Practical Challenges when Implementing a Distributed Population of Cloud-Computing Simulators Controlled by a Genetic Algorithm

K. Mills, C. Dabrowski & D. Santay
National Institute of Standards & Technology
Gaithersburg, MD, USA
{kmills, cdabrowski, dsantay}@nist.gov

***Abstract*—The recent explosion of affordable multicore, multichip systems, coupled with cluster management software, encourages the development of novel distributed applications for exploring large parameter spaces. We expect many such applications will soon appear. For example, we recently applied a genetic algorithm to steer a population of cloud-computing simulators toward low-probability, costly failure scenarios. We aim to provide a design-time tool that system engineers can use to identify and mitigate such scenarios. We found that our idea was much simpler in theory than in practice, largely due to implementation challenges that arose. In this paper, we describe the design and deployment of our application, and we identify and discuss the practical challenges that we faced. We outline pragmatic solutions that we adopted to overcome those challenges. We believe many near-future applications will face similar challenges, so we hope that our experiences prove instructive.**

***Index Terms*— Computational steering, cloud computing, cluster computing, discrete event simulation, distributed systems, fault tolerance, genetic algorithms, software for parallel and distributed systems**

I. Introduction

As the microprocessor industry increases production of chips with numerous cores [1], deployed in servers containing multiple such chips [2], the scientific and engineering research communities are becoming awash in affordable, available computing power of unprecedented scale [3]. Harnessing such raw computational power requires software frameworks that envelop many multicore, multichip servers into clusters that can provision operating system images onto nodes, can monitor the availability of node resources, can allocate computationally intense tasks onto available cores and can monitor task execution. Already, a cluster computing market is forming [4], as commercial vendors offer cluster management software for high performance computing. Some such products focus on single operating systems and some support multiple operating systems. In addition, a range of open-source cluster management systems exist.

This growing availability of processing power, packaged in conveniently accessible form, holds potential for advanced computational approaches to applications that have not been attempted routinely in the past. For example, numerous software packages exist that embody advanced search techniques, such as genetic algorithms [5], evolutionary computation [6] and simulated annealing algorithms [7]. Such search algorithms are being employed increasingly in novel applications, particularly in software engineering [8], hardware design [9], and materials research [10]. The combination of advanced search algorithms with compute clusters appears likely to extend the range of novel applications that could be attempted. While most advanced search algorithms aim to optimize some selected trait, we plan to invert the search process to seek anti-optimal solutions in cloud-computing systems, as a few other researchers have attempted when investigating potential for reliability problems in concrete structures [10] and aerodynamic codes [11].

To achieve our aims, we recently adapted a genetic algorithm to steer a population of sequential cloud-computing simulators, executing in parallel on a compute cluster, in an effort to discover low-probability, costly failure scenarios. We aim to provide a design-time tool that system engineers can use to identify and mitigate such scenarios. We identified genetic algorithms (GAs) as a search technique that might be well suited for our problem. GAs can find good solutions within a large, ill-defined search space, and can be readily adapted to a wide variety of search problems [5]. Fig. 1 illustrates how we intend to apply GAs to search for failure scenarios in a cloud-computing model.

The cloud-computing model [12-14] encompasses a search space of about 10^{101}, which exceeds the estimated number of atoms in the visible universe [15]. The GA represents the parameter space as a binary encoding of 334 bits, referred to as chromosomes. We adapted the GA to convert any given set of chromosomes into parameter files that can be read by the cloud-computing simulator. Initially, the GA generates a random set of chromosomes and a population of simulators executes the derived parameter files in parallel. Each simulator returns an anti-fitness value. Anti-fitness measures the degree to which the simulator is failing on some performance objective; so higher anti-fitness represents lower performance. For example, we might equate anti-fitness to the proportion of users that could not be served.

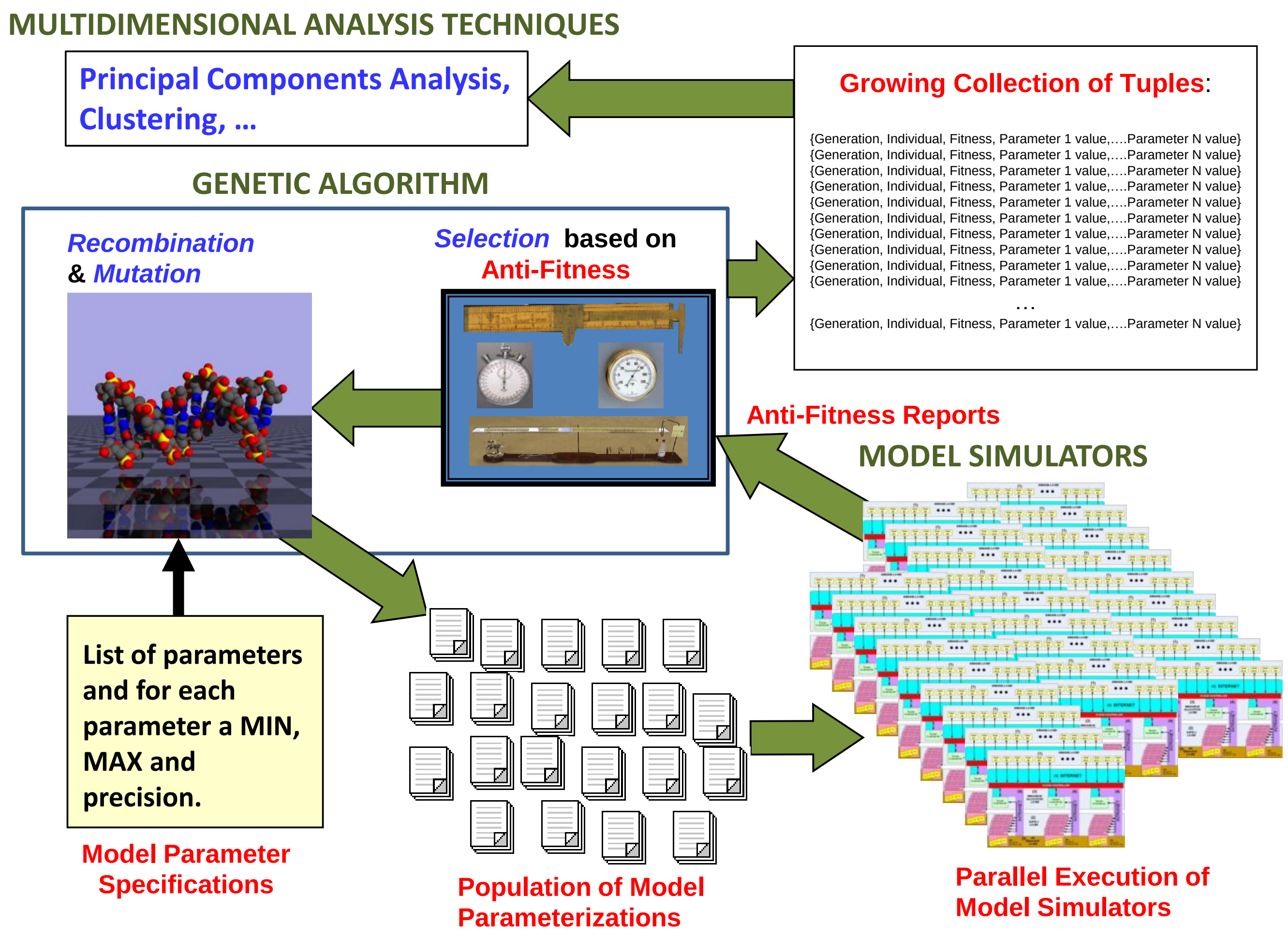
MULTIDIMENSIONAL ANALYSIS TECHNIQUES
Principal Components Analysis, Clustering, …
Growing Collection of Tuples:
{Generation, Individual, Fitness, Parameter 1 value,….Parameter N value}
…
GENETIC ALGORITHM
Recombination & Mutation
Selection based on Anti-Fitness
Anti-Fitness Reports
MODEL SIMULATORS
List of parameters and for each parameter a MIN, MAX and precision.
Model Parameter Specifications
Population of Model Parameterizations
Parallel Execution of Model Simulators

Figure 1. Schematic of a genetic algorithm steering a parallel population of cloud-computing simulators toward regions of failure and degraded performance.

Once an entire population of simulators reports their anti-fitness values, the GA employs an algorithm to construct the next generation of parameter files. As generations advance, the population of simulators explores more challenging parameter combinations, uncovering an increasing number of scenarios with degraded performance. Over time, we accumulate tuples containing anti-fitness and related model parameter values. After a sufficient number of generations, we apply selected data analysis techniques to partition the tuples into classes suggesting various causes for failed outcomes in the simulated cloud. As described in Sec. 3, we deployed this novel search application on a cluster of multicore, multichip nodes.

Computer scientists have observed [16] that multicore, multichip systems are difficult to program, because parallel, distributed systems are notoriously prone to synchronization problems and failure scenarios (as we attest to in Sec. 4). Due to this, computer scientists advocate [17] for increased spending on research and education in programming for parallel systems. In the meantime, practitioners must attempt to leverage the growing availability of computational power and advanced search algorithms to solve problems based on current knowledge and available software components. This typically requires combining existing programs written for sequential operation into cooperating assemblies that can be orchestrated to solve today's difficult and interesting problems. In Sec. 2, we review some available implementation options, and identify and justify our choice, which we selected based on conceptual simplicity.

We found that our chosen approach was much simpler in concept than in practice, largely due to implementation challenges that arose. In Sec. 3, we describe the design and deployment of our application. In Sec. 4, we identify and discuss the practical challenges that we faced, and outline the pragmatic solutions that we adopted to overcome those challenges. We believe many near-future applications will face similar challenges, so we hope that our experiences prove instructive to practitioners.

II. Related Work

As noted by Silva and Buyya [18], converting an existing sequential program to exploit parallel computing resources can be approached using three general strategies: (1) automated parallelization of a sequential program, (2) integrating parallel communication and synchronization libraries into existing source code for a sequential program or (3) major recoding of a sequential program to become parallel. The second strategy matches well to our application, where a population of sequential simulators executes in parallel, while synchronizing with a master GA process. The communication requirements for our application are rather modest: exchanging commands and status between the GA and simulators and passing parameter files from the GA to simulators. For that reason, we considered various library-based approaches.

The Message-Passing Interface (MPI) [19-20] is the most well-known standard for parallel communication libraries, with bindings to FORTRAN and C, which provides communication among cooperating sequential processes. MPI provides subroutines for sending and receiving both blocking and non-blocking messages, either point-to-point, within a group or globally. MPI can support a wide range of distributed systems, built as collaborating sequential processes. Both our cloud simulator and GA can incorporate dynamic link libraries (DLLs) using C bindings, and our available cluster system supports processes communicating via MPI, so it would be feasible to use MPI in our application, provided we modified our processes to send and receive MPI messages and data.

We also considered GridRPC [21], which mates standard remote procedure call semantics with asynchronous, coarse-grained parallel processes and provides for exception reporting. GridRPC is primarily motivated by Grid computing applications [22], which often involve parallel processing among programs distributed across remote computing facilities in various administrative domains. A more comprehensive approach can be found in the form of the PACE toolkit [23], which enables deployment and management of distributed agent-based systems in Grid computing environments. Like MPI, GridRPC and PACE could probably be used in our application, provided we made significant modifications to our GA and simulation processes.

As we mentioned in the introduction, research in parallel programming has garnered great interest, so there are many other approaches that could be considered for our application. Needham and Hansen [24] describe PVM, a parallel virtual environment based on MPI, and they also identify nine competing research initiatives, all aimed at providing parallel programming environments for grids, clusters or networks of workstations. Kee and colleagues [25] discuss ParADE, which provides a wide range of high-level functions to support parallel programming over MPI. Cilk-NOW [26] provides a runtime system for deploying parallel programming applications written in the Cilk language onto a network of workstations. Cap and Strumpen [27, 28] describe a parallel programming system that focuses on dynamic, real-time load balancing on networks of workstations. Adopting any of these approaches would require reprogramming our cloud simulator and GA for deployment in a specific distributed computing environment, as well as configuring and managing the environment ourselves.

In addition to general parallel programming strategies, we considered parallel search strategies conceived as refinements to GAs. Muhlenbein and colleagues [29] defined a genetic algorithm with multiple searches conducted in parallel, where each parallel search process runs its own GA on a defined subspace of the global search space, and the search processes occasionally exchange information about their most fit outcomes. The exchanged information is used to guide the parallel search processes. Gordon and Whitely [30] compare nine parallel genetic search algorithms, in three categories: global models, island models and massively parallel models. The global models exploit parallelism only in the selection process, while island models run several subpopulations in parallel, allowing individuals to migrate. Massively parallel models assign one processor per individual in a cellular grid, and permit mating only among nearby individuals. Unfortunately, none of the GAs compared by Gordon and Whitely were actually implemented in parallel, so adopting them would require us to expend significant effort to construct, test and verify such a parallel GA.

We decided that adopting an experimental parallel programming environment would prove too costly, requiring us to reprogram the existing GA and cloud simulator, as well as deploy and operate the environment. We also discarded, for now, the idea of using a parallel GA because we would have to rewrite the existing GA to operate as one among a cooperating set of GAs, using some algorithm identified by Gordon and Whitely. Using MPI or GridRPC would require us integrate message passing subroutines into the GA and cloud simulator, potentially introducing bugs unrelated to our problem. Subsequently, we would have to test and verify this code, which must be made extremely robust. The available libraries leave exception handling to the user, so we would need to program suitable routines. In the end, we decided that using MPI or GridRPC for communication and synchronization would not be cost effective because, as discussed below, our available cluster contained a shared file store that we could use as a common location to signal information and exchange data among the GA and simulator processes. Using a shared file store is conceptually simple, requiring development of minimal interface code based on existing file input/output functions.

III. System Design & Deployment

We began our project with three elements given: (1) an implementation of a GA [31], (2) Koala, a discrete-event simulator of an Infrastructure-as-a-Service (IaaS) cloud-computing system [12-14], and (3) a compute cluster managed by commercial software. The GA was tested previously against a wide variety of numeric optimization problems. The Koala IaaS simulator was used for a couple of years to study virtual-machine (VM) placement algorithms. Though the simulation environment, cluster management software and underlying hardware platforms consist of specific commercial products, the challenges we faced would arise under any similar setup, regardless of the source of the components. For that reason, we keep our description generic.

While Koala is written as a sequential simulator, the previous VM-placement studies were carried out with many independent Koala processes executing in parallel on various parameter combinations, orchestrated as a typical parameter-

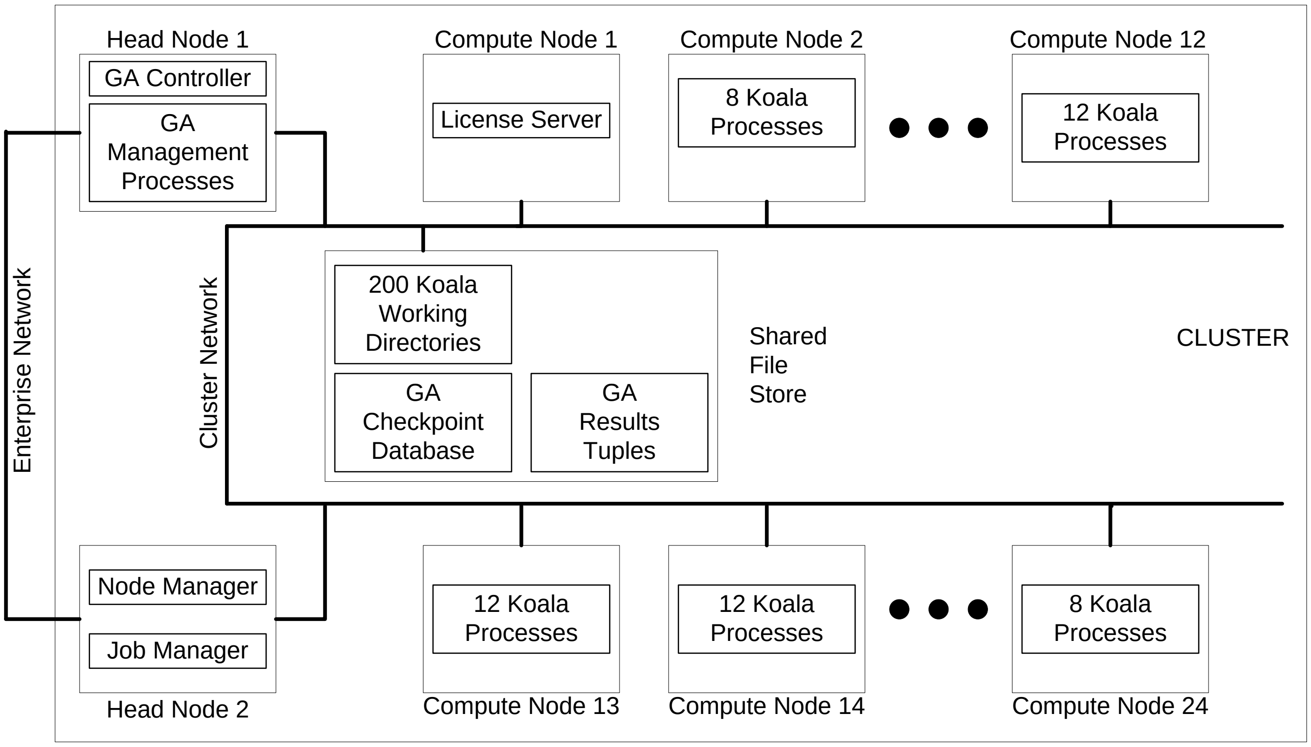

Figure 2. Schematic of GA Controlled Population of Koala Simulators Deployed on a Cluster.

sweep application. Before executing, each simulation process must obtain a license from a network license server.

The previous Koala simulations were executed on the same cluster that was made available to us. The cluster (see Fig. 2) consists of two redundant head nodes, coordinating through a shared file store, and 24 compute nodes (224 cores total), which came in different varieties, some outfitted with dual, quad-core processors and some with dual, hex-core processors. The head nodes, file store, and compute nodes share a cluster network, while the head nodes are also accessible from the enterprise network.

We adapted the GA to control Koala simulations, inserting functions: to convert specific genetic encodings understood by the GA into parameter files understood by Koala, to instruct a population of Koala simulators that a new set of parameter values was available, and to wait for the Koala processes in each generation to complete their simulations and report back fitness values. We also modified the GA to report tuples, each containing a generation identifier, individual identifier, fitness value, and for each of the 130 Koala parameters the values that led to the reported fitness. Finally, we added code so that, after each generation, the GA state and population of individuals are saved to a checkpoint database, which allows us to restart the GA from any previously completed generation. We call this modified GA the GA Controller, which we deployed on one of the head nodes.

The head nodes also executed cluster software components, as shown in Fig. 2. The node manager monitors and reports the instantaneous status and resource usage of all compute nodes. The job manager allows users to configure and submit batch jobs for execution. We use the job manager to configure and begin an entire population of Koala simulators under a single job. Also available on the head nodes are some specific GA management processes that we developed. These processes allow us to set running Koala simulators to specific states, such as started, restarted and stopped. While Fig. 2 shows various processes on specific head nodes, the head nodes are redundant, so the processes can be running on either.

We also needed to alter the Koala simulator to allow repeated executions under direction from the GA Controller. We modified Koala to query a signal file for specific instructions. The GA Controller and GA management processes can update signal files to effect control of Koala simulators. We disabled the normal results writing portion of Koala, replacing it with a function to report anti-fitness achieved at simulation completion. We began our experiments with anti-fitness defined as the proportion of arriving users who could not be served by the cloud being simulated.

The main communication channel among processes consists of a shared file store, which can be mounted by any node on the cluster network. The GA Controller is started within a specified working directory, which contains one subordinate working directory for each deployed Koala simulator. The GA Controller’s directory houses the GA checkpoint database and the results tuples, while each Koala working directory houses a simulator’s signal file and parameter input files generated by the GA Controller.

We deployed the license server on the first compute node, refraining from placing Koala simulators there. We preferred to place the license server on one of the head nodes, but each of those nodes had another license server, which allowed interactive users from the enterprise network to conduct simulations, and no node can run more than one license server.

The cluster job manager makes all decisions regarding allocation of Koala processes to specific compute nodes. The job submitter can identify which compute nodes are available for running Koala processes and can specify how many cores are required for each Koala process, along with various other job characteristics. The job manager has no knowledge of the GA Controller or license server, and the GA Controller and Koala simulators have no knowledge of the job manager. The GA Controller and GA management processes know about the Koala simulators, but are unaware of the license server.

IV. Challenges & Pragmatic Solutions

While designing and deploying our system, we faced a significant set of implementation challenges that hindered successful achievement of our objectives. Below we address those challenges in five categories: computational feasibility, robustness, coordination, failure recovery and forensics. In each category, we discuss the specific impediments we experienced, and we outline pragmatic solutions we adopted.

A. Computational Feasibility

We intended to steer a population of 200 individual Koala simulators through 500 generations of simulations, where each simulator in a given generation must finish before the GA Controller has sufficient information to construct parameter files for the next generation of simulations. This meant that the longest simulations within each generation would determine the elapsed time before the next generation could commence. Ultimately, then, the time to simulate 500 generations would be the sum of the maximum simulation time for any individual in each generation.

Suppose we chose to have each individual simulate three months in the life of a cloud. Some parameter combinations could be completed within minutes, but others could take a week or more. Further, the GA is intended to steer the population into failure scenarios, which take much more CPU time to simulate than normal operating ranges. So, if we did not further constrain simulations, then completing 500 generations could take a decade or more, which is impractical for any useful investigation of design alternatives.

We added a constraint to each Koala simulator that limited the CPU time available to any given simulation to no more than 90 minutes. This constraint ensured that 500 generations of simulations could complete within about one month. For simulations that were terminated prior to normal completion, anti-fitness results reflect the state at termination. While this ensured our system would be computationally feasible, abrupt, early terminations introduced other challenges, as we discuss in the next section on robustness.

B. Robustness

Executing 10^5 simulations (200 individuals x 500 generations) over the course of a month requires taking great care to ensure individual simulations do not produce run-time exceptions, such as NULL pointers and memory exhaustion. Previously, the Koala simulator had been used under a wide range of parameter combinations [12-14], with individual simulations lasting two weeks or longer. Based on this history, we were confident that Koala would produce few such failures. Unfortunately, we were proved wrong on this point.

To avoid biasing a search, the first population of parameter inputs is generated randomly by the GA Controller. This randomization subjected Koala to a diverse set of parameter combinations not previously experienced. Subsequently, numerous unexpected issues were uncovered including NULL pointer exceptions; excessive logging behavior; set removal exceptions; and memory exhaustion. We attempted to use normal debugging processes to uncover the source of these errors, and remove them. Our attempts were complicated by the fact that the simulations were executed as tasks in a batch job, where each task logged standard outputs to a file.

The simulation run-time environment was intended to log sufficient information to identify what error occurred and where in the source code the error arose. Unfortunately, the log files only specified what error occurred. Thus, to find the location of particular errors, we were required to rerun simulations from an interactive user interface, which could pinpoint error locations. While batch and interactive executions were supposed to produce the same error behavior, we found that in about 20% of cases, interactive executions did not fail when rerunning a parameter combination that had failed in a batch run. These reproducibility issues arose because simulator random number streams are initialized each time an interactive execution starts, which could not match the state of the random number streams leading to the failure of a batch simulation that had iterated through many generations. Fortunately, 50 to 100 crashed simulations were caused typically by only a handful of specific errors, so by interactively executing a sample of failed batch simulations, we were able to uncover the errors crashing the simulators. Subsequently, the developer of the simulation environment has ensured that batch simulation processes report both event data and code location related to exceptions.

Once we ensured that Koala was robust against randomly generated parameter combinations, new robustness issues arose as Koala finished a first generation of simulations. After a simulation completes, we needed to ensure that all residual objects are eliminated, and Koala is reinitialized, and then waits for the GA Controller to provide the next generation of parameter combinations. Without ensuring this, latent memory leaks can accumulate over the course of many generations, leading simulations to crash due to memory exhaustion. Even if a crash is avoided, accumulating unused memory objects could slow down the simulations, allowing less efficient use of CPU time. Since we were aware of these issues, we had taken the necessary steps to ensure that end-of-simulation memory leaks did not occur.

Unfortunately, when simulations were terminated abruptly due to the expiration of CPU time, a second wave of exceptions, mainly NULL pointers, arose. These exceptions resulted from situations where simulation processes had been waiting for some delay to expire during the time that the simulation was told to terminate. When this occurred, processes that did not check for a stop instruction immediately following a delay instruction became vulnerable to NULL pointer exceptions, as parent simulation processes stopped and began reclaiming memory.

To solve these problems, we inserted, where feasible, checks for a stop instruction immediately following a delay instruction. In selected cases, inserting stop checks was determined to be sufficiently difficult that we also implemented a higher level approach, which we call "grow quiet". In the grow-quiet approach, rather than immediately terminating subordinate simulation processes, a parent component would stop creating new subordinates, and then wait for sufficient time to elapse for on-going subordinate processes to complete. Subsequently, the parent component terminated itself. This approach was coupled with carefully placed wait statements in the main simulation loop, which appropriately ordered the stopping of simulation processes by type, and prevented final reclamation of any residual, unused simulation objects until all major simulation processes had terminated.

C. Coordination

Koala is built on a simulation environment that provides a single operating system process for each running simulation instance. For that reason, we implemented each Koala individual as a separate process with a unique working directory (e.g., simulator1, simulator2 and so on). Similarly, we implemented one process for the GA Controller and one process for each supporting GA management function.

The GA Controller and the management processes execute within the directory containing the subdirectories for the individual Koala simulations. The underlying simulation environment provides no inherent mechanism to coordinate among simulation processes or with other processes. As explained in Sec. 2, we opted to take a conceptually simple approach, coordinating via shared files instead of adding a communication and synchronization library to each process.

We implemented a signal file for communicating between each Koala simulation and the GA Controller or management functions. There is one signal file for each Koala process, and that file resides in the process working directory. To determine when it is time to start, a Koala process checks the signal file for a start command. Upon completing a simulation, each Koala process writes its status and anti-fitness to the signal file, and subsequently monitors the signal file for further instructions.

Initially, we imagined that normal file procedures (open, read, write, close), provided by the simulation environment and mapped to underlying operating system functions, could mediate access to the signal file. Unfortunately, we found that was not the case. Initial tests demonstrated that multiple processes could open the same file for reading and writing simultaneously, and with undesirable results. For example, if the GA Controller opened a process for reading that was simultaneously opened for writing by a Koala process, then the GA Controller experienced a read exception, while the Koala process hung. Though we could change the GA Controller to handle read exceptions, there was nothing we could do (short of manually restarting) to recover a hung Koala process.

Fortunately, the simulation environment has the ability to create, inspect and delete directory paths. We used these primitives to implement a locking scheme, where any process that wants to write or read a signal file must first obtain access to a lock. To obtain a lock, a process first checks to see if the lock directory exists. If so, the process seeking the lock waits a bit and then checks again. When the lock directory does not exist, the seeking process attempts to make the lock directory. If unsuccessful, then the process waits a bit and tries again. If the lock directory cannot be accessed or created after several tries, then a failure is logged. Releasing a lock simply requires deleting the lock directory. When processes are required to wait for a lock, we offset the waiting times so that, for example, a Koala process waits five seconds when attempting to acquire an unavailable lock, while a GA Controller or management process waits longer. We found this locking approach to effectively mediate access to shared signal files.

While our locking approach requires that processes be able to wait for a specified amount of wall-clock time to elapse, the simulation environment we used does not provide that capability as a native part of its functionality. Fortunately, the environment does permit a simulation process to launch an external operating system command, and then resume processing once the command finishes. We exploited this capability to launch an external "sleep" command with a parameter defining the time to wait. This approach became the main means of coordinating the operations of the simulation processes with the GA Controller and management processes.

When first executing (or after completing any simulation), a Koala process checks its signal file to learn what to do next. Typically, Koala will wait until instructed to begin simulating the next generation of parameters, but it could also be told to stop or restart. Absent specific instructions, Koala will use an external sleep command to suspend for a specified time (e.g., five minutes) and then check the signal file again. Here, using sleep prevents Koala from consuming CPU time while it is suspended. This cycle repeats forever until Koala is given specific instructions. We call this "lazy coordination" because it provides (bounded) signaling latency with little computational overhead.

Koala also checks for instructions during simulations, when a simulator might be told to stop or restart. To accomplish this, a Koala process periodically (e.g., every 2 ½ minutes) reads its signal file. We implement this by having Koala check every simulated hour to see if sufficient wall-clock time has passed to warrant checking for an external command. If so, the signal file is locked, opened, read, closed and unlocked. We call this "aggressive coordination" because signal checking is interleaved with normal, CPU-intensive simulation.

In all other situations, we adopt lazy coordination. For example, after seeding a population of parameters and signaling Koala processes to begin simulating, the GA Controller uses a sleep command to periodically (every five minutes here) scan signal files to determine which processes have completed simulations. After all processes complete, the GA Controller uses the resulting anti-fitness values to generate the next population of parameters, then signals the simulations to start, and uses lazy coordination to monitor for completion. Similarly, GA management processes update the signal files for designated Koala simulators and then use lazy coordination to monitor progress.

Typically, we might begin a search by starting the GA Controller, which seeds the directory for individual simulators with random parameter files. Subsequently, we use the cluster's job management software to start the required number of Koala simulators, one in each individual directory, and those simulators would grab a network license, begin executing, consult the signal file, read the designated parameter file and then begin simulating. However, lazy coordination allows components to start in any order, so we can also first start the simulators, which then wait until the GA Controller signals them to start.

D. Failure Recovery

Having implemented solutions to problems related to computational feasibility, model robustness and inter-process coordination, we expected our GA-steered simulators to work effectively, possibly experiencing intermittent failures of the GA Controller and/or Koala simulation processes. We also expected that twice a year, our compute cluster would be shut down, and any executing GA search would have to be stopped. Given these expected failures, we devised solutions to address them. What we had not anticipated was that the cluster's job manager could fail and require restarting (with unexpected results), that simulation processes could be reallocated to different compute nodes when their original compute nodes failed, that license acquisition could be temporarily blocked when a failed and restarted compute node included the license server, and that many compute nodes could fail simultaneously and reboot, after which all simulations in a given search might be relocated and restarted. Unfortunately, we experienced all of these situations and we had to devise means to cope with them. We begin our discussion with the semi-annual planned shutdowns, required for maintenance of the air conditioning system in the building housing the cluster.

GAs naturally support the ability to checkpoint and resume a search process. We implemented a function so that the GA Controller checkpoints its own chromosome map at the beginning of each execution. The chromosome map defines the allocation of Koala parameters to bit positions in the GA's genome and includes sufficient information to convert binary encoded genes into Koala parameter values. Then, at the completion of each generation of Koala simulations, the GA checkpoints the chromosomes of every individual in the generation, along with associated historical fitness information, the position of the random number stream and the values of any randomized GA control parameters. Subsequently, the GA Controller can be restarted from a specified generation. When that occurs, the GA Controller restores its state from the previously saved checkpoint and resumes from there. Including this functionality in the GA Controller allows a search to be interrupted for scheduled or unexpected cluster outages, and then to resume, losing only the generation that was in process at the time of the outage.

Unfortunately, we found that this checkpoint-restart process was insufficient to handle all outages that we experienced. For example, the cluster job manager might fail due to an unexpected software exception, and then be restarted. Upon restart, individual simulation processes could be removed from their initially allocated nodes and restarted elsewhere. Similarly, when a user tells the cluster job manager that a job requires a specified number of cores (say one per Koala simulation), then when enough compute nodes fail simultaneously, an entire job will be suspended, only to be restarted when sufficient failed nodes recover. This sort of node failure/restart scenario is opaque to the GA Controller, which has no insight into the operations of the cluster running the Koala simulations. As a result, restarted simulations would check their signal file only to find that they were already in a simulating state. Our original protocol had Koala simulations begin only when told to start by the GA controller. After determining that Koala simulations could be restarted by the cluster manager, we changed the procedures for starting Koala so that a simulation would begin when the GA Controller tells a simulator to start or when Koala finds that the simulator is supposed to be in the simulating state (i.e., had been told to start at some previous time, but had not yet completed). This change allowed Koala simulators to be robust to node reassignments and restarts.

Particularly difficult situations arose when a set of failing and restarting compute nodes included the node containing the license server. Two cases appeared. First, a Koala simulator might be restarted, after reallocation to a new compute node, and then attempt to acquire a network license when the compute node containing the license server was not operational. In such cases, the simulation environment was intended to try for a license for a period of time, and then, if unsuccessful, cause the simulation process to fail. Recovery from such a situation would require identifying and manually restarting individual Koala simulations after the license server reappeared. Unfortunately, the actual behavior of the simulation environment was different, apparently hanging the simulation process when a license could not be obtained. The developer of the simulation environment that we used was notified of this situation, and provided an updated version of the license query procedures.

A more difficult situation arose when a large set of compute nodes failed, perhaps due to a power failure, and the license server was among the failed nodes. The failed nodes rebooted independently and automatically, and once sufficient cores became available, the cluster manager restarted the suspended population of simulators. This scenario led to a race condition, where some simulation processes started before the license server and others started after. Those simulation processes that started before the license server could not obtain a license and hung, while those that started after obtained a license and started successfully. With the updated version of the simulation environment, the hung simulation processes would instead have failed, and then a manual process would be required to identify and restart the failed Koala simulations.

After discussing this issue with the developer of our simulation environment, we were provided an update that allowed us to specify a longer retry regime, so that simulation processes could be more persistent in seeking a network license. This enabled us to allow all simulations to start successfully after failure and restart of the node containing the license server.

Finally, we should mention two expected potential failures and our solution for them. We expected that the GA Controller might crash due to software errors. Recovering from such crashes involved two steps. First, we used a GA management process to restart all simulations. Second, we restarted the GA Controller with instructions to begin from the last generation that had been completed. This involves restoring the checkpoint information saved by the GA Controller after the previous generation had been completed. We also expected that latent software errors in Koala might cause occasional simulator crashes. To recover from such failures, we could diagnose and fix the fault, then stop and restart all Koala simulators with a new version of code. This would also involve restarting the GA Controller from the previously completed generation. Alternatively, we could replace the parameter file that caused the Koala simulator crash with a different parameter file, and then restart the simulator, which could allow an ongoing generation of simulations to complete.

The combination of robustness, coordination and failure-recovery techniques we adopted proved effective. For example, a trial run of 200 simulations over 500 generations experienced only 11 Koala simulator crashes. Unfortunately, those crashes proved costly in terms of increased latency, as our original schedule of one month expanded to 52 days. The increased latency arose because the GA, which executes individuals in parallel during each generation, operates sequentially between generations, i.e., the GA Controller can advance to the next generation only after all simulations from a previous generation complete. When a single simulation crashes at inopportune times, such as during weekends or when we are away on a business trip or vacation, the GA Controller waits patiently for the crashed simulator to be restarted. During these periods, the GA makes no progress. To counter such delays, we added logic to enable the GA Controller to monitor the state of Koala simulators and to automatically reassign individual parameter files from a failed simulator to an operating simulator.

We modified the Koala simulator to write a *tick* file at suitable intervals. We modified the GA Controller to check periodically for the presence of Koala tick files, deleting any that are found. This ping-ponging between writing and erasing tick files establishes a heartbeat exchange between Koala processes and the GA Controller. After missing a specified number of heartbeats, the GA Controller declares a related Koala process dead. When heartbeats resume, the related Koala process is resurrected. These changes allow the GA Controller to monitor Koala processes, which can be (1) dead, (2) alive and available for assignment, or (3) alive and in use, simulating some assigned individual.

We also modified the GA Controller lazy-coordination procedures to consult the status of a Koala process prior to checking for the fitness of an assigned individual. If the Koala process is dead, then the individual is added to a set of pending individuals; otherwise, a check is made, as normal, to see if fitness is available for the individual. Then, prior to suspending for the next lazy-coordination interval, the GA Controller cycles through the set of pending individuals, assigning each to an available Koala process. If there are insufficient Koala processes for all pending individuals, then the residual individuals wait until some future time when additional Koala processes become available. If an individual has been reassigned too many times, then a default fitness value is set, so that the GA Controller can continue with the next generation. We take this last step to prevent parameter combinations causing a Koala crash from indefinitely delaying the GA Controller from proceeding to the next generation.

Monitoring the state of simulation processes allows the GA Controller to automatically reassign individuals from dead to living Koala simulators, which enables generations to advance when simulations fail and no one is around to restart them. By starting more simulation processes than required to handle the population of individuals, the excess processes are able to pick up work from failed processes. Alternatively, a population of individuals can be simulated by a smaller number of simulation processes at the cost of increased delay from sequentially executing some number of individuals in a population.

E. Forensics

We anticipated that we would face some difficulties assessing the operational state of the system. We knew that we had available tools provided by the cluster manager, such as a node monitor that could report instantaneous resource usage on the compute nodes and a job manager that could report the operating state (e.g., dispatching, running, failed, cancelled) of individual simulations. We also understood that this would be insufficient to reflect the operating state of the entire population of simulators.

To augment the available cluster monitoring tools, we constructed a status reporting process that scanned the signal files and reported the state of specified Koala simulators. We also inserted code to report significant state changes to the console for the GA Controller and to the console files for batch Koala simulations. All of these forensic tools proved to be quite helpful, but they were insufficient. Typically, the myriad failure, restart and outage scenarios that arose left us guessing about the precise state of the distributed system, and about the trajectory leading there. As examples, we will discuss some specific situations, and then describe how we enhanced the recording of forensic information to reduce the amount of guesswork required to determine system state and trajectory.

While running a search, the GA Controller would periodically report the number of pending simulations that were still running in a given generation. We knew that any given simulation would be restricted to only about 90 minutes of CPU time, thus when we found that numerous simulations were still not finished after a prolonged period (e.g., overnight) we needed to determine the state of the simulators. To explore the state, we would typically use the status reporting process we developed to scan the signal files of all Koala simulation processes. A usual report would find that some number of simulators had finished (consistent with the report of the GA Controller) and that some number were either simulating or had been told to start, but had not responded. Consulting the cluster job manager we found that all simulations in the job were still in a running state. Consulting the node manager revealed that no significant CPU time was being used on any of the compute nodes on which the Koala simulations were allocated.

To further investigate, we initiated remote logons to individual compute nodes and then used the local task manager to inspect the status of individual processes. In some cases, we found that Koala simulators were in a suspended state (through an external sleep command). This would surely include simulators that had finished simulating, as identified by our status monitoring process, but it also included simulators that were in the simulating state. Only after consulting the job activity log on the cluster job manager did we discover that these simulators had been restarted after being moved between nodes. Further investigation with remote logons to the originally assigned compute nodes revealed that those nodes had rebooted after some failure.

A second set of Koala simulators were not executing due to a completely different cause. This set was also only revealed after remote logons to specific compute nodes. Inspecting the task manager, we found that these Koala instances were suspended, but not through a sleep command. The only other cause we could think of for such suspensions would be failure to obtain a network license. After remotely logging on to the node containing the license server we learned that the node had rebooted, from which we inferred that a set of reallocated and restarted Koala simulations had attempted and failed to obtain a network license during the period that the compute node containing the license server was rebooting. This finding was what led as to consult the developer of our simulation environment and initiated the process of improving the robustness of procedures for acquiring a network license.

As you can see, investigating the state of the entire distributed system required a significant amount of digging, using the available tools, which were useful but still left us making inferences and guesses about the system state and trajectory. This led us to rethink our approach to logging forensic information.

We decided to augment the Koala simulator to include both an event log and a lock log. Subsequently, we found these logs to be insufficient, so we added a simulation progress log. The event log records each significant change in state (along with a time stamp) for a Koala simulator through its entire operating history, including all restarts. The lock log records each failure to obtain or release a signal-file lock. The progress log reports simulator status every simulated hour, which provides clues when an event log shows that a simulation is underway, yet the process appears idle. We augmented the signal file information to include the time and component that last changed the file. We also created an additional management process to clear event and lock logs under user direction. With all significant simulator history recorded in logs, the operating state and trajectory of the entire distributed system became quite transparent, and we could easily and effectively conduct post mortems for most situations, though understanding the precise cause of idle simulations remains a difficult task.

V. Conclusions

The scientific research and engineering community is awash in computing power available through multicore, multiprocessor servers deployed in clusters and clouds. Exploiting such vast resources presents difficult challenges because tools and techniques for programming parallel, distributed systems are lagging growth in raw hardware power. Nevertheless, parallelizable search algorithms exist, and might be combined with increasing numbers of processors, available in clusters and clouds, to implement novel approaches to difficult and significant problems. We demonstrated the feasibility of one such approach, applying a genetic algorithm to steer a population of cloud-computing simulators into directions that reveal low-probability, costly behaviors that might otherwise lurk unforeseen until they appear in a deployed system.

While our approach proved feasible, we encountered a number of practical implementation challenges likely to arise in many parallel, distributed systems built from existing sequential processes deployed on clusters. We identified and discussed the challenges we encountered and the pragmatic solutions that we adopted to overcome those challenges. We believe many near-future applications will face similar challenges, so we hope that our experiences prove instructive to practitioners who attempt to deploy parallel search algorithms on today's compute clusters.

Acknowledgment

We appreciate funding for this research, provided by the Cloud Computing and Complex Systems programs, operating under the auspices of the National Institute of Standards and Technology. We acknowledge the timely and helpful assistance received from Jim Henriksen, the developer of the simulation environment underlying Koala. Without his assistance, we could not have resolved the issues encountered with network license server procedures in the face of failures and restarts. We also thank Antonio Izquierdo and Frederic de Vaulx, along with the anonymous external reviewers, who helped us improve our manuscript. We accept responsibility for any errors remaining in the paper.

References

[1] D. Geer, Chip makers turn to multicore processors, *Computer*, *38*(5), 2005, 11-13.

[2] C. Keltcher, K. McGrath, A. Ahmed & P. Conway, The AMD Opteron processor for multiprocessor servers, *IEEE Micro*, *23*(2), 2003, 66-76.

[3] A. Weiss, Computing in the clouds, *netWorker Magazine*, *11*(4), 2007, 16-25.

[4] M. Baker, G. Fox & H. Yau, A review of commercial and research cluster management software, *Northeast Parallel Architecture Center*, Paper 19, 2006, 65 pages.

[5] M. Mitchell, *An introduction to genetic algorithms* (Cambridge, MA: MIT Press, 1998).

[6] K. De Jong, *Evolutionary computation: a unified approach* (Cambridge, MA: MIT Press, 2006).

[7] D. Bertsimas & J. Tsitsiklis, Simulated annealing, *Statistical Science*, 8(1), 1993, 10-15.

[8] M. Harman, Software engineering meets evolutionary computation, *Computer*, *44*(10), 2011, 31-39.

[9] B. Hounsell & T. Arslan, A novel genetic algorithm for the automated design of performance driven digital circuits, *Proceedings of the Congress on Evolutionary Computation*, *1*, 2000, 601-608.

[10] L. Catallo, Genetic anti-optimization for reliability structural assessment of precast concrete structures, *Computers & Structures*, *82*, 2004, 1053-1065.

[11] J. Cruz, An application of anti-optimization in the process of validating aerodynamic codes, Virginia Tech PhD dissertation, 2003, 223 pages.

[12] K. Mills, J. Filliben & C. Dabrowski, Comparing VM-placement algorithms for on-demand clouds, *Proc. of IEEE CloudCom* Nov. 29-Dec. 1, Athens, Greece, 2011, 91-98.

[13] K. Mills, J. Filliben & C. Dabrowski, An efficient sensitivity analysis method for large cloud simulations, *Proc. 4th International IEEE Cloud Computing Conf.*, Washington, DC, July 5-9, 2011.

[14] C. Dabrowski & K. Mills, VM leakage and orphan control in open-source clouds, *Proc. of IEEE CloudCom*, Nov. 29-Dec. 1, Athens, Greece, 2011, 554-559.

[15] C. Impley, *How It Began: A Time-Traveler's Guide to the Universe*, (W.W. Norton and Company), 2012.

[16] D. Patterson, The trouble with multicore, *IEEE Spectrum*, July 2010.

[17] S. Fuller & L. Millet (eds.), *The future of computing performance: game over or next level*? (Washington, DC: The National Academies Press, 2011).

[18] L. Silva & R. Buyya, Parallel Programming Models and Paradigms, Chapter 1 in *High Performance Cluster Computing: Programming and Applications*, Vol. 2, Prentice Hall, 1999, 4-27.

[19] J. Dongarra & D. Walker, MPI: A Standard Message Passing Interface, *Supercomputer* 12(1), 1996, 56-68.

[20] J. Dongarra, S. W. Otto, M. Snir, & D. Walker, A Message Passing Standard for MPP and Workstations, *Communications of the ACM* 39(7), 1996, 84-90.

[21] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee & H. Casanova, Overview of GridRPC: A Remote Procedure Call API for Grid Computing, Proceedings of the 3rd International Workshop on Grid Computing, 2002, 274-278.

[22] M. Bote-Lorenzo, Y. Dimitriadis & E. Gomez-Sanchez, Grid Characteristics and Uses: a Grid Definition, Proceedings of the 1st European Across Grids Conference, LNCS 2970, 2004, 291-298.

[23] J. Cao, D. Spooner, J. Turner, S. Jarvis, D. Kerbyson, S. Saini & G. Nudd, Agent-based Resource Management for Grid Computing, *Proceedings of the 2nd International Symposium of Cluster Computing and the Grid*, 2002, 350-351.

[24] S. Needham & T. Hansen, Cluster Programming Environments, 2002.

[25] Y. Kee, J. Kim & S Ha, ParADE: An OpenMP Programming Environment for SMP Cluster Systems, *Supercomputing* '03, 2003.

[26] R. Blumofe & P. Lisiecki, Adaptive and Reliable Parallel Computing on Networks of Workstations, *Proceedings of USENIX*, 1997.

[27] C. Cap & V. Strumpen, Efficient parallel computing in distributed workstation environments, *Parallel Computing*, 19(11), 1993, 1221-1234.

[28] V. Strumpen, Coupling Hundreds of Workstations for Parallel Molecular Sequence Analysis, Software-Practice and Experience, 25(3), 1995, 291-304.

[29] H. Mühlenbein, M. Schomisch, & J. Born, The parallel genetic algorithm as function optimizer, *Parallel Computing*, 17(6–7), 1991, 619-632.

[30] V. Gordon & D. Whitely, Serial and Parallel Genetic Algorithms as Function Optimizers, *Proceedings of the 5th International Conference on Genetic Algorithms*, 1993, 177-183.

[31] A. Haines, Determining important control parameters of a genetic algorithm, *Summer University Research Fellowship Presentation,* Gaithersburg, MD, Aug. 7, 2012.