Scalable and Fast Simulation of Peer-to-Peer Systems Using SimGrid

Martin Quinson, Cristian Rosa, Christophe Thiery

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scalable and Fast Simulation of Peer-to-Peer Systems Using SimGrid

Martin Quinson, Cristian Rosa, Christophe Thiéry

N° 7653

2011

Domaine 3

ISSN 0249-6399 ISRN INRIA/RR--7653--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Scalable and Fast Simulation of Peer-to-Peer Systems Using SimGrid

Martin Quinson, Cristian Rosa, Christophe Thiéry

Domaine : Réseaux, systèmes et services, calcul distribué
Équipe-Projet AlGorille

Rapport de recherche n° 7653 — 2011 — 18 pages

Abstract: Simulation is a common experimental methodology in distributed systems since it allows to easily and quickly test ideas. In some domains such as Peer-to-Peer (P2P) systems or Volunteer Computing, most of the studies rely on simulation. Despite several simulators now available, many researchers still choose to develop their own custom tool. This can certainly be explained by the apparent simplicity of doing so, but this task becomes tedious when trying to simulate quickly very large systems in a realistic way.

In this paper we present the new architecture of the general-purpose simulation framework SimGrid, which provides significantly more realistic and flexible simulation capabilities than the aforementioned simulators. Our key contribution is a new implementation of the simulation core that enables the parallel execution of the user code during simulation, achieving faster and more scalable simulations. SimGrid now outperforms the reference simulator in the area in speed (one order of magnitude faster) and scalability (10 times bigger scenarios) while providing a better simulation accuracy. We discuss the key issues of implementing the parallelism, we analyze its trade-offs, and we give a criterion to understand in what kind of scenario a speedup can be expected. Finally we present several experiments to evaluate its performance and scalability in different domains, in particular a simulation instance of the Chord peer-to-peer protocol with two million nodes using a single computer.

Key-words: Peer-to-peer systems simulation, simulation performance

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Simulation extensible et rapide de systèmes pair-à-pair avec SimGrid

Résumé : La simulation est une approche populaire de prédiction des performances des systèmes distribués car elle permet une évaluation simple et rapide. Dans certains domaines comme le pair-à-pair ou le volunter computing, la plupart des études reposent sur la simulation. Bien que plusieurs simulateurs soient maintenant disponibles, de nombreux chercheurs choisissent de développer leur propre outil. Cela s'explique sans doute par le fait que cette entreprise semble simple au premier abord, mais cette tâche s'avère difficile lorsqu'il s'agit de simuler rapidement et de façon réaliste des systèmes de très grande taille.

Dans cet article, nous présentons la nouvelle architecture du framework généraliste de simulation SimGrid, capables de simulations plus rapides et flexibles que les simulateurs mentionés précédement. Notre contribution principale est l'implémentation d'un noyau de simulation permettant l'exécution parallèle du code utilisateur pendant la simulation, permettant ainsi des simulations plus rapides et extensibles. Les performances de SimGrid sont maintenant supérieures à celles du simulateur phare du domaine, à la fois en vitesse (un ordre de magnitude plus rapide) et en extensibilité (des simulations 10 fois plus grosses) tout en offrant une meilleure précision.

Nous discutons les principales difficultés liées à cette implémentation, nous analysons les compromis réalisés, et nous donnons un critère permettant de comprendre les types de scénarios expérimentaux les plus à même de profiter de ces changements. Enfin, nous présentons différents résultats expérimentaux pour évaluer les performances et l'extensibilité de notre solution dans différents domaines. En particulier, nous évaluons une simulation de deux millions de nodes suivant le protocole Chord sur une seule machine.

Mots-clés : Performance de simulation, Simulation de systèmes Pair-à-pair

imulation is a common experimental methodology in distributed systems since it allows to easily and quickly test ideas. In some domains such as Peer-to-Peer (P2P) systems or Volunteer Computing, most of the studies rely on simulation. Despite several simulators now available, many researchers still choose to develop their own custom tool. This can certainly be explained by the apparent simplicity of doing so, but this task becomes tedious when trying to simulate quickly very large systems in a realistic way.

In this paper we present the new architecture of the general-purpose simulation framework SimGrid, which provides significantly more realistic and flexible simulation capabilities than the aforementioned simulators. Our key contribution is a new implementation of the simulation core that enables the parallel execution of the user code during simulation, achieving faster and more scalable simulations. SimGrid now outperforms the reference simulator in the area in speed (one order of magnitude faster) and scalability (10 times bigger scenarios) while providing a better simulation accuracy. We discuss the key issues of implementing the parallelism, we analyze its trade-offs, and we give a criterion to understand in what kind of scenario a speedup can be expected. Finally we present several experiments to evaluate its performance and scalability in different domains, in particular a simulation instance of the Chord peer-to-peer protocol with two million nodes using a single computer.

1 Introduction

Distributed systems are difficult to get right because they mix difficulties of parallel and multi-threaded systems, such as race conditions and deadlocks, with specific ones, such as the absence of shared memory and time. Assessing their correctness and performance is thus very important. Unfortunately, it is also very difficult, specially at large scale. Direct execution of the tested applications would require a platform as large as the target platform, which is usually the whole Internet. Even in the cases where direct experimentation is actually possible, it remains a very time and labor consuming task, and the results remain difficult to reproduce by other scientists.

That is why simulation is one of the most widely used technique to test and assess the quality of distributed systems. Often, researchers develop their own custom simulator due to the apparent simplicity of doing so, but this becomes a tedious task when trying to simulate very large systems in a realistic way. According to the survey in [12], from 141 P2P papers based on simulation reviewed by the authors, 30% use a custom simulation tool while half of them do not even report which simulation tool was used. Moreover, most ad-hoc simulators lack a proper validation of their methodology that hinders the reliability of the results, and poses serious complications when trying to compare them.

The SimGrid framework [8] is a framework for the simulation of distributed computer systems that provides significantly more realistic and flexible simulation capabilities than the aforementioned simulators. SimGrid was conceived as a scientific instrument, thus the validity of its analytical models were thoughtfully studied [16], ensuring the realism of the results.

In this paper, we present the new design of SimGrid that enables the parallel execution of the user application during simulation and boosts the simulation speed and scalability. It is to be noted that the models constituting the simulation core are not parallelized, but that the simulation toolkit is. Our key contribution is a new implementation of the tool that permits faster and bigger simulations than reference *peer-to-peer* (P2P) simulators. First we discuss the limitations of the current design, then we introduce the main aspects of the new implementation, such as the threading

model, the synchronization primitives, and the interaction with the host's operating system, that are the key of an efficient parallel execution of the user code. Though the parallelism might seem an always winning scheme, it is not true because of the high level of optimization of the sequential execution. We thus give a criterion over the usage scenario to understand whether a speedup is to be expected. Finally, we present several experiments that justify the parallelism criterion and show that SimGrid now can handle simulations at peer-to-peer scale. The experiments rely on two distributed programs, one from the cluster computing domain, and one from the peer-to-peer computing area. The first is a classical parallel matrix multiplication (PMM) algorithm that notably benefits from the parallel execution due to the symmetry in its communication and computation pattern. The second is more challenging and uses the Chord [14] look-up protocol to show that SimGrid now can handle large simulation instances, like the ones required by the P2P community. We simulate Chord scenarios up to 2 millions nodes in about 3.3 hours on a single computer where comparable simulators of the area can only simulate up to 300 thousands nodes in 10 hours.

The rest of this article is organized as follows: Section 2 contextualizes our work by presenting the state of the art on simulation and highlighting the simulator internals relevant to our work. Section 3 presents the contributions of this article: we analyze the limitations of the current design, then we present the new architecture, we detail the threading model, and we discuss the importance of the interaction of the simulator with the operating system of the host machine. Section 4 shows the results of the experiments, and finally Section 5 concludes this article by summarizing the results and presenting the envisioned future work.

2 Context

2.1 State of the Art

Simulations are widely used by physicists, mathematicians, engineers, computer scientists, and even video game designers. They offer a mean to predict the probable outcome of experiments and aid to understand large and complex systems, facilitating their performance analysis and optimization.

Simulation always relies on models of the reality. A simulation model is an abstract representation of the physical system under study. In this article we understand *simulation* as Discrete Event Simulation [17] (DES), where the state of the model describing the system changes instantaneously at discrete points in the *simulated* time. There are two different notions of time when we speak about simulations, one is the real world time (or wall-clock time), the second one is the simulated time, that is the time elapsed in the simulated world. In this article, we assume that the simulated time advances triggered by the next earliest occurring event, thus it can advance faster or slower than the wall-clock time.

Over the last decade, several simulation solutions for distributed systems emerged as reference. They can be classified according to the type of system they allow to study. In the area of P2P networks, PeerSim [10] is a widely used simulator, conceived to be scalable and simple to adapt to the users needs. To achieve maximum scalability, it implements a query-cycle mode in addition to the classical event-driven simulation, on which the main loop iterates over every node to execute one event at each step, avoiding the need of allocating memory for the machine-state of the processes. It was reported to simulate up to one million nodes in this mode, at the price of an important

loss of realism due to the simplifications done for sake of extensibility and scalability. OverSim [3] is another P2P simulator that tries to address this problem by leveraging the handling of communications to a discrete event simulation kernel OMNet++ [2], a packet-level simulator comparable to the classical NS3 [1]. It was reported to simulate up to 100,000 nodes, but only when replacing OMNet++ by other internal mechanisms, and the validity of the results was never clearly demonstrated.

In the area of grid computing, numerous tools were produced but most of them were almost never used outside their developers. ChicSim [13] and OptorSim [4] are specifically designed to study data replication on grids, but they are discontinued. GridSim [6] is another widely used grid simulator which was initially conceived for studying grid economy and was later used in a more general-purpose way.

For studying cloud computing applications, only CloudSim [7] arises as a reference. It is based on GridSim, and exposes specific interfaces to study Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) settings. This lack of reference simulators in this domain is probably due to its recent nature.

2.2 The SimGrid Framework

The SimGrid Framework [8] is a collection of tools for the simulation of distributed computer systems. To our knowledge, it is the only tool that intends to provide a generic solution for the simulation of any kind of distributed system. It uses validated analytical models of the platform's resources, that translates in precise but yet fast simulations [16] for a wide spectrum of applications [9].

SimGrid's experimentation work-flow requires the user to provide three input elements:

Application to simulate. It must use one of the communication APIs offered by the framework. The main APIs are written in C, with bindings available for Java, Ruby and Lua.

Resource models. These are analytical models of the hardware like CPUs or network links, and they are used to compute the timings according to the resources in the platform to simulate. Users can provide new models or choose amongst the ones provided by SimGrid.

Experimental setup. This includes the description of the platform to simulate (hosts, links, the network topology, and routing information), the external workload experienced by the platform during the experiment and the deployment information (application's processes to run on the simulated hosts and their parameters).

Provided these elements, the simulation is then performed by executing the distributed application in a controlled environment, where the simulated processes are associated to the hosts in the virtual platform using the deployment information, and the applicative events (typically, emissions and receptions of messages) are intercepted and delayed according to the resource sharing timings computed by the models.

The redesign of SimGrid's internals constitutes the major contribution of this article. Figure 1 presents the the overall architecture of the framework to ease the forthcoming explanation of this contribution. The framework is composed of three major layers:

Simulation core (SURF). This module has an abstract view of the simulation where there are only resources (CPUs, network links, etc.) and actions that compete to consume these. It provides the analytical models of the hardware and uses them to share the resources in the virtual platform and compute the earliest finishing action in a given setting.

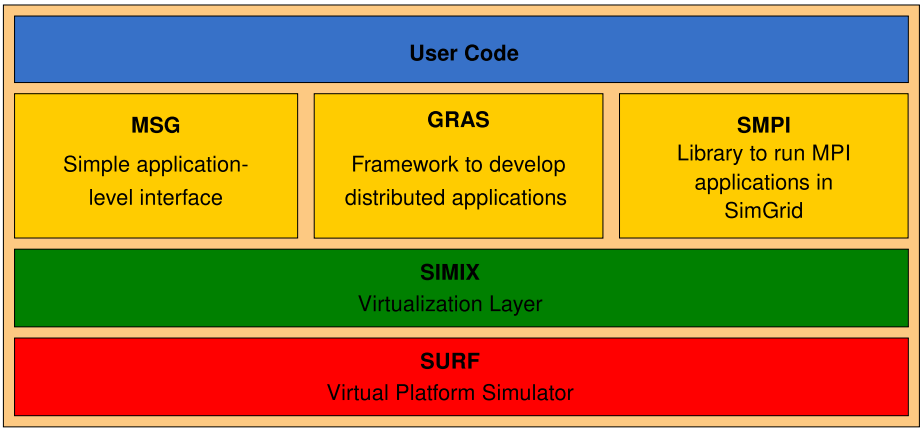

Figure 1: SimGrid's Layout.

Execution environment (SIMIX). It introduces a virtualization layer introducing processes, synchronization primitives and scheduling control. SIMIX executes the user processes and interrupts them when they make an action that require to use a SURF resource, like sending a message over the simulated network. It is thus binding the user processes to the simulated world.

Communication APIs (MSG, GRAS, SMPI). These modules provide user-level communication facilities with a specific syntactic sugar intended to ease the use of the simulator in a given context.

2.3 Formal Description of the Simulation

This section introduces an abstract formal notion of simulation used below to simplify the presentation.

A distributed system is described by the set of processes P that composes it. Each $p \in P$ has a local state that evolves by the executions of events (or actions) $e \in E$, and we denote e^p to refer to the events executed by the process p.

The execution of a distributed system can be formally described by the potential causality relation of the events produced by the concurrent processes that compose it, often known as the *happened-before* relation [11] (denoted as $\rightarrow$). Each $\rightarrow$ defines a partial-order in the set of local states, therefore the behavior of a distributed system can be formally characterized by the set of $\rightarrow$ relations that it can generate.

Simulation. *Given a distributed system composed of P processes, a description of the platform R, and analytical models of the resources M, then a simulation is an algorithmic procedure to compute one possible $\rightarrow$ of P with timestamps t_i associated to each of its events that represent the* simulated time *on which each event finishes. These timestamps are calculated using the models M subject to the restrictions imposed by the resources R.*

$$S(P, R, M) = <\rightarrow, \{e_{t_0}, e_{t_1}, \ldots, e_{t_k}\} >$$

with $t_0 = 0$.

Note that with this definition, it is possible to have more than one event with the same timestamp if the models determine that several events finish at the exact same time.

Scheduling Round. *Given a simulation $S(P, R, M)$, for each timestamp t_i we define the set P_{t_i} of processes which have a blocking event that finishes at time t_i. We call this set a* Scheduling Round *as these are the processes that can execute at t_i.*

$$P_{t_i} = \{p \in P \mid \exists\, e^p_{t_i} \in S(P, R, M)\}$$

2.4 Prior Simulation Algorithm

We now detail how we used to compute a simulation $S(P, R, M)$ before the work presented in this article, which were introduced in SimGrid v3.6. Algorithm 1 shows a very abstract implementation of the main loop. The variable $time$ represents the simulated time, and P_{time} is the scheduling round corresponding to the timestamp $time$, in other words it is the set of processes ready to be executed at $time$. Processes not ready at any given time are blocked waiting for the end of a specific action, such as a message delivery or reception. At each iteration, the virtualisation layer (SIMIX) schedules the processes in the set P_{time} (line 4). They execute until an interaction with the simulated platform is required, like sending a message, point on which SIMIX blocks them and creates the respective action in the simulation core (SURF). Once all the processes are blocked, the loop calls the simulation core (line 5) to compute and advance the simulated time to the next earliest ending actions (the next timestamp). In return, it gets the list of finished actions which uses to compute the next scheduling round (line 6). The simulation is over when there is no more processes to run.

Algorithm 1 Main Loop in SimGrid v3.5.

```
1: time ← 0
2: P_time ← P
3: while P_time ≠ ∅ do
4:     schedule(P_time)
5:     time ← surf_solve(&done_actions)
6:     P_time ← process_unblock(done_actions)
7: end while
```

From the implementation point of view, SimGrid runs the entire simulation as a single process in the host machine. To achieve this, the virtualization layer (SIMIX) folds the user's processes into *execution contexts* composed by a stack and storage space to save the CPU state (registers). The simulator itself runs in the default execution context of the process in the host machine, called *maestro*. It is responsible of executing the computations of the core (SURF), and it controls the scheduling using subroutines that swap execution contexts.

Figure 2 depicts a macroscopic description of the execution contexts of the modules in SimGrid for one iteration of Algorithm 1, where U_1, U_2 are the contexts of the user processes, and M is the maestro context. The loop schedules sequentially U_1 and then U_2 until they block waiting for their actions to finish (USER+API+SIMIX). Then, it calls SURF to share the resources and compute the time of the next earliest ending actions, in this case t_{n+1}. The scheduling round finishes when SIMIX determines the list of contexts to schedule in the next iteration of the loop.

It is interesting to recall the difference between wall-clock time and simulated time. In Figure 2 the simulated time advances discretely on each call to SURF and remains constant during the scheduling rounds, while the wall-clock time advances normally.

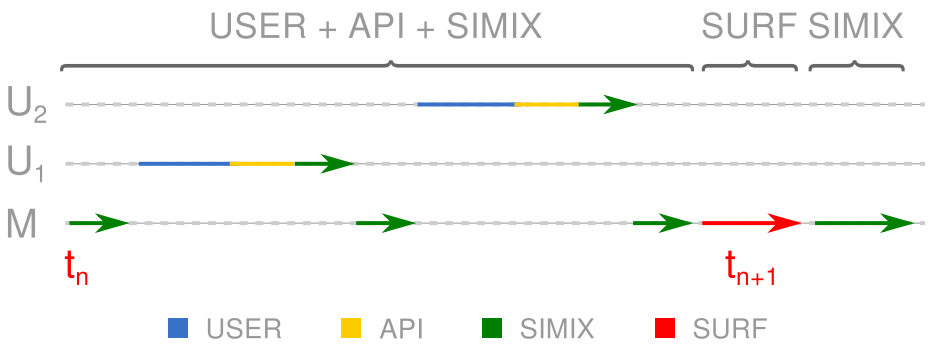

Figure 2: Simulation Main Loop in SimGrid v3.5.

3 Contributions

Though the architecture of SimGrid v3.5 seems adapted to the needs of a simulator, it is limited by design decisions that do not correctly address the new requirement of parallel execution. In addition, its implementation has shortcomings that hinder the overall scalability and performance of the simulator.

In this section, we detail the limitations of SimGrid v3.5, we introduce the design of SIMIX2, a new virtualisation layer that replaces SIMIX in SimGrid v3.6 and above, and addresses the requirements of simulation with parallel user code execution. It is to be noted that the model resolution constituting the simulation core is not parallelized by itself, but the parallelization only occures for the user code running in each simulated process. The reason for this is that most of the time is spent in the user code, and also to reduce the burden of developping new models in SimGrid.

Our contributions are built from the observation that the services offered by the simulator are similar to those provided by an operating system (OS): processes, inter-process communication, and synchronization primitives. Next we present the multi-threading architecture used to execute the user code and show that the interaction with the operating system of the host is crucial to the performance of the simulations.

3.1 The Shared State

In SimGrid v3.5, the data structures that compose the state of the simulation are scattered across the whole software stack. For example, SIMIX maintains the list of processes to execute on each scheduling round (P_{time}), the association of the simulator's core actions with the blocked processes, etc. In particular, the communication APIs control the network state, used to determine the source and destinations of messages (and thus the $\rightarrow$ relation to execute).

The problem with the organisation depicted in Figure 2 is that the user contexts do not only execute the user code which modify data that are private to the process they simulate. In addition, they also run the calls to the communication APIs that affect the shared network state plus the code of SIMIX that creates the actions on the simulation core (USER+API+SIMIX arrows). Under sequential simulation, this poses no problem, as the mutual exclusion is trivially guaranteed. However, the parallel execution of the user contexts under that design would require fine-grained locking across the entire software stack. This would be both extremely difficult to get right, and prohibitively expensive in terms of performance. Moreover, each communication API implements its

own logic to affect the state of the network. So each API would require its own locking mechanism with little possible logic factorization between APIs.

Finally, even if these difficulties were solved to ensure the internal correction of the simulator, the parallel execution with this design would hinder the reproducibility of simulations, because the chosen execution depends on the scheduling order of the processes during a given scheduling round, that can potentially vary between simulations that run them in parallel. In other words, race conditions could occure at the applicative level too, resulting in differing simulated results. Assume for example that three processes A, B and C are scheduled in a given round. A issues a receive request while B and C issue send requests. To ensure the reproducibility of the applicative scenario, whether A receives the message of B or the one of C must not vary from one run to another. Under the design of SimGrid v3.5, if the user contextes were executed in parallel, the ordering would unfortunately vary and A would receive the message from the first ending context between B and C.

All these reasons explain that distributing the code modifying the simulation's shared state across the execution contexts of the user code clearly hinders the possibility of running these contextes in parallel efficiently.

3.2 Kernel-mode Emulation

A distinctive characteristic of most modern operating systems is the use of a system call interface to provide services that the program does not normally have permission to run. The user processes execute in a virtual address space, isolated from the rest, and the system calls are the only interface with the rest of the system. On the other hand, the kernel runs in a special supervisor mode, and has a complete view of the system state. This clear separation between the user process in one side, and the kernel in the other, permits the independent and parallel execution of the processes, as all the potential access to the shared state are mediated by the kernel, that is responsible of maintaining the coherence.

Inspired by these ideas, we designed SIMIX2, a new virtualization layer that emulates a system call interface called *requests*. It completely separates the execution context of the user code and the simulation core (maestro), ensuring that the shared state can only be accessed from the maestro execution context. When a process performs an action that requires the interaction with the platform (like executing a computing task or sending a message), it issues the corresponding request through the interface which stores the request and its arguments in a private memory location, and then blocks the current execution context until the answer is ready. SIMIX2 executes in the *maestro* context, and handles the requests in a deterministic order (using the process IDs of issuers as a key).

Figure 3 schematizes a scheduling round following the new design. The main difference is that now, the user contexts only execute the user processes and the portions of the user APIs not involving any modifications to the shared state.

The proper split enforced by the request interface overcomes most of the limitations of the old design as the modification of the simulation shared state is now always done by the request handlers, that execute sequentially in the maestro context. This removes the need of the fine-grained locking scheme, and allows a simpler parallelization implementation presented in the next section.

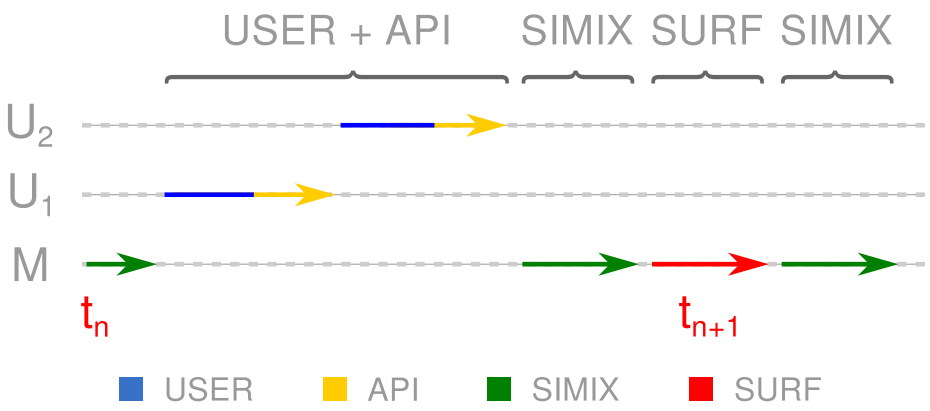

Figure 3: Simulation Main Loop in SimGrid v3.6.

3.3 The Multi-threading Architecture

The previously presented simulation loop relies on co-routines to handle each execution contextes. Full featured threads can naturally be used to that extend, but other lighter solutions such as `ucontext`es, which are part of the POSIX [15] standard, are also possible. They were not designed for concurrent nor parallel operation. Their interface only provides a few functions to get, set, and swap contexts, as they were originally conceived as an evolution of the setjmp/longjmp functions. Moreover, only recent operating systems allow to use ucontextes in conjunction with regular threads.

Full featured threads are not adapted to our case for several reasons. First, every operating system has a hard limit in the number of threads that can be owned by a user (in the order of thousands), where we envision simulations with millions of processes. In addition, even if the simulation is small enough to not encounter system limits, this design is inefficient since the number of CPUs available to run the threads is usually lower than the amount of threads to execute, resulting in a performance loss due to the contention.

Instead, in SIMIX2 we follow a different approach that mixes contexts and threads, taking advantage of the best features of each alternative. The new design still uses a ucontext for each simulated process, however it relies in a thread pool-like construction that distributes the scheduling jobs among worker threads, whose number depends in the amount of available CPUs or cores.

Algorithm 2 shows the new main loop found in SIMIX2. The sequential schedule function in line 4 of Algorithm 1 is now replaced by a parallel one that relies on the thread pool.

Algorithm 2 SIMIX2 Parallel Main Loop.

1: $time \leftarrow 0$
2: $P_{time} \leftarrow P$
3: **while** $P_{time} \neq \emptyset$ **do**
4: parallel_schedule(P_{time})
5: handle_requests()
6: $time \leftarrow$ surf_solve($\&done\_actions$)
7: $P_{time} \leftarrow$ process_unblock($done\_actions$)
8: **end while**

Figure 4 depicts a parallel scheduling round P_{t_n} with four user processes that execute in the contexts $U_1, \ldots, U_4$, scheduled onto two threads T_1 and T_2. Inside each thread the scheduling mechanism remains the same except now the context swap is done between the thread's context and the one to run.

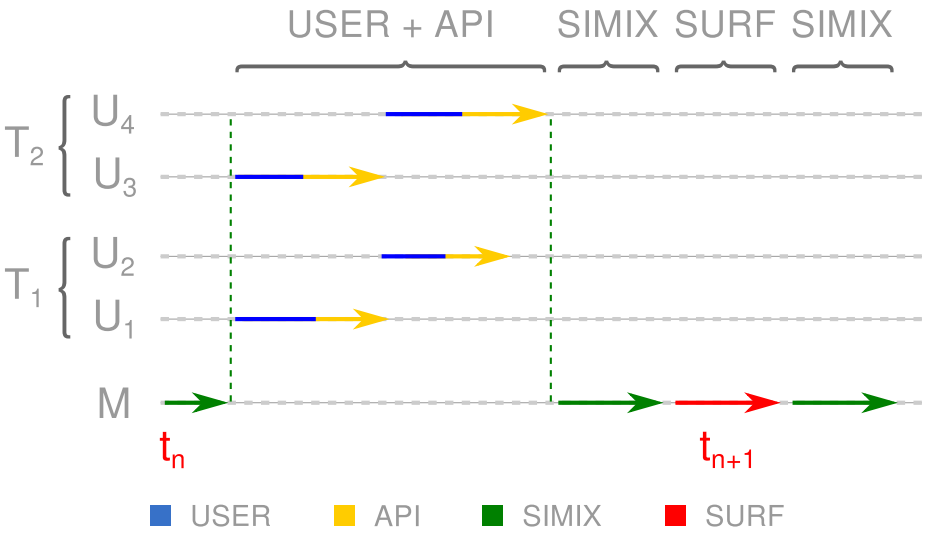

Figure 4: Parallel Scheduling Round

3.4 Managing Concurrency Efficiently

System programming design practices are not only useful as a guideline for the architecture of a simulator, they are also crucial to optimize its performance. The interaction with the operating system of the host involves issuing expensive system calls that might go unnoticed at first, but they produce a performance hit when scaling-up to larger simulations instances.

An interesting case is the virtualization for the user processes, that is in the critical path of the simulator. As mentioned before, the user code runs in execution contexts that emulate a cooperative multitasking environment entirely in user space. Using the POSIX's `ucontexts`, the execution is transferred from one ucontext to another using a `swapcontext` function, which is passed a pointer to the stack to restore and a pointer to a storage where the current stack should be saved. At the first glance, this function runs entirely in user space without requiring intervention of the OS kernel, but it turns out that this is not true. Indeed, POSIX allows to specify a different signal

$$
\begin{aligned}
C_{seq} &= \sum_{t_i \in S(P,R,M)} \Big(C_{surf}(R,M) + C_{smx}(|P_{t_i}|) + C_{usr}(P_{t_i}) \Big) \\
C_{par} &= \sum_{t_i \in S(P,R,M)} \Big(C_{surf}(R,M) + C_{smx}(|P_{t_i}|) + C_{thr}(|T|) + \max_{w \in T}(C_{usr}(P^w_{t_i})) \Big)
\end{aligned}
$$

Figure 5: Computational cost of sequential and parallel simulations.

mask for each ucontext, which induces an operation involving a system call during the swap. Since we do not need such feature, SimGrid now offers an alternative ucontext implementation that is syscall free. Because the swap routine modifies specific registers, it is programmed in assembly language and it is architecture dependent. For the moment this option is only available for x86 and x86_64 hardware, and other architectures fall back to the standard ucontexts.

The thread pool used to implement the parallel execution of the user code is another point in the critical path of the simulator. The synchronization of the worker threads involves system calls that must be minimized as much as possible, otherwise the parallelization could reveal counter-productive. The POSIX standard defines several high level primitives such as mutexes or condition variables to this end. This allows to write portable applications even if each operating system implement different system calls as the building blocks for more abstract constructions. For example, Linux has Futexes, and BSD has Spin-locks, but their direct usage remains tedious and should be reserved to advanced users.

However, *barriers* like the ones used at the end of each scheduling round, are not directly part of POSIX. Although it is possible to construct one from the pthreads primitives, this poses serious difficulties from the performance point of view. For example, one possible way of instrumenting a barrier is using a condition variable, where the last entering process signals it instead of blocking. Despite being correct from the functional point of view, it would generate unnecessary system calls due to the specification of the condition variable, that requires a mutex to protect it even if there is no need of mutual-exclusive access.

In SimGrid, the implementation of the thread pool's barrier uses a combination of the operating system primitives and some atomic operations available as compiler extensions. Thanks to this design it achieves the optimal cost in the number of system calls. On each scheduling round there is one system call to unblock all the worker threads, one to block maestro, then one for each worker when entering the barrier, and finally one to resume maestro. For the moment this functionality is only implemented in Linux, and fall back implementations using classical POSIX synchronization primitives are provided for other systems.

3.5 Cost of the Parallelism

It is a common belief that parallel execution always lead to performance improvement. The simulation is however an inherent sequential problem, and it is important to know the different costs at play to understand the implementation trade-offs and predict the scenarios for which a benefit can be expected compared to the standard sequential execution.

Figure 5 shows an approximation of the computational cost of the sequential (Algorithm 1) and parallel (Algorithm 2) execution. In both cases it is estimated adding up the costs of each individual scheduling round (loop iterations). C_{surf} represents the cost of computing the time of the next ending actions (the call to *surf_solve()*) and depends on both the size of the platform R and the precision of the models M. C_{smx} is the cost of SIMIX to handle the requests issued by the processes. This is a function of the amount of processes since each process issues one request per scheduling round. C_{usr} is the cost of the user code plus its scheduling (swap of execution context), and depends on the set of processes in the round since the execution time may differ from one user process to another. Particular to the parallel execution, T is the set of worker

threads, C_{thr} is the cost of their synchronization that depends on their amount, and $P_{t_i}^w$ is the subset of processes in P_{t_i} scheduled by the thread w.

The following equation details the criterion to decide whether the parallel execution can outperform the sequential one ($C_{par} < C_{seq}$) in a given setting.

$$\sum_{t_i \in S(P,R,M)} \left(C_{thr}(|T|) + \max_{w \in T}(C_{usr}(P_{t_i}^w)) \right) < \sum_{t_i \in S(P,R,M)} C_{usr}(P_{t_i})$$

Intuitively, if the proportion of blue arrows (user code) is dominant in Figure 4, then a significant improvement can be expected from the parallel execution. In other words, the size of the user code greatly impacts the potential gain of the parallel execution.

A simulation of a few processes with very small local computations would not benefit from the parallel execution since the performance cost of the thread synchronization would not be amortized by the enabled parallel execution. When $C_{usr} \to 0$, the performance penalty of parallelism for a simulation of K scheduling rounds is given by $C_{par} - C_{seq} \approx K \cdot C_{thr}(T)$

Another observation is that the precision of the models have also a great impact on the potential gains of the parallelism. The more precise M is, the larger the amount of timestamps t_i in $S(P, R, M)$ gets. Then, if the total amount of work done by the application remains constant (i.e. the amount of exchanged messages), with a precise model it can possibly be distributed across more timestamps, resulting in more numerous but smaller scheduling rounds P_{t_i}, and thus less user code executed in parallel and a higher cost of thread synchronization (K gets bigger).

This characterization allows to analyze where the performance losses come from. First, multi-threading does not comes for free, locking/unlocking the threads has a cost that depends on their amount, so potential parallelism between the code segments to execute in parallel should be large enough to amortize this cost. Second, the barrier that synchronize the threads at the end of each scheduling round is an idling point, and the simulation can continue only when the last thread reaches the barrier. Load balancing between threads is thus an important aspect.

4 Evaluation

4.1 Experimental Settings

The forthcoming experiments rely on two use cases. The first one consists of a classic parallel matrix multiplication algorithm (PMM) for a grid of processors using a *double-diffusion* communication pattern at each iteration. It was chosen because of its symmetric characteristics and because of the large amount of computation run by each process at each step.

The second case uses the well-known Chord [14] peer-to-peer DHT (Distributed HashTable) protocol. It is designed to be scalable and capable of functioning even with nodes leaving and joining the system. The nodes form a logical ring and maintain local routing information called a *finger table*. Each node keeps information about $O(\log n)$ other nodes, where n is the total number of nodes in the system. Periodically, the nodes exchange messages to update their knowledge about the logical ring and their neighbors, and eventually converge to a consistent global vision. It has been showed that any lookup request issued by a node is resolved with only $O(\log n)$ messages generated in the network. Chord was chosen because it is representative of a large body of algorithms studied in the P2P community.

We performed an experiment similar to the one of [3]. We consider n nodes that all join the Chord ring at time $t = 0$. Every node performs a *stabilize* operation every 20 seconds, a *fix_fingers* operation every 120 seconds, and an arbitrary *lookup* request every 10 seconds. When the simulated time has reached 1000 seconds, each node stops its process and the simulation ends. To ensure that experiments are comparable between different settings we tuned the parameters to ensure that the amount of applicative messages exchanged during the simulation, and thus the workload onto the simulation kernel, remains comparable (with $100,000$ nodes, about 25 millions messages are exchanged in this scenario).

Each experiment was run on a machine of Grid'5000 [5] with two AMD 12-core CPUs at 1.7 GHz and 48 GB of RAM. SimGrid v3.6 beta (git version 8d32c7) were used for these experiments.

4.2 Optimizing the Simulations

4.2.1 Cost of Switching Contexts

As mentioned in Section 3.4, the scheduling of the user processes is in the critical path of the simulator, therefore a custom optimized implementation is provided to maximize the performance. To compare it with the other alternatives available in SimGrid (ucontext and threads) we now present the results of experiments using Chord. For $n = 100,000$ nodes, the simulation lasts 471 seconds using optimized contexts vs. 582 seconds using ucontexts, and system limitations make it impossible to run this experiment using pthreads. However, downscaled versions of the experiment show a 10x factor slowdown for pthreads over our optimized contexts. The gain obtained with our custom implementation over ucontexts is relatively constant around 20%, showing the clear benefit of avoiding unnecessary system calls on the simulation critical path.

4.2.2 Cost of Thread Synchronization

The overhead introduced by the synchronization primitives that control the thread-pool is a determinant factor of the potential achievable speedup of the parallel execution. In this experiment we try to measure this overhead ($K \cdot C_{thr}(T)$) by comparing the standard sequential simulation time to a parallel execution over a single thread.

For PMM, the cost of synchronization is negligible because the number of scheduling rounds is relatively low and so the number system calls involved. However, in the case of the Chord it exhibits an overhead of 16% (*i.e.* an increment of 76 s in a simulation of 471 s). This cost, which remains relatively high despite the high level of system optimization we did, clearly demonstrates that parallel execution is only beneficial in some specific conditions that we now try to better characterize.

4.3 Characterizing when $C_{par} < C_{seq}$

We now present a set of experiments that summarizes the scenarios where the parallelism can outperform the sequential execution. In cases like PMM where C_{usr} is large, the parallel execution trivially outperforms the sequential one. For PMM using 9 nodes with matrices of size 1500, the sequential simulation takes 31 s and the parallel with 4 threads takes 11 s, representing a speedup of more than 50%.

The case of Chord is more challenging as C_{usr} is very small, as typical of most peer-to-peer applications: processes exchange a lot of messages and perform few calculations.

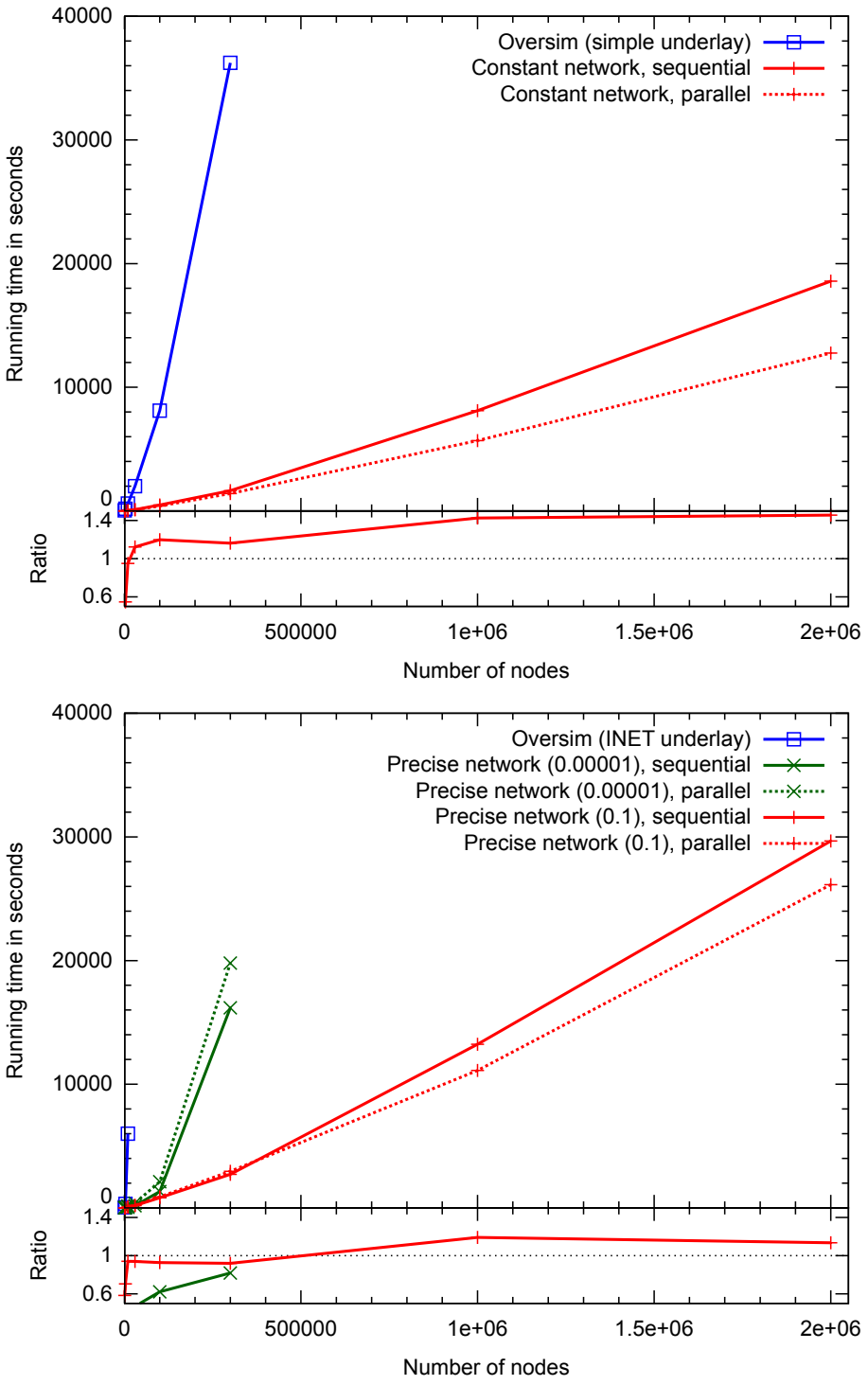

Figure 6: **Left:** Running times of the Chord simulation with a constant network model on SimGrid, compared to Oversim with a simple underlay. **Right:** Running times of the Chord simulation with a precise network model on SimGrid, compared to Oversim with the INET underlay. The bottom part of each graph shows the ratio between the parallel and the sequential modes (when the curve is above 1, the parallel mode is faster than the sequential mode).

In this case, we study the impact of the simulation precision over the potential parallelism in the simulation by measuring the average amount of user processes ready to run at each scheduling round ($|P_{t_i}|$) for several values of the precision ε. Results presented in Table 1 clearly show that small values of ε (corresponding to important precisions) lead to reduced scheduling rounds. This can be explained by the dispersion of actions across more timestamps. Better speedups are thus expected for big values of ε, *i.e.* for moderated simulation accuracy.

ε	10^{-5}	10^{-3}	10^{-1}	Constant network
$\lvert P_{t_i}\rvert$	10	44	251	7424

Table 1: Average scheduling round size in Chord as a function of ε, the simulation precision.

Figure 6 reports the obtained running times as a function of the node amount (from 1000 to 2 millions nodes). The left graph represents the results of a constant network model, which consists in adding a constant delay of 0.1 second for each communication. We were able to simulate 2 million nodes in 3h18 with the sequential mode, and in 2h24 in parallel mode. The ratio depicted below the graph shows that the parallel mode is preferable to the sequential mode even for small scenario sizes and provides a speedup of up to 40%.

The right graph shows the same experiment with the precise network model described in [16]. As explained before, the numerical accuracy used to represent the time has an impact on the performance of parallel execution. Using the default value ($\varepsilon = 10^{-5}$), we could only simulate up to 300k nodes in less than one night of computation. When the precision is reduced to $\varepsilon = 10^{-1}$, the events are less dispersed across timestamps, making the scheduling rounds bigger, and parallelizing the user processes becomes efficient. In these conditions, we were able to simulate 2 million nodes in 8h15 in sequential mode, and in 7h15 in parallel mode. The relative gain of the parallel mode is smaller than previously: it becomes beneficial only for 500,000 processes and more and the speedup remains under 20%.

A more disturbing result is the fact that the sequential simulation performance is also highly impacted by ε. This is unexpected since the amount of work remains the same in each case, and understanding the cause of this phenomenon remains to be done in future work.

The memory usage for simulating 2 million nodes was about 36 GB, that represent 18 kB per node, including 16 kB for the user stack.

4.4 Comparison to OverSim

Figure 6 also reports the simulation timing obtained with OverSim [3] on the same machine. On the left, OverSim was used with the simplest network underlay, comparable to our constant time model. We could not to simulate more than $300,000$ nodes in less than a night of computation with this settings.

On the right, we also plot the running times we obtained with the INET network model of OverSim. Relying on a packet-level simulation, it offers the best accuracy of the framework and is thus comparable to the precise model of SimGrid. In this setting, the simulation failed to reach $30,000$ nodes: the initialization phase was not finished after a night of computation.

Those results show that the new design of SimGrid reaches high scalability and performance, including with precise network models. In particular, SimGrid's worst

case (precise model, $\varepsilon = 10^{-5}$) outperforms OverSim's best case (constant model) while the offered simulation accuracy is not comparable.

5 Conclusion and Future Work

In this paper, we presented several improvements to SimGrid, which allows researchers to study distributed systems through simulation. We have shown that the old design of the framework was limited to the sequential execution of the user code, due to the simulation's shared state being disaggregated across the entire software stack. We proposed a new architecture that solves these issues following the design guidelines of operating systems. It abstracts all the modifications of the shared state using an emulation of the system call mechanism called *requests*. We presented a threading model mixing ucontexts and threads, and thus avoiding wasting resources in unwanted contention resolution, yet allowing the simulation of millions of processes. We implemented a custom synchronization scheme relying directly on the operating system primitives to reduce the cost of the threads synchronization. We analyzed the cost of sequential and parallel execution and gave an analytical criterion characterize situations in which a speedup can be expected from parallel execution. We validated this criterion experimentally alongside with the other design choices made in the newest version of the SimGrid framework. Finally, we showed that SimGrid is capable of simulating peer-to-peer systems to an unprecedented scale, one order of magnitude faster and with better precision than OverSim.

Overall, we think that this paper demonstrates the difficulty to get a parallel version of a P2P simulator faster that its sequential counterpart, provided that the sequential version is optimized enough. During the work leading to this paper, we faced several situations were the parallel implementation offered nearly linear speedups, but it always resulted from blatant performance mistakes in the sequential version. We hope that this return of experience and our work on the design of the framework will be applicable to other simulation toolkits as well.

As future work, we are considering to further parallelize the framework by executing parts of Simix and Surf in parallel too. We are also planning to study the performance of other parts of the SimGrid framework (such as support for churn) at very large scale.

References

[1] The Network Simulator (ns3). http://nsnam.org/.

[2] The Omnet++ Simulator. http://www.omnetpp.org/.

[3] I. Baumgart, B. Heep, and S. Krause. OverSim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium*, pages 79–84, Anchorage, USA, May 2007.

[4] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies. *International J. of High Performance Computing Applications*, 17(4), 2003.

[5] R. Bolze and Al. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

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

[7] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[8] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[9] B. Donassolo, H. Casanova, A. Legrand, and P. Velho. Fast and scalable simulation of volunteer computing systems using simgrid. In *Workshop on Large-Scale System and Application Performance (LSAP)*, 2010.

[10] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. PeerSim. `http://peersim.sourceforge.net`.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. Towards yet another peer-to-peer simulator. In *Proc. Fourth International Working Conference Performance Modelling and Evaluation of Heterogeneous Networks (HET-NETs)*, 2006.

[13] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing – HPDC'02*, page 352, Washington, DC, USA, 2002. IEEE Computer Society.

[14] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, Feb. 2003.

[15] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. IEEE, New York, NY, USA, 2004.

[16] P. Velho and A. Legrand. Accuracy study and improvement of network simulation in the simgrid framework. In *Proc. of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10, 2009.

[17] V. yee Vee and W. jing Hsu. Parallel discrete event simulation: A survey. Technical report, 1999.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399