Fast and Scalable Simulation of Volunteer Computing Systems Using SimGrid

Bruno Donassolo
blmdonassolo@inf.ufrgs.br
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500, Porto Alegre, Brazil

Henri Casanova
henric@hawaii.edu
Information and Computer Sciences Dept.
University of Hawai'i at Manoa
1680 East-West Rd, POST 317, Honolulu, HI 96822, USA

Arnaud Legrand Pedro Velho
arnaud.legrand@imag.fr pedro.velho@imag.fr
CNRS – University of Grenoble
INRIA MESCAL Project
LIG Laboratory - ENSIMAG - 51 Avenue Jean Kuntzmann - 38330 MontBonnot Saint-Martin - France

ABSTRACT

Advances in internetworking technology and the decreasing cost-performance ratio of commodity computing components have enabled Volunteer Computing (VC). VC platforms aggregate tens or hundreds of thousands of hosts. These hosts are typically volatile, which raises difficult research questions. Most research in this area relies on simulation. The main issue when developing VC simulators is scalability: How to perform simulations of large-scale VC platforms with reasonable amounts of memory and reasonably fast? To achieve scalability, state-of-the-art VC simulators employ simplistic simulation models and/or target on narrow platform and application scenarios. In this paper we enable VC simulations using the general-purpose SimGrid simulation framework, which provides significantly more realistic and flexible simulation capabilities than the aforementioned simulators. Our key contribution is a set of improvements to SimGrid so that it brings these benefits to VC simulations while achieving good scalability.

1. INTRODUCTION

Advances in internetworking technology in the last decade have made it possible to establish distributed computing platforms at a global scale. Capitalizing on the decreasing cost-performance ratio of commodity computing components, Volunteer Computing (VC) platforms aggregate tens or hundreds of thousands of hosts. VC platforms are attractive as they provide enormous amounts of computational power at low cost. However, the hosts are typically individually owned, and thus heterogeneous as well as volatile because subject to frequent downtimes and reclaims. To cope with host volatility and heterogeneity, production VC uses a centralized server and a simple master-worker computing model for large numbers of independent, compute-intensive tasks. The deployment of new classes of VC applications and the development of alternate designs for VC systems mandate that several difficult research questions be answered. In general, research in this area is experimental in nature. Since experimenting with real-world VC platforms is a challenging proposition, due in part to the need for repeatable experiments, most researchers resort to *simulation*.

Several simulators have been proposed and developed specifically for VC systems and applications [3, 32, 15, 8]. A paramount concern is *scalability*, i.e., the ability to simulate large-scale platforms and applications with low time and space complexity. To achieve good scalability these simulators make allowances that afford scalability at the expense of realism and/or flexibility. For instance, they may opt for not simulating network resources, for simulating host availability using standard probability distribution functions instead of real-world availability traces, or for simulating only one particular scenario without providing a generic and programmable simulation framework. In this paper we take a different approach and enable VC simulations using the general-purpose SimGrid simulation framework, which provides significantly more realistic and flexible simulation capabilities than aforementioned simulators. Our key result is that it is possible to improve the SimGrid simulation core to achieve high scalability. We make the following contributions: (i) we propose, justify, and implement improvements to SimGrid in a view to supporting VC simulations; (ii) we evaluate the impact of our improvements on simulation scalability via simple benchmarks; and (iii) we compare our improved SimGrid version to competing approaches.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents the SimGrid simulation core and Section 4 describes improvements to it. Section 5 presents experimental results. Section 6 concludes the paper and highlights directions for future work.

2. BACKGROUND AND RELATED WORK

2.1 Volunteer Computing

Volunteer Computing is a form of distributed computing that allows *volunteers* to donate their computers' idle CPU times to a given application, or *project*. VC became famous thanks to the SETI@home project [2]. Since then, SETI@home has been refactored and open-sourced, culminating in the release of BOINC (Berkeley Open Infrastructure for Network Computing) [1]. BOINC is the most pop-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LSAP '10, Chicago, USA
Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

ular VC infrastructure today with over 580,000 hosts that deliver over 2,300 TeraFLOP per day. In such VC infrastructures, each project (e.g., SETI@home, Climateprediction.net, Einstein@home, World Community Grid) is hosted on a server that provides hosts with work units. Once a host has fetched one or more work units, it disconnects from the server and computes work unit results. Several mechanisms and policies determine when the host may perform these computations, accounting for volunteer-defined rules (e.g., caps on CPU usage), for the volunteer's activity (e.g., no computation during keyboard activity), and for inopportune shutdowns.

Salient characteristics of VC systems are thus their scale, their heterogeneity, and their volatility and unpredictability [22]. One way in which to cope with these characteristics is to run applications that consist of large numbers of independent work units (i.e., orders of magnitude larger than the number of available hosts). Supporting new application classes mandates that open research questions be addressed so that the effects of heterogeneity and volatility can be mitigated intelligently. For instance, scheduling techniques have been proposed for VC applications that consist of small numbers of work units [19]. Other works have proposed departing from the traditional centralized server approach and bring in ideas from peer-to-peer computing [11, 7].

In the vast majority of the cases, VC research must be done empirically. Unfortunately, studying VC systems *in vivo* is a challenging proposition. Production VC systems are rarely available for experimentation so as not to interfere with their workload. Deploying a VC system for the purpose of experimentation is possible at best at moderate, non-representative scale. Besides, to be convincing, experiments should be conducted for more than one platform scenario. Finally, and perhaps most important, non-deterministic host volatility makes experiments inherently non-repeatable even when run back-to-back. Given these challenges, most research in this area is done through simulation.

2.2 Volunteer Computing Simulation

2.2.1 Design Issues

Simulating the hosts – The most generic and flexible approach for simulating the execution of a VC client is to model each host as a lightweight process that executes arbitrary source code and that uses a simulation API to simulate data transfers, downtime, computation, etc. The scalability challenge is space complexity: for a VC simulation of one million hosts running on a single machine with 4GB of memory, each host can only be described with about 4KB. A popular alternative is to represent each host by a finite-state automaton. While this approach greatly limits the flexibility of the simulation and takes it further from a real-world system, it may allow better scalability. More elaborate techniques, some of them used for simulating peer-to-peer systems (e.g., fluid simulation [28, 23]), may be viable but, to the best of our knowledge, have yet to be used for VC simulation.

Simulating the network – One option is to assume that data transfers take a fixed (possibly zero) amount of time, or an amount of time proportional to the data size. Such a naïve model may be justified for simulating compute-intensive applications when server bandwidth is plentiful. For other scenarios, realistic network simulation is needed. Although simulation of large-scale networks can be extremely costly (e.g., using packet-level simulation), simulation based on mathematical models of network flows [24] may be feasible.

Simulating volatility – The most realistic approach for simulating volatility is to "replay" availability traces collected on VC systems [21]. This approach raises a scalability challenge as traces may be numerous and large, which makes storing and processing them expensive. The alternative, which affords much higher scalability, is to rely on statistical characterizations of host availability [22, 20]. Although some simple candidate models have been identified [27], it is not clear how closely they represent typical real-world platforms.

Choosing a time scale – The time granularity of the simulation is a difficult issue because researchers may be interested in long-term notions (e.g., average host idle time) as well as in short-term notions (e.g., the number of time a particular work unit was preempted). The most precise, but also less scalable, approach is discrete-event simulation. The simulation proceeds as a sequence of events, and each event triggers one or more subsequent events. A less precise alternative, which affords better scalability, is time-driven simulation. In this approach one estimates the number of events for several event types that occur in a relatively long simulated time interval (e.g., one hour). Such time intervals are processed in sequence. With time-driven simulation one can trade off increased scalability for reduced accuracy by increasing the interval size.

Different choices for the above design issues can lead to widely different simulators, going from simulators that use naïve models and simulate very restricted scenarios, to general-purpose simulators that use sophisticated models and are flexible enough to simulate a wide range of scenarios. The ideal simulator would be in the latter category while still affording high scalability. Developing such a simulator is the ultimate goal of this work.

2.2.2 BOINC Simulators

The BOINC distribution comes with a simulator that simulates a single BOINC client interacting with one or more projects. This simulator uses time-driven simulation, uses a simple host model, and does not simulate the network. It has been used to evaluate and improve the BOINC client scheduler [3, 18]. Because it uses original BOINC source code, faithful simulations should be guaranteed. A drawback is that it is difficult to extend as a new scheduling algorithm would require a full-fledge implementation in the BOINC source code.

In [14], Estrada *et al.* investigate threshold-based scheduling policies in BOINC with the use of a custom simulator called SimBA [32]. SimBA is a discrete-event simulator that models hosts as finite-state automata parameterized with several characteristics (e.g., compute rate, error rate, timeout rate), and that models the failure or success of each task using a uniform probability distribution. Volatility is modeled using a Gaussian probability distribution. This simulator, because it opts for the simple and inexpensive options for the design issues in the previous section, provides high scalability at the potential expense of simulation realism. Furthermore, it is limited to the simulation of a single project.

A few years later, Estrada *et al.* developed a new simulator, EmBOINC [15]. They modified the BOINC server source code so that it could run in emulation mode. In this way, fictitious clients can send fictitious requests to a

server executing real BOINC source code. These modifications were integrated in the original BOINC source code. Here again, this approach is interesting for studying a single BOINC project server but cannot accommodate a multi-project BOINC system.

In [8], Kondo *et al.* use a simulator called SimBOINC [17]. This simulator, based on modified BOINC source code, emulates BOINC clients using the general-purpose SimGrid [31] simulation framework as a back-end. One weakness of this simulator is that the modified BOINC source code must be maintained alongside the main BOINC distribution. At the moment, the simulator is compliant with BOINC 5.5.11, which is two years behind the current 6.6.40 distribution, and is no longer supported. Unlike the previously described simulators and emulators, SimBOINC allows for the simulation of a complete BOINC system, i.e., with many projects and many clients with complex and diverse behaviors.

2.2.3 General-Purpose Simulators

Many custom simulators are built by researchers in the area of distributed computing, but most are short-lived [26]. Two that have withstood the test of time are SimGrid [10, 31] and GridSim [6]. Both simulators are general-purpose and provide ways for a user to program arbitrary simulations using a convenient API. Scalability limitations have been reported for GridSim [13, 4], which casts doubts on its potential use for VC simulation. For instance, its use of Java threads prevents it from simulating more than 10,992 active hosts. Instead, SimGrid has been used in recent VC studies [16] and has been shown capable of simulating up to 200,000 hosts on a machine with 4GB of memory [31]. SimGrid has also been used by the OurGrid project [30, 25] and in a recent Bag-of-Tasks application scheduling study [12]. In spite of these encouraging SimGrid results, in a recent study with a few hundred hosts [16] Heien *et al.* identified simple scenarios in which SimGrid leads to unacceptably high execution times. These observations motivate the improvements to SimGrid presented in this work.

3. SIMGRID SIMULATIONS

3.1 SimGrid APIs

The SimGrid project was initiated for simulating distributed applications in grid computing environments [9]. In such environments, the underlying platform is complex (resource heterogeneity, hierarchical network topology, dynamic resource availability, etc.), and scheduling algorithms must be designed based on a much simpler and tractable model of the platform. The original goal of SimGrid was to provide a tool for evaluating such scheduling algorithms using realistic simulation of complex grid platforms. SimGrid now provides four APIs (MSG, GRAS, SimDAG, and SMPI) through which users can develop simulations that run on top of SimGrid's simulation core, each of these API being suited to a particular use (e.g., study of parallel applications structured as directed acyclic graphs, of already existing MPI applications). For the work in this article we use the MSG API, which allows the easy prototyping and study of generic distributed applications, and is the one most commonly used by SimGrid users.

3.2 The SIMGRID Simulation Core

The SimGrid simulation core implements and provides interfaces to a number of simulation models that vary in sophistication and can be used to simulate different types of resources (network resources, computational resources). It consists of two main layers: the SURF layer implements the simulation models, and the SIMIX layer provides a low-level API to these models upon which user-level APIs can be developed. Both layers are described hereafter. Some simulation models share the same structure, which is implemented as an additional layer, called LMM, which is called by SURF and which is briefly described hereafter as well.

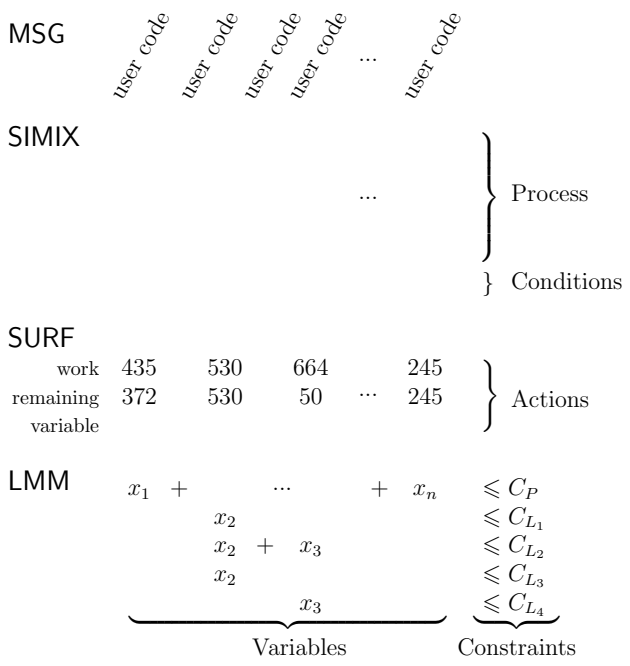

Figure 1: Illustrating the SimGrid layers and the main data structures.

3.2.1 SIMIX

SIMIX provides a Pthread-like API to manage concurrent simulated processes. More precisely, it provides the following abstractions: *processes*, *locks*, *condition variables*, and *actions*. Processes correspond to threads of control of the simulated application, locks and condition variables are used for synchronizing these threads of control, and actions are used to represent resource consumption generated by these threads of control. We briefly illustrate these abstractions via a simple example.

Consider a simulation of a computation on a host. This computation is embedded within a SIMIX process and launched via a user-level API call, e.g., `MSG_task_execute` if using the MSG API. This call creates a SIMIX action that corresponds to the amount of computation to be performed (specified by the user-level API call). This action is associated with a SIMIX condition variable on which the process blocks. Once the action is completed, as dictated by the simulation models after some elapsed simulated time, the condition variable is signaled. The user-level API call returns control to the user, thereby providing the simulation of the delay incurred for performing the computation.

Most simulations consist of many SIMIX processes. All processes run in mutual exclusion and SIMIX is responsible for controlling their execution. Essentially, all processes run in round-robin fashion until all of them block on condition

variables to wait for action completions. At a given simulated time t, SIMIX has thus a list of blocked processes. SIMIX then calls the lower layer of the simulation core, SURF, through the `surf_solve` function. SURF, discussed in the next section, is responsible for handling the simulation clock and the usage of simulated physical resources. `surf_solve` advances the simulation clock to time $t + \delta$ at which at least one of the actions waited upon has completed (or failed). A list of the completed (or failed) actions at time $t + \delta$ is returned to SIMIX. SIMIX then wakes up the corresponding processes. The same procedure is repeated, advancing the simulated time from task completion to task completion until all processes terminate.

The execution of the simulated application is handled by SIMIX, and is fully separated from the simulation of the underlying platform, which is handled by SURF. The two layers communicate solely via the condition variable and action abstractions, as shown in the top part of Figure 1.

3.2.2 SURF

SURF provides several models for determining simulated action execution times and resource consumptions. These models can be selected and configured at runtime, and each model is responsible for actions and resources of a given type (e.g., CPU, network, timer). For instance, in terms of network resource models, the current implementation provides a default model of TCP networks [33], a model that offloads all simulation to the GTNetS packet-level network simulator [29], a simple model based on uniform random distributions, and more advanced models that use Lagrangian optimization and gradient descent [24]. By picking an appropriate model the user can trade off speed/scalability for accuracy, with no change to user source code.

All simulation models are accessed via the `surf_solve` function, which proceeds in the following steps:

1) Query each active simulation model for the next action completion/failure date among all the actions managed by that model. This is done through the `share_resources` function, which each model must implement. For many models, this function relies on an extra layer, LMM, via which resource usage is represented as a a set of linear constraints, as seen in the next section. This general approach enables to represent very complex situations. LMM uses a sparse representation of this linear system and uses a simple max-min allocation algorithm by default but also implements more sophisticated models based on the work in [24]. As seen in Figure 1, the models in SURF keep track of the amount of work remaining for each action, and can therefore determine when each action will complete based on current simulated resource usage.

2) Compute t_{min}, the minimum of these completion dates. Examine user-provided traces used to describe dynamically changing resource conditions to see whether a resource state change occurs before t_{min} (e.g., the available bandwidth of a network link increases, a host is shutdown). If such a state change occurs, then call the `update_resource_state` function of the model in charge of the resource. Each model must implement `update_resource_state`. Update t_{min} to be the earliest time of next resource change.

3) Ask each active model to advance the simulation time to t_{min} and to update every action state accordingly. This is done through the `update_action_state` function, which each model must implement.

4) Return the set of actions that have finished or failed.

3.2.3 The LMM Layer

Many of the simulation models in SimGrid represent actions and resources as variables and constraints in a linear system. For example, given a set $\mathcal{L}$ of network links defined by their bandwidths and a set $\mathcal{F}$ of network flows defined by the set of links they use, we can represent each flow f by a variable x_f (representing the bandwidth allocated to it). For each link l we have the following constraint:

$$\sum_{f \ni l} x_f \leqslant C_l, \text{ where } C_l \text{ is the bandwidth of link } l,$$

which states that the bandwidth capacity of the link cannot be exceeded. For instance, in Figure 1, variable x_2 and x_3 correspond to two flows using respectively $\{L_1, L_2, L_3\}$ and $\{L_2, L_4\}$. Many allocations x can satisfy the set of link capacity constraints and different network protocols lead to different allocations [24]. SimGrid uses a simple max-min allocation by default [5] but also implements more sophisticated models based on the work in [24].

For such models, the LMM layer uses a sparse representation of the above constraints. The problem is solved by the `lmm_solve` function, which is efficient because its complexity is *linear* in the *system size*, where the system size depends on the number of actions, the number of active resources, and the complexity of the resource usage. For example, the system corresponding to a set of N CPUs running each an action would be of size $\Theta(N)$. The system corresponding to F flows going each through L links would be of size $\Theta(F.L)$.

If the system needs to be modified it is invalidated and the allocation must be recomputed with possibly new variables and constraints. For example, in Figure 1, removing variable x_2 would force recomputation of variable x_3, removing variable x_1 would force recomputation of variable x_n, etc. More generally, such invalidations occur based on the action life-cycle (e.g., action creation, action termination, action suspension/resumption), i.e., between two successive calls to `surf_solve`, or based on resource state changes, i.e., when function `update_resource_state` is called. Although we have used network resources as an example, the same approach is applicable to other, arguably less challenging, resource types.

3.2.4 The Default CPU Model

Like many other models, the default CPU model relies on a LMM system and associates each CPU with a constraint whose bound is the rate of the simulated CPU (in MFlop/s). We detail the components of this model along with their complexities:

Action creation − An action is defined by its *remaining amount* of work (in MFlop), which is initialized upon creation, and by a corresponding variable in the LMM system. The resource consumption rate allocated to the action varies over time depending on the value of this variable.

`share_resources`− To compute the next action completion date, this function first computes a new solution of the LMM system, if needed. Then, it goes through the list of all active actions to compute when each would complete, based on its current resource share and remaining amount of work, assuming that the system remains unchanged. The complexity of this function is thus $\Theta(|\text{actions}|)$ plus possibly the complexity of `lmm_solve`, which is also $\Theta(|\text{actions}|)$.

`update_resource_state`– When the state of a resource is changed, one needs only to update the bound of the corresponding constraint, which is done with complexity $\Theta(1)$.
`update_action_state`– This function advances simulation time. To do so it goes through the list of all actions to update their remaining amounts of work, which leads to a $\Theta(|\text{actions}|)$ complexity.

3.3 Analyzing a Simple VC Example

In this section, we consider a platform with N hosts with compute speeds sampled from the speeds found in SETI@home traces [21]. On each host a worker process sequentially computes P tasks whose work amounts, in MFlop, are uniformly sampled between 0 and 8.10^{12} (i.e., up to roughly one day on a standard machine). This minimalistic example was submitted to the SimGrid developers by Eric Heien after using SimGrid in [16].

Whenever an action ends, the whole resource allocation is recomputed (through the call to `share_resources`) and all pending actions are updated (through the call to `update_action_state`). However, there is very little modification to the system. Indeed when an action ends on a host, it does not affect the other hosts nor the other action completion dates. Still, the remaining amount of all actions is updated so as to always have a consistent system, which takes a time $\Theta(|\text{pending actions}|) = \Theta(N)$, since at a given time there is one pending action per host. There will be as many calls to `surf_solve` as the total number of actions in the lifetime of the simulation. Since there are N hosts and P tasks per host, there is a total of NP actions. Thus the overall complexity of SimGrid to run such a simple example is $\Theta(N^2P)$. One would hope for a complexity that is much lower than quadratic with respect to N, since in VC scenarios N is routinely as high as tens or hundreds of thousands.

In our example, the computational power of the hosts does not change over time. Assume that each host is annotated with a trace with T state change events, where each state change could be a change in the host's available computation power. In this case, there would be a total of NT state change events and the remaining amount of work of all actions would be updated every time. The time needed to retrieve an event would be $\Theta(\log N)$ (as we use a heap to store the next event for each resource), which is negligible when compared to the calls to `update_action_state`. Therefore the overall complexity for running such an example would be $\Theta(N^2(P+T))$, which is also unacceptably high for large N and large traces.

4. IMPROVING THE SIMGRID CORE

4.1 Lazy Action Management

The main reason why SimGrid has such a high complexity is that it was initially designed to handle hierarchical heterogeneous networks, where an action may interfere with many others. However, when simulating a VC system, most resources are independent from each other. In such a situation, a careful management of LMM makes it possible to amortize the complexity of `lmm_solve` but the complexity of `share_resources` is still $\Theta(|actions|)$ as the completion date of each action is recomputed after the call to `lmm_solve`. Yet, only the actions whose resource shares have just been modified in `lmm_solve` need to be updated. We introduce a future event set, implemented as a heap, in which we store the completion date of the different actions. When a resource share is modified, all corresponding actions are removed from the set. The completion date of each such action is then updated and the action is reinserted into the heap. Removing and inserting elements in the heap has $\Theta(\log(|actions|))$ complexity and computing the minimum completion date to return to SURF's main loop is now $O(1)$.

The last remaining expensive function is `update_action_state`. This function is supposed to update the state of all actions, namely remaining work amounts, and return completed and failed actions. There is thus no hope to reduce its $\Theta(|actions|)$ complexity if all actions need to be updated. Instead, we update remaining work amounts lazily. More precisely, we update a remaining work amount only when needed for recomputing a completion date or when requested from a use-level API call. In `update_action_state` we only remove from the future event set the actions that complete, which enables us to reduce the complexity of `update_action_state` to $\Theta(\log(|actions|))$

In our simple example, the complexity of `surf_solve` is thus $\Theta(\log N)$ and the overall complexity of the simulation $\Theta(NP.\log(N))$, instead of $\Theta(N^2P)$.

In a more complex example (e.g., involving a network model), if the completion date of most actions need to be updated then our new algorithm may be slower than the previous algorithm ($\Theta(|actions|.\log(|actions|)$ compared to $\Theta(|actions|)$). Therefore, although our improvements should be beneficial in most situations, the SimGrid user can deactivate them per model if needed.

4.2 Trace Integration

Let us assume now that our CPU resources are associated with traces. At each trace event, we call `update_resource_state`, which leads to recomputing the sharing of the corresponding resource during the next call to `surf_solve`. This is unavoidable when simulating complex situations (e.g., a network) but again, when resources do not interfere with each others it may be avoided.

Let us consider a CPU and denote by $p(t)$ its computational power (in MFlop/s) at time t. Let us define $P(x) = \int_0^x p(t).dt$. Let us consider n actions $A_1, \ldots, A_n$ running on this CPU, each with R_i remaining MFlop to do at time t_0. At any time $t \geqslant t_0$, each action receives a share of the CPU's computational power equal to $p(t)/n$. Therefore, during the time interval $[a, b]$, action A_i will advance by $\int_a^b p(t)/n.dt = 1/n \int_a^b p(t).dt$.

The value returned by `share_resources` is the largest t_1 such that $1/n \int_{t_0}^{t_1} p(t).dt \leqslant R_i$, i.e., t_1 is such that $\int_{t_0}^{t_1} p(t).dt = n \min R_i$. The goal is to compute the first completion of all active tasks on each resource, which then amounts to solving $P(t_1) = P(t_0) + n \min R_i$ for all resources.

By pre-integrating the trace, t_1 can be computed using a simple binary search. The time needed to pre-integrate the trace is the same as parsing the trace (instead of storing the trace p, we simply store its integral P as cumulative sums) and finding t_1 is thus logarithmic in the trace size.

In our VC simple example, the sharing needs to be recomputed for only one host at each call to `surf_solve` and the cost of `share_resources` is thus only $\Theta(\log(T) + \log(N))$, when combined with the previously described lazy action management. Hence, the overall complexity of the simulation is $\Theta(NP(\log(N) + \log(T)))$ (instead of $\Theta(N^2(P+T))$

with no improvement and $\Theta(N(P+T)\log(N))$ with lazy action management).

Note that unlike the lazy action management mechanism described in the previous section, this improvement works only for CPU resources. However, it is fundamental for VC simulation since host states are often described by large traces.

5. PERFORMANCE EVALUATION

5.1 Simple Case Study

We consider a simple platform with N hosts with compute speeds sampled from the speeds found in SETI@home traces [21]. In the first experiment, these hosts are available at all times, whereas in the second experiment we use the SETI@home availability traces. On each host a worker process sequentially computes tasks whose work amount, in MFlop, is uniformly sampled between 0 and 8.10^{12} (i.e., up to roughly one day on a standard machine). The source code for the simulator can be found at http://mescal.imag.fr/membres/pedro.velho/lsap10 along with scripts for running experiments and analyzing experimental results. All our improvements have been implemented in the SimGrid source code and integrated in the last stable version (3.4).

The timings reported in this section were obtained on an AMD Opteron 248 Dual Core (2.2 GHz) with 1MB of L2 cache and 2 GB of RAM. Experiments were performed at least 10 times each and the 95% confidence interval based on Student's distribution is plotted on all graphs (even though they are generally so small that they can hardly be seen).

We compare three versions of SimGrid:

- **Default CPU Model:** original SimGrid implementation, as described in Sections 3.2 and 3.3;
- **Lazy Action Management:** as above, but using the improvement described in Section 4.1;
- **Trace Integration:** using the improvement described in Section 4.1 and 4.2.

5.1.1 Experiments Without Traces

In these experiments the number of hosts, N, ranges from 10 up to 10,240 and a week of computation is simulated. No availability traces are used. Figure 2 (a) plots the execution time as a function of N for the three versions of SimGrid.

For the default model, simulations were limited to 2,560 hosts due to extremely long execution times (e.g., 16 hours for 5,120 hosts with the default model). As expected, the new two SimGrid versions scale significantly better. The seemingly linear shape of the curves is due to the $O(N^2P)$ complexity of the default model, and the $O(NP\log(N))$ of the new versions. Both new versions have similar performance because no host availability traces are used. They are able to simulate one week of computation with 10,240 hosts in about 1 minute.

5.1.2 Experiments with Traces

Figure 2 (b) shows results for experiments with host availabilities described by SETI@home traces. These traces are from the Failure Trace Archive [21] and total from 148KB (for 10 hosts) to 177MB (for 10,240 hosts). The trace parsing time may thus be a substantial part (about one third when using trace integration) of the simulation executions time.

The results with the default model are limited to 2,560 hosts due to scalability issues, and results with lazy action management as limited to 5,120 hosts. Indeed, the lazy action management optimization is not sufficient to achieve scalability in the presence of availability traces. This is because its complexity is linear in the number of trace events, i.e., $O(N(P+T)\log(N))$. The trace integration reduces this complexity to $O(NP(\log(N)+\log(T)))$, which explains its better scalability. The last version of SimGrid is the version of choice for VC simulations. In this example it manages to run simulations essentially as fast as in the no-trace case.

Project	Einstein	SETI	LHC
Process Time (hours)	24	1.5	1
Deadline (days)	14	4.3	4

Table 1: BOINC project characteristics

Time	SimGrid			BOINC client simulator		
(hours)	Einstein	SETI	LHC	Einstein	SETI	LHC
100	1	23	27	1	21	33
500	7	112	163	7	108	166
1,000	12	220	272	14	220	331
5,000	70	1106	1565	70	1103	1652
10,000	139	2221	3327	139	2214	3319

Table 2: Number of completed tasks for the SimGrid and the BOINC client simulator.

5.2 A BOINC-like Simulation

As a proof of concept of SimGrid's ability to simulate VC systems, we have implemented a simulator for a BOINC architecture. This simulator uses the MSG API to SimGrid and consists of about 800 lines of C source code. Our simulator implements the BOINC client scheduling algorithm as described in [3]. The source code of the real BOINC client consists of 20,000+ lines and its scheduler core (`work_fetch.cpp` and `cpu_sched.cpp`) of 2,000+ lines. Our version is thus simplified. For example, we do not simulate multi-core hosts or application checkpointing. However, we implement most important features of the real scheduler (deadline scheduling, long term debt, fair sharing, exponential back-off).

In Section 5.2.1, we compare our simulator with the BOINC client simulator, which allows the simulation of one BOINC host. We run this simulator using its default settings. The goal of this comparison is to validate the simulation results produced by our simulator. In Section 5.2.2 we compare the scalability of our simulator to that of the SimBA simulator [32]. We conducted our experiments on a machine similar to that used in [32], a 3.0 GHz Intel Pentium 4 with 1GB RAM, so that our execution time measurements can be compared fairly to those published for SimBA.

For all these experiments, we used SimGrid with all the improvements described in Section 4 and using the simple network model based on uniform random distributions.

5.2.1 Comparison with the BOINC Client Simulator

We simulate three projects that we deem representative of the diversity of BOINC projects: Einstein@home, SETI@home, and LHC@home. Table 1 gives the time to process a project task and project-imposed task deadlines, as found in the official catalog of active BOINC projects [1].

Table 2 shows the number of tasks completed for each project within a given time period ranging from 100 to 10,000 hours according to both our SimGrid simulator and the

[1] http://www.boinc-wiki.info/Catalog_of_BOINC_Powered_Projects

Figure 2: Execution time vs. the number of simulated hosts using the different proposed improvements on a log-log scale, (a) without availability traces and (b) using availability traces.

BOINC client simulator. In all experiments, no task deadline were missed. More importantly, we see that the numbers of completed tasks are very close. This is true even for large time frames, which shows that the cumulative "error" introduced by our simulator is small. More thorough experiments would be needed to fully validate our simulator, but based on these results we simply say that our simulator does a reasonable job of simulating BOINC.

In terms of execution times we found that, surprisingly, the SimGrid simulator is faster than the BOINC client simulator, even though the latter is specific-purpose and time-driven and thus should be extremely fast. For instance, for 10,000 hours of simulated time, our simulator runs in about 1 second while the BOINC client simulator runs in 1 minute.

5.2.2 Comparison with SimBA

For results in this section we configured SimGrid so that the stack size of the simulated processes is reduced from the default 128KB to 16KB. This is so that the simulation can fit on a machine with 1GB of RAM. By default SimGrid implements simulated processes using System V contexts. While SimGrid supports POSIX threads, System V contexts are more lightweight and better suited for the SimGrid core implementations. The execution times measured with our simulator and those reported by Estrada *et al.* in [32] are contrasted as follows:

- They simulated 15 days of computation with 7,810 hosts working on the Predictor@Home project in 107 minutes. Our simulator simulates the same configuration in under 4 minutes (for 10 runs the 95% confidence interval is $[208.13, 223.71]$ in seconds).
- They simulated 8 days of computation with 5,093 clients working on the CHARMM project in 44 minutes. Our simulator simulates the same configuration in under 1.5 minutes (for 10 runs the 95% confidence interval is $[80.38, 80.87]$ in seconds). Note that in both experiments, we used host availabilities described by SETI@home traces and about one third of the simulation time was spent parsing these traces.

Our hope was that our simulator would not be "too slow" when compared to SimBA, but we found it to be faster. This is surprising given that SimBA opts for many of the simplest options for addressing the design issues outlined in Section 2.2.1. Instead SimGrid, because its goal is to be a powerful and general-purpose simulation framework, opts for more sophisticated but costly options (e.g., use of host availability traces, simulation of processes that execute arbitrary code using a simulation API). As a result, a VC simulator build with SimGrid benefits from all the advantages that SimGrid has afforded to many simulators in the past. Among these, and perhaps most important, is the fact that the simulator can be programmed and thus extended easily. Consequently, it would be straightforward to implement new client scheduling strategies using the MSG API of SimGrid. Similarly, the implementations of the server could be be extended using the same API to implement various scheduling policies. Another immediate extension, which would require essentially no extra work, would be to activate one of SimGrid's network model to take network contention into account.

6. CONCLUSION AND FUTURE WORKS

Given the scale and nature of VC systems, it is often thought that they cannot be simulated efficiently by general-purpose simulators. As a result, most existing VC simulators use simple (and often naïve) design options for achieving scalability at the expense of simulation realism and extensibility. In this work we have presented and analyzed the internals of the SimGrid general-purpose simulation framework, which affords simulation realism and extensibility, and have identified performance and scalability bottlenecks that come into play for VC simulations. We have proposed and implemented improvements to remove these bottlenecks. These improvements have been evaluated and, using simple benchmarks, have been shown to lead to better performance than the original SimGrid version by orders of magnitude. We have also implemented a simulator for a simple BOINC system. Our key result is that this simulator is significantly more scalable than a previously implemented BOINC simulator. Since SimGrid is a programmable simulation framework, this simulator can easily be extended to conduct complex studies that are simply not possible with previous approaches. For example, it is now possible to study a scenario in which projects have interest in response time rather than

throughput and thus develop aggressive scheduling/replication strategies. SimGrid is thus not only a viable, but in fact an attractive platform for VC simulation both in terms of the power of the simulation and of its scalability.

A clear future direction is to focus on improving the scalability of some of SimGrid's network models. All results in this article are for simulations that use very simple network models since the focus is on compute-intensive scenarios, as done in most previous VC research. However, due to recent advances in internetworking technology, it is conceivable to run communication-intensive applications on VC systems. Most volunteers use DSL connections that have very specific behaviors. SimGrid provide complex but realistic communication models for grid platforms. These models may not be adequate to DSL connections. Moreover, these network models as implemented in SimGrid currently do not scale well beyond a few thousand hosts. We are in the process of designing both realistic and scalable network models in this context.

Acknowledgments

This work is partially supported by ANR (*Agence National de Recherche*), project reference ANR 08 SEGI 022 (USS SimGrid) and ANR 07 JCJC 049 (DOCCA). Special thanks to CAPES/Brazil, which supports Pedro Velho's doctoral research, and to INRIA, which supports Bruno Donassolo's internship.

Many thanks to Martin Quinson for the many insightful discussions about scalability, and to Eric Heien for submitting the simple example that motivated this work. Last but not least, many thanks to Derrick Kondo who has provided us with many references and information on BOINC.

7. REFERENCES

[1] D. P. Anderson. BOINC: a system for public-resource computing and storage. In *Proc. of the 5th IEEE/ACM Intl. Workshop on Grid Computing*, 2004.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11), 2002.

[3] D. P. Anderson and J. McLeod VII. Local Scheduling for Volunteer Computing. In *Proc. of the Intl. Parallel and Distributed Processing Symp. (IPDPS 2007)*. IEEE Intl., 2007.

[4] M. Barisits and W. Boyd. MartinWilSim Grid Simulator. Summer internship, Vienna UT and Georgia Tech, CERN, Switzerland, 2009. Available at http://www.slideshare.net/wbinventor/slides-1884876.

[5] D. Bertsekas and R. Gallager. *Data Networks*, chapter 6. Prentice-Hall, 1992.

[6] R. Buyya and M. Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13-15), 2002.

[7] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Neri, and O. Lodygensky. Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. *Future Generation Computer Systems*, 21(3), 2004.

[8] F. Cappello, G. Fedak, D. Kondo, P. Malécot, and A. Rezmerita. *Handbook of Research on Scalable Computing Technologies*, chapter Desktop Grids: From Volunteer Distributed Computing to High-Throughput Computing Production Platforms. IGI Global, 2009.

[9] H. Casanova. Simgrid: a Toolkit for the Simulation of Application Scheduling. In *Proc. of the 1st IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, 2001.

[10] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proc. of the 10th IEEE Intl. Conf. on Computer Modeling and Simulation*, 2008.

[11] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3), 2006.

[12] F. A. B. da Silva and H. Senger. Improving scalability of Bag-of-Tasks applications running on master-slave platforms. *Parallel Computing*, 35(2), 2009.

[13] W. Depoorter, N. Moor, K. Vanmechelen, and J. Broeckhove. Scalability of grid simulators: An evaluation. In *Proc. of the 14th Intl. Euro-Par Conf. on Parallel Processing*, 2008.

[14] T. Estrada, D. A. Flores, M. Taufer, P. J. Teller, A. Kerstens, and D. P. Anderson. The Effectiveness of Threshold-Based Scheduling Policies in BOINC Projects. In *Proc. of the Second IEEE Intl. Conf. on e-Science and Grid Computing (e-Science)*, 2006.

[15] T. Estrada, M. Taufer, K. Reed, and D. P. Anderson. EmBOINC: An emulator for performance analysis of BOINC projects. In *Proc. of the 3rd Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid)*, 2009.

[16] E. M. Heien, N. Fujimoto, and K. Hagihara. Computing low latency batches with unreliable workers in volunteer computing environments. In *Proc. of the Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2008.

[17] D. Kondo. SimBOINC: A Simulator for Desktop Grids and Volunteer Computing Systems. http://simboinc.gforge.inria.fr/, 2007.

[18] D. Kondo, D. Anderson, and J. V. McLeod. Performance Evaluation of Scheduling Policies for Volunteer Computing. In *Proc. of the 3rd IEEE Intl. Conf. on e-Science and Grid Computing (e-Science)*, Bangalore, India, 2007.

[19] D. Kondo, A. A. Chien, and H. Casanova. Rapid application turnaround on enterprise desktop grids. In *Proc. of the ACM Conf. on High Performance Computing and Networking (SC)*, 2004.

[20] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova. Characterizing resource availability in enterprise desktop grids. *Journal of Future Generation Computer Systems*, 23(7), 2007.

[21] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In *Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid)*, 2010.

[22] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. A. Chien. Characterizing and Evaluating Desktop Grids: An Empirical Study. In *Proc. of the Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2004.

[23] R. Kumar, Y. Liu, and K. Ross. Stochastic fluid theory for p2p streaming systems. In *INFOCOM*, 2007.

[24] S. H. Low. A Duality Model of TCP and Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 11(4), 2003.

[25] M. Mowbray, F. Brasileiro, N. Andrade, J. Santana, and W. Cirne. A reciprocation-based economy for multiple services in peer-to-peer grids. In *Proc. of the Sixth IEEE Intl. Conf. on Peer-to-Peer Computing (P2P)*, Washington, DC, USA, 2006. IEEE Computer Society.

[26] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. Towards Yet Another Peer-to-Peer Simulator. In *Proc. of HET-NETs*, 2006.

[27] D. Nurmi, J. Brevik, and R. Wolski. Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments. In *Proc. of the Intl. Euro-Par Conf. on Parallel Processing*, 2005.

[28] D. Qiu and R. Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

[29] G. F. Riley. The Georgia Tech Network Simulator. In *Proc. of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, 2003.

[30] E. Santos-Neto, W. Cirne, F. Brasileiro, and A. Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *Proc. of the Workshop on Job Scheduler Strategies for Parallel Processors (JSSPP)*, 2004.

[31] The SimGrid project. http://simgrid.gforge.inria.fr.

[32] M. Taufer, A. Kerstens, T. Estrada, D. Flores, and P. J. Teller. SimBA: A Discrete Event Simulator for Performance Prediction of Volunteer Computing Projects. In *Proc. of the 21st Intl. Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2007.

[33] P. Velho and A. Legrand. Accuracy study and improvement of network simulation in the simgrid framework. In *Proc. of the 2nd Intl. Conf. on Simulation Tools and Techniques (SIMUTools)*, 2009.