Parallel Simulation of Peer-to-Peer Systems

Martin Quinson, Cristian Rosa, Christophe Thiéry
LORIA / Nancy Université

***Abstract*—Discrete Event Simulation (DES) is one of the major experimental methodologies in several scientific and engineering domains. Parallel Discrete Event Simulation (PDES) constitutes a very active research field for at least three decades, to surpass speed and size limitations. In the context of Peer-to-Peer (P2P) protocols, most studies rely on simulation. Surprisingly enough, none of the mainstream P2P discrete event simulators allows parallel simulation although the tool scalability is considered as the major quality metric by several authors.**

This paper revisits the classical PDES methods in the light of distributed system simulation and proposes a new parallelization design specifically suited to this context. The constraints posed on the simulator internals are presented, and an OS-inspired architecture is proposed. In addition, a new thread synchronization mechanism is introduced for efficiency despite the very fine grain parallelism inherent to the target scenarios. This new architecture was implemented into the general-purpose open-source simulation framework SimGrid. We show that the new design does not hinder the tool scalability. In fact, the sequential version of SimGrid remains orders of magnitude more scalable than state of the art simulators, while the parallel execution allows to save up to 33% of the execution time on Chord simulation.

***Index Terms*—Parallel Discrete Event Simulation, Peer-to-Peer.**

I. Introduction

Researchers from the Peer-to-Peer (P2P) community aim at constituting distributed systems comprising millions of processes collaborating to a common goal. Studies on real systems being near to impossible at this scale, most of the literature relies on simulation studies.

The used simulator is however often seen as a technical detail, and it is even common to see P2P researchers developing their own custom tool. According to the survey in [11], 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. This situation is problematic since most ad-hoc simulators lack a proper validation of their methodology and this hinders the reliability and the reproducibility of the results. Moreover, it poses serious complications when trying to compare these results. Finally, the performance of such tools certainly restricts the scale of the studies conducted.

Despite the clear importance of scalability in the community, all mainstream P2P simulators remain single threaded [4], [15]. This is surprising given the huge activity in Parallel Discrete Event Simulation (PDES) research community for over three decades (see for example [8] for a survey).

This observation constitutes the starting point of this paper. To address this challenge, we introduce a novel parallelization approach specifically suited to the simulation of distributed applications. Actual implementation of this parallelization poses extra constraints on the simulator internals that we propose to overcome with a new architecture, highly inspired from the Operating Systems concepts. In addition, we propose a specifically crafted inter-threads synchronization design to maximize the performance despite the very fine grain exposed at best by the simulation of typical P2P protocols.

As a proof of concept, we implemented this approach in SimGrid[1] [6], an open-source, generic distributed systems simulation framework providing very realistic and flexible simulation capabilities. SimGrid was conceived as a scientific instrument, thus the validity of its analytical models was thoughtfully studied [14], ensuring their realism. Experimental results show that the proposed parallelization schema do not hinder the tool efficiency. On the contrary, the sequential version of SimGrid proves to be orders of magnitude more scalable than state of the art simulators despite the complexity that was added to the simulator's internals to make the parallel execution possible. For example, sequential simulations of Chord [13] scenarios comprising 2 millions nodes last 5.5 hours on a single computer using SimGrid, whereas comparable simulators of the area can only simulate up to $300,000$ nodes in 10 hours. In addition, activating the parallel execution reduces further the execution time by 28%, down to 4 hours.

This article is organized as follows. Section II reviews the state of the art regarding P2P simulators, and revisits the classical PDES approaches in the light of P2P protocols simulation. Section III details the contributions of this article, namely how we enable parallel simulation of P2P protocols and how we ensure its efficiency. Section IV presents some experimental results, and Section V concludes this article by summarizing our findings and presenting some future work.

II. Context & State of the Art

From the vast amount of P2P simulators proposed in the recent years, a large majority were only intended to be used by their authors, sometimes for a unique study. Even when they are published, most of them reveal to be short lived [10], [4]. This situation poses a serious threat on researcresearch using these tools, as works may get jeopardized when their tool gets abandoned by its authors. As a result, several authors hope that a tool will emerge as a *de facto* standard to improve the community organization [15].

PeerSim [9] is probably the most used P2P simulator nowadays. A recent survey [4] concludes by "From these surveyed simulators PeerSim is best for p2p researchers due to very high scalability". Another strength is its simplicity:

[1]SimGrid is freely (LGPL) available from http://simgrid.gforge.inria.fr/

Simulation Workload	• Granularity, Communication Pattern • Events population, probability & delay • #simulation objects, #processors
Simulation Engine	• Parallel protocol, if any: – Conservative (lookahead, ...) – Optimistic (state save & restore, ...) • Event list mgnt, Timing model...
Execution Environment	• OS, Programming Language (C, Java...), Networking Interface (MPI, ...) • Hardware aspects (CPU, mem., net)

Figure 1: Performance Factors for PDES (after Fig. 2 of [2]).

users can easily adapt it to their needs. For sake of scalability, it implements a query-cycle mode where the processes are represented as simple state machines. At each time step, the simulator executes one event for each node of the system. This allows the simulator to reach the unprecedented scalability of 10^6 nodes on a single machine, but puts an extra burden to the user. Since the machine state formalism may reveal counter-intuitive for users, the authors of PeerSim introduced a Discrete Event Simulation mode in addition to the original query-cycle mode. It was reported to simulate thousands of nodes using the DES mode, although the lack of documentation was criticized in the literature [15]. Another drawback of PeerSim is the fact that the network is completely abstracted, preventing studies that would mandate some network realism [4].

OverSim [3] is another widely used P2P simulator. It was reported to simulate up to 100,000 nodes in an event-driven approach. Another recent survey [15] concludes by "OverSim may be the best for P2P researchers due to very high scalability". When the network needs to be precisely simulated, its scalable models can be replaced to use the OMNet++ [1] packet-level simulator.

PlanetSim [12] is one of the rare mainstream P2P simulators that permits parallel simulation. Its query-cycle approach makes its parallelization rather straightforward: at each simulation step, each process has to be processed separately. Since they only do local computations, handling them in parallel with several threads does not lead to any complication. The authors report a speedup of 1.3 on two processors.

dPeerSim [7] is an extension to PeerSim that allows distributed simulations of very large scenarios using classical PDES techniques. However, the overhead of distributing the simulation seems astonishingly high. Simulating the Chord protocol [13] in a scenario where 320,000 nodes issue a total of 320,000 requests last about 4h10 with 2 logical processes (LPs), and only 1h06 with 16 LPs. The speedup is interesting, but this is to be compared to the sequential simulation time, that the authors report to be 47 seconds. For comparison, this can be simulated in 5 seconds using SimGrid with a precise network model.

Interestingly enough, none of the P2P discrete-event simulators reviewed in the cited surveys [10], [4], [15] seem to support parallel simulation to exploit multi-core architectures. Given that scalability is one of the main goals of any P2P simulator, one could expect that three decades of intense research in parallel and distributed discrete-event simulation would have been leveraged to reach maximal scalability.

Simulation Workload	User Code
	Virtualization Layer
	Networking Models
Simulation Engine	
Execution Environment	

Figure 2: Layered View of classical Discrete-Event Simulations of Distributed Applications (such as P2P Protocols).

In our opinion, the relative failure of PDES in this context comes from the fact that the simulation of distributed systems (such as P2P protocols) is different from the simulations classically parallelized in the PDES literature. The current work aims at demonstrating that these specificities shift the optimization needs.

The classical performance factors of a PDES system are depicted in Figure 1, similarly to Figure 2 of [2]. Most works of the PDES literature focus on the simulation engine itself. Proposed improvements include better event list management or parallel protocols (either conservative or optimistic) to distribute this component over several computing elements (logical processes).

However, this trend does not match the experience that we gained with the SimGrid simulator over the years. According to our findings, most of the time is not spent in the simulation engine, but in the layers built on top of it. These layers are depicted in Figure 2. The event timings are given by some hardware models of the network and computing resources. These models can be really simplistic such as in classical P2P simulators, or rather complex in packet-level simulators. On top of it comes a virtualization layer, in charge of executing the user code in separate contexts, and converting user actions into requests that the hardware models change into events for the engine. In OverSim for example, the network models are clearly separated from the application under the name *Underlay* [3]. In SimGrid, the hardware models are grouped in a module called SURF, while the virtualization module is called SimIX (for Simulated POSIX) [6].

During the discrete-event simulation of a distributed system, two main phases occur alternatively: the simulation models are executed to compute the next occurring events, and the virtualized processes unblocked by these events are executed until they issue another blocking action (such as a simulated computation or communication). Equation 1 presents the distribution of time during such an execution, were SR is a simulation round, $model$ is the time to execute the hardware models, $engine$ is the time for the simulation engine to find the next occurring event, $virtu$ is the time spent to pass the control to the virtualized processes executing the user code, and use is the time to actually execute the user code.

$$\sum_{SR}(engine + model + virtu + use) \quad (1)$$

The timing resulting from the classical parallelization schema is presented in Equation 2. Grossly speaking, the time to execute each simulation round is reduced to the maximum of execution time on a logical process LP for this simulation round, plus the costs induced by the synchronization protocol, noted $proto$.

$$\sum_{SR}\left(\max_{LP}(engine + model + virtu + use) + proto\right) \quad (2)$$

To be beneficial, the protocol costs must be amortized by the gain of parallelization. According to Figure 1, this gain highly depends on the computation granularity and on the communication pattern (to devise a proper spatial distribution of user processes over the LPs reducing the inter-LPs communications). Unfortunately, in the context of P2P protocols, the computational granularity is notoriously small, and good spatial distributions are very hard to devise since most P2P protocols constitute application level small-worlds, where the diameter of the application-level interconnection topology is as low as possible. If such a distribution exists, it is highly application dependent, defeating any attempt to build a generic simulation engine that could be used for several applications. That is why $proto$ is expected to remain too high to be amortized by the classical parallelization schema.

Our proposition is instead to keep the simulation engine centralized and to execute the virtualization and user code in parallel[2]. This is somehow similar to the approach followed in PlanetSim and other query-cycle simulators, where the iteration loop over all processes is done in parallel. The resulting timing distribution is presented in Equation 3, where WT represents one of the worker threads in charge of running the user code in parallel and $sync$ is the time spent to synchronize the threads.

$$\sum_{SR}\left(engine + model + \max_{WT}(virtu + use) + sync\right) \quad (3)$$

III. Contributions

We now present an approach to implement the specific parallelism scheme proposed in the previous section. We first propose an alternative multi-threading architecture that enables the parallel execution of the user processes and the virtualization layer, while keeping the simulation engine sequential. We discuss the new constraints on the simulation's internals posed by the concurrency of the user code, we detail the new simulation main loop and how we optimized the critical parts of the parallelization code. Then, we introduce a new synchronization schema inspired from the usual worker pool but specifically fitted to our usage.

Our contributions are built from the observation that the services offered by a simulator of distributed systems are similar to those provided by an operating system (OS): processes, inter-process communication and synchronization primitives. We also show that the tuning of the interactions between the (real) OS and the simulator is crucial to the performance.

A. Parallel Simulation made Possible

The actual implementation of a simulator of distributed systems mandates complex data structures to represent the shared state of the system. These structures not only include the future event list of the simulation engine, but also data for hardware models and for the virtualization layer. Shared data is typically modified on each simulation round both by the simulation engine to determine the next occurring events, and by the user code to issue new future events in response to these events.

This poses no problem under sequential simulation as the mutual exclusion is trivially guaranteed. But enabling the parallel execution that we envision requires to prevent any possible concurrent modifications between working threads.

Shared data could be protected through fine-grained locking scattered across the entire software stack. This would be both extremely difficult to get right, and prohibitively expensive in terms of performance. In addition, even if these difficulties were solved to ensure the internal correction of the simulator, race conditions at the applicative level could still happen for event occurring at the exact same simulated time. Consider for example a simulation round comprising three processes A, B and C. A issues a *receive* request while B and C issue *send* requests. Ensuring that applicative scenarios remain reproducible mandates that whether A receives the message of B or the one of C is constant from one run to another. But if B and C are run concurrently, the order of their request is given by the ordering of their respective working threads. In other words, the simulated timings tie is solved using the real timings. This clearly makes the simulation non reproducible as real timings naturally change from one run to another.

Since the concurrent modifications between working threads executing the user code would be near to impossible to regulate efficiently through locking, they must be avoided altogether. The design of modern operating systems is very inspiring here: The user processes are completely isolated from the rest of the system in their virtual address space. Their only way to interact with the environment is to issue requests (*system calls*) to the kernel that then interact with the environment on their behalf. 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 processes and the kernel permits the independent and parallel execution of the processes, as any potential access to the shared state is mediated by the kernel, responsible of maintaining the coherence. Applying this design to distributed systems simulation enables the parallel execution of user code at each simulation round.

[2] The same concept could be used to distribute the user processes over several machines in addition to several threads, in order to address scenarios too large to fit in memory, but this is outside the scope of this paper.

Algorithm 1 Parallel Main Loop.

1: $t \leftarrow 0$ # *t: simulated time*
2: $P_t \leftarrow P$ # *P_t: ready processes*
3: **while** $P_t \neq \emptyset$ **do** # *some processes can be run*
4: parallel_schedule(P_t) # *resume processes*
5: handle_requests() # *answer their requests*
6: $(t, events) \leftarrow$ models_solve() # *find next events*
7: $P_t \leftarrow$ processes_to_wake($events$)
8: **end while**

As a proof of concept, we implemented a new virtualization layer in SimGrid that emulates a system call interface called *requests*. In the rest of this article, we will use the term *request* to designate a call issued by a user process to the simulation core. The term *system call* will now refer to a real system call to the OS.

Our proposition completely separates the execution of the user code contexts from the simulation core, ensuring that the shared state can only be accessed from the core execution context. When a process performs an interaction with the platform (such as a computing task execution or message exchange), it issues the corresponding request through the interface. The request and its arguments are stored in a private memory location, and the process is then blocked context until the answer is ready. When all user processes are blocked this way, the control is passed back to the core context, that handles the requests in an arbitrary but deterministic order based on process IDs of issuers. To the best of our knowledge, it is the first time that this classical OS design is applied to distributed system simulation, despite its simplicity and efficiency. As the simulation shared state only gets modified through request handlers that execute sequentially in the core context, there is no need for the fine-grained locking scheme to enable the parallel execution of the user code. Algorithm 1 presents the resulting main simulation loop. The sequential execution of the simulated processes is replaced by a parallel schedule on line 4, followed by a sequential handling of all issued requests.

B. Parallel Simulation made Efficient

The very fine grain computation that P2P protocols typically exhibit results in a huge amount of very short simulation rounds. In these conditions, ensuring that the parallel execution runs faster than its sequential counterpart mandates a very efficient handling of these rounds. This section details where synchronization is involved in our design, and the solutions that we propose to make these synchronization points as efficient as possible.

1) User Code Virtualization: As explained earlier, it is expected that the simulated protocol is expressed directly using a classical programming language, and conceptually executed in a way that is similar to multi-threading or co-routines. Each user process runs normally until it starts a computation or a communication in the simulated world. At this point, it is blocked in the real world until the simulation reaches the point where this action ends.

Several options are possible to implement this mechanism. Full featured threads can naturally be used, but their scalability would be too limited in our context. First, the amount of existing threads that can co-exist in the system process is limited; launching millions of threads on a given machine seems near to impossible. Then, the features of threads are too rich in our context. Thread containers are still provided in SimGrid for portability, but when available, simpler and thus more efficient solutions are activated.

The first such solution relies on the `ucontexts` that are part of the POSIX standard. The execution flow is transferred from a context to another using the `swapcontext` function, which saves the current stack and restores another one. At first glance, this function seems to run entirely in user space without any intervention from the OS kernel, but this is not true. Actually, POSIX allows to specify a different signal mask for each `ucontext`, forcing a system call during the swap to exchange the masks. Since SimGrid does not use system signals at all, we provide an alternative implementation of contexts that is free of any system calls. Because the swap routine modifies specific registers, it is programmed in assembly language and is architecture dependent. For the moment, this option is available for x86 and x86_64 hardware, and other architectures fall back to the standard `ucontexts`.

However, these lightweight contexts (either `ucontexts` or the assembly ones) are inherently sequential and don't provide any multi-threading support. They were originally conceived as an evolution of the `setjmp` and `longjmp` functions, not to handle multiple cores or processors. In SimGrid, we therefore mix the approaches to leverage both the advantages of contexts and the ones of threads. The user code is virtualized into lightweight contexts to reduce the cost of context switches. In a given simulated round, each working thread runs a subset of the user processes. The workload is then split among N working threads, where N is typically the amount of cores in the host machine.

2) Threads Pool Synchronization: We redesigned the thread pool synchronization to leverage our very specific workload. Indeed, we never add work to the pool while its workers are active. Instead, the simulation core passes a batch of processes to be handled concurrently and then waits for the complete handling of this batch. As a result, the thread control scheme that we need is composed of two asymmetric barriers: one where the calling thread unblocks all worker threads, and one where the last terminating worker unblocks the calling thread.

This can easily be implemented using the primitives provided by the POSIX standard. For the first barrier, a condition variable is broadcasted by the calling thread. For the second one, an integer is used to count the amount of remaining working threads and detect when the current thread is the last terminating one. Then, another condition variable is signaled to unblock the calling thread. Of course, these operations have to be protected by a mutex according to the POSIX semantic. While portable across the operating systems speci-

Algorithm 2 **signal(parmap pm)**

1: $myround \leftarrow pm.done\_round$
2: $pm.amount\_blocked \leftarrow 0$
3: $pm.curr\_round \leftarrow pm.curr\_round + 1$
4: **futex_wake**$(\&pm.curr\_round, pm.total\_amount)$
5: **futex_wait**$(\&pm.done\_round, myround)$

Algorithm 3 **wait(parmap pm)**

1: $myround \leftarrow pm.curr\_round$
2: **atomic**$(myrank \leftarrow {+}{+}pm.amount\_blocked)$
3: **if** $myrank = pm.total\_amount$ **then**
4: $\quad pm.done\_round \leftarrow pm.done\_round + 1$
5: $\quad$ **futex_wake**$(\&pm.done\_round, 1)$
6: **end if**
7: **futex_wait**$(\&pm.curr\_round, myround)$

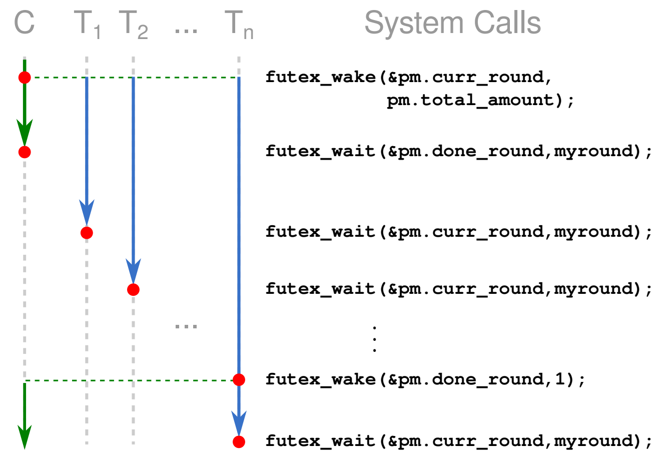

Figure 3: An execution of the parmap abstraction.

ficities (where Linux builds upon *futexes* while BSD uses *spin locks*), the resulting code poses serious drawbacks from the performance point of view. A first improvement is to use an atomically incremented value instead of a variable protected by a mutex. This can be done using the *Fetch-and-add* construct provided directly by most modern hardware. Thanks to the compiler support, this remains reasonably portable.

The protection of the condition variable through a mutex is mandated by the standard, but introduces an unnecessary overhead due to the system calls involved in solving the access contention. Instead, we introduce specialized synchronization abstraction named *parmap* (parallel map). Conceptually, a parmap is synchronized by a combination of a condition variable and a barrier. It has has two primitives: `signal` (where the caller unblocks all working threads to get them process the data in parallel) and `wait` (where the caller waits for the working threads' completion). Upon completion, the working threads are blocked again, waiting for the next `signal`.

The most efficient implementation of the parmap is built directly on top of the futexes provided by Linux, and atomic operations. Futexes ("Fast Userspace muTexe") are the building blocks for every synchronization mechanism under Linux. Their semantic is similar to semaphores, where a counter is incremented and decremented atomically in user space only, and processes can wait for the value to become positive at the price of context switches to the kernel mode. On top of it, the operations that we use take two arguments: a *reference* to an integer variable and an integer value. $futex\_wait(\&a, b)$ atomically verifies that $*a == b$ and if true it gets blocked on $\&a$. If $*a \neq b$, the call fails and an error code is returned. $futex\_wake(\&a, b)$ wakes at most b threads blocked at $\&a$.

Algorithms 2 and 3 show the pseudo-code of the `signal` and `wait` functions while Figure 3 depicts a typical execution cycle with a calling thread C and $T_1, \ldots, T_n$ worker threads. Both functions accept a parmap pm as the only argument. A parmap has four integer fields associated: $total\_amount$, $amount\_blocked$, $curr\_round$ and $done\_round$. The first two are respectively a constant field that indicates the total number of working threads in the parmap, and the count workers that called `wait` on the parmap. $curr\_round$ and $done\_round$ are used to ensure that no signal gets lost in the interactions between the caller thread and the worker threads.

After the parmap initialization, all worker threads are blocked at line 7 of `wait` in $futex\_wait$. When the caller thread calls `signal` it unblocks the worker threads (at line 4), and then blocks itself in the next line $futex\_wait(\&pm.done\_round, myround)$ (the two first system calls in Figure 3). When unblocked this way, the workers perform their computations and call `wait` upon completion. This function atomically increments the counter of worker threads waiting for a new signal and block by calling $futex\_wait(\&pm.round, myround)$ again. In addition, the last worker thread entering the barrier wakes the caller (at line 5) before blocking.

The local variable $myround$ ensures that every `signal` issued gets received. They could be missed if the caller thread issues a new `signal` right after being awaken by the last terminating worker. This could lead to situations where the caller thread tries to awake the workers before that the last worker goes to sleep for the previous round. This would result in a deadlock since that worker would never receive the missed wait signal. Since every call to `signal` increases the value of $round$, the comparison $pm.round == myround$ would fail in the call to $futex\_wait(\&pm.round, myround)$ by the delayed worker (because it $myround$ variable would still store the old value of $round$ saved at line 1 of `wait`). The delayed worker would thus detect the situation, and simply proceed its execution: the unmatched call to wait is simply canceled.

Thanks to this design, the synchronization is achieved in only $N+3$ system calls (with N being the number of worker threads). For each scheduling round, there is one system call to unblock all the worker threads, one to block the signaler, then one for each worker when finishing the computations and waiting again, and finally one to resume the signaler. For the moment this functionality is Linux dependent because it relies on futexes, but fallback implementations using POSIX synchronization primitives are provided for portability.

3) Load Balancing: The new design introduced in §III-A ensures that no data from the hardware models or simulation engine is shared between the working threads of the parmap. But the actual work to be distributed, that is the list of user processes to schedule, remains shared.

A simplistic approach is to protect the list of tasks with a mutex, but it reveals very inefficient. To fetch a task, every worker has to acquire the lock, remove the element from the list, and release the lock. Given the tasks are typically extremely fine grained, the induced contention on this mutex would be so high that this list management would constitute a severe bottleneck in the whole simulation.

Another option is to statically assign tasks to the workers beforehand. The task list is split into several sub-lists, one for each worker. This removes the last shared data since each worker only accesses its own private sub-list, but at the price of potential load imbalances. Indeed, the execution time of each user process can vary significantly, depending on what each process does in reaction to each event received. According to our informal preliminary experiments, the resulting imbalance can be rather high on typical P2P scenarios, leading to important time waste while some working threads wait for the remaining ones to finish.

Our approach strives to both minimize the idling threads and avoid any additional synchronization mechanism requiring system calls. Instead, we rely on hardware atomic operations: the tasks are stored in an array that is shared and directly accessed by the working threads. The next task to be processed is indicated by an integer counter, that is atomically incremented by each working thread when fetching some more work to do. This approach is both *wait-free* and very efficiently balanced between threads. Idling threads can only exist when no more work remains to be processed, so it will only suffer of imbalance if the last task of the list is much larger to process than the other ones. In that case, the balance could theoretically be improved by starting the last task of the list as soon as the parmap begins. But since there is no way to predict the length of each task before executing them, our approach remains as efficient as possible in practice.

IV. Experiments

A. Methodology and Experimental Settings

This section presents experimental evidences of our approach's efficiency. First, we present several microbenchmarks characterizing the performance loss in sequential simulation due to the extra complexity mandated by the introduction of parallel execution. This loss is then characterized at macroscopic scope through the comparison of the sequential SimGrid and several tools of the literature on Chord [13] simulations. Finally, we characterize the gain of parallel executions.

Chord was chosen because it is representative of a large body of algorithms studied in the P2P community, and because it is already implemented in all P2P simulators studied. Using an implementation of the simulator's authors limits the risk of performance error in our experimental setup.

We ran all experiments on one machine of the Parapluie cluster in Grid'5000 [5], with 48 GB of RAM and two AMD Opteron 6164 HE at 1.7 GHz (12 cores per CPU) and under Linux. The versions used for the main software packages involved were: SimGrid v3.7-beta (git revision 918d6192); OverSim v20101103; OMNeT++ v4.1; PeerSim v1.0.5; Java with hotspot JVM v1.6.0-26; gcc v4.4.5. All experiments were interrupted after at most 12 hours of computation. We were unable to test dPeerSim: it is only available upon request, but over a bogus email address.

For SimGrid and OverSim, the used experimental scenario is the one proposed in [3]: n nodes join the Chord ring at time $t = 0$. Once joined, each node performs a *stabilize* operation every 20 seconds, a *fix_fingers* operation every 120 seconds, and an arbitrary *lookup* request every 10 seconds. The simulation ends at $t = 1000$ seconds. To ensure that experiments are comparable between different settings, we tuned the parameters to make sure 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). What we call a message here is a communication between two processes (which may or may not succeed due to timeouts).

For PeerSim, the implementation provided on the project web page does not follow this experimental scenario: there is no *stabilize* nor *fix_fingers* operations, and only one *lookup* is generated every 120 seconds (instead of one per node).

The whole experimental settings and data is available at http://simgrid-publis.gforge.inria.fr

B. Microbenchmarks of Parallelization Costs

The first microbenchmark compares the advantages of each implementation of the virtualization layer (see §III-B1). We ran several Chord scenarios with each container implementation. The `pthread` containers prove to be about ten times slower than our custom contexts and hit a scalability limit by about 32,000 nodes since there is a hard limit on the amount of semaphores that can be created in the system. Such limit does not exist for the other implementations, that are only limited by the available RAM. Compared to `ucontext`, our implementation presents a relatively constant gain around 20%, showing the clear benefit of avoiding any unnecessary complexity such as system calls on the simulation critical path.

The second microbenchmark assesses the efficiency of the synchronization primitives that control the thread pool (see §III-B2). For that, we compare the standard sequential simulation time to a parallel execution over a single thread. This cost is naturally highly dependent on the user code granularity: a coarse grain user code would hide these synchronization costs. In the case of Chord however, we measure a performance drop of about 15%. This remains relatively high despite our careful optimization, clearly showing the difficulties of efficiently simulating P2P protocols in parallel.

C. Sequential SimGrid Scalability in the State of the Art

Figure 4 reports the simulation timing of the Chord scenario as a function of the node amount. It compares the results of the

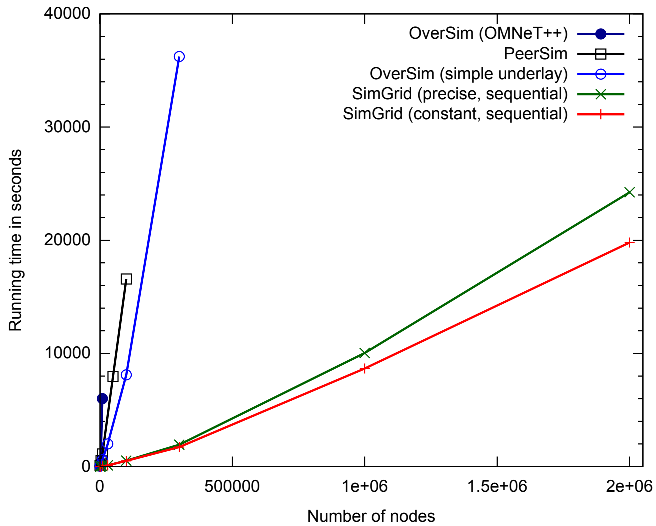

Figure 4: Running times of the Chord simulation with constant and precise network models on SimGrid, compared to OverSim with a simple underlay, OverSim with OMNeT++ and PeerSim.

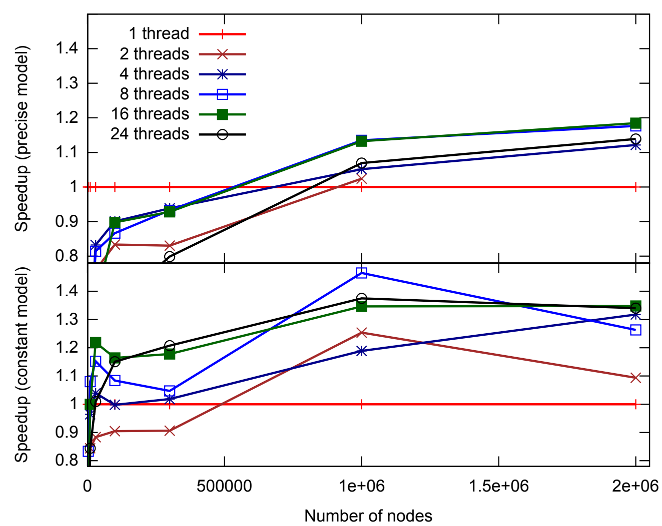

Figure 5: Parallel speedups observed for the precise (above) and constant (below) models of SimGrid, as a function of both the system size and the amount of worker threads.

main P2P simulators of the literature: OverSim when using a simple and scalable network underlay, OverSim using its complex but precise network overly based on OMNet++, PeerSim, SimGrid using the precise network model (that accounts for contention, TCP congestion avoidance mechanism and cross traffic – [14]), and SimGrid using the simple constant network (that applies a constant delay for every message exchange).

The largest scenario that we managed to run in less than 12 hours using OMNeT++ was $10,000$ nodes, in 1h40. With PeerSim, we managed to run $100,000$ nodes in 4h36 (but with a much lighter workload, as noted previously). With the simple underlay of OverSim, we managed to run $300,000$ nodes in 10h. With precise model of SimGrid, we ran $2,000,000$ nodes in 6h43 while the simpler model of SimGrid ran the same experiment in 5h30. Simulating $300,000$ nodes with the precise model took 32mn. The memory usage for 2 million nodes in SimGrid was about 36 GiB, that represent 18kiB per node, including 16kiB for the stack devoted to the user code.

Those results show that the extra complexity added to SimGrid to enable parallel execution does not hinder the sequential scalability, as it is the case with dPeerSim (see §II). On the contrary, SimGrid remains order of magnitude more scalable that the best known P2P simulators. It is 15 times faster than OverSim, and simulates scenarios that are ten times larger. This trend remains when comparing SimGrid's precise model to the simplest models of other simulators, while the offered simulation accuracy is not comparable.

D. Characterizing the Gain of Parallelism

We now present a set of experiments assessing the performance gain of the parallelism on Chord simulation. As expressed in §II, such simulations are very challenging to run efficiently in parallel because of their very fine grain: processes exchange a lot of messages and perform few calculations between messages. This evaluation thus presents the worst case scenario for our work, that could trivially be used on simulations presenting a coarser grain.

Figure 5 reports the obtained speedups when simulating the previous scenario in parallel. The speedup is the ratio of the parallel and sequential timings: $S = \frac{t\_seq}{t\_par}$. A higher speedup denotes an efficient parallelism while a ratio below 1 denotes that the synchronization costs are not amortized, making the parallel execution slower than its sequential counterpart.

The first result is that for small instances, parallelism actually hinders the performance. The constant model benefits from parallelism only after about $30,000$ nodes while the precise model has to reach about $500,000$ nodes for that. This can be explained by the differences in the code portions that do not run in parallel: it is much higher with the precise model since we compute the hardware models sequentially. The observed differences are thus due to the Amhdal's law.

Another result is that the speedups only increase up to a certain point when increasing the amount of working threads. That is, the inherent parallelism of these simulations is limited, and this limit can be reached on conventional machines. The optimal amount of threads varies from one setting to another, denoting similar variations in the inherent parallelism. For the precise model, the maximal speedup for 2 million nodes is obtained with 16 threads. The execution time is reduced from 6h43 in sequential to 5h41mn with 16 threads. But it remains more efficient to use only 8 threads instead of 16, since the execution time is only 2 minutes longer (less than one 1%) while using only half of the resources. Reducing further to 4 threads leads to a notable performance drop, as the execution lasts 6h. Conversely, increasing the amount of threads beyond 16 threads to 24 leads to a speedup decrease, at 5h55mn. Although less polished, the results for the constant models show similar trends, with an optimal amount of threads of approximately 16 workers. This difference in the optimal

Model	*4 threads*	*8 threads*	*16 threads*	*24 threads*
Precise	0.28	0.15	0.07	0.05
Constant	0.33	0.16	0.08	0.06

Table I: Parallel efficiencies achieved for 2 million nodes.

between the models is also due to the Amhdal's law.

Table I presents the parallel efficiency achieved in different settings for 2 million nodes. The parallel efficiency is $\frac{S}{p}$ where S is the speedup and p the amount of cores. Our results may not seem impressive under this metric, but to the best of our knowledge, this is the first time that a parallel simulation of Chord runs faster than the best known sequential implementation. In addition, our results remain interesting despite their parallel efficiency because the parallelism always reduces the execution time of large scenarios. The relative gain of parallelism seems even strictly increasing with the system size, which is interesting as the time to compute very large scenarios becomes limiting at some point. For example, no experiment presented here failed because of memory limits, but some were interrupted after 12 hours. This delay could arguably be increased, but it remains that given the amount of memory available on modern hardware, the computation time is the main limiting parameter to the experiments' scale. Leveraging multiple cores to reduce further the timings of the best known implementation is thus interesting.

V. Conclusion and Future Work

In this paper, we present a new parallelization schema for the Parallel Discrete Event Simulation of P2P protocols. The classical approach proves difficult to apply to these applications mainly because their very fine grain of computation makes it difficult to amortize the synchronization costs. In addition, most P2P protocols present a very dense applicative topology that complicates the simulation spatial separation, resulting in inordinate amounts of inter-LP communications. In addition, the rollback operation of optimistic protocols is challenging since the user code runs as classical code contained in system-level co-routines. This mandates difficult and slow system-wide checkpoints and rollbacks.

Instead, we propose to parallelize the execution of the user code while keeping the simulation engine sequential. This is enabled by applying classical concepts of OS design to this new context: every interaction between the user processes and the simulated environment is mediated by a specific layer that acts as the OS kernel. The performance is ensured by a simplistic and thus efficient container to virtualize the user code in the simulator, and by a specifically tailored parmap implementation that minimizes the system calls while evenly distributing the work among the worker threads.

This new architecture was implemented into the open-source general-purpose SimGrid simulation framework. Experimental results show that the new design does not hinder the tool scalability. In fact, the sequential version of SimGrid remains orders of magnitude more scalable than state of the art simulators. A simulation of the Chord protocol encompassing 2,000,000 nodes lasts 6h43 in sequential mode using the precise models of SimGrid while the previously best known simulator, OverSim, can only simulate 300,000 nodes in 10h. When activating the parallelism, SimGrid manages to precisely simulate 2,000,000 nodes in 4h with 16 threads.

Overall, 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 presented work, 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 think that this work can be useful to understand and improve the performance in other simulation toolkits as well.

As future work, we will enable automatic tuning of the working thread amount to reduce the synchronization costs, and test our approach on other P2P protocols.

Acknowledgment

The authors would like to thank their colleagues of the SimGrid project for their support and feedback on this work: A. Legrand, A. Giersch, P.-N. Clauss and all others.

References

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

[2] V. Balakrishnan, R. Radhakrishnan, D. M. Rao, N. Abu-Ghazaleh, and P. Wilsey. A performance and scalability analysis framework for parallel discrete event simulators. *Simulation Practice and Theory*, 8(8), 2001.

[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] R. Bhardwaj, V. Dixit, and A. K. Upadhyay. An Overview on Tools for Peer to Peer Network Simulation. *International Journal of Computer Applications*, 1(1):70–76, Feb. 2010.

[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] 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*, 2008.

[7] T. T. A. Dinh, M. Lees, G. Theodoropoulos, and R. Minson. Large scale distributed simulation of p2p networks. In *2nd International Workshop on Modeling, Simulation, and Optimization of Peer-to-peer Environments (MSOP2P 2008), in conjunction with PDP*, 2008.

[8] J. Liu. *Wiley Encyclopedia of Operations Research and Management Science*, chapter Parallel discrete-event simulation. 2009.

[9] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, Sept. 2009.

[10] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A Survey of Peer-to-Peer Network Simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium (PGNet), Liverpool, UK*, 2006.

[11] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. Towards yet another peer-to-peer simulator. In *Proc. of 4th Intl Conf. on Performance Modelling and Evaluation of Heterogeneous Networks*, 2006.

[12] J. Pujol-Ahulló, P. García-López, M. Sànchez-Artigas, and M. Arrufat-Arias. An extensible simulation tool for overlay networks and services. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2072–2076, New York, NY, USA, 2009. ACM.

[13] 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. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[14] 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.

[15] H. XU, S.-p. WANG, R.-c. WANG, and P. TAN. A survey of peer-to-peer simulators and simulation technology. *JCIT: Journal of Convergence Information Technology*, 6(5):260–272, May 2011.