (Figure 6.10), experiments 1, 2, and 4 could have reused each others' results, as could experiment 5, 7, and 8. Unfortunately, because SimX scheduled them to start together, they all have to start from scratch, because none of their simulation results are available at the time of their starting. The reuse potential is thus lost. Configuration D schedules the first members of each reuse class concurrently, resulting in higher reuse potential. In Figure 6.11, experiments 3 to 8 reuse the results from experiments 0 to 2, and experiments 12 to 23 reuse the results from experiments 0 to 2 and 9 to 11. By maximizing reuse potential, SimX improves the average per-experiment run time on 32 workers by 37%. Combining the benefits of increased utilization rate and maximized reuse potential, the knowledge of reuse classes results in the overall run time showing a further 16% (32 workers) and 10% (64 workers) improvement.

6.4.2.6 Preemption

Comparing Configurations D and E shows the benefits of preemption. Even though separating start-from-scratch or start-from-checkpoint experiments alleviates the problem of uneven run times of concurrent experiments, the scheduling graph of Configurations D still shows that many worker processes are kept idle while waiting for some long-running experiment to finish on another worker process group. In Configuration E, these long-running experiments may be preempted by the idle worker processes so they may be "spread out" to

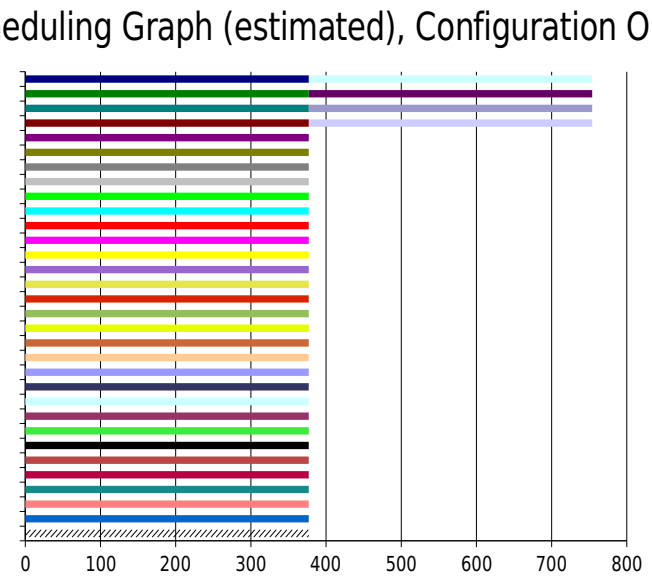


Figure 6.9: Helium validation study scheduling graph, Configuration O1

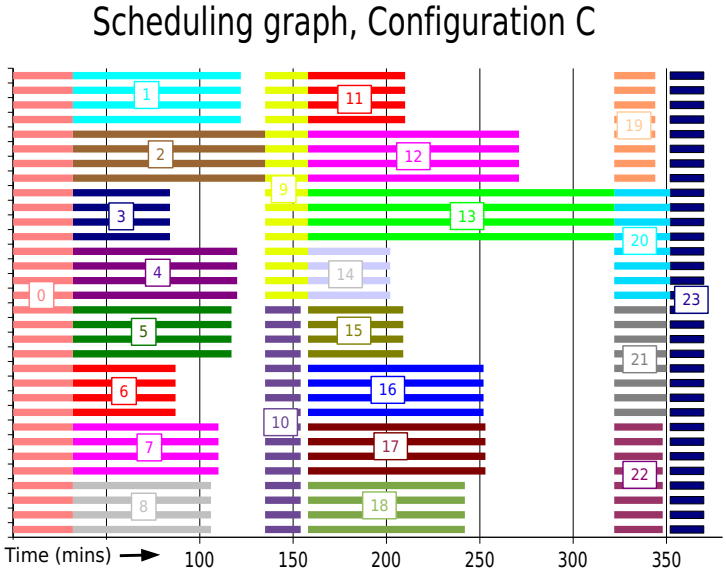


Figure 6.10: Helium validation study scheduling graph, Configuration C

those idle worker processes. Configuration E's scheduling graph (Figure 6.12) shows that, long-running experiments can fill in the "holes" in the scheduling graph, like experiment 16, which started on 4 workers and eventually spread itself out to all 32 workers. Configuration E improves processor utilization rate back above 90%. On 32 workers, this translates into a 14% improvement of the overall study run time. On 64 workers, however, this improvement is offset by the overhead required to perform the reconfigurations, and Configuration E shows no improvement over Configuration D.

6.4.3 Conclusion

In this section, we showed that, by adding application-specific knowledge to the scheduling and resource allocation decisions in our helium model validation study, the system can achieve a 78.4% and 64.6% reduction in overall run time over the baseline configurations on 32 worker processes, and a 66.8% and 42.4% reduction in overall run time over the baseline configurations on 64 worker processes.

6.5 Summary

This chapter has demonstrated that Multi-Experiment Studies can be performed efficiently on a parallel runtime system that exploits their characteristics. Interactive MESs can respond to user input at an interactive rate, while the run time of non-interactive MES can be reduced by several times to several orders of magnitude. To enable these gains, the system makes use

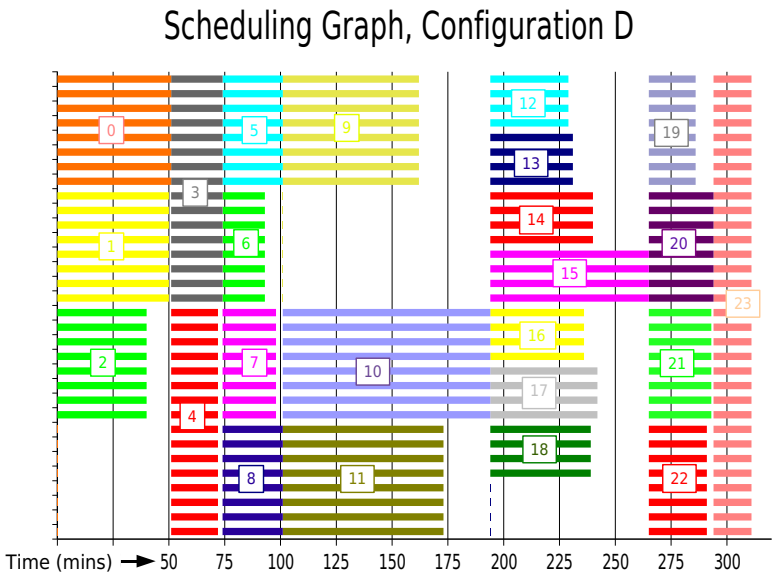


Figure 6.11: Helium validation study scheduling graph, Configuration D

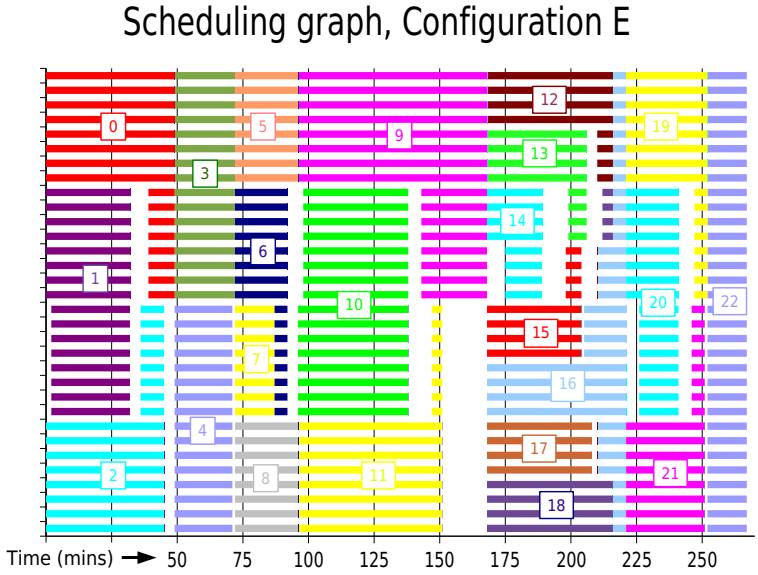


Figure 6.12: Helium validation study scheduling graph, Configuration E

of multiple techniques that leverage the knowledge of the application: active sampling uses the goal of the MES to automatically prune the design space; user interactivity leverages the user's input to guide the study; result reuse uses the result from earlier experiments to speed up the execution of later experiments; parallelism-driven resource allocation maximizes resource utilization while minimizing parallelization overhead within each experiment; and reuse-driven scheduling ensures maximum result reuse potential. These techniques work in tandem to reduce the number of experiments required, reduce the run time of individual experiments, and improve resource utilization rate, resulting in a 400-fold run time improvement in the bridge design study, up to a 100-fold response time improvement in the defibrillator design study, interactive steering for the animation design study, and up to a 75% improvement in the run time of the helium model validation study.

Chapter 7

Summary and Future Work

7.1 Summary

In this thesis, we have motivated why Multi-Experiment Studies (MESs) are expected to become more prominent users of parallel clusters in the near future (Chapter 1). We have identified the challenges posed by the problem of conducting MESs on clusters (Chapter 3) and techniques that can be borrowed from existing systems to deal with this problem (Chapter 2). We have developed a new approach to the problem: by viewing the entire MES as a single entity, and taking advantage of application-specific knowledge when making system-level decisions (Chapter 5). We have presented the SimX system (Chapter 4), a run time system that conducts MESs on parallel clusters using that approach. We have also quantified the benefits of each type of application-specific knowledge: by demonstrating the performance benefits obtained in four example MESs (Chapter 6), we have shown the importance of applying application-level knowledge in their executions.

7.1.1 Performance

The techniques proposed in this thesis yielded a multi-order of magnitude improvement in the run time of two of the example MESs, achieved multi-fold improvement in the run time of one example MES, and realized interactive steering of two example MESs.

In order to achieve this, it was essential to incorporate different types of application-level knowledge in system decisions:

- **Reuse**: The system can improve its performance by reusing the intermediate result or simulation result of earlier experiments to speed up later experiments (Sections 5.3 and 5.4). For interactive studies, the system can reuse aggregate results from early units-of-exploration to inform the conduct of later units-of-exploration. SimX demonstrates that result reuse can achieve an order of magnitude improvement in the bridge design study's response time, an 8x to 100x improvement in the defibrillator design study's response time, and a 26%–28% improvement in the helium code validation study's response time.

- **User interactivity**: By allowing the user to steer the entire MES instead of one single experiment at a time, the system allows the user to explore complicated design spaces by breaking them down into slices, or to guide the exploration of a search tree by using his subjective judgement (Section 5.2). The defibrillator design study and animation design study show that, by coupling SimX's interfaces with SCIRun, it

is possible for a user to interactively steer entire computational studies. SimX achieves study-level response rate in the order of seconds.

- **Sampling**: In sampling (Section 5.1), the system schedules experiments in a way such that the result of early experiments can inform the system as to whether some of the later experiments are needed, thereby cutting down the total number of experiments required by the study. By using sampling strategies that conforms to the goal of the MESs, we have demonstrated a 33% to a two-orders-of-magnitude reduction in the number of experiments needed by the Pareto frontier discovery studies.

- **Resource Allocation**: When conducting an MES composed of simulation code that themselves can run in parallel, the system can allocate processing elements to experiments according to the scaling behavior and reuse properties of the simulation code to minimize parallelization overhead within individual experiment runs while maximizing resource utilization, and at the same time maximize result reuse potential. Using application-aware resource allocation techniques, SimX is able to bring about a 42.4% to 4.6 times improvement in the run time of the helium validation study.

These improvements are only possible because SimX is able to take advantage of application-specific knowledge and use it in its system policy decisions.

7.1.2 Applicability

We have demonstrated that application-aware techniques can be implemented in a convenient fashion using an architecture that can be easily integrated with existing execution frameworks and problem solving environments. The standalone implementation of SimX presents SimX as a set of libraries; a user needs to re-write and re-compile his simulation code with calls to the SimX library in order to incorporate it into an MES. The SimX/SCIRun version packages SimX as a set of SCIRun modules; the user can include those modules into existing SCIRun applications to turn their SCIRun nets into MESs. Finally, the SimX/Uintah version packages SimX as modified Uintah components; the user can turn his Uintah workflow into an MES by attaching the SimX component into the existing Uintah workflow.

The four examples MESs used in this thesis cover a large range of properties that one expects to find across different computational studies. The examples cover MESs that: 1) are composed of both serial and parallel experiment code, 2) have per-experiment runtime that range from milli-seconds to hours, 3) require experiment counts that range from dozens to tens of thousands, 4) have varying reliance on user input, 5) have both concrete and subjective definitions of their study objectives, and 6) have dimensionality of the exploration space that range from two to over a hundred. Therefore, the results obtained in this thesis is expected to be applicable to other MESs.

7.2 Future Work

As described in Chapter 2, this thesis was built upon numerous previous efforts related to parallel steering, parameter sweep, and parallel runtime systems. This thesis also opens up a number of future research areas.

Interaction between search algorithm, resource allocation, and reuse: One important insight presented in this thesis is that the scheduling and resource allocation strategies often can inform each other. For example, in active sampling, by scheduling coarse grid experiments first, one can narrow the search space of finer levels. In studies with moldable experiments, this insight presents more interesting implications. We have shown in the helium study, for example, that it is actually worthwhile to incur the parallelization overhead of the Arches code by running the code on all 32 worker processes instead of running each experiment on their own worker process, because the speed up due to result reuse and active sampling can outweigh the parallelization penalties. In general, when running an MES, one may want to assign "importance" to some experiments over another, due to these experiments' reuse potential, or due to them being expected to give important information for the search algorithm. How such importance should be assigned is not immediately clear. For example, when conducting an MES to search a large design space using a genetic algorithm, the user may want to give more computational resources to offsprings of well-performing parents, or he might want to give more resources to offsprings formed from mutation in order to explore new strands.

Study-level steering interface: Much previous work has been done to show how a user can steer a single parallel code. The animation study and defibrillator design studies showed that it is possible for a user to steer an entire computational study consisting of thousands of simultaneous individual simulations. While these results are promising, much work remains in understanding how users should interact with MESs in general. In particular, what are the modes with which humans perceive aggregated information from multiple sources, and how do they steer multiple executions at once? It is likely that eventual solutions to this problem will require participation from multiple sub-areas within computer science, including systems and Human-Computer Interaction.

Automated collection of application-specific information: On the more system-oriented side, some of the application-specific knowledge used in this thesis can conceivably be automatically generated. The scaling behavior required for optimzal batching in the helium model validation MES, for example, can conceivably be collected by SimX automatically. Self-optimizing code like FFTW or Atlas ([39, 83, 82]) conducts test runs to determine the information about the hardware it is running on. When FFTW is being installed on a platform, for example, it performs test runs to determine the hardware's characteristics, such as the cache size, register bank size, cache line block size, etc. It then installs a version of itself optimized for that particular platform based on those characteristics. In a similar vein, a SimX-like system should be able to perform test runs on the application code to determine information such

as the code's scaling behavior or even the expected benefits of result reuse, and adapt itself to the application as the computational study is being conducted. In order to accomplish this automated collection of application-specific information, a well-defined API has to be designed, and additional work is needed to identify what type of information can be obtained automatically, and what has to be supplied by the user.

Using application-domain knowledge in storage systems: The current TCP server-based SISOL implementation is only one way to implement a shared object system. In all four of our example MESs, the SISOL has not become a bottle-neck — when it did we simply increased the number of SISOL data servers. However, other MESs may put more pressure on their shared object layer. In those MESs, a more efficient shared object layer may be needed. There, the shared object layer could make use of application-domain knowledge to optimize its performance. For example, it could use the notion of a priority region, where some objects are known to be more important than others (more likely to be reused, for example), and give special treatment to them (cached in SISOL clients, has extra index for lookup in SISOL servers, etc) in order to improve the performance of operations pertaining to high-priority objects. Also, in the current implementation of SISOL, the user has to set aside a number of processing elements on the cluster to run the SISOL servers. These processing elements thus cannot be used to run worker processes. To put these processing elements into the worker process pool, the feasibility of a distributed cooperative store implementation (e.g., [51]) of SISOL can be investigated. Here, SISOL has

to contend with the simulation code for CPU power and with FUEL for network bandwidth, and the knowledge of the simulation code — when the code enters phases of intense CPU usage or intense network usage — can be useful in wisely scheduling the resources of a single processing element. Furthermore, the shared object layers could be made to be non-volatile, i.e., the objects stored can survive beyond the life time of the entire computational study. That way the user can checkpoint *entire studies*. This will come in useful for conducting long-running computational studies where each individual run of an experiment itself could last beyond the length of an allocation of nodes on the cluster.

Defining simulation-level and system-level component interfaces: The Common Component Architecture (CCA) [4, 3] is a set of standard interfaces that allow application developers to package common scientific operations into components. Users can then take these components to compose applications. One possible use of CCA is to design components that perform operations to conduct Multi-Experiment Studies. Such MES components would require an interface similar to that of SimX's. The component-based versions of SimX — SimX/SCIRun and SimX/Uintah — represent one set of possible interfaces for such an MES component. An MES component would take SimX's interface and make them compliant with CCA. The SimX experience, including the type of application-specific knowledge that are found to be beneficial, can inform the design of such a component.

7.3 Conclusion

Incorporating application domain knowledge to guide its decision-making process, a parallel runtime system can efficiently manage the collection of experiments in a Multi-Experiment Study, achieve user-interactivity at the study-level and reduce the overall run time. With an end-to-end system view that encompasses the execution platform, the application being executed, and the requirement of the computational study, the system can take advantage of optimization opportunities that are not available when it focuses on one or two of these elements alone.

In particular, the study's goal can dictate scheduling decisions with an end result of reducing the number of experiments needed, user interactivity can help drive the study and allow subjective steering, the experiment code's internal intermediate states can be reused to reduce the per-experiment run time, and the experiment code's scaling behavior and reuse pattern can influence the resource allocation decisions to maximize resource utilization and reuse potential. For example, without knowing that the bridge design study is Pareto Frontier exploration, active sampling would not be available; without knowing the internal structures of DefibSim, intermediate result reuse would not be applicable; without user interactivity, the animation design study could not make meaningful progress; and without knowing about reuse classes, the system would not be able to schedule the experiments in a way that maximizes reuse opportunities.

Therefore, it is important for system designers not to treat each execution of the code as separate black boxes, but rather to consider the inner structures

of the application codes, as well as the purpose and intent of the users who run those application codes on their systems, often multiple times. To do so, they need to rethink how the system-application interface are designed: how to design a system API that allows the system to obtain and take advantage of application-specific knowledge? What system-level optimization opportunities are afforded by such application-level knowledge? With a well-defined interface, the system designer can obtain the view of the entire end-to-end system, rather than just one or two sub-parts of it, and then take advantage of optimization opportunities that are only present with that view.

SimX is only one example of such a system. In order to efficiently take advantage of the amount of parallelism available both in massively parallel architectures as well as small-scale parallel interactive applications, more SimX-like systems are likely to be designed and deployed.

Bibliography

[1] David Abramson, Andrew Lewis, Tom Peachey, and Clive Fletcher. An automatic design optimization tool and its application to computational fluid dynamics. In *SuperComputing 2001*, 2001.

[2] David Abramson, Rok Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *Proceedings of 4th IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 112–121, 1995.

[3] Benjamin A. Allan, Robert Armstrong, David E. Bernholdt, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. Mclnnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, 20(2):163–202, 2006.

[4] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 13, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Jean-Francois Aubry, Frederic Beaulieu, Caroline Sevigny, Luc Beaulieu, and Daniel Tremblay. Multiobjective optimization with a modified simulated annealing algorithm for external beam radiotherapy treatment planning. *Medical Physics*, 33(12):4718–4729, 2006.

[6] Chuck A. Baker, Layne T. Watson, Bernard Grossman, William H. Mason, Steven E. Cox, and Raphael T. Haftka. Study of a global design space exploration method for aerospace vehicles. In *Proceedings of the 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, September 2000.

[7] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[8] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[9] Mikls Bergou, Saurabh Mathur, Max Wardetsky, and Eitan Grinspun. Tracks: Toward directable thin shells. In *SIGGRAPH (ACM Transactions on Graphics)*, 2007.

[10] Jim Blythe, Ewa Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[11] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chirs Ferris, and David Orchard. Web services architecture. Technical report, W3C, 2004.

[12] J. Branke, H. Schmeck, K. Deb, and Reddy. M. Parallelizing multiobjective evolutionary algorithms: Cone separation. In *Proceedings of the Congress on Evolutionary Computation (CEC-2004)*, pages 1952–1957, 2004.

[13] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational steering in realitygrid. In *Proceedings of the UK e-Science All Hands Meeting*, 2003.

[14] Erick Cantu-paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10:141–171, 1997.

[15] Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, 2001.

[16] Heather A. Carlson and J. Andrew McCammon. Accommodating protein flexibility in computational drug design. *Molecular Pharmacology*, 57:213–218, 2000.

[17] Henri Casanova, Thomas Bartol, Francine Berman, Adam Birnbaum, Jack Dongarra, Mark Ellisman, Marcio Faerman, Erhan Gockay, Michelle Miller, Graziano Obertelli, Stuart Pomerantz, Terry Sejnowski, Joel Stiles, and Rich Wolski. The virtual instrument: Support for grid-enabled scientific simulations. *International Journal of High Performance Computing Applications*, 18(1):3–17, 2004.

[18] Henri Casanova and Jack Dongarra. Netsolve: A network server for computational science problems. *International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[19] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in cool. *ACM SIGPLAN Notices*, 28(7):249–259, 1993.

[20] Carlos A Coello, David A Van Veldhuizen, and Gary B Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Kluwer, 2002.

[21] Scientific Computing and Imaging Institute. SCIRun Web page, 2008. http://software.sci.utah.edu/scirun.html.

[22] Peter Coveney, Jamie Vicary, Jonathan Chin, and Matt Harvey. Introducing weds: a wsrf-based environment for distributed simulation. Technical Report UKeS-2004-07, University College London, 2004.

[23] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. *ACM SIGARCH Computer Architecture News*, 19(2):164–175, April 1991.

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

[25] J. Davison de St. Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. Uintah: a massively parallel problem solving environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing, 1-4 Aug. 2000*, Proceedings the Ninth International Symposium on High-Performance Distributed Computing, pages 33–41, Pittsburgh, PA, USA, 2000. IEEE Comput. Soc.

[26] J. Davison de St. Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A massively parallel problem solving environment. In *HPDC '00: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, page 33, Washington, DC, USA, 2000. IEEE Computer Society.

[27] J.D. de St. Germain, J. McCorquodale, S.G. Parker, and C.R. Johnson. Uintah: A massively parallel problem solving environment. In *Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 33–41. IEEE, Piscataway, NJ, Nov 2000.

[28] Paul E. DesJardin, Timothy J. O'Hern, and Sheldon R. Tieszen. Large eddy simulation and experimental measurements of the near-field of a large turbulent helium plume. *Physics of Fluids*, 16(6):1866–1883, 2004.

[29] R. Diekmann, R. Luling, and J. Simon Y. Problem independent distributed simulated annealing and its applications. In *Applications, Applied Simulated Annealing, Lecture Notes in Economics and Mathematical Systems, Springer LNEMS 396*, pages 17–44. Springer, 1993.

[30] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[31] E.Laure and H.Moritsch. Portable parallel portfolio optimization in aurora financial management system. In *Proceedings of SPIE ITCom Conference: Commercial Applications for High-Performance Computing*, August 2001.

[32] S. Stuecke et al. Open grid services infrastructure (ogsi), 2003. Global Grid Forum. http://www.ggf.org/documents/GFD.15.pdf.

[33] Marcio Faerman, Adam Birnbaum, Henri Casanova, and Francine Berman. Resource allocation for steerable parallel parameter searches. In *Grid Computing - GRID 2002*, pages 157–168, 2002.

[34] Center for Applied Scientific Computing (CASC). Hypre Web page, 2008. http://acts.nersc.gov/hypre/.

[35] Organization for the Advancement of Structured Information Standards (OASIS). WSRF Web page, 2008. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[36] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal on Supercomputer Applications*, 15(3), 2001.

[37] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[38] Robert J. Fowler and Leonidas I. Kontothanassis. Improving processor and cache locality in fine-grain parallel computations using object-affinity scheduling and continuation passing. Technical Report TR-411, University of Rochester, Department of Computer Science, Rochester, NY, USA, 1992.

[39] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[40] Fujita and Yamashita. Approximation algorithms for multiprocessor scheduling problem. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 2000.

[41] G. A. Geist, Ii James, Arthur Kohl, and Philip M. Papadopoulos. Cumulvs: Providing fault tolerance, visualization, and steering of parallel applications. *The International Journal of Supercomputer Applications and High Performance Computing*, 11:224–236, 1997.

[42] J. Davison De St. Germain, Alan Morris, Steven G. Parker, Allen D. Malony, and Sameer Shende. Performance analysis integration in the uintah software development cycle. *International Journal of Parallel Programming*, 31(1):35–53, 2003.

[43] Rony Goldenthal and Michel Bercovier. Design of curves and surfaces using multi-objective optimization, 2004.

[44] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[45] Network Working Group. Xdr: External data representation standard, 2006. http://www.ietf.org/rfc/rfc4506.txt.

[46] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[47] Jeffrey Horn, Jeffrey Horn, Nicholas Nafpliotis, Nicholas Nafpliotis, David E. Goldberg, and David E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *In Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, pages 82–87, 1994.

[48] Ken Brodlie Jason Wood and Jeremy Walton. Gviz - visualization and steering for the grid. In *Proceedings of e-Science All Hands Meeting*, Nottingham, UK, 2003.

[49] Christopher R. Johnson and Steven G. Parker. A computational steering model applied to problems in medicine. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 540–549, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[50] C.R. Johnson, R.S. MacLeod, S.G. Parker, and D.M. Weinstein. Biomedical computing and visualization software environments. *Communications of the ACM*, 47(11):64–71, 2004.

[51] V. Karamcheti and A. Chien. Architectural Support and Mechanisms for Object Caching in Dynamic Multithreaded Computations. *Journal of Parallel and Distributed Computing*, 58(2), 1999.

[52] Vijay Karamcheti and Andrew A. Chien. A hierarchical load-balancing framework for dynamic multithreaded computations. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society.

[53] J. Kettenbach, A. Schreyer, S. Okuda, M.O. Sweeney, S. Eisenberg, C.-F. Westin, F. Jacobson, R. Kikinis, B.H. KenKnight, and F.A. Jolesz. 3d modeling of the chest in patients with implanted cardiac defibrillator for further bioelectrical simulation. In *Proceedings of the 12th International Symposium and Exhibition on Computer Assisted Radiology and Surgery*, volume 12, pages 194–198. Computer Assisted Radiology and Surgery, 1998.

[54] Pansoo Kim and Yu Ding. Optimal engineering system design guided by data-mining methods. *Technometrics*, 47(3):336–354, August 2005.

[55] James A. Kohl, Torsten Wilde, and David E. Bernholdt. Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance. *International Journal of High Performance Computing Applications*, 20(2):255–285, 2006.

[56] Robert Van Liere, Jurriaan D. Mulder, and Jarke J. van Wijk. Computational steering. *Future Generation Computer Systems*, 12(5):441–450, 1997.

[57] F. Luna, A.J. Nebro, and E. Alba. A globus-based distributed enumerative search algorithm for multi-objective optimization. Technical Report LCC 2004/02, Departamento de Lenguajes y Ciencias de la Computacin. Universidad de Mlaga, March 2004.

[58] R.S. Macleod, D.M. Weinstein, J.D. de St. Germain, D.H. Brooks, C.R. Johnson, and S.G. Parker. Scirun/biopse: Integrated problem solving environment for bioelectric field problems and visualization. In *Proceedings of the International Symposium on Biomedical Imaging*, pages 640–643, April 2004.

[59] V. Mann, V. Matossian, R. Muralidhar, and M. Parashar. Discover: An environment for web-based interaction and steering of high-performance scientific applications. *Concurrency and Computation: Practice and Experience*, 13:737–754, 2001.

[60] J. McCorquodale, J.D. de St. Germain, S. Parker, and C.R. Johnson. The uintah parallelism infrastructure: A performance evaluation on the sgi origin 2000. In *Proceedings of The 5th International Conference on High-Performance Computing*, Mar 2001.

[61] Achille Messac. Physical programming: Effective optimization for computational design. *American Institute of Aeronautics and Astronautics Journal*, 31(4):149–158, 1996.

[62] Michelle Miller, Charles D. Hansen, Steven G. Parker, and Christopher R. Johnson. Simulation steering with scirun in a distributed memory environment. In *In Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7*, 1998.

[63] G. Moore. Nanometers and gigabucks - moore on moore's law, 1996.

[64] National Institute of Standards and Technology. Verification and validation, 2008. http://fire.nist.gov/fds/verification_validation.html.

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

[66] Steven G. Parker and Christopher R. Johnson. Scirun: a scientific programming environment for computational steering. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 52, New York, NY, USA, 1995. ACM.

[67] G. Ch. Pflug, A. Swietanowski, E. Dockner, and H. Moritsch. The aurora financial management system: Model and parallel implementation design. *Annals of Operations Research*, 99:189–206, 2000.

[68] G.Ch. Pflug and A. Swietanowski. Dynamic asset allocation under uncertainty for pension fund management. Technical Report AURORA TR1998-15, University of Vienna, 1999.

[69] Beth Plale, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. *IEEE Concurrency*, 6(2), 1998.

[70] Gunter Rudolph and Fachbereich Informatik. Massively parallel simulated annealing and its relation to evolutionary algorithms. *Evolutionary Computation*, 1:361–383, 1994.

[71] J. Schmidt and C. Johnson. DefibSim: An interactive defibrillation device design tool. In *Proceedings of the IEEE Engineering in Medicine and Biology Society Conference*, 1995.

[72] J.A. Schmidt, C.R. Johnson, and R.S. MacLeod. An interactive computer model for defibrillation device design. In *International Congress on Electrocardiology*, pages 160–161, 1995.

[73] M. Scott and E. Antonsson. Preliminary vehicle structure design: An industrial application of imprecision in engineering design.

[74] Russell Smith. Open Dynamics Engine Web page, 2000. http://ode.org/ode.html.

[75] J.P. Spinti, J.N. Thornock, E.G. Eddings, P.J. Smith, and A.F. Sarofim. *Transport Phenomena in Fires*, chapter Heat Transfer to objects in pool fires. Witpress, 2008.

[76] Srividya Srinivasan, Vijay Subramani, Rajkumar Kettimuthu, Praveen Holenarsipur, and P. Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *High Performance Computing - HiPC 2002*, pages 174–183, 2002.

[77] Sudha Srinivasan, Savitha Krishnamoorthy, and P. Sadayappan. A robust scheduling strategy for moldable scheduling of parallel jobs. *IEEE International Conference on Cluster Computing (CLUSTER'03)*, 00:92, 2003.

[78] The BlueGene/L Team. An overview of the bluegene/l supercomputer. In *Proccedings of ACM Supercomputing Conference, 2002*, 2002.

[79] The Condor Team. Condor DAGMan Web page, 2008. http://www.cs.wisc.edu/condor/dagman.

[80] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[81] Christopher D. Twigg and Doug L. James. Many-worlds browsing for control of multibody dynamics. *ACM Transactions on Graphics (SIGGRAPH 2007)*, 26(3), aug 2007.

[82] C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. Technical Report UT-CS-00-448, University of Tennessee Computer Science Department, 2000.

[83] R. Whaley and J. Dongarra. Automatically tuned linear algebra software (atlas). In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE, 1998.

[84] Clemens Wiesinger, David Giczi, and Ronald Hochreiter. An open grid service environment for large-scale computational finance modeling systems. *Springer Lecture Notes in Computer Science*, 3036:83–90, 2004.

[85] Benjamin Wilson, David Cappelleri, Timothy W. Simpson, and Mary Frecker. Efficient pareto frontier exploration using surrogate approximations. *Optimization and Engineering*, 2(1):31–50, 2001.

[86] P. R. Woodward. Perspectives on supercomputing: Three decades of change. *IEEE Computer*, 29(10):99–111, 1996.

[87] S. Yau, K. Damevski, V. Karamcheti, S. Parker, and D. Zorin. Result reuse in design space exploration: A study in system support for interactive parallel computing. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (22th IPDPS'08)*, Miami, FL, April 2008. IEEE Computer Society.

[88] S. Yau, K. Damevski, V. Karamcheti, S. Parker, and D. Zorin. Application-aware management of parallel simulation collections. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), February 2009*, Raleigh, NC, February 2009. ACM.

[89] S. Yau, E. Grinspun, V. Karamcheti, and D. Zorin. Sim-X: Parallel system software for interactive multi-experiment computational studies. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)*, Rhodes Island, Greece, April 2006. IEEE Computer Society.

[90] Siu Man Yau, Eitan Grinspun, Vijay Karamcheti, and Denis Zorin. Simx meets scirun: A component-based implementation of a computational study system. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (21th IPDPS'07)*, pages 1–6, 2007.