Application-Aware Management of Parallel Simulation Collections

Siu-Man Yau Vijay Karamcheti
Denis Zorin
Courant Institute of Mathematical Sciences,
New York University
{smyau,vijayk,dzorin}@cs.nyu.edu

Kostadin Damevski Steven G. Parker
Department of Computer Science, University of Utah
{damevski,sparker}@cs.utah.edu

Abstract

This paper presents a system deployed on parallel clusters to manage a collection of parallel simulations that make up a computational study. It explores how such a system can extend traditional parallel job scheduling and resource allocation techniques to incorporate knowledge specific to the study.

Using a UINTAH-based helium gas simulation code (ARCHES) and the SimX system for multi-experiment computational studies, this paper demonstrates that, by using application-specific knowledge in resource allocation and scheduling decisions, one can reduce the run time of a computational study from over 20 hours to under 4.5 hours on a 32-processor cluster, and from almost 11 hours to just over 3.5 hours on a 64-processor cluster.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design—Batch processing systems, Distributed systems, Hierarchical design; C.5.m [*Computer System Organization*]: Computer System Implementation—Miscellaneous

General Terms Design, Performance

Keywords Parallel System, High-throughput computing

1. Background

Computer simulation has become an integral part of the scientific method, often delivering deeper insights into complex physical processes than possible using only the traditional dyad of theory and experiment. Often, such insights manifest themselves in the scientific exploration process in the form of *computational studies* built out of multiple *computational experiments* corresponding to individual runs of simulation software. Examples of such studies range from exploration of design spaces in engineering to molecular simulations for drug design.

Traditionally, much research effort has been put into discovering ways to speed up the performance of a single simulation under high levels of parallelism. However, relatively less effort has been directed toward improving the performance of *entire computational studies*. As gains in computation power is increasingly being delivered on ever-higher levels of parallelism, the ability to exploit parallelism inherent in computational studies becomes more important.

One example that demonstrates this rift is in Design Space Exploration (DSE). DSE is a type of computational study used in various disciplines such as automotive design, mechanical engineering, electrical engineering, chemical engineering and medicine (15; 21; 9; 5; 14). An example of multi-experiment studies, DSE relies on multiple executions of a simulation code using different input parameters to discover a region of interest. Each execution would simulate, for example, the dynamics of a car crash under variously-shaped body frames, or the accuracy of a chemical simulation software under different values of model parameters. Out of all these simulations, a subset is then identified as the 'best' and singled out as the belonging to the region of interest.

Since the simulations in a DSE are completely independent of each other, DSE can be a trivially parallel problem. However, as shown in (1) and (23), if the runtime system understands the characteristics of the underlying DSE, the system can use the results of earlier simulations to guide the exploration in later simulations, significantly reducing the overall size of the search space. Additionally, (22) shows that, if the runtime system can store the results and internal states of some early simulations, they can be reused by later simulations, thus reducing the runtime of later simulations and the overall run time of the DSE.

Hence, not all simulations are created equal: some simulations may be more important because their results and internal states have more reuse potential, or because their results are beneficial to trimming the design space in later exploration. Thus, if the system can devote more computational resources to these higher-priority simulations, it could improve the runtime of the DSE.

(23) and (22) have shown how application-specific knowledge can be used to improve the performance of computational studies made up of multiple execution of serial simulation codes. This paper expands on that work by looking at how application-specific knowledge can be used to schedule and allocate resources for computational studies made up of multiple execution of *parallel* simulations codes. In particular, we examine how to expand on the work in traditional parallel job scheduling, described in Section 6, to incorporate application-specific knowledge. This knowledge helps the system decide which simulations to run, and in what order, as well as to decide which processing elements to devote to execute each simulation.

Section 2 describes the formulation of the multi-experiment computational study problem, and presents the helium model validation study (5), which serves as a running example throughout the paper. Section 3 introduces the SimX system, a runtime system designed to conduct multi-experiment studies. Section 4 describes how application-level knowledge can influence system decisions

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

made by SimX. In Section 5, we evaluate the effectiveness of various types of application-level knowledge in the helium model validation study. Section 6 provides an overview of previous work, and we conclude in Section 7.

2. Problem Description

This section presents the helium model validation study, which serves as the running example for this paper.

2.1 Helium Model Validation

To understand how gases mix during a fire, a computational model for simulating turbulent mixing of gases of different densities was developed at the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE). The model is implemented in an application known as ARCHES (16), which is built using the UINTAH Problem Solving Environment (PSE) (4).

To validate the model, the results computed by ARCHES are compared against the results obtained from a real-life experiment (5). In the experiment, helium is pumped into a cube-shaped container from an inlet at the bottom of the container. The centerline gas velocity profiles are measured at three heights. These measured results are then used to compare against that computed by ARCHES.

ARCHES, however, contains many input parameters whose exact values are not precisely known. 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 helium inlet's gas velocity, or the size of the inlet. The goal in the validation study is to find the values of these parameters such that the difference between the centerline velocities produced by the simulation code and the centerline velocities measured in the real-life experiment are minimized. Figure 1 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 0.2, as its velocity profile is a better match for the measured profile, but at height 0.4, the two configurations are more evenly-matched.

Validation study is a *multiple objective* optimization problem, where the user searches the space spanned by the input parameters in order to optimize for multiple objectives, in our case, the difference of the centerline velocity profile at each height.

For this paper, we consider a simplified version of the study. We only explore two input parameters: the Prandtl number and inlet velocity, and measure the "goodness" of each set of inputs using two centerline velocity profiles instead of three.

To conduct the simplified study, the user executes ARCHES multiple times, each time using a different combination of Prandtl number and inlet velocity. At the end of each execution, the user takes the centerline velocities from the execution at two heights, and calculates the difference between the measured and simulated velocities at each heights. He then looks for the set of input parameters that minimize those differences.

This simplified version preserves the properties of the full version; the only difference is that there are fewer objectives and fewer input parameters. The principles learned from designing the runtime system to support this simplified version are applicable to the full system.

2.2 Pareto Optimization

Keeping with the terminology used in (23), (24), and (22), we refer to the space spanned by Prandtl number and inlet velocity as the *Design Space*, each combination of Prandtl number and inlet velocity values as a *design*, and the optimization problem as *Design Space Exploration* (DSE). The velocity profile difference on each height is known as a *Performance Metric*, and the space spanned by performance metrics is known as the *Performance Space*.

As discussed above, the validation study is a multi-objective optimization problem. A common strategy for multi-objective optimization is *Pareto optimization* (e.g., (10), (21)). Pareto optimization seeks to find a *set* of designs, with each *Pareto-optimal* design corresponding to a different trade-off of optimization objectives, e.g., two designs may be equally desirable, one for its good velocity profile match at a lower height, and the other for its good match on a higher height.

A Pareto-optimal design has the property that improving one measure can only be achieved at the expense of another, e.g., a design is Pareto-optimal if the velocity difference on one height cannot be improved while holding the velocity difference on the other height fixed. This set of optimal designs is the *Pareto frontier*.

Figure 2 shows the result of the simplified validation study. A region of the design space is sampled on a regular grid (left), and their performance metrics are plotted on the performance space (right). The simulation code can be thought of as a function that maps a point on the design space to a point on the performance space. The North-East most designs on the performance space (circled in red) form the Pareto Frontier. If a point is on the Pareto Frontier, there is no other point on the design space that yields a lower value on both performance metrics. The goal of the Pareto Optimization is to discover where on the design space is the Pareto Frontier.

2.3 Implementation Details

Like all UINTAH applications, ARCHES is an MPI-based timestepping application. It is designed to run on parallel clusters, and uses a shared file system to store the simulation result and checkpoints.

ARCHES simulates gas flow in three dimensions. For the initial condition, the simulation domain is filled with air with zero velocity. A boundary condition is placed in the gas inlet, where helium gas is introduced at constant velocity. At each timestep, the pressure, velocity, and gaseous mixture in the simulation domain is updated. Due to the buoyancy of helium and the velocity with which the helium gas is pumped into the system, the simulated domain undergoes a "puffing" motion (Figure 3), where the gas at the central part of the domain is pushed upwards, resulting in a velocity profile like Figure 1. ARCHES writes checkpoints (the pressure, velocity, and gaseous mixture) to disk periodically, and continues to timestep until the total kinetic energy of the simulation domain reaches a steady state.

The time to execute an ARCHES run thus depends on the per-timestep run time and the number of timesteps 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.

For the simplified version of the study, an ARCHES run typically takes more than a half-hour on 32 processors, which is prohibitively long: it would take over 20 hours to complete a study that requires achieving the grid resolution shown in Figure 2. Consequently, the user can only explore a few points in an ad hoc fashion.

However, this paper shows that, by combining knowledge specific to the validation study with traditional scheduling techniques in parallel systems, the user could explore the design space automatically, systematically and efficiently, and complete the study in a few hours.

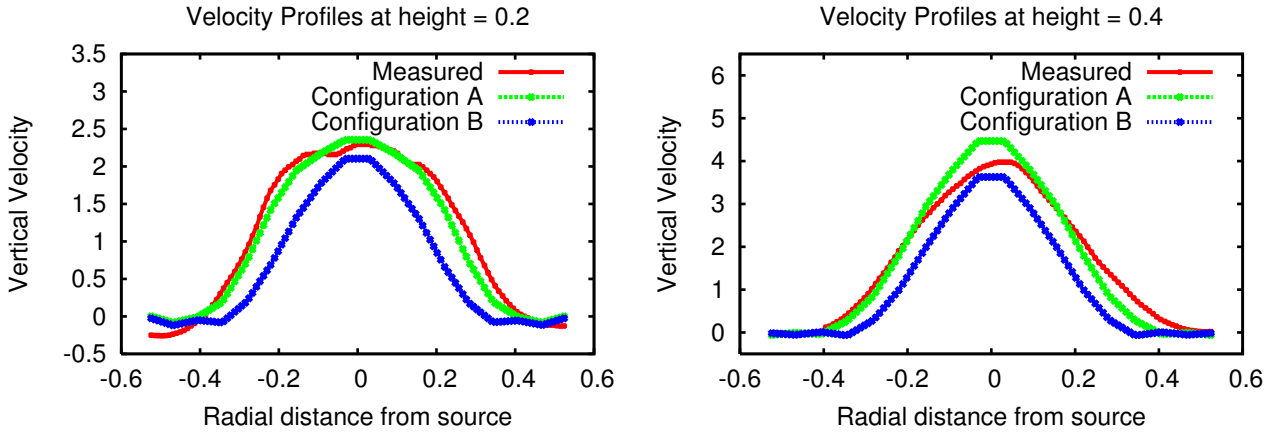

Figure 1. Comparing the velocity profiles of two input configurations at two heights. Configuration A's velocity profile is a better match with the measured profile than Configuration B's at height = 0.2. At height = 0.4, the two configurations are more evenly-matched.

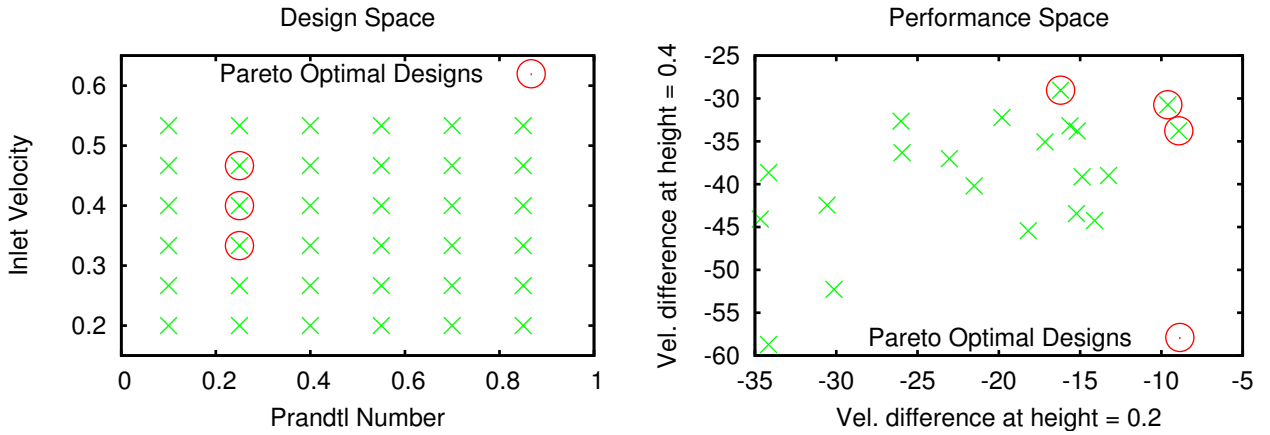

Figure 2. Pareto Optimization Validation: the design space (left) is sampled on a regular grid, and the performance metric is plotted on the performance space (right). The *Pareto Optimal* designs are circled in red in both plots.

While this work focuses on the validation study, the techniques used here can easily be generalized to other Pareto optimization studies or studies involving multiple parallel malleable or moldable simulations.

3. SimX

The Parallel System Software for Interactive Multi-Experiment Computational Studies (SIMECS, or SimX for short) is a runtime system designed for conducting interactive computational studies on parallel clusters (23). We use SimX as a test bed to experiment with incorporating application-specific knowledge in the running of the helium model validation study. This section offers an overview of the SimX architecture and interface.

3.1 SimX architecture

The SimX architecture is presented in Figure 4. 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.

A typical workflow in SimX proceeds as follows.

After the user is allocated a partition of the cluster, he starts up the SISOL servers on one or more of the assigned cluster nodes, and launches a single MPI job on the rest. Each of these MPI processes is a worker process. He then starts up the manager process on his front-end machine.

The user uses the UI module in the manager process to specify the DSE he wants to conduct. A module known as the *Sampler* decides which computational experiments need to be executed, and puts those experiments into the *Task Queue*. A third module, the *Resource Allocator*, decides which worker processes should be assigned each particular task, and sends this task over to the worker processes, instructing them to form into a processor sub-group and run the simulation.

Each worker process is part of two MPI communicators: the global communicator (i.e., MPI_COMM_WORLD), used by the SimX system to form processor groups, and a sub-communicator, used by the simulation code to run the application. Initially, the sub-communicator is a duplicate of MPI_COMM_WORLD. When

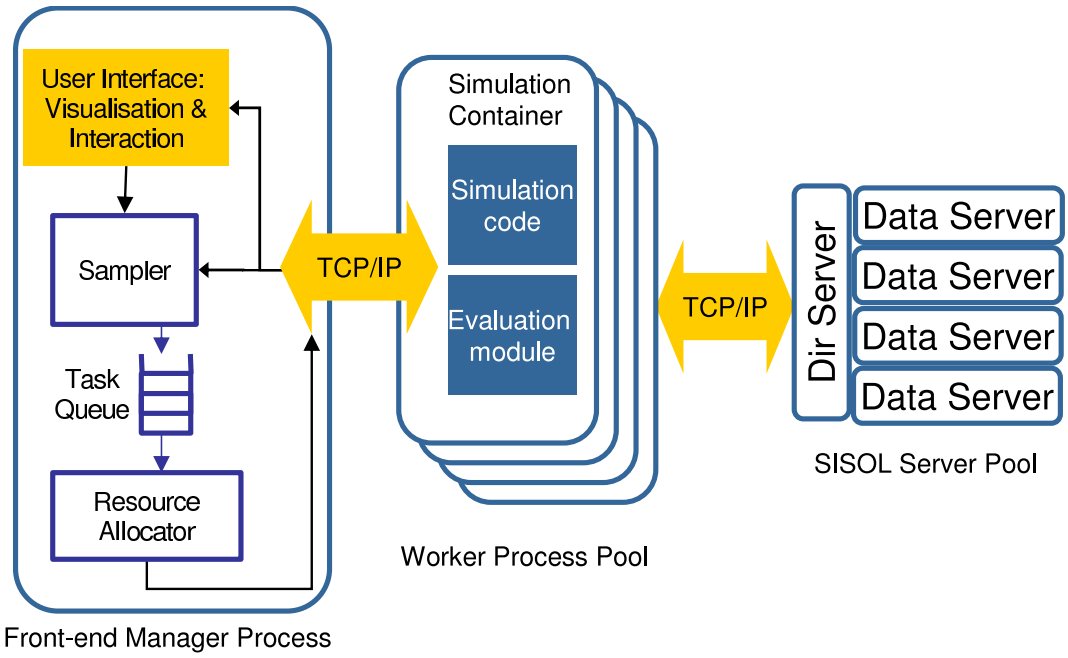

Figure 4. An overview of the SimX architecture.

a worker process receives an instruction from the Resource Allocator to form a group, it destroys its old sub-communicator, and rebuilds a new MPI sub-communicator with the other processes making up the same 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 worker processes are essentially *Simulation Containers*; they do little more than to serve as a substrate on which the simulation code is run. When the Simulation Container receives a simulation to run, it will consult the SISOL first to see if there are opportunities to optimize the simulation run (see Subsection 4.1). Then it hands over the thread of execution over to the simulation code. The simulation code may periodically return control to the Simulation Container to check if there is a need to terminate the simulation early (see Subsection 4.3). When the simulation is finished, the Simulation Container writes the simulation result into SISOL to enable their use in optimizing future simulation runs. It also executes an evaluation module to calculate the performance metric for the experiment just conducted. It then sends the performance metric back to the manager process.

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 (see Subsection 4.2).

The modular structure of SimX separates out key functionalities that optimization techniques can be adapted onto each module independently as dictated by the needs of the particular DSE being conducted.

3.2 SimX interface

In the workflow described above, only the simulation code (in our case, ARCHES) and the evaluation code (in our case, the velocity profile comparator) are provided by the user. The UI module is part of SimX/SCIRun (24; 13; 12). The Sampler, Task Queue, Resource Allocator, and Simulation Container modules, as well as the SISOL servers, are all part of the SimX package. However, these modules have a set of APIs that allows the user to customize their behaviors. This set of APIs is the mechanism through which the user can provide application-specific knowledge to SimX.

Table 1 shows a simplified version of the **Sampler** module's API. The Sampler module converts user specifications of the DSE into sample points in the design space for which simulations need to be run. In our case, the Sampler module issues a list of <Prandtl Number, Inlet Velocity> tuples, each one representing a set of input parameters to an execution of ARCHES.

The Sampler module occupies an intermediate position between system and application software. Adding domain knowledge to the Sampler is likely to enhance its performance, but narrow the applicability of the system; making the Sampler completely application-independent may result in suboptimal sampling strategies in some cases. In our case, we use a sampler module specialized in Pareto optimization. This Sampler module is discussed at length in Subsection 4.2.

The **Task Queue** represents a list of experiments issued by the Sampler. There, a subset is selected into a *batch*. The batch consist 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 user can employ the Task Queue's API (Table 2) 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. Subsection 4.1 discusses how the Task Queue is used. As is shown by the next section, the user can adapt the system's scheduling policy to the needs of his application by controlling how experiments are batched.

The **Resource Allocator** is responsible for assigning the batched tasks from the Task Queue to the worker processes. It is also responsible for configuring the worker processes so that the sizes of

Inferface	Explanation
`struct StudyID {vector<range> DesignSpaceRanges, SamplerType, InitialSamplerOptions}`	A data structure to identify a DSE study.
`SetCurrentStudy(StudyID)`	Called by the UI module to initialise a study.
`registerResults(Experiment, Values)`	Registers the result (performance metric) of an experiment.
`getNextPointToRun(hasPointReady)`	Gets the next experiment to issue from the current study.

Table 1. SimX interface to the Sampler.

Inferface	Explanation
`addTask(Experiment e)`	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 2. SimX interface to the Task Queue.

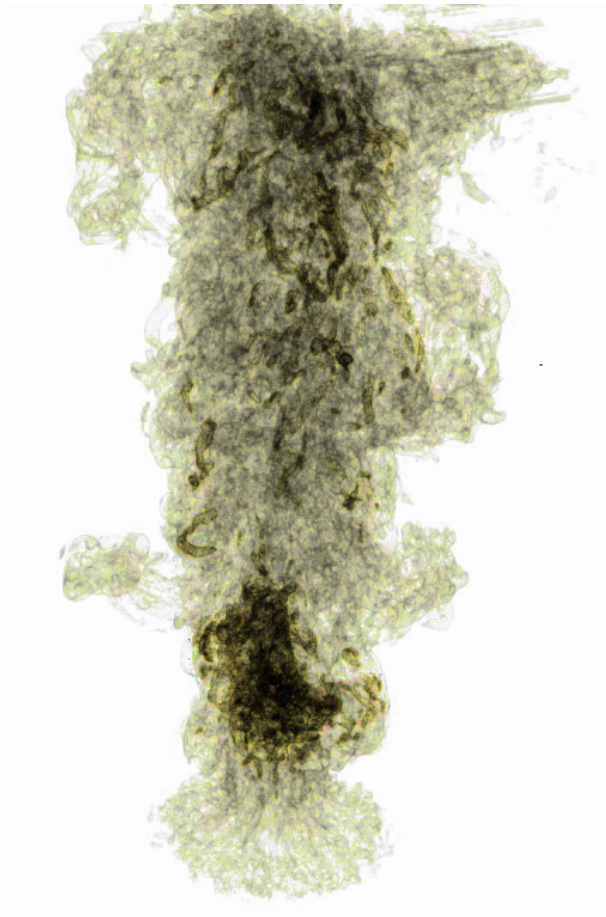

Figure 3. Visualizing the helium plume. Image courtesy of Chemical Reaction Simulation group, University of Utah

process groups available will be as close to the requirements of the task (corresponding to the return value of GetIdealGroupSize()) as possible. As discussed above, 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 destroys old ones. The `assign` array is an array of length N, where N is the total number of worker processes involved in the 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. Subsection 4.3 discusses how application-specific knowledge is used in the reconfigurations of worker process groups.

The **Spatially Indexed Shared Object Layer** (SISOL) provides implicit communication between simulation workers. In the current implementation, SISOL is implemented as a variable number of stand-alone data servers and one directory server communicating through TCP/IP. The data servers store the actual objects, in volatile memory, while the directory server stores the mapping from spatial coordinates to data servers. Table 3 shows the SISOL interface. Typically, objects on the SISOL are identified by their associated point on the design space, hence the objects are spatially-indexed. Subsections 4.1 and 4.3 discuss how SISOL is used.

Section 4 explains how these interfaces are used in concert to incorporate application-specific knowledge into running the helium model validation study.

4. Application-Aware Allocation

To conduct the helium model validation study efficiently, SimX makes use of various pieces of knowledge specific to the study. This section explains how such knowledge is exploited and how the required elements are introduced into SimX via its interfaces described in Subsection 3.2.

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 the 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 3. Interface of the spatially-indexed shared object space layer (SISOL).

4.1 Reuse classes

As discussed in Subsection 2.3, ARCHES run time depends on the per-timestep run time and number of timesteps required to reach a steady state. An inherent characteristic of ARCHES 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. This is a type of *result reuse*, where the result from one simulation can be used to speed up the execution of another. On average, without reuse, ARCHES takes 2900 timesteps; 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*.

SimX's Task Queue uses the knowledge of reuse classes to select its batches. When selecting a batch, it takes one simulation from each reuse class without a completed simulation (of the set referred to by the current tasks in the Task Queue). This way, the Task Queue can ensure that all subsequent runs have a result to reuse.

Another benefit to this batching policy is that the simulations issued within the same batch are either all starting from scratch, or all starting from a checkpoint. This way, the simulations issued together will have more comparable run times, limiting the sizes of "holes" on the scheduling graph.

To enable result reuse, a database of checkpoints is stored in SISOL. When ARCHES finishes, it will have written the simulation result on the disk as the last checkpoint. The Simulation Container writes this disk location into SISOL, using the <Prandtl number, Inlet Velocity> tuple as its spatial index. When a new simulation starts, the Simulation Container checks against SISOL to see if a checkpoint with the same inlet velocity is already on disk. If so, it instructs ARCHES to start from that checkpoint instead of starting from scratch.

4.2 Active Sampling

Another type of result reuse is *Active Sampling*. In our simplified validation study, we issue experiments on a 6x6 grid on the design space (see Figure 2). However, as explained in (23), it is possible to explore the space without running all 36 simulations. The Sampler can first issue experiments on a sparser 3x3 grid to identify a coarse version of the Pareto Frontier. This version represents a "promising" region of the design space. Then the Sampler issues experiments only for the finer grid points that are neighbors to the coarse Frontier. Using this technique, we can reduce the number of simulations needed from 36 to 24. Note that the active sampler would provide an even larger benefit for real-life computational studies involving a larger base number of experiments.

For SimX to take advantage of this application-specific knowledge, we specify the Active Sampler as the sampler type when we initiate the study. Note that the Active Sampler module (provided by SimX) is not only specific to the helium model validation study: it can be used for any Pareto Optimization studies.

This sampling technique is only possible because SimX knows the DSE being conducted is a Pareto Optimization. Unlike traditional parallel job scheduling systems, SimX is not interested in the results of the individual simulations - it is only interested in the aggregate, i.e., the Pareto Frontier. Therefore, it has freedom not to run some jobs if it is confident that they will not alter the aggregate.

4.3 Scaling behavior

Like most parallel codes, ARCHES does not scale perfectly because of parallelization overhead. The time it takes to execute a single timestep on different numbers of processors in a cluster environment is shown in Table 4.3.

The non-linear scaling behavior has several implications. First, in order to minimize parallelization overhead within each process group, SimX would like to schedule "long-and-thin" scheduling graphs - many executions running concurrently on relatively small process groups. Knowing the exact parallelization overhead can help Task Queue 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 4.3 need to be run. Assume for the moment that they all take the same exact number of timesteps, n. The optimal batching 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 is 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 4.3.

Then, for $2 \leq m$:

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

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

Table 4. ARCHES scaling behavior: run time per timestep

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.

To realize the benefits from scaling behavior-based batching, during initialization, the Task Queue loads the scaling behavior of the ARCHES code, and calculates the ideal batch sizes. When it is asked to create batches, it tries to batch the tasks according to the ideal batch sizes.

In the current implementation, SimX requires the user to explicitly specify the scaling behavior of ARCHES. A more sophisticated implementation of SimX can conceivably infer the scaling behavior automatically by performing test runs of ARCHES during the study's initialization.

4.4 Preemption

Unfortunately, the "ideal" batch sizes assumes that every simulation executes the same number of timesteps. Since simulations do require different timesteps, there will always be "holes" left on the scheduling graph. Separating checkpointed runs from non-checkpointed runs can mitigate this problem, but does not solve it.

Luckily, 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.

The scaling behavior can help 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., assuming 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 inequality must hold:

$$\alpha < 1 - \sigma.$$

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 information about its processors are written into the database.

As discussed previously, while ARCHES runs, it may periodically return control to the Simulation Container. The Simulation Container can look for idle worker processes on the SISOL database and decides whether to claim them. If it claims them, 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. Evaluation

To evaluate the performance of application-knowledge-enabled SimX, we use a 128 processor cluster of 2.4GHz Intel Xeon nodes each with 2GB memory and 1 Gigabit Ethernet interconnect to conduct the simplified version of the helium model validation study. We used various configurations of SimX, enabling subsets of the application-specific knowledge elements in order to measure the impact of each piece of knowledge. The configurations are as follows:

Configuration A: Use Active Sampler. No result reuse. The Task Queue issues one simulation at a time to 32 processors. No preemption.

Configuration B: Same as Configuration A; enable result reuse.

Configuration C: Same as Configuration B; the Task Queue issues simulations according to the optimal batch size based on ARCHES's scaling behavior.

Configuration D: Same as Configuration C; the Task Queue gives the first member of each reuse class higher priority.

Configuration E: Same as Configuration D; enable preemption.

In addition to the five configurations above, we estimate the time it would take to conduct the study with no application-specific knowledge, i.e., a brute force search with no reuse, no preemption, and always using 32 processors per simulation. We call this **Configuration O**.

For each configuration, we conduct the study on a set of 32 or 64 worker processes, 1 manager process, and 1 SISOL data server. The same problem size is used on all runs. We record the total wall clock time it takes to complete the study, the worker process utilization rate, the average run time of a single simulation, the percentage improvement over the previous configuration. The results are listed in Table 5.

Comparing Configurations O and A shows that Active Sampling cuts down the total run time by 33% and 27% on 32- and 64-processor configurations, respectively. The benefit is an artifact of how the run time of Configuration O is estimated: the brute force search requires 36 instead of 24 experiments. So, on 32 workers, Configuration A is estimated to be 33% faster. On 64 workers, however, since Configuration A always uses 32 processors per simulation, 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.

Comparing Configurations A and B shows the benefits of reusing results from prior checkpointed runs. The benefit is derived from the reduced average per-simulation run time: down to 24.7 mins for 32 processors and 26.8 mins for 64 processor runs. That translates to a 28% (32 processors) and 26% (64 processors) reduction in the study's run time.

Comparing Configurations B and C shows the benefits of ideal batching. 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% and 33% on 32- and 64-processor configurations respectively.

Comparing Configurations C and D shows the benefits of using the knowledge of reuse classes in batching decisions. Figure 5

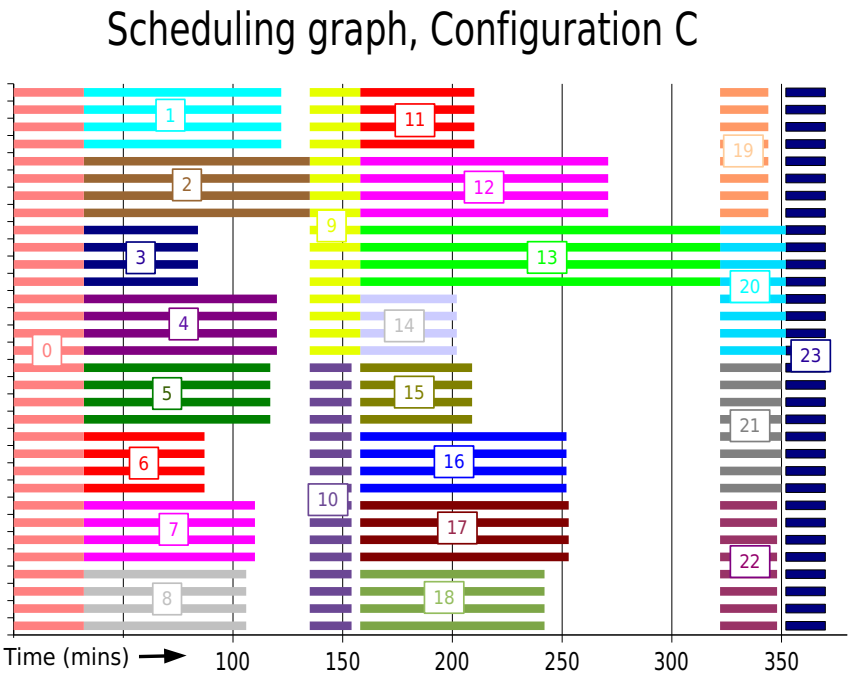

Figure 5. Helium validation study scheduling graph, Configuration C

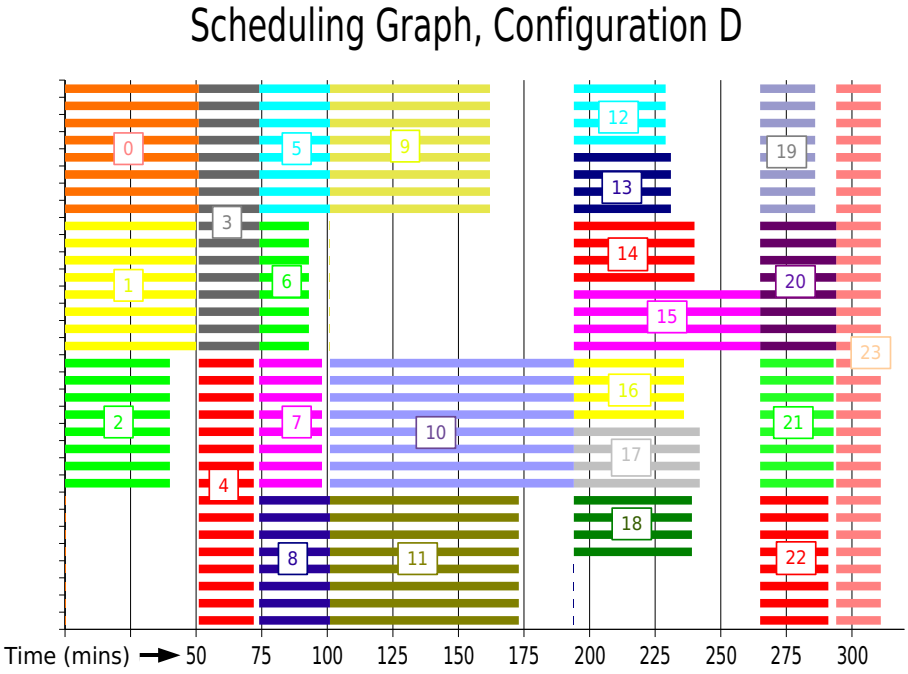

Figure 6. Helium validation study scheduling graph, Configuration D

	32 workers				64 workers			
Config-uration	Total time	Utilization Rate	Average simulation run time	Incremental improvement	Total time	Utilization Rate	Average simulation run time	Incremental improvement
O	20 hr 35 min	100%	34.3 min	N/A	10 hr 53 min	100%	36.3 min	N/A
A	13 hr 44 min	100%	34.3 min	33%	7 hr 55 min	92.74%	36.3 min	27%
B	9 hr 52 min	100%	24.7 min	28%	6 hr 0 min	93.33%	26.8 min	26%
C	6 hr 10 min	71.08%	63.4 min	38%	4 hr 3 min	60.60%	36.1 min	33%
D	5 hr 10 min	71.29%	39.7 min	16%	3 hr 37 min	70.00%	27.0 min	10%
E	4 hr 27 min	91.81%	33.5 min	14%	3 hr 41 min	93.67%	25.8 min	0%

Table 5. SimX performance on simplified helium model validation study

shows the scheduling graph of Configuration C on 32 workers. The X-axis shows the time, and the Y-axis lists the 32 worker processes. The graph shows which experiment each worker process is working on at any moment, with each experiment shown in different colors, so the workers in the same group show up as a same-colored block. Experiment IDs are annotated on the graph. While waiting for long-running experiments like Experiment 13 to finish, the system leaves idle the other worker processes, which have finished their shorter-running experiments. Separating reuse classes ensures that concurrent simulations are all start-from-scratch or all start-from-checkpoint and thus have comparable run times. It also reduces the average per-simulation run time by maximizing reuse potential. Experiments 1, 2, and 4 could have reused each others' results, as could Experiment 5, 7, and 8, but are prevented by doing so because Configuration C issued them concurrently. In contrast, Configuration D, whose scheduling graph on 32 workers is shown in Figure 6, achieves better process utilization and shorter average per-experiment run time. The overall run time shows a further 16% (32 processors) and 10% (64 processors) improvement.

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 so they may be "spread out" to those idle worker processes. Configuration E's scheduling graph (Figure 7) shows that long-running experiments can fill in the "holes" in the scheduling graph. Experiment 16, for example, 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, the improved utilization rate is offset by the overhead required to perform the reconfigurations, and Configuration E shows no improvement over Configuration D.

In summary, adding application-specific knowledge to the scheduling and resource allocation decisions in our helium model validation study resulted in a 4.6- and 3- fold improvement in overall run time.

6. Related Work

SimX builds on a body of related work, some of which are mentioned here.

The issue of resource allocation in a multi-processor environment is not new. (7) surveys conventional parallel scheduling techniques such as dynamic re-partitioning to minimize the turn-around time of individual tasks. (18) and (17) advocate "fairness" in distributing processing elements: the proportion of processing elements out of the whole cluster assigned to a task is decided by comparing its expected run time to that of the other tasks on the task queue. (19) reduces the parallel overhead of individual tasks by ensuring they are run on co-located processors. However, as is apparent, these systems differ from SimX in that SimX aims to minimize the overall run time of the entire study, while these systems focus on the turn-around time of individual tasks.

Theoretical results of the Multi-Processor Scheduling problem (MPS) are surveyed in (8). 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. However, the MPS model does not capture application-specific information such as result reuse and scaling overhead.

Grid-based parameter sweep infrastructures provide scheduling support for running the same application with a change in parameter values across distributed resources. Example systems include Nimrod (2), Nimrod/O (1), Condor (20), Globus (11), Virtual Instruments (6), and NetSolve (3). Like SimX, they aim to minimize the makespan of an entire computational study, but they primarily focus on fault tolerance, resource management, and load balancing issues on grid computing infrastructures, rather than the use of application-level knowledge in resource allocation and scheduling.

7. Conclusion

This paper described system support for and the benefits from exploiting application-specific knowledge of a multi-experiment study involving dozens of executions of a parallel simulation code. We have demonstrated that, by taking into account information such as reuse classes, scaling behavior, and study objectives in its sampling strategies, batching policies, and preemption policies, the SimX system can reduce the overall run time of the helium model validation study by more than four times.

While this paper has demonstrated a number of improvements, additional opportunities remain. For example, the current SimX system does not have knowledge of expected run times - only that runs with reuse are expected to take less time. An implementation of SimX that takes advantage of the simulation software's performance model (provided by the user or inferred from SISOL) can construct more sophisticated execution plans to further reduce the run time of the study. Also, SimX currently treats all worker processes as equal, but by being topologically-aware, SimX can co-locate nearby processors in the same worker processor groups, thereby further improving the performance of parallel simulations.

Acknowledgments

This study is funded in part by NSF grants CSR-0615255, CSR-0614770, CSR-0615194, CCR-0312956 and DMS-05-28402, and AFOSR grant F49620. The authors would also like to thank Jeremy Thornock (University of Utah) for his valuable discussions and insights on ARCHES and the helium model validation study.

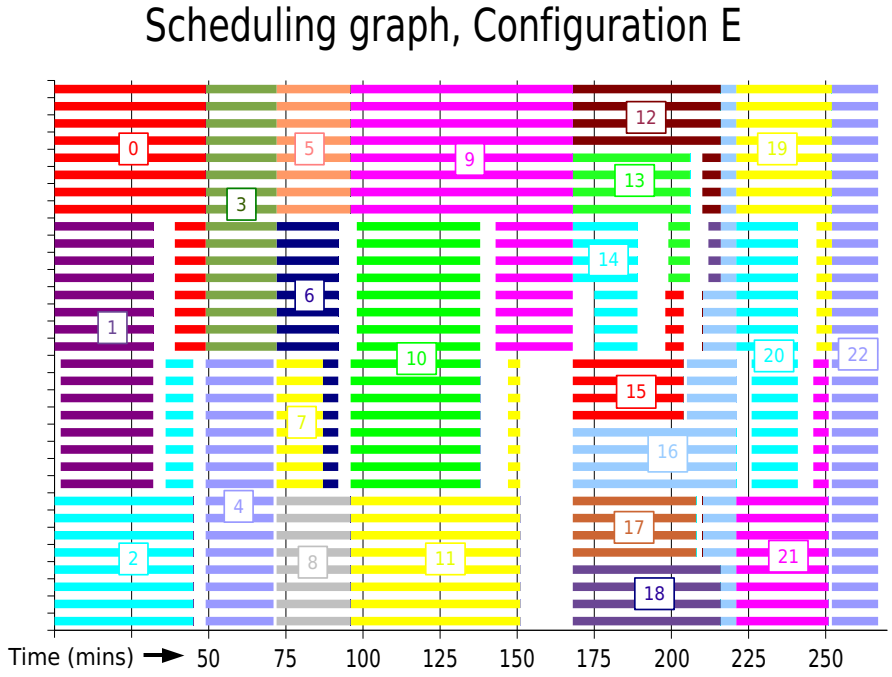

Figure 7. Helium validation study scheduling graph, Configuration E

References

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

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

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

[4] J. Davison de St. Germain, J. McCorquodale, S. Parker, and C. Johnson. Uintah: a massively parallel problem solving environment. In *Proc. HPDC*, pages 33–41, 2000.

[5] P. E. DesJardin, T. J. O'Hern, and S. 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.

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

[7] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems. IBM Research Report RC 19790 (87657), Aug 1997.

[8] Fujita and Yamashita. Approximation algorithms for multiprocessor scheduling problem. *TIEICE*, 2000.

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

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

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

[12] S. Parker and C. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proc. SC'95*, pages 1419–39, 1995.

[13] S. Parker, M. Miller, C. Hansen, and C. Johnson. An integrated problem solving environment: the SCIRun computational steering system. In *Proc. HICSS*, volume vol.7, pages 147–56, 1998.

[14] J. Schmidt and C. Johnson. DefibSim: An interactive defibrillation device design tool. In *Proc. EMBS Conf.*, 1995.

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

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

[17] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. A robust scheduling strategy for moldable scheduling of parallel jobs. *Cluster 2003*, 00:92, 2003.

[18] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *HiPC*, pages 174–183, 2002.

[19] V. Subramani and R. Kettimuthu. Selective buddy allocation for scheduling parallel jobs on clusters. In *Cluster 2002*, pages 107–116, 2002.

[20] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *CC-PE*, 2004.

[21] B. Wilson, D. Cappelleri, T. W. Simpson, and M. Frecker. Efficient pareto frontier exploration using surrogate approximations. *Optimization and Engineering*, 2(1):31–50, 2001.

[22] 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 *Proc. IPDPS*, 2008.

[23] S. Yau, E. Grinspun, V. Karamcheti, and D. Zorin. Sim-X: Parallel system software for interactive multi-experiment computational studies. In *Proc. IPDPS*, 2006.

[24] S. M. Yau, E. Grinspun, V. Karamcheti, and D. Zorin. SimX meets SCIRun: A component-based implementation of a computational study system. In *NSFNGS Workshop, IPDPS*, pages 1–6, 2007.