UNIVERSITÉ DE LORRAINE

Département de formation doctorale

École doctorale IAEM Lorraine

UFR STMIA

Méthodologies d'expérimentation pour l'informatique distribuée à large échelle

Mémoire

présenté et soutenu publiquement le 8 mars 2013

pour l'obtention d'une

Habilitation à Diriger les Recherches
de l'Université de Lorraine
(spécialité Informatique)

par

Martin Quinson

Composition du jury:

Président:	Thierry Priol, Inria Rennes – Bretagne Atlantique.
Rapporteurs:	Jean-François Méhaut, Université de Grenoble. Pierre Sens, Université Paris 6. Gabriel Wainer, Carlton University, Ottawa, Canada.
Examinateurs:	Isabelle Chrisment, Université de Lorraine. Jens Gustedt, Inria Nancy – Grand Est.

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria

Table of Contents

Chapter 1

Introduction

Learning and doing is the true spirit of free software;
Learning without doing gets you academic sterility;
Doing without learning is all too often the way things are done in proprietary software. — *Raph Levien*

THIS document presents the research activities that I conducted after my PhD thesis in December 2003. These activities took place at the University of California at Santa Barbara (UCSD) where I hold a temporary post-doctorate position in 2004, at the computer science laboratory of Grenoble (LIG), where I hold another temporary post-doctorate position (ATER) the same year, and in the LORIA laboratory where I hold a permanent associate professor position since February 2005.

This short chapter provides a general introduction to this work, including the research motivations, the problem statement and the proposed methodology. The overall organization of this document is also detailed.

1.1 Scientific Context

Recent and foreseen technical evolutions allow to build information systems of unprecedented dimensions. The potential power of the resulting distributed systems stirs up most domains of informatics. Simulations made possible by *High Performance Computing* (HPC) or *Grid Computing* systems are now considered as a new pillar of science, alongside with Theory and Experimentation. *Cloud computing* and the associated virtualization technologies allow to completely externalize the computing infrastructure of any company to large dedicated data centers, which physical location is transparent to the end users. Peer-to-peer technologies (P2P) federate the computational and storage resources located at the network edge, directly in users facilities. As demonstrated by the Hadopi law in France, the Pirate Bay trial in Sweden, the threat posed by P2P-controlled BotNets or the role of the Tor protocol in the Arab revolutions, the societal impact of the peer-to-peer systems matches their extraordinary dimensions.

While large-scale production platforms are nowadays used routinely in all these various domains, evaluating the correction and performance of computer systems of such scale raises severe methodological challenges. Several approaches are classically used: **direct execution** of real applications onto real platforms (comparable to the field experiments that are common in other sciences) seems an obvious approach for obtaining sound experimental performance results. It is however not always possible as it requires to build the complete system beforehand. Even when possible, these experiments provide only a limited experimental control, hindering the exploration of *what-if* scenarios, or even the experimental reproducibility. This lake of control makes this approach hardly applicable to correction, as no exhaustive answer can be provided this way. Virtualization techniques can be used to run **emulation** tests, where real applications are run on virtual machines (similarly to *in vitro* experiments elsewhere). The experimental control is greatly increased, but the process is very technically demanding, and thus reveals even more time and labor consuming.

Simulation is an appealing alternative to study such systems. It consists in the prediction of the system's evolution through numerical and algorithmic models. It may not be sufficient in some cases to capture the whole complexity of the phenomena, but *in silico* studies have already proven their values to most scientific and engineering disciplines. Indeed, simulation allows to capture some important trends in an easy and convenient way, while ensuring the controllability and reproducibility of experiments. This appearing simplicity should not hide the methodological challenges raised by the realization of a simulation framework. The realism of the used models should be carefully ensured, while the fast simulation of large systems mandates careful optimization. Simulation can also be used to assess the correction of the systems, for example using a model checking approach such as the dynamic formal verification.

These problems, and other raised by the study of large-scale distributed applications, constitute the general context in which I conducted most of my research in the last decade.

1.2 Problem Statement and Methodology

My work is mainly methodological. Most of my contributions aim at improving the general method to study a given distributed algorithm, application or system. I also strive to compare the existing methodological approaches by applying them to my own work.

I strive to find practical answers to the multiple problems that arise in such studies. Most tools in this domain unfortunately remain prototypical, appealing and potentially useful but hardly usable outside of the authors' team or constituting only a partial answer to the difficulties. This unfortunate propensity is probably reinforced by the exaggerated use of the amount of publications as an evaluation metric for the quality of science and scientists. This may prompt a rush into a multiplication of superficial analyses. I was given the opportunity (and the tenacity) needed to conduct deep studies despite this pressure. While doing so, I had no qualms about doing the engineering work needed. I aspire to pragmatic tools, that are usable by a large community, without requiring the users to understand all the subtleties at stake.

The SimGrid simulator is certainly the best example of this orientation, as I integrated most of my work to this framework over the years. I am the main software architect of this project since the end of my PhD, and I was given the opportunity to lead two large scientific project funded by the ANR about this tool, that thus constitutes the practical context of my research since a decade. These efforts, and the work of my colleagues, secured a large community of users around SimGrid, mostly composed of scientists working on large scale distributed systems. SimGrid is now one of the two or three most used tools in this context. Such a large community is a big responsibility, but it also constitute an incredible opportunity to assess in practice that the scientific contributions that I present in this document constitute effective answers to the considered problems.

Through these technical realizations and beyond them, I have the constant concern of improving the experimental habits of my community by studying the experimental methodologies per see and easing their practical applications, adapted to each context.

1.3 Structure of the Document

This document highlights the global coherence of my work. In spite of appearances, I made every endeavor to keep this text concise. I think that the technical details are best demonstrated through the source code, that is always freely available. Instead, I only include the technical elements that are needed to understand the overall contribution. I also hope to point out my continuous trend toward pragmatic answers to the challenges.

Overall, this document opens with the broad generalities constituting the context of my work in §2. It then focuses on my work that is specific to the simulation kernel in §3. §4 widens back horizons by presenting my modeling efforts around the simulation kernel. §5 further enlarges the scope by considering some perspectives on how the simulation could become part of a coherent ecosystem of experimental methodologies.

More specifically, Chapter 2 aims at putting my work back in its context. §2.1 and §2.2 introduce the distributed systems that constitute the context of my work such as Grids, Clouds or HPC and P2P systems. §2.3 then contrasts the methodologies allowing to study the performance of these systems while §2.4 categorizes some actual tools doing so. §2.5 presents similarly some ways to assess the correction of these systems. §2.6 concludes this chapter by justifying my methodological orientations under an epistemological light.

Chapter 3 is focused on the simulation kernel. §3.1 introduces the SimGrid framework that constitutes the technical context of my research. Two works on the framework's performance are then presented: §3.2 improves the simulation speed through parallelization while §3.3 introduces a scalable network representation toward larger simulations. Before a conclusion, §3.4 presents how a model-checker were added to SimGrid for the formal evaluation of correction properties about the studied systems.

Chapter 4 presents several modeling efforts to constitute a complete framework around the simulation kernel. §4.1 discusses how the real applications can be used as models; §4.2 presents an effort to automatically map real platform, stressing the methodology and workbench that we developed to assess such solutions; §4.3 shows the methodology used to improve the accuracy of network model one step further.

Finally, Chapter 5 concludes this document with further research directions.

Chapter 2

State of the Art

Manuscripts containing innumerable references are more likely a sign of insecurity than a sign of scholarship.
– William C. Roberts

THIS CHAPTER PROVIDES A COHERENT VISION of my research domain. This comprehensive survey is essential to the comprehension of the following chapters, that describe in more technical detail specific points of my work.

Studying distributed applications reveals extremely challenging. Two major elements are to be examined: first, the correction of the system must be ensured so that no deadlock or other fault occur in operation. Then, the performance (throughput, response time, etc.) is to be assessed and optimized.

This chapter begins in §2.1 by an introduction to the modern distributed systems such as HPC, Grids, P2P and Cloud systems. These systems constitute the natural context of my work, although my own research is more specifically about the methodologies that can be leveraged to study these systems. §2.2 presents the differing ways to specify the distributed system, depending on the intended study. §2.3 contrasts the possible approaches to study the performance of distributed applications. This section provides some additional insight on the simulation, justifying its prevalent use in this context. §2.4 categorizes and presents the abundant literature presenting simulation frameworks dedicated to distributed applications. §2.5 contrasts the methodologies that can be leveraged to study the correction of distributed systems, and presents an overview of the existing tools. Finally, §2.6 puts an epistemological light on my work in order to answer some methodological criticisms that were addressed at my work.

This coherent presentation of my research domain is the result of fruitful collaborations and interactions with numerous colleagues, among which my team members and the partners of the ANR projects *Ultra Scalable Simulation with SimGrid*[1] and *Simulation Of Next Generation's Systems*[2] that I lead. It also builds upon the elements established during the doctoral work of Cristian Rosa, that I had the chance to co-advise.

[1]USS-SimGrid (ANR project 08 SEGI 022): http://uss-simgrid.gforge.inria.fr/
[2]SONGS (ANR project 11 INFRA 13): http://infra-songs.gforge.inria.fr/

2.1 Distributed Systems Taxonomy

HPC and Grids: Computational Science. Science In Silico has become the third pillar of science through the *simulation* of the phenomena in study. These simulations now constitute a crucial tool in disciplines such as particle physics, cosmology, biology or material engineering. Initially, large supercomputers were developed specifically to that extend, but it revealed more economically efficient to interconnect off-the-shelf processors through fast ad-hoc networks. This has contributed to achieve high performance in commodity components for both home PC market and for computational science.

Since the advent of microprocessors in the 70s, the computer performance had doubled every 18 months. In the early times, this was achieved by reducing the transistors size. In the 90s, physical limits of wires was reached, having only a few dozens atoms wide. To continue with the exponential performance increase, constructors started raising the clock frequency of the processors. In the last decade, power consumption and heat dissipation issues made it impossible to further increase the frequency. Indeed, the computational power of a computer increases nearly sub-linearly with clock frequency while the energy consumption increases more than quadratically. To push the performance further despite these difficulties, processor constructors have increased the amount of computing units (or cores) per processor. Modern **High Performance Computing** (HPC) systems comprise thousands of nodes, each of them holding several multi-core processors. For example, one of the world fastest computers, the Jaguar Cray XT5 system [jag] at Oak Ridge National Laboratory (USA), contains 18,688 dual-processor compute nodes with six-core each, for a total of 224,256 cores. Recent evolutions amongst the world's fastest machines [top] confirm the trend of massive hardware parallelism. Researchers envision systems with billions of cores (called ExaScale systems) for as early as the coming decade, which will tackle through simulation major issues such as the characterization of the abrupt climate changes, understanding the interactions of dark matter and dark energy or improving the safety and economics of nuclear fission [exa]. The **classical research questions** in this domain are naturally led by performance, and the traditional quality metric is the percentage of the peak performance achieved. Recently, the extreme scale of the targeted systems raised important feasibility issues. The typical mean time to failure (MTTF) of each hardware component and their amount induces for example that the supercomputers can no longer be considered robust nor reliable, leading to major difficulties on the software side, which are quite new in HPC.

Research on **Grid Systems** also relates to the applications now used by scientists in their day to day work. However, instead of designing massive systems with the focus on raw computational performance, they are more concerned by interoperability and data exchanges within virtual organizations. Even if Grid and HPC research communities are very close in practice, they still differ. For example, most computations on the LHC Grid infrastructure are sequential parameter sweeps applications and not highly tuned MPI applications as this is often the case in HPC. **Classical research questions** in grid systems encompass trust forwarding between virtual organizations, accountability, or the dimensioning of the system (computational elements, networks and storage) so that it accepts the load and ensure that all scientist jobs get handled under reasonable delay, and that all scientific data can be stored and retrieved afterward.

P2P: Mainstream Distributed Systems. Nowadays, similar hardware is used in home PC and in HPC nodes, with a time gap measured only in months. With the popularization of DSL and other high speed personal connections, it becomes tempting to leverage this large potential located at the edges of the network, by individuals. Researches in this area was initially led by the demand for easy (but often illegal – although not always) file sharing solutions between individuals. Meanwhile, Volunteer Computing platforms emerged. These are scientific applications that harness the idle cycles of individually owned computers. Another emerging application is the streaming of popular video to its consumers while minimizing the impact on the network infrastructure.

The main challenge to address in P2P computing is the lack of organization in the resulting aggregated system. Adequate solutions must be completely distributed, without any central point, justifying their name of **Peer-to-Peer** systems. Beyond the removal of any centralization points in the proposed algorithms, the **classical research questions** encompass the adaptation to node volatility (called churn in this context), that get off-line without notice when individuals halt their computers. Because of the experienced latencies and of the burden induced by P2P applications on the ISP networks, it is also crucial to discover and leverage the physical infrastructure underlying the P2P network.

Clouds: Industrial Data Centers. Recent improvements of virtualization techniques permit the encapsulation of program executions within *virtual machines* (VM) with little or no performance loss. The key enabling feature of this technology is that VMs can be freely mapped on the actual physical resources.. This way, the hardware owner can co-locate and migrate users' virtual machines to maximize the hardware utilization and thus the return on investment. Such *elastic computing* is also highly attractive from the user point of view in terms of efficiency and agility. The datacenters' operation is completely externalized to specialists (allowing economy of scale on the operator side) while the users only temporarily rent the computational power they need for their computations as they happen. Since it is often impossible to know where the rented machines are located, the computations are said to be sent in **the Cloud**. Although recent, this trend will not fade in the near future. The federal state of the USA is for example currently moving all the agencies' systems to such settings [Kun11].

This explains that a large momentum of research emerged recently to better understand and optimize these systems. The **classical research questions** are split in two categories. From the provider point of view, the placement of virtual resources upon the physical ones is to be optimized in for application performance, resource efficiency and platform profitability. From the client point of view, the selection cloud resources among providers and available settings is to be optimized with respect to the price paid and according to the targeted service level.

2.2 Specifying Distributed Systems

Depending on the kind of study to be conducted, distributed systems can be expressed in very differing ways. **Abstract processes' representation** is well adapted to algorithmic and theoretical studies. This is usually done using a specific formalism (which can be seen

as a *Domain Specific Language* – DSL), classical automatons, process algebras, petri nets or a CSP-like language. The main advantage of these representations comes from their simplicity, as they omit many details. This allows a good scalability in the case of simulation studies, or a more throughout formal analysis thanks to more complex automatic handling allowed by the simplicity of the application models. Existing formalisms include DEVS [ZPK00] to specify any discrete system to simulate, MACE [KAB+07] to represent distributed P2P-like applications, GOAL [HSL09] to represent parallel HPC applications, TLA$^+$ [Lam02] for the formal analysis of any timed system, and PlusCal [Lam07] for the specification of concurrent and parallel algorithms. The main drawback is that a manual translation may be required to get an executable system, often a transformation that is very error prone. Some of these systems (such as Mace) thus allow to automatically extract an executable implementation to tackle this issue.

A diametrically opposed approach is to rely directly on the **executable code of the program**. It allows for more realistic studies at the expense of higher computational costs. It also hiders the use of some formal analysis methods that rely on a precise definition of application's semantic, as even static analysis cannot automatically provide such insight on real code. Most of the time, this is however limited to applications written using the specific interfaces provided by the tool, but it is possible in some rare cases to study legacy applications, not originally designed for the simulator. This can be done either by using complex system-level interception and interposition mechanisms (as with MicroGrid [XDCC04]), or by reimplementing a given real-world APIs (such as MPI – see §2.4.3) on top of the simulator.

An alternate option is to replace the program with a **trace of events** previously extracted or artificially generated. This is known as *offline* simulation, and permits to decouple the execution of the systems from the simulation procedure. It is probably of little interest for correction studies, but can reveal very useful for performance studies, *e.g.* to compute the timings of the application on another platform. In that case, this provides more flexibility, and can for example be used to simulate programs that are too big to run in a single simulation.

2.3 Performance Evaluation Methodologies

Scientifically assessing the quality of competing algorithmic solutions with respect to a particular metric (*e.g.* task throughput achieved by a scheduling heuristic, probability of service availability, response time of lookup queries) is a key issue. Three classical scientific approaches are used to address this issue: theory, experimentation and computer simulation. In most cases, assessment through pure theoretical analysis can at best be obtained for stringent and ultimately unrealistic assumptions regarding the underlying platform and/or the application. As argued in §2.6.2, the limited applicability of pure theory to our context does not undermine the quality of the conducted studies, but only mandates the use of other epistemological approaches. That is why, most research results in these areas are obtained via empirical experiments, be they through real-world experiments or simulations.

2.3.1 Direct execution and Emulation

The direct execution of the tested applications on production platforms or large testbeds seems an obvious approach for obtaining experimental results. Unfortunately, this approach often proves infeasible. Real-world platforms may not be available for the purpose of experiments, so as not to disrupt production usage. Moreover, experiments can only be conducted for the platform configuration at hand, not on yet-to-be-built next generation systems. Such experiments may also be prohibitively time consuming especially if large numbers of experiments are needed to explore many scenarios with reasonable statistical significance. Finally, conducting reproducible experiments on real-world platforms proves very difficult because of the lack of control over experimental conditions. The experimental control can be improved through the use of emulation techniques, but this increases even further the technical expertise mandated to run such experiments [BNG11, EG11].

Even when such direct experiments reveal possible in practice, other considerations also tend to limit this approach because of the classical power consumption of supercomputers. The aforementioned Jaguar XT5 supercomputer consumes 7Mw/h [gre, FS09] (counting only the computational nodes and excluding storage, network and cooling which can represent 90% of the total consumption [Koo08]). Cloud data centers can reveal even more energy hungry [QWB+09] (up to an astonishing 180 Mw/h for the Chicago's Microsoft data center [Mil]). The total power used for IT data centers is estimated to at least 1% of world energy consumption [Koo08]. This naturally raises ecological considerations, but also financial ones since 1Mw/h costs about 1M$ per year [QWB+09]. Under these settings, using the platform for performance testing and application tuning can be considered an unacceptable waste of resources. At the same time, running under-optimized applications on these platforms is also highly criticized, for the exact same reasons, thus leading to an antinomy.

2.3.2 Simulating Computer Systems

Given the difficulties to run field experiments and the inherent limitations of the resulting studies, the majority of published results in the field are obtained through simulations, even though researchers always strive to obtain at least some experimental results in real-world systems. Simulation allows researchers to evaluate competing solutions in an easy, reproducible and efficient manner. But at the same time, it raises methodological issues since the induced experimental bias must be carefully assessed and controlled. However, note that simulation has become the main approach to scientific investigation in many domains, which, incidentally, has led to the development of ever larger Grids and HPC platforms. Simulation has also been used commonly in several areas of computer science for decades, *e.g.* for microprocessor and network protocol design. It is also widely used for the study of distributed applications, but there is no acknowledged standard simulation frameworks. This may be explained by the relative simplicity of the platforms used until recently. When using a dozen of homogeneous computers running standard CPU bound applications, there is no real need for complex simulation frameworks. The ongoing increase of complexity of distributed computer systems explains why their study

through simulation is evolving into a scientific field on its own.

Simulation as a Scientific Methodology. As stated earlier, simulation is so widely used in science and engineering that it is now considered as the third scientific pillar alongside with theory and experimentation. In a nutshell, **simulation allows to predict behavioral aspects of a system using an approximate model of the system**. In some cases, the model is so simple that the simulation reduces to a simple trace replay, but most of the time it goes through the animation of an *algorithmic model* of the reality [Var10].

A first categorization of all simulations studies can be based on how the model's state evolves [Fer95, LK00]. In **Continuous Simulations**, the state of the model changes continuously in time, usually according to differential equations. This is well suited to simulate evolutive systems such as chemical reactions or physical phenomena such as electric circuits, hydraulics, motors or structural deformation. The simulation consists in solving these equations numerically on each point of the mesh for each time interval, possibly adding some randomness according following the Monte Carlo approach when fully deterministic algorithms reveal too complex or not robust enough.

On the contrary, **Discrete-Event Simulation** (DES) considers that the state of the model changes instantaneously at discrete points in time. It is better suited to study the interaction among components presenting a discrete behavior. Conceptually, DES consists in evaluating the state of each component to predict the next event of the system, and then applying the effects of this event onto the system. The application field is equally vast. It allows to optimize the interactions among the production lines the industry, or to study the behavior of the clients in a bank.

In most cases, the choice between these two types of simulation is given more by the kind (and granularity) of the envisioned study rather than by the type of system under analysis. Hence, the traffic in a city can be modeled like a discrete-event system, capturing the cycles of each traffic light, and the behavior of each car, or it can be described as a continuous system to study the global flow and predict the overall circulation conditions.

Taxonomy of Distributed Systems Simulations. Simulations of Distributed Applications can be further categorized depending on the desired granularity of the study.

Since computer systems are inherently discrete and human-made, it is tempting to capture the behavior of all the components in **Microscopic Discrete-Events Simulation**, without any kind of abstraction. For example the network can be modeled at packet level as a sequence of events such as packet arrivals and departures at end-points and routers; cycle-accurate models of the CPU can be devised; the behavior of a disk drive can be described by imitating the mechanics of the read/write heads and plates rotation. As argued in §2.6.1, overly detailed models do not necessary lead to better studies, and it is common to numerically approximate some phenomenons instead of describing them in all details. Such **Macroscopic Discrete-Events Simulations** are for example classically used in network modeling. The treatment of network packets is then abstracted with mathematical functions that represents the data streams as fluids in pipes [MSMO97, OKM97, PFTK98]. If these functions are correctly chosen, it is possible to obtain an approximation that is

reasonably precise, yet being lighter and faster than a microscopic discrete-events simulation [VL09a].

Constant Time Simulations (also called **query-cycle** simulation) constitute the next step in model abstraction, neglecting the duration of events. The time is discretized in steps or phases of similar lengths. This simplification is comparable to the Big-O studies. It allow to simulate millions on nodes on a low-range host computer by letting the simulation loop over every node to execute one request at each step. This reveals useful to study the interactions among components without considering the underlying platform. This approach is often used in studies of large scale systems such as P2P systems, allowing to study the algorithmic complexity of the proposed solutions at extreme scale.

When the interacting entities of the system become too numerous, studying them at individual scale becomes intractable. To overcome this, **Mean Field Simulations** exploit the behavioral symmetries of the entities, abstracting them into groups of homogeneous behavior. This can be done using a classical DES at class level, with events representing the interaction among groups of entities, but it may reveal difficult to achieve in practice. A simpler alternative is to assume that the amount of entities goes to infinity and study the convergence of the model through *continuous simulation*, where the system evolution are given by probabilistic laws that are integrated over the time. This approach was used in various contexts, from theoretical queuing systems [Mit01] and epidemic models [BMM07] to practical network congestion protocols in TCP [BMR02]. It was also successfully used to efficiently characterize Large-Scale Distributed Systems [JKVA11, LFHKM05].

2.4 Performance Evaluation Simulation Frameworks

This section presents the most prominent simulation frameworks that can be used to predict the performance of parallel and distributed applications from the abundant literature. A first classification of this body of work could be on whether the studies deal with algorithms and prototypes (as it is common in P2P) or directly on applications (as classically done in HPC). This may be related to the fact that P2P systems raise strong algorithmic issues, but their actual implementation is then relatively straightforward. On the other end, technical issues deserve much more attention in HPC, resulting in the emergence of highly sophisticated dedicated runtimes such as MPI for communications. Since most of the HPC applications are written using this standard, it opens specific solutions to study their performance.

2.4.1 Simulation of Distributed Algorithms

The commonly agreed quality criteria for P2P simulators are *scalability* (aiming at 300,000 to 10^6 nodes), *genericity* (ability to use query-cycle and DES, to model churn and to simulate both structured and unstructured overlays) and *usability* (availability to the community, documentation, support) [NBLR06, BDU10, XWWT11].

PeerSim [JMJV] may be the most widely used simulator for theoretical P2P studies. It allows both query-cycle and discrete event simulations, and was reported to simu-

late up to one million nodes in the first mode. Another major advantage of this tool is its relative simplicity, allowing users to modify it to fit their needs. Its main drawback remains its lack of realism, due to the simplifications done for sake of scalability. OverSim [BHK07] tries to address this by leveraging an external discrete event simulation kernel called the OMNeT++[Var]. This later tool is mainly a packet-level simulator comparable to NS2 [ns2], but with extensions to model the computational resources. These extensions are not used in OverSim, which only represents the communication between peers. It was reported to simulate up to 100,000 nodes, but by replacing OMNeT++ by other internal mechanisms. We compare these tools and SimGrid with regard to their scalability in §3.2.3. The simulation validity was never clearly demonstrated, but this is probably due to the fact that it is not considered as a major quality criteria in many P2P studies.

Over the last decade, numerous other simulation projects were also proposed from the P2P community, such as P2PSim [GKL+] or PlanetSim [PGSA09]. But these projects proved to be short lived and are no longer maintained by their authors. Similarly, it is hard to say for now whether the new player of the fields such as D-P2P-Sim [STP+09] will be maintained in the future, which constitutes an important risk for any scientist deciding to base his/her work on these projects.

2.4.2 Simulation of Distributed Applications' Prototypes.

Numerous simulation tools were produced in the recent years in the Grid research community, most of them only intending to be used by their own developers. Several were published but revealed to be short lived and targeting very specific community, such as the ChicSim [RF02] and OptorSim [BCC+03] projects, both specifically designed to study data replication on grids but discontinued since then. SimGrid, the framework that constitutes the technical context of most of my research work, was also created in this context. Its genericity was greatly improved since then, as shown below. Another widely used grid simulator is GridSim [SCV+08], which was initially intended for grid economy and later became used in other areas of grid computing.

To the best of our knowledge, the only freely-available simulators for Cloud studies is CloudSim [CRB+11], a GridSim sequel exposing specific interfaces. In addition, GroudSim [OPF10] is a simulation toolkit allowing both Grid and Cloud studies and Koala [Lea10] is a cloud simulator developed by the NIST, but neither of these tools are available outside the research groups that develop them.

All these tools seek a compromise between execution speed and simulation accuracy. To this end, they rely on rather simple models of performance. The CPU models are macroscopic: tasks costs are measured in MFlops while computer power is measured in MFlop/s. Only three tools model the disk: OptorSim does not take access time into account, but only available space; SimGrid and GridSim provide a fine grain model with latency, seek time and max transfer rate but does not model the file system whose effect on performance is crucial. For the network, ChicSim and GridSim mimic the flow fragmentation into packets that happens in real network, but they do not take TCP flow management mechanisms into account. This approach (called wormhole) makes them

potentially as slow as packet-level simulators such as NS2 [ns2], but not quite as accurate [FC07]. GridSim and SimGrid are compared performance-wise in §3.3.3.

OptorSim and SimGrid rely on analytical models of TCP where flows are represented as flows in pipes [MR99]. Unfortunately, the bandwidth sharing algorithm used by OptorSim is clearly flawed when the platform is not homogeneous: the bandwidth share that each flow receives on a congested network link depends only on the number of flows using this link. This does not take into account the fact that some of these flows may be limited by other links in their path, preventing them to use the whole share on the aforementioned congested link. In such case, the unused link share is wasted while in the real-world it would be split between other (not otherwise limited) flows. This blatant shortcoming is even acknowledged in the documentation of OptorSim, but not fixed in the last release. Such validity issues are unfortunately not uncommon, and GroudSim also exhibits the same network simplifications as OptorSim that hinders its validity on heterogeneous infrastructures.

2.4.3 Simulation of Parallel Applications.

One option for simulating parallel applications is **off-line simulation**. A log, or trace, of computation and communication events is first obtained by running the application on a real-world platform. A simulator then replays the execution of the application as if it were running on a *target platform*, i.e., with different hardware characteristics. This approach is used extensively, as shown by the number of trace-based simulators described in the literature since 2009 [HGWW09, HSL10b, NnFG$^+$10, TLCS09, ZCZ10]. The typical approach is to compute the durations of the time intervals between communication operations, or "CPU bursts". Upon replaying the application, the CPU bursts are modified to account for the performance differential between the host and target platforms, either using simple scaling [HGWW09, NnFG$^+$10, TLCS09] or more sophisticated techniques [SCW$^+$02] accounting for both memory hierarchy characteristics and application memory access patterns.

Network communications are simulated based on the communication events recorded in the trace and on a simulation model of the network. A challenge for off-line simulation is the large size of the traces, which can hinder scalability. Mechanisms have been proposed to improve scalability, including compact trace representations [TLCS09] and replay of a selected subset of the traces [ZCZ10]. Another challenge is that if the application execution is defined by many parameters (e.g., block size, data distribution schemes), a trace may be needed for each parameter configuration. Finally, it is typically necessary to obtain the trace on a platform that has the same scale as the target platform. However, trace extrapolation to larger numbers of nodes than that of the platform used to obtain the trace is feasible in some cases [HSL10b, NnFG$^+$10].

An approach that avoids these particular challenges, but that comes with challenges of its own, is **on-line** simulation. In this approach, the actual code, with no or marginal modification, is executed on a *host platform* that attempts to mimic the behavior of the *target platform*. Part of the instruction stream is then intercepted and passed to a simulator. LAPSE is a well-known on-line simulator developed in the early 90's [DHN96]. In LAPSE, the parallel application executes normally but when a communication operation

is performed a corresponding communication delay is simulated on the target platform using a simple network model (affine point-to-point communication delay based on link latency and bandwidth). MPI-SIM [BDP01] adds I/O subsystem simulation in addition to network simulation. Another project similar in intent and approach is the simulator described in [Rie06]. The BigSim project [ZKK04], unlike MPI-SIM, allows the simulation of computational delays on the target platform. This makes it possible to simulate "what if?" scenarios not only for the network but also for the compute nodes of the target platform. BigSim provides several ways to simulate the computation delays. They can be based on user-supplied projections for the execution time of each block of code (as done also in [GC05]). Another provided approach is to scale the execution times measured on the host platform by a factor that accounts for the performance differential between the host and the target platforms. Finally sophisticated execution time prediction techniques such as those developed in [SCW+02] can be leveraged. The weakness of such approaches is that since the computational application code is not executed, the computed application data is erroneous. The simulation of irregular applications (e.g., branch-and-bound) becomes questionable at best. Aiming for high accuracy, the work in [LRM09] uses a cycle-accurate hardware simulator for computation delays, which leads to a high ratio of simulation time to simulated time.

The complexity of the network simulation model has a high impact on speed and scalability, thus compelling many authors to adopt simplistic network models. One simplification is to use monolithic performance models for collective communications [TLCS09, BLGE03]. Another simplification is to ignore network contention. The work in [TLCS09] proposes the use of simple analytical models of network contention for off-line simulation. An exception is MPI-NetSim [PWTR09], which provides full-fledged contention simulation via a packet-level discrete-event network simulator. As a result, the simulator may run more slowly than the application, which poses time coherence problems for on-line simulation. Another exception is PEVPM [GC05] that provides probability distributions of communication times to model network contention phenomenons.

2.4.4 Simulation and Open Science

The *Open Science* approach is to publish and share between researchers the simulator, simulation scenario and simulation tools used for the studies. Since the experiments in our domain are made through computer simulations, one would have high expectation on the scientific quality in general, and on reproducing the experiments in particular. The reality is however completely different. In [NBLR06], the author reviewed 141 papers based on simulation about peer-to-peer systems. They find that 30% use a custom simulation tool while half of them do not even report which simulation tool was used! Given the potentially large impact on semantic and performance of only a slight change in a distributed protocol [CKV01], reproducibility of results in the current state of the art is questionable at best. This is particularly ironical given that every simulated experiment ought to be perfectly reproducible by design. Indeed, this remains true only if the experimental methodology is meticulously documented.

This sorry situation is not exactly glorious for our community, and one would expect that at least the authors of experimental instruments such as the simulators used for such

studies would blaze a trail. This is not true either, and most the tools presented in previous sections are very difficult (and sometimes impossible) to use by other researchers than their authors. For example, when trying to compare the PSINS framework [TLCS09] with the SMPI framework (see §4.1.2), we stumbled upon the fact that the distributed version of PSINS had undocumented dependencies and missing parts, which we had to ask to the authors. Even worse, the techniques used in [SCW+02] relies on the MetaSim tracer and the pmac convolver, which are not available and whose internals are only superficially described in related publications. Under such conditions, it is impossible to reproduce results based on such frameworks and to have confidence in their applicability to context even slightly different from those studied by their authors. Building upon such work is thus very hard and many performance studies have to be restarted from scratch.

To conclude this depressing section, I must confess that our work on SimGrid is not trail blazing either although we try to address these issues. We distribute our framework as free software, exposing every intermediate versions in the Git version management system. We also strive at making our experimental settings available on the Internet, such as http://simgrid-publis.gforge.inria.fr/ that collects the scripts, sources and data of [QRT12] and [BLD+12]. This is unfortunately not sufficient, as we did not manage to rerun our own experiments one year after to evaluate the impact of some recent changes. As an answer, we made the Git version management of these scripts public[3], and reworked them to make them easier to use by others. I still think that much remains to be done for Open Science in our community, as expressed in §4.4 and §5.2.

2.5 Correction Evaluation Methodologies

Performance and correction mandate very different techniques. This is because most of the time, performance studies reason about representative or average executions. On the contrary, correction must be assessed over all possible executions. Exceptions to this rule naturally exist. Worst case performance analysis for example reason about all executions, but it remains that most performance studies on distributed systems are experimental (as detailed in §2.6.2), and are thus limited to subsets of the possible executions. Also, *tests* can be used to assess the correction of a system over a fix subset of scenarios. This approach is however almost never used in scientific publications, as a test plan cannot give a definitive answer on the correctness. The authors either do not discuss the correction of their solution (assuming that it works since no defect was found experimentally), or they tackle the issue using more formal methods.

Usually, the distributed algorithms published in the literature are shown correct through **classical mathematical proofs**, that are formal declarative statements. This process being manual, it remains possible that errors crept into the arguments. Peer reviewing is however usually seen as a sufficient way to reach a reasonable level of confidence. A possibility to further reduce the odds of errors on long and technical proofs is to use automated **proof assistants** such as Coq [dt04] or Isabelle [NWP02]. These tools do not

[3] https://github.com/mquinson/simgrid-scalability-XPs/

replace the expert, as they must be manually *guided* to the proof objective through cleverly crafted intermediary proof objectives.

Model checking is another technique to establish the correction of systems. Instead of trying to inductively demonstrate that the model meets its specification, it strives at verifying that the system cannot evolve from the considered initial states to any states where the properties would be violated. This is done by computing all possible execution trajectories, starting from the given initial states. The search continues until a property violation is detected, or until the whole state space is explored, or until the host system runs out of resources. If the properties are shown to be invalid, a trajectory leading to this violation is provided as a counter example. This methodology is thus much more practical (or even *experimental*) than the usual mathematical proofs. But unlike classical tests, it leads upon success to a formal proof of the assessed properties.

Proof obtained through model checking are marginally less generic than declarative proofs, as it is only known that the system cannot evolve to faulty states *from the considered initial states*. Nothing is known when the system starts from another state.

Also, the applicability of model checking is more restricted than regular proofs, as it remains difficult to handle infinite state systems this way. Explicit-state model checking then becomes inapplicable, mandating the use of symbolic representations that are implicitly explored. In turn, this is not universal as for example deciding on the correctness of recursive and multithreaded programs is another form of the halting problem, that is not decidable [JM09].

The main downside is that this approach sometimes fail to produce an answer due to the potential size of the state space. Complex reduction techniques are thus introduced to fight the state space explosion. They try to detect redundant or symmetric states and trajectories. The difficulty is to prove that these reduction techniques do not endanger the validity of the model checking, that is, that the state space exploration remains sound and complete. The effectiveness of these techniques depends on specificity of the model and properties, and some may only be applied under some conditions.

Beyond these restrictions, the model checking presents strong advantages, justifying reasons why I became interested in this technique in the first place. Foremost, its exploration process can be automatized. This fully automatic nature allows non expert users to verify systems with less effort than with proof assistants. The exhaustiveness of the conclusions is also very appealing to me, as I usually have to manually decide whether my test plan is sufficient or not to draw the conclusion from my simulations.

Formal methods were not part of my initial area of expertise after my PhD program. I use model checking since 2006 only, striving to adapt them to research domain of distributed systems, but I cannot pretend to discuss the interest of model checking with regard to other formal methods any further. The rest of this document thus focuses on model checking, evacuating any other methods to assess the correction of a system.

Safety and Liveness Properties. In linear-time temporal logic, formulas are interpreted over sequences of states, and can be classified in two categories. The **safety properties** express facts of the type *"nothing bad can happen"*, such as absence of deadlock, or race

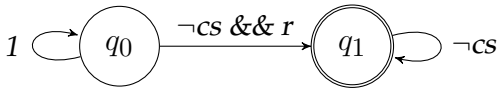


Figure 2.1: Büchi Automaton accepting any histories matching $\neg\phi$, with $\phi = \Box(r \Rightarrow \Diamond cs)$. It is used to search for counter examples of ϕ. If r denotes whether the request is issued and cs whether the critical section is granted, ϕ denotes the property "Any process asking the critical section will eventually obtain it".

conditions are examples of such properties. Their truth value depends only on finite histories, and can be evaluated by inspecting single states of the system.

The **liveness properties** denote that something must (or must not) occur in the future. For example, when the breaks is pressed, the car must eventually slow down; when the button is pressed, the elevator must eventually come. Determining the truth value of liveness properties is much more complicated than for the safety ones, because they depend on infinite histories of computations. More precisely, such property can only be falsified over an infinite execution. Hence, one possibility to verify a liveness property is to determine if the system is free of strongly connected components that contain a cycle along which the liveness property is violated.

Mathematical Notations. Model checking is usually built upon *Linear Temporal Logic* (LTL), that allow to reason about the history of the system, that is the successive states that it is in. At each step of history, the value that a variable x will take in the next state is noted as x'. Time quantifiers are also introduced, such as $\Box$ (for *always*, where $\Box \boldsymbol{P}$ means that P will stand true in any subsequent history) and $\Diamond$ (for *eventually*, where $\Diamond \boldsymbol{P}$ means that there exist a point of future history where P be becomes true).

Büchi automatons are used to represent the model and liveness properties, as they can represent infinite words. A Büchi automaton is a 5-tuple $B = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is a finite alphabet; δ is the state transition relation ($\delta : Q \times \Sigma \to Q$); q_0 is the initial state ($q_0 \in Q$); F is the acceptance condition, *i.e.*, the set of final states ($F \subseteq Q$).

A finite sequence of states $s_1, \ldots, s_n$ (with $s_i \in \Sigma$) is *accepted* by the automaton B if there is set of transitions in δ starting at q_o so that each transition is labeled by the $s_1 \ldots s_n$ and ending with one of the states of F. An infinite word is accepted if it starts at q_0 and visits infinitely often to some states of F.
I refer the reader to [CGP99] for further information on the model checking foundations.

The Model Checking Algorithms. The verification algorithm greatly depends on the type of property to be verified, liveness properties introducing specific complexities. Let's assume a model M, an initial state s_0, and a property ϕ to be verified.

The verification of **safety properties** is also called the reachability analysis problem. If ϕ is an assertion, the problem is reduced to determining if it is possible to reach a state s' from s_0 where ϕ won't hold. The model checking algorithm in this case consists of an exhaustive exploration of the model's state space, searching for a counter-example. At every visited state, the validity of the properties specified by ϕ is tested.

This approach is not sufficient if ϕ is a **liveness property**, since such properties can only be falsified over infinite histories. Instead, we rely on a Büchi automaton B for the negation of ϕ, and search for an execution leading to an infinite history accepted by the automaton. An example of such automaton is depicted in Figure 2.1. In practice, the classical verification algorithm explores $M \times B$: it tries to advance alternatively M and B. At any point, all enabled transitions (*i.e.* in B, edges that go out of the current state, and that are labeled with a property evaluating to true in the current state) are explored one after the other. For that, the first possibility gets explored in depth-first. Then, when the exploration of this branch is terminated (either because M reached a final state, or because the exploration reached the configured maximal depth of exploration), it rollbacks to the decision point and explore the second possibility, and then the subsequent ones.

Any path p explored this way in $M \times B$ denotes a counter-example to ϕ iff it contains at least *twice* the same accepting pair $\{q, s_A\} \in M \times B$ such that $s_A \in F$. Indeed, if p manages to return to $\{q, s_A\}$ from $\{q, s_A\}$, it can do so *infinitely often*. This is the necessary condition to have a Büchi automaton accepting a word of infinite length, so we conclude that B accepts p, that thus constitutes a counter example to ϕ.

If the algorithm ends without finding any such counter example execution, we conclude that the property ϕ hold for M, when starting from s_0.

To explore all possible executions, model checkers rely on the ability to checkpoint and rollback the model each decision point of their exploration, so that they can explore all possibilities one after the other. Two main approaches are possible for that: **State-full** model checkers actually checkpoint the model on each decision point. **State-less** model checkers take a unique checkpoint, at the beginning of the execution. The backtracking is then replay-based: it simply restarts the program from the beginning and executes the previous scheduling until the desired backtracking point. This avoids exhausting the memory with countless checkpoints, at the price of more computational intensive rollbacks. Intermediary solutions are possible, taking checkpoints only every N steps.

The State Explosion Problem. Model checking intrinsically relies on the exploration of a transition system, which size grows exponentially with the sizes of the model and the properties. This problem, known as the **state explosion**, is one of the biggest stumbling block to the practical application of the model checking. As a result, a great body of work is dedicated to cope with this problem. This literature can be classified in two main categories ([JM09]). First, *composition techniques* mitigate the state explosion by verifying properties on subprograms so that then properties can then combined to deduce global properties on the whole system. For example, the DeMeter framework [GWZ+11] allows to check separately the components of a distributed system, and then combines the interface behaviors automatically.

Then, *Reduction techniques* aim at identifying equivalence classes in system histories, and explore only one history per class. One solution to that extend is to leverage symmetries, *e.g.*, between processes. One of the most effective techniques for handling asynchronous systems is the **partial order reduction** (POR) [God91]. The idea is to not explore several traces that differ only in the ordering of events that are independent. Assume for example two transitions t_1, t_2 that only access the local state of two separated processes.

Their relative order have then no impact on the final state, *i.e.*, applying t_1 and then t_2 leads to the same result than applying t_2 and then t_1, so the model checker should explore only one of the orders. This is the approach used in my work, as presented in §**??**. Another approach is to generate an abstraction of the model that has a smaller state space, but that *simulates* it. The approach presented in [CGJ+03] ensures that properties that are valid in the abstract model hold in the concrete one, but a property valid in the concrete system may fail to hold in the abstraction. This requires to test the counterexamples found in the abstraction against the original model. *Spurious* counterexamples are then used to improve the abstraction, thus the approach's name: *Counter-example Guided Abstraction Refinement* (CEGAR).

Software Model Checking. As its name indicates model checking was initially geared towards the verification of abstract models. Rapidly, it got used in other domains. Hardware designs for example were getting more complex, making traditional simulation techniques unable to ensure the correctness. As an answer, the model checking was leveraged give formal guarantees about the correctness of these systems. The idea of applying model checking to actual programs originated in the late 1990s [VHBP00, MPC+02], and still remains a challenging area of research.

The first difficulty to overcome is the availability of the models themselves. Traditional model checkers require the models to be written in specification languages like TLA^+ for TLC [Lam02] or Promela for SPIN [Hol97]. In contrary to the common practice of hardware design, this modeling phase is often skipped in software engineering, or conducted using less formal languages (such as UML) that are not well adapted to formal verification. Even if such formal models exist for the software to be verified, they result of a manual modeling process. Conversion between this formalism and program source code are however known to be very tedious, and many bugs can be introduced while doing so.

To overcome these limitations several recent model checkers accept source code as input. This in turn raises specific difficulties, as most programming languages are not designed for model checking. In particular, there is no trivial way to get the state space of the model to verify from the program code. Two main approaches exist in the literature.

Dynamic Formal Verification consists of exploring the state space of the program by executing it under the control of the model checker [God97, VVGK09]. The model is thus explored in practice, but remains inaccessible, making some optimization techniques inapplicable. Dynamic verification techniques provide the most detailed representation of the systems, where the states are the real running memory of the program, and the counterexamples point to bugs in the implementation. But they also suffer from the complexity of the real running state. For example, saving the program state for further rollback is naturally more complex when dealing with real programs than when dealing with abstract models which state is clearly identified. As model checkers also need to compare states for equality, which is made complex by the dynamic nature of the system heap. As a result, many heap configurations correspond to the same observable state of the program [Ios01]. Finally, many programs have an infinite state space and thus it is not possible to fully verify them using a dynamic approach, as every state has to be explicitly

visited. However, it is still possible to bound the exploration and verify the executions of the program up to the bound. In general, dynamic model checkers are designed more as a bug hunting tool than as a mean to ensure full correctness.

Automatic abstraction consists of generating an abstract model of the program automatically using static analysis, or symbolic execution, and then following a classical model checking approach on the abstraction. Because the abstraction is automatically computed, it is usually an approximation of the program's real behavior. Also, some aspects of the system are particularly challenging to automatically retrieve through the static analysis of the source code, and are only available at run time.

Combining these two approaches would allow to benefit both from statically gathered information and dynamically retrieved knowledge. This would certainly allow to propose further reductions techniques, but to the best of my knowledge, this was never attempted in the literature. This is probably because of the technical difficulties raised, although modern compilation tools such as LLVM [LA04] could help alleviating them.

Software Model Checking Tools. **Verisoft** [God97] was one of the first software model checker. It allows to verify safety properties on arbitrary concurrent programs using semaphores, mutexes and other system level IPC (but no multi-threading nor shared memory). This work first introduced the state-less approach to verification. It addresses the state space explosion using POR techniques based on the detection of independent transitions. For that, transition concerning differing synchronization objects are said to be independent. **Java PathFinder** [VHBP00] allows to verify safety properties on Java bytecode relying on a custom Java virtual machine to control the bytecode execution, the scheduling, and to store the state in a fast and compact way. Early version generated automatic abstraction through static analysis, and passed them to the SPIN external tool. POR techniques based on static analysis are proposed.

CMC [MPC+02] allows to verify safety properties on C programs interacting over the network. It alleviate the state explosion by detecting the already visited states to not re-explore them. To reduce the amount of false inequality, it uses a canonical representation of the memory. To reduce the memory requirements, only a hash of each state is used in the comparison. Hash conflicts can then void the soundness of the verification process, but the probability remains low. In addition, CMC is designed as a bug hunting tool, meaning that it cannot be used to prove the correctness of the algorithms, but revealed to actually find defect in the tested programs. The main author of CMC now leads the **CHESS** [MQ08] project. This is a state-less bug hunting tool for multithreaded software based on Win32 or .NET APIs. It refines the approach of Verisoft by introducing fairness constraints on the schedules, avoiding many interleavings that are unlikely to happen with the real OS scheduler. **Line-Up** [BDMT10] builds upon Chess to check the *linearizability* of concurrent data structures. This condition ensures that the structure behaves similarly in sequential and concurrent settings, remaining easily understandable by the users. This constitutes a particular category of liveness properties. Line-Up is a bug hunting tool working by ensuring that the execution of user-provided scenarios lead to the same result when run in sequential or in parallel. No specific support to address state explosion seems to be provided, beside of the randomization of the algorithm. Several

instances can then run in parallel, hopefully accelerating the discover of some defects.

MoDist [YCW+09] verifies safety properties over unmodified distributed systems that use the WinAPI. An interposition layer exposes all the interactions of the nodes with the operating system, and communicates with a centralized model checking engine that explores all relevant interleavings of these actions. This approach allows to work at binary level, without the source code of the application being verified. To cope with the state space explosion, it uses a DPOR based exploration algorithm.

MaceMC [KAJV07] is a model checker for the MACE domain specific language that can be compiled into C++ programs that execute on real platforms (as presented in §2.2). MaceMC exploits the fact that MACE programs are structured as state transition systems to perform their state space exploration. It can verify safety properties by combining a stateless approach with state hashing. MaceMC can also verify the liveness properties that can be expressed as $\Box\Diamond P$, with P being a property without any time quantifier. This properties can thus be seen as invariant that are allowed to be temporarily violated. For example "all sent messages are delivered" can be false at some point of the execution, but it is expected that it is true infinitely often over any execution. Properties such as $\Box\Diamond P$ are verified through an heuristic where the state space is explored exhaustively only until a rather limited depth. From each state of the periphery, a long random walk is conducted, searching to states where the P becomes true. If no such state can be found, the execution is suspected of being a counter example of $\Box\Diamond P$. As these executions can be very long (up to hundreds of thousand transitions), the developers need some support to isolate the interesting parts, denoting a bug in the tested application. To that extend, an heuristic searches for the *critical transition*, that is the transition which plunged the property from a transient violation (*i.e.*, there exist some subsequent state where P is true) to a permanent violation (*i.e.*, no such subsequent state can be found). This is achieved through random walks starting from the states of suspected execution, searching for the last state of the suspected trace from which no random walk validating P can be found. Although the soundness of the exploration is by no mean enforced, this technique allowed to find many bugs in practice in various algorithms.

CrystalBall [YKKK09] is consistency enforcement tool implemented on top of MACE. Nodes continuously run a state exploration algorithm up to a limited depth, starting on a recent snapshot taken from the real live system. This allows to mitigate the explosion problem by exploring only regions that actually happen at runtime. Additionally, if certain invalid states are detected, the system execution is steered to avoid these problems.

ISP [VVD+09] verifies safety properties over HPC applications written with the MPI library. It works by intercepting the MPI calls of the application through the PMPI interface, originally intended to allow profiling tools to capture information on the running application. The ISP "profiler" can rewrite the MPI call parameters or delay the process execution if instructed so by a central scheduler that run the exploration algorithm. State explosion is implemented through DPOR depending on an independence predicate over the MPI functions [VVGK09]. **DAMPI** [VAG+10] builds upon the same interception techniques, but greatly reduce the amount of explored interleavings. It leverage the fact that many communications of typical MPI applications are deterministic (the sender and receiver is fixed by the programmer). DAMPI matches any non-determinist receive with all sends that are not *causally after* it, using Lamport vector clocks to determine the causality.

The amount of interleavings to test for a sound coverage can greatly reduced this way, down to a unique interleaving in most of the applications tested by the authors.

For further information on Software Model Checking tools and techniques, the reader is referred for example to [JM09].

2.6 Methodological Considerations

Since my own scientific objects are methodologies used to conduct scientific studies, I had several heated debates about the scientific methodology in the past. This section summarizes my position on these considerations. In §2.6.1, I fight a common misconception stating that increasing the level of details in the models is the way to go to increase their realism. Indeed, I was often told that our simulations could never lead to accurate answers as we do not even *try* to capture all effects of the real world. In §2.6.2, I go after a much more profound criticism sometimes expressed against my work, stating that pure theory should be enough to understand any human-made system, and that my approach is a scientific renouncement. Answering this criticism requires to situate the computer science in the whole picture of existing sciences. Interestingly, this argumentation also justifies the overall organization of this document, separated in two main chapters of contributions.

I am well aware that these debates are part of the philosophy of computer science and that I am not an expert of this domain. I still hope that the practical point of view accumulated "on the field" can contribute to an endogenous Philosophy of Computer Science. Any contribution, even as small as mine, seems welcomed because the philosophy of computer science is not well established yet [Ted07], certainly not on the same level than the huge the societal impact of our discipline. For further topics, I advise the reading of [Ted07] and [Var09], very inspiring on the Philosophy of Computer Science.

2.6.1 Precision and Realism of Simulation's Models

It is commonly accepted that there is an unavoidable trade-off between the amount of details (and thus the lack of scalability) and the model's realism trough the reduction of the experimental bias [BCC+06]. This common belief is very ancient as it was stated by René Descartes almost four centuries ago: "Divide each difficulty into as many parts as is feasible and necessary to resolve it" [Des34, part II]. Nevertheless, I see basically three reasons to strongly disagree with this statement.

The first reason is captured by this quote of George E. P. Box "*essentially, all models are wrong, but some are useful.*" [Box87, p. 424]. In particular, the most detailed model of a networking system will fail to capture the random noise due to the imperfections of the considered physical hardware. This is because the models are ideally perfect objects while the modeled systems are from the real world, thus subject to causal relations [Fet88]. Failing to capture all the imperfection details may lead to phenomenons such as the *Phase Effect*, that was first presented by Floyd et al in [FJ91]. It happens when events occurring at the exact same time in the simulator induce resonance effects that do not happen in reality because of the system noise. The system noise can be integrated to the model

using a random function, but this constitutes an abstraction. That is why fixing the model this way contributes to my point stating that reducing the level of abstraction and adding more details does not always increase the predictive power of a model.

I am therefore very critical about tools such as ICanCloud [NVPC+11] that try to incorporate every details of cloud systems, usually composed of thousands of computers (from disks' head placement strategies to the process scheduling policies of the CPU and Operating Systems!) in the hope that it will automagically increase the model's prediction power. I feel easier to design macroscopic models properly capturing the effects that I want to study instead of building utterly detailed models in the hope of seeing these effects emerging *per se*.

Secondly, complex models tend to be too fragile to be usable in practice. The work presented in [GKOH00] is a retrospective on a six years project in which several differing simulators were used to design a processor chip that were effectively produced afterward. This gives to the authors the special opportunity to discuss the quality of all the simulators that they designed during the project course. Doing so, they come to the conclusion that "*a more complex simulator does not ensure better simulation data*". The authors explain this by the difficulty to properly configure complex models. In addition, my personal experience would advise that simpler models are also intrinsically more robust than complex ones that tend to diverge rapidly.

Thirdly, even when their realism and robustness is not discussed, more detailed models do not automatically lead to better scientific studies as they tend to produce "*a wealth of information [that] create a poverty of attention*" [Sim71, p. 40-41], requesting their users to sort out the relevant information out of the flow of overabundant data, which complicates the observation of the phenomenons under study.

2.6.2 Limitations of Purely Theoretical Approaches

My methodological approach is based on the statement that pure theory is not sufficient to fully understand large scale distributed systems. That is why I explore the possibility to *test* such systems (using the methods presented in §2.3 and §2.5) instead of trying to *prove* their properties. I had a similar argumentation during my PhD [Qui03, p37-38]. This goes against a traditional belief stating that computer science is merely a spin-off of mathematics. For example, Hoare stated that "Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning" [Hoa69]. Knuth stated that computer science were all about algorithms and that it should have been called *algorithmics* instead. Under this premise, my methodological statement comes down to the affirmation that pure theory is not sufficient in this branch of mathematics, which makes no sense. The remaining of this section shows that this apparent contradiction comes from a *misrepresentation* of the computer science.

There is no established definition of this science. Surprisingly, the popular representation of computer science greatly differs around the world, as one can see by browsing the Wikipedia pages on "computer science" in the differing languages (using some automatic translation tool). Portuguese contributors insist on the heritage of Shannon's information

theory; Germans (and Russians to some extend) put a lot of emphasis on the linguistic heritage, listing Chomsky as one of the pioneers and formal languages as the main theoretical topic in computer science; English-speaking contributors point the importance of engineering background; French speaking volunteers list calculability, algorithmic and cryptography (all very similar to mathematics) as major parts of the "science of informatics"; The Hebrew and English pages notes that the academic discipline is mainly hosted in the buildings of Mathematics and Electrical Engineering of universities.

The philosophy of computer science is not well established either. This should not be very surprising given how young our discipline is. Even sociology, which still refines its epistemological foundations, was established as an academic discipline over one century ago while the first department of computer science was established in the early 1960s only. In addition, cultural variations are also to be observed in the philosophical foundations of our discipline. Most of the French-speaking literature on the epistemology of the informatics goes about defining this science. It thus explores the objects under study, the means and the methods used to do so [LM90] while the English speaking literature on the Philosophy of Computer Science focuses on the historical heritage of the discipline.

The French-speaking debate rages to define whether the list of scientific objects can be limited to *information*, *algorithms* and *machines* or whether other notions such as languages, complexity or simulation must be added to the list [Var09]. Concerning the mean and methods, [Dow11] notes that computer science occupies a very specific place in the epistemological taxonomy of sciences. **Mathematics** use *a priori* judgments, and prove them right through demonstrations. That is, they study the things that are true by *necessity* in any *possible world* and not only in our reality, and they prove them true using proofs, that are declarative statements. On the other hand, natural and **empiric sciences** such as physics use *experiments* to confirm or infirm *a posteriori* judgments. They thus study the things that are *contingent*, maybe true only in our world (there may well be another world where the light does not travel at 300,000 km/s)[4].

This gives **computer science** a very interesting position, since it traditionally uses experiments to assess *a priori* judgments. This specific position is confirmed by the fact that the ultra classic IMRAD article organization (Introduction, Material and Methods, Results And Discussion) is almost never used in our discipline, even if it is prevailing in empiric sciences [DG06]. Instead, most of our work is about taking a given problem, proposing a solution (that is often a *necessary* truth, not one that is *contingent* to our world) and proving that this solution works, most of the time through *experiments*. This very specific position of computer science in epistemology is not limited to the french-speaking scientific community. For example, the well known university of Heidelberg in Germany have a faculty named "Natural Sciences, Mathematics and Computer Science", demonstrating that computer science is neither a natural science nor a branch of mathematics, although it shares elements with both.

But this categorization is not completely satisfactory, as there is numerous scientific works in our discipline that study *a posteriori* judgments (such as the ones dealing with chip designs or energy savings strategies: their results are highly contingent to our

[4]Necessary and contingent truths in cartoon: http://abstrusegoose.com/316

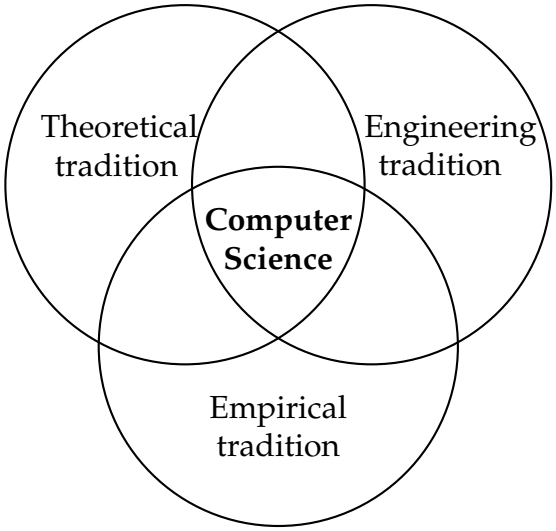


Figure 2.2: Historical heritages of Computer Science (cf. [CGM+89]).

world), or that come to their point through *demonstrations* instead of experiences (such as the ones categorizing the problems depending on their time- and space-complexity by reduction to other problems). The epistemology of computer science thus includes every possible judgments and methodologies, raising the need for further categorization, within computer science itself.

When considering computer science from the historical perspective, most of the authors focus on the fact that some algorithms are used since centuries (*e.g.*, by al-Khwarizmi or Euclid, respectively eleven centuries and two millennium ago) and present the evolution of mechanical calculators that predated computers, such as the Babbage's difference engine.

But to understand the underlying science, it is much more interesting to refer to the *scientific heritages* of computer science, as done in [CGM+89]. The authors identify three main cultural styles (or traditions) in our discipline as depicted in Figure 2.2. **Theory** is actually a spin-off of mathematics, aiming at establishing coherent theories. The process is to freely define some ideal objects through axioms, hypothesize theorems on the relationships between these objects, and then assessing whether these theorems are true using *proofs*. The **empiricism** follows the epistemology of natural sciences, aiming at understanding our world as it is. It builds theories trying to explain various aspects of our world, predict facts according to these theories, and then strive to find *experiences* invalidating the theories. This is for example how most of artificial intelligence research are conducted. It can also be paralleled to the abstractions that occupy a central place in computer science [Var09]. Finally, **engineering** aims at realizing working systems. They identify the requirements, propose a set of specifications that are sufficient to fulfill the requirements, and then design and implement the systems that follow these specifications. They evaluate their realization through *tests* and *benchmarks*.

This categorization is very instructive for the epistemology of computer science. Computer science uses both *a priori* and *a posteriori* judgments, as well as both demonstration

and experiences because it relies on several epistemological approaches to study its scientific objects. The relative lack of diffusion of this bright idea may be due to the fact that engineering not being a science, most computer scientists feel this connections to the machines as demeaning (especially in France, see [Dow09]). But this cultural background cannot be questioned given how *problem solving* is utterly important to our discipline [Win06].

The perspective proposed by [CGM+89] is also enlightening when coming back to my research: When I state that pure theory is not sufficient to fully understand large scale distributed systems, I only state that the other cultural heritages of computer science must also be leveraged to study these systems, which seems perfectly sensible. The engineering heritage is leveraged in most sections of §3 to enable the efficient simulation of distributed applications; the theoretical heritage is leveraged in §3.4 (to demonstrate the soundness of our reduced state space exploration); the empirical heritage is also leveraged in several parts, such as §4.3 to propose a performance model of MPI runtims. Finally, I often mix these approaches, such as §4.2 that leverage engineering and empiric methodologies to propose a workbench to study network tomography algorithms or §3.4 that rely on a mathematical modeling of the network semantic that was introduced in §3.1.2 under the engineering light.

Chapter 3

Simulation of Large-Scale Distributed Applications

THIS CHAPTER PRESENTS MY WORK on the simulation of Large-Scale Distributed Applications, aiming at improving the evaluation framework itself. §3.1 introduces the SimGrid framework, that constitutes the technical context of my work since almost ten years. It presents the software architecture that I devised (in collaboration with A. Legrand and others) to ensure the versatility and performance of the framework. I then detail several works improving the simulator's performance: §3.2 presents a novel approach to the parallelization of fine grain simulations allowing to improve the simulation speed while §3.3 introduces a scalable manner to represent large-scale distributed networks that permits the simulation of larger scenarios. §3.4 details how SimGrid were modified in order to allow correction studies through model-checking in addition to the performance studies through simulation that are usually done with this tool. Somehow unexpectedly, this reveals very related to the parallel simulation.

3.1 The SimGrid Framework

The SimGrid project was initiated in 1999 to allow the study of scheduling algorithms for heterogeneous platforms. SimGrid v1 [Cas01] made it easy to prototype scheduling heuristics and to test them on a variety of applications (expressed as task graphs) and platforms. In 2003, SimGrid v2 [CLM03] extended the capabilities of its predecessor in two major ways. First, the realism of the a simulation engine was improved by transitioning from a wormhole model to an analytic one. Second, an API was added to study non-centralized scheduling and other kind of concurrent sequential processes (CSPs).

I came into this project in 2004, working on SimGrid v3.0, which was released in 2005. Since then, I am the principal software architect of this framework in collaboration with A. Legrand. I am also the official leader of two nation-wide projects aiming at improving the framework that were funded by the ANR: USS-SimGrid (Ultra Scalable Simulation with SimGrid — 2009-11, 800k€, 20 man-years of permanent researchers) and SONGS (Simulation Of Next Generation Systems — 2012-16, 1.8M€, 35 man-years of permanent researchers). I conduct this task of animating the scientific community and orienting the scientific directions in tight collaboration with A. Legrand and F. Suter.

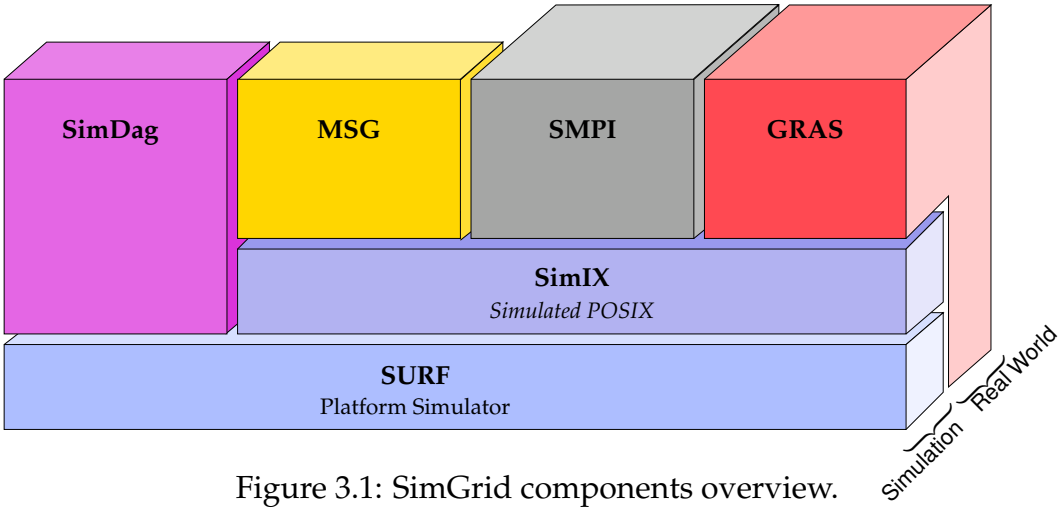


Figure 3.1: SimGrid components overview.

Over the years, SimGrid grounded the experiments of more than 100 scientific publications, in domains ranging from Grid Computing [BHFC08], Data Grids [FM07], Volunteer Computing [HFH08], and Peer-to-Peer systems [CBNEB11]. Its scalability has also been demonstrated independently by others in [GR10, BB09, DMVB08], while providing a reasonable level of accuracy [VL09b]. This remarkable versatility is a strong argument for the chosen software design.

SimGrid From 30,000 Feet. As depicted in Figure 3.1, SimGrid is composed of three layers: the lowest one, called SURF for historical reasons, contains the platform models. Basically, **SURF allows to map actions onto resources** (such as communications onto links), and compute when these actions are to be terminated. At the surf level, a message exchange is merely a communication action that consume the availabilities of a given link. The intermediate layer that builds upon Surf is called Simix (a portmanteau of "Simulated" and "Posix"). Its main goal is to let **Simix let processes interact with each other through specific synchronizations**. At this level, a message exchange is a specific synchronization schema that interconnect two processes. The processes can decide to block onto that communication (*i.e.* sleep until the communication is terminated), or regularly check whether the communication is done. Naturally, when that communication terminates is computed by Surf. There is thus a strong connection between Simix synchronizations and Surf actions, although it may not be a simple 1:1 mapping. The upper layer introduces the public interfaces with which the users interact directly. These **user interfaces constitute the syntactic sugar needed to improve the user experience**. It uses the lower layers and provide an interface that sound familiar to the users. Several such interfaces exists to match the need of several user communities, and several bindings also exist to allow the usage of SimGrid from Java, Ruby or lua in addition to the C language.

The following sections detail this overview by briefly presenting how each layer is architectured to fulfill its goal, underlining my contributions to this design.

3.1.1 SURF: The Modeling Layer

Basically, SURF is the part of SimGrid in charge of mapping actions onto resources (be it communications onto links, computations onto CPUs or I/O requests onto disks), computing when the actions terminate, and of keeping track of the simulated clock. This design is mainly the realization of A. Legrand and presented here for completion only.

The Big Picture. Any action can be placed onto several resources at the same time, such as communications that typically spawn over several links. Less common action kinds are also possible, such as *parallel tasks* that are both communications and computations (thus mapped onto CPUs and links at the same time) allowing specific studies such as [DS10].

Once all wanted actions are placed onto resources, the simulation can be launched for a round, after which surf returns the first terminating action(s) to the upper layers. For that, Surf first computes the resource's sharing, that is how each resource's power is shared between the actions that are mapped on it. With this sharing information alongside with the power of each resource and amount of work remaining for the completion of each action, Surf can then easily compute the termination date of the first terminating action. It updates the clock accordingly and updates the remaining work of all other actions in preparation of the next simulation rounds.

Computing the Sharing. The exact sharing naturally depends on the used model: it can be a fair sharing where all actions on a given resource gets the same power (as for the CPUs in most cases), or more complex coefficients can be computed (as it is often the case for the network, where long connections are penalized over short ones). In practice, computing this sharing comes down to the resolution of a linear system. The share devoted to a given action is represented by a variable, and an equation is added to the system for each constraint. The most common ones are capacity constraints, that state that the share of all actions located on a given resource cannot exceed the resource's power.

Figure 3.2 represents an example of such a linear system with four resources (two CPUs and two links) and six ongoing actions: x_1, x_2 and x_3 are compute actions while y_1, y_2 and y_3 are communication actions. Equation 3.1a denotes that the action corresponding to x_1 is located on CPU_1 while Equation 3.1b denotes that x_2, x_3 and x_4 are located on CPU_2. Concerning the communications, y_1 traverses both links while y_2 and y_3 only use one link each. The corresponding platform is depicted in Figure 3.3.

$$x_1 \leq Power\_CPU_1 \quad (3.1a)$$
$$x_2 + x_3 \leq Power\_CPU_2 \quad (3.1b)$$
$$y_1 + y_2 \leq Power\_link_1 \quad (3.1c)$$
$$y_1 + y_3 \leq Power\_link_2 \quad (3.1d)$$

Figure 3.2: Example of Linear System.

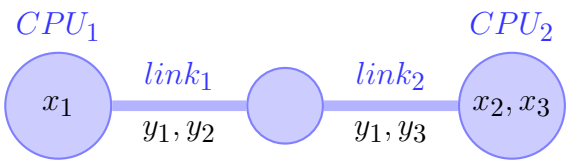


Figure 3.3: Corresponding Platform.

Such linear systems can be solved very easily: (a) Search for the bottleneck resource, that is the one with the smallest ratio $\frac{\text{resource power}}{\text{amount of actions}}$. (b) The sharing of all actions lo-

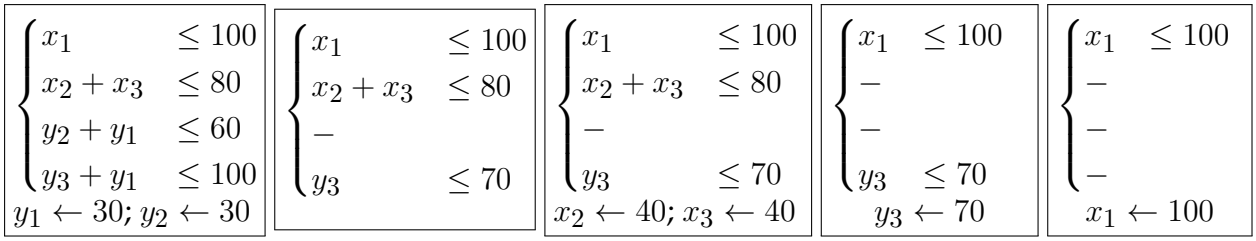


Figure 3.4: Example of Linear System Resolution. (1) Initially, the limiting resource is the third one. This sets the values of y_1 and y_2. (2) These actions are removed and the other equations updated accordingly. (3) The second resource is now the limiting one, setting x_2 and x_3. (4) The fourth resource sets y_3. (5) The first resource sets x_1.

cated on this resource is then set to the value of this ratio. (c) These actions are removed from any other equations in the system, and the share they got is substracted from the available power of the relevant resources. (d) the process iterates to search the next bottleneck resource until all actions' share are set. Figure 3.4 presents an example of system resolution using some arbitrary values as resources' power.

Advanced Features in Surf. This only scratches the surface of Surf, that contains much more subtleties. Arbitrary coefficients can be attached to each variable in the system by the models to improve the modeling quality [VL09b], without major changes to the resolution algorithm. The availability of resources can change according to a trace or to a specified random function. For this, Surf only advances the clock to either the termination date of the next action, or the date of the next trace event, which ever occurs first. The remaining work of unfinished actions can be *lazily* updated, leading to huge performance improvements in decoupled scenarios [DCLV10]. In addition, not all existing SimGrid models rely on such a linear system: the constant model assumes that every actions last the same duration, and thus computes the next finishing action using a simple heap structure; the GTNeTs and NS3 models rely on the corresponding external tools to compute the next finishing action [FC07].

3.1.2 SIMIX: The Virtualization Layer

The main goal of the Simix component is to introduce the concept of *simulated processes* in a portable manner, and allow the inter-processes interactions within the simulated world.

Context Factories. A central constituent of Simix is the *context factories* that allow to create, start, pause, resume, stop and destroy execution contexts. There is one such context per simulated process, executing the user-provided code. These contexts are cooperative threads that may run or may not run in parallel. When running in sequential, *resuming* a context actually yields the execution flow to that context, pausing the current context. Then, *pausing* a context returns the execution flow to the context that previously resumed the paused context.

To day, there is seven different factories, providing this execution context feature with different system-level abstractions. The most common one relies on system threads (be they pthreads or windows threads). Pause and resume are then implemented with two semaphores per context. Full-fledged threads being too feature rich for our usage, we implemented another factory based on System V Unix contexts for sake of efficiency. Pause and resume are implemented using `swapcontext()`. The main advantage of this approach is that the maximal amount of contexts is only constrained by memory, while the system limits the available amount of threads and semaphores. Because the POSIX standard mandates a system call per context switch in order to restore the signal masks, Pierre-Nicolas Clauss implemented another context factory (that we call "raw contexts") during his postdoc with me. Diverging from the standard and adapting the implementation to our use-case leads to important performance improvement. This factory is used by default when available (it is only implemented for the x86 and AMD processors so far).

Other factories are used in the SimGrid bindings to other languages. Indeed, the main difficulty when writing such a binding is to allow SimGrid to take the control over the threads' scheduling in the language runtime. The Ruby bindings (written by Mehdi el Fekari under my supervision) use a safe but somehow inefficient design: the threads of the target runtime are created and controlled in the target language. When the SimGrid kernel needs to pause or resume a context (often in response to a request coming from ruby), it uses an upcall in ruby. The Java bindings (written by Samuel Lepetit under my supervision) avoid these countless traversals of the language boundaries by creating and controlling the Java threads directly from the C without any upcall. This is possible because the JVM maps Java treads onto system threads, and because it provides an API to control this mapping. As shown by the experimental evaluation below, the performance of this factory is comparable to the one of the C thread factory. For sake of performance, we provide another Java factory (also written by Samuel Lepetit under my supervision), approaching the performance of the C contexts. It is more demanding on the users as it requests a non-standard JVM, such as the Da Vinci machine[1] (the user code remains unchanged). This is the only way so far to get the Continuation API that should be integrated in Java 9. This API allows direct context switching in the JVM without relying on full featured threads. Finally, the factory for the lua language is rather simple since there is no notion of thread in this language, but the VM is fully reentrant. We use thus the regular C factory in that case.

Experimental Evaluation. To evaluate the performance offered by each context containers, we ran several Chord scenarios with each implementation. The `pthread` containers prove to be about ten times slower than our custom `raw` 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.

[1] Da Vinci JVM project: `http://openjdk.java.net/projects/mlvm/`.

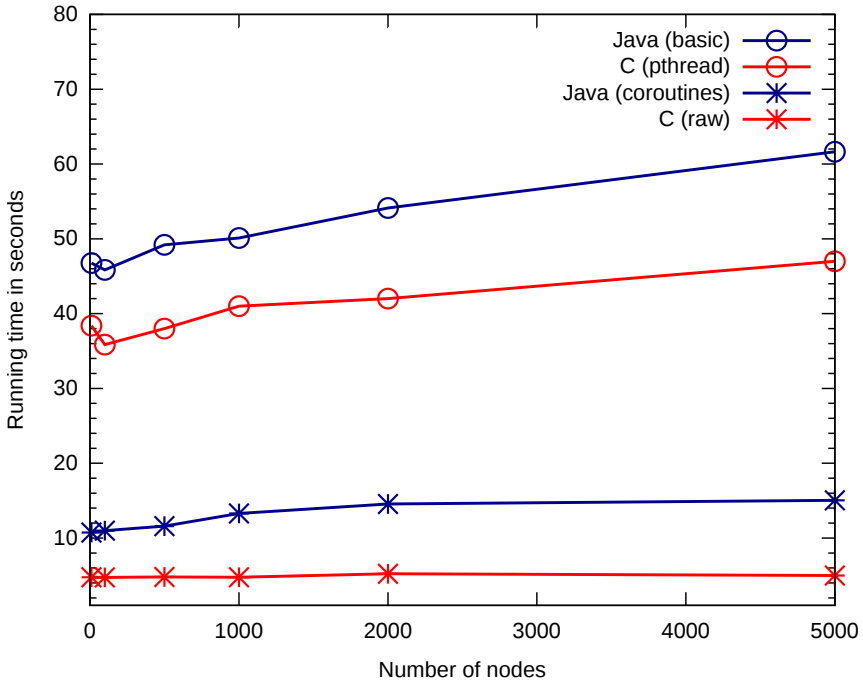


Figure 3.5: Performance Comparison between C and Java.

Figure 3.5 presents a performance comparison of the Java and C contextes. We used a simple master-workers as a benchmark, where the worker dispatches a fixed amount of tasks over a varying amount of workers. C raw contexts are faster than any other solution, and their timing does not depend on the amount of nodes. This is not the case of the pthread contextes, which timing increases with the amount of simulated nodes. Concerning Java, both continuations and regular Java contextes exibit a constant degradation factor: Basic Java contexts are about 30% less efficient than pthreads while coroutines are three times less efficient than raw contextes. Coroutines remains very interesting when considering that they are about four time faster than basic Java contexts.

Inter-Processes' Interactions in Simix. Another role of Simix is to allow the interactions of each process with the simulated environment and with the other processes. Simix models any such interaction as a synchronization object. When for example a process wants to send a message to another process, Simix creates a synchronization object of type *communication*, and then the process use that object to `test` whether the communication is terminated yet (and whether it successfully terminated or failed), or to `block` until the completion of that communication. Conceptually, this is very close to the usage of other, more traditional, synchronization objects such as semaphores, condition variables and mutexes (that Simix also provides). Naturally, a specific API is provided for the creation and handling of these objects, depending on whether they represent communications, computations or I/O requests, but considering them in an unified manner as synchronization objects greatly simplifies both implementation and usage.

As a side note, it should not be assumed that there is a 1:1 mapping between Surf's ac-

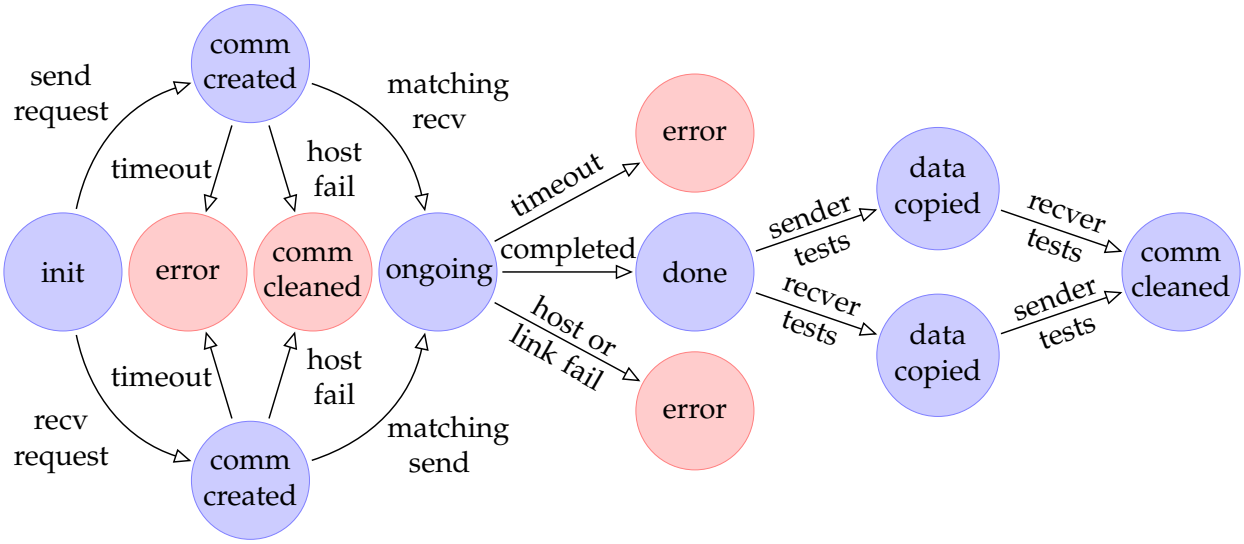


Figure 3.6: Lifespan of a Communication Object in Simix.

tions and Simix's synchronizations. Communication synchronizations for example are naturally associated to a communication action, but also to two *sleep* actions that are placed on the end hosts. They ensure that Simix will be notified if these machines are requested to fail by the experimental scenario (either programatically or through a trace file).

Network Semantics in Simix. It is not sufficient for SimGrid to provide models predicting the duration of each communication. It must predict how the messages are matched beforehand, that is decide which sender is associated to which receiver for each communication. To that extend, Simix provides a generic *Rendez-Vous* mechanism that proved sufficient in practice to encode the semantic of all user interfaces provided by SimGrid (ranging in particular from the BSD sockets-oriented interface provided by GRAS to the MPI interface, where the message matching is controlled by message tags and communicators). To allow the expression of all these semantics, I introduced a concept of *mailbox* that act as a rendez-vous point in Simix. Their semantic is similar to the Amoeba [TKVRB91] transparent port system. When initiating a communication, the sender searches for a potential receiver in such a mailbox. If no matching receive were posted to the mailbox yet, a new send request is created and queued. The symmetric scenario occurs when the receiver post a receive request. Each mailbox is identified by its name, that can be any character string. For example, the BSD socket semantic is captured by using mailboxes' names of the form "<hostname>:<port number>". In addition, filtering functions can be used to further restrict the matching. This could for example allow a receiver to express that it accepts only the messages coming from a specific host. The MPI message tags are expressed as filtering functions.

Once both the sender and receiver are registered in a communication object, it is removed from the mailbox and the simulated communication starts (*i.e.*, the relevant Surf actions are created). Figure 3.6 represents the lifespan of a communication object in Simix.

At any point, the processes can *test* the communication to assess its status (that can be ongoing, done or failed). Processes can also *wait* for the object, *i.e.* block until the communication's completion. Internally, the data is not copied to the sender until one of the processes *test* or *wait* for the object. The sender can also *detach* from the object, indicating that the data should be automatically copied and the object cleaned upon completion. Allowing the receiver to detach from the communication is still to be implemented.

This clean semantic were crucial in the addition of a model checker into SimGrid, as described in §3.4, and is further detailed in this section.

3.1.3 User Interfaces: The Upper Layer

A disconcerting aspect of SimGrid is that it exposes several user interfaces. It may be hard for newcomers to choose the right API matching their needs, but this richness allows to build specific APIs that are particularly adapted to a use case or research community. Disconnecting the internals from the user interface also eases the maintenance of backward compatibility in the API.

The most used interface currently is **MSG**, allowing to study Concurrent Sequential Processes in distributed settings. It was designed by A. Legrand around 2002, as an answer to the itch of its author to study decentralized algorithms in good conditions. This interface targets a good compromise between realism and usage comfort. It does not exposes the full knowledge of the simulated platform to the processes to avoid the temptation of writing an algorithm relying on data that are impossible to gather on real platform, but allow simulated processes to migrate between machines although this feature is very technically challenging to implement in reality (yet not impossible). MSG is also usable from other languages through appropriate bindings: Java, Ruby and lua.

The MSG interface exposed in SimGrid v3.7 (released in June 2012) is still backward compatible with the version introduced in SimGrid v2 ten years before. This continuation in time is very important to us: it allows our users to compare their current developments with prototypes that were written one decade ago. This does not mean that MSG is frozen in the past. We for example recently extended the interface to ease the manipulation of virtual machines and other typical Cloud concepts. It merely means that once a function is exposed in any version of the MSG API, it will be maintained in any future ones. Even deprecated constructs are still present in the recent versions of the interface (provided that a specific configuration option is activated at compilation), although their use is not advised anymore. This feature would be somehow harder to achieve if the kernel interface (surf and simix) were directly exposed to the users.

The spirit of the SimGrid v1 original interface still exists under the name **SimDAG**. It makes it easy to study scheduling algorithms working with direct acyclic graphs (DAGs) of parallel tasks. This interface were removed when transitioning from SimGrid v2 to SimGrid v3, but were later reintroduced and greatly improved by F. Suter.

I introduced **GRAS** (Grid's Reality And Simulation) in 2002 as a way to benefit of the simulator's comfort for the development of applications that can then be deployed on real platforms. This interface is presented in more details in §4.1.1. The main limit of this approach is that it forces users to develop their applications within this system from the

scratch. **SMPI** (Simulated MPI) constitutes a major step to overcome this limitation: it is a reimplementation of the MPI standard on top of the SimGrid kernel, allowing to study the applications developed with this interface through simulation. Introduced by Mark Stillwell during his PhD under H. Casanova's supervision and pushed further during the postdoc of Pierre-Nicolas Claus with me, this environment is detailed in §4.1.2.

This separation between the simulation kernel and the user interface would make it possible to use the kernel in new contexts. It would be possible to design new interfaces such as a theory-oriented interface inspired from the Bulk Synchronous Parallel (BSP) model, or a practical one mimicking the OpenMP interface (just like SMPI mimicks MPI). It would also be possible to use the simulator in new contexts, such as online simulation of real platforms to guide the decisions of runtimes and middlewares such as [CD06].

3.2 Parallel Simulation of Peer-to-Peer Applications

This section presents a work aiming at improving the speed of very large simulations. The main goal is here to reduce the time taken by the simulation on very large simulations such as the typical P2P simulations, targeting millions of processes. To that extend, we strive to allow the parallel simulation of very fine grain, very large scale applications such as P2P protocols. The work presented here builds upon [QRT12] and proposes new improvements.

This section presents a work that I did in 2011 in collaboration with Cristian Rosa during his PhD and Christophe Thiéry during his postdoc with me. A. Legrand and A. Giersch provided very precious feedback during this research. This effort is partially supported by the ANR projects USS-SimGrid (08-SEGI-022) and SONGS (11-INFRA-13).

3.2.1 Motivation and Problem Statement

The motivation for this work comes from the fact that despite the clear importance of scalability in the community, all mainstream P2P simulators (presented in §2.4.1) remain single threaded. This is surprising given the huge activity in Parallel Discrete Event Simulation (PDES) research community for over three decades (see *e.g.* [Liu09] for a survey).

This observation constitutes the starting point of this work. 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.

To the best of our knowledge, the only tools allowing the parallel simulation of P2P protocols are not Discrete-Events Simulators, but Discrete-Time Simulators. As detailed in §2.3.2, this means that at each time step, the simulator executes one event for each simulated process, that are represented as simple state machines. This can trivially be executed in parallel since the execution of each process only involves local data. The authors of

Simulation Workload	• Granularity, Comm. pattern • Evts population, proba. & delay • #simulation objects, #processors
Simulation Engine	• Parallel protocol, if any: Conservative / Optimistic • Event list mgnt, Timing model
Execution Environment	• OS, Prog. Language (C, Java...), Networking Interface (MPI, ...) • Hardware aspects (CPU, net)

Figure 3.7: Performance Factors for PDES [BRR+01, Fig. 2].

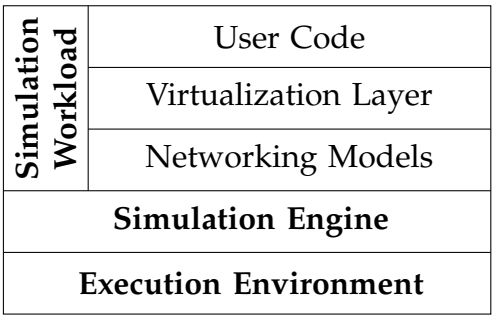


Figure 3.8: Classical DES of Large-Scale Distributed Applications.

PlanetSim [PGSA09] report for example a speedup of 1.3 on two processors while doing so. dPeerSim [DLTM08] 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 [Sto03] 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.

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. 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. Our goal in this section is to demonstrate that these particularities shift the optimization needs.

The classical performance factors of a PDES system are depicted in Figure 3.7, from [BRR+01, Figure 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).

This does however not match our experience with the SimGrid simulator. Our intuition is that most of the time is not spent in the simulation engine, but in the layers built on top of it, that [BRR+01] groups under the term *Simulation Workload*. As depicted in Figure 3.8, we split this in three layers, corresponding to the internal architecture of SimGrid presented in §3.1. The platform models compute the event timings using networking and computing models. On top of it comes a virtualization layer, that executes the user code in separate contexts, and mediates the interactions between this code and the platform models. Such separation also exists in other simulators, even if the boundaries are often

more blurred than in SimGrid. In OverSim for example, the network models are clearly separated from the application under the name *Underlay* [BHK07]. Even if this separation is not clearly marked in a given tool, we believe that the proposed parallelization schema remains applicable, preserving the general applicability of our contributions.

A Novel Parallelization Schema for DES. 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 3.2 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.

$$\sum_{SR}(engine + model + virtu + use) \tag{3.2}$$

The timing resulting from the classical parallelization schema is presented in Equation 3.3. 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 $proto$.

$$\sum_{SR}\left(\max_{LP}(engine + model + virtu + use) + proto\right) \tag{3.3}$$

To be beneficial, the protocol costs must be amortized by the gain of parallelization. According to Figure 3.7, 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. 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 3.4, 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.

$$\sum_{SR}\left(engine + model + \max_{WT}(virtu + use) + sync\right) \tag{3.4}$$

Algorithm 1 Parallel Main Loop.

```
1: t ← 0                          # t: simulated time
2: P_t ← P                        # P_t: ready processes at time t
3: while P_t ≠ ∅ do               # some processes can be run
4:     parallel_schedule(P_t)     # resume all processes in parallel
5:     handle_requests()          # answer their requests sequentially in kernel mode
6:     (t, events) ← models_solve() # find next events
7:     P_t ← processes_to_wake(events)
8: end while
```

3.2.2 Toward an Operating Simulator

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.

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 con-

Figure 3.9: Enabling Parallel Execution with simcalls for Process Isolation.

stant 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 are 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 or *syscalls*) 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.

We implemented a new virtualization layer in SimGrid that emulates a system call interface called **simcalls** (as an analogy to the syscalls of real OSes). In the following, we will use the term *simcall* to designate a call issued by a user process to the simulation core. The term **syscall** 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 *simcall* 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, that is depicted in Figure 3.9. 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.

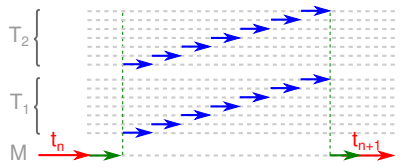


Figure 3.10: Logical View of a parmap.

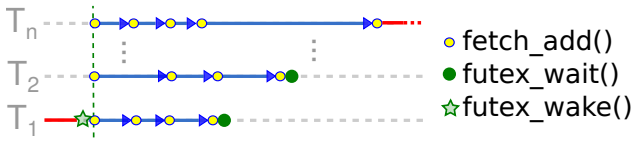


Figure 3.11: Ideal parmap Algorithm.

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, as we shall see in this section.

As detailed in §3.1.2, simulated processes are executed in a way that is conceptually similar to multi-threading. Since system limits make it difficult (at best) to launch millions of regular threads, we rely on cooperative threading facilities (such as swapcontext, specified in Posix, or manually crafted similar solutions) to virtualize the user code. However, these solutions are inherently sequential and don't provide any parallel execution support. They were originally conceived as an evolution of the `setjmp` and `longjmp` functions, not to handle multiple cores or processors.

Our proposal therefore mix the approaches to leverage both the advantages of contexts and the ones of threads. As depicted in Figure 3.10, the user code is virtualized into lightweight contexts to reduce the cost of context switches, and we use a thread pool to spawn the execution of these contexts onto available cores at each simulated round. Since the duration of each user context's execution is impossible to predict, probably very short and possibly very irregular, we opted for a dynamic load balancing: 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 (using the hardware-provided primitive *fetch and add)*) by each working thread when fetching some more work to do.

Our workload onto the thread pool is very specific: we never add work to the pool while the 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. For sake of efficiency, we designed a thread pool variant (named *parmap* – parallel map) that is particularly adapted our case. The parmap provides a unique primitive `apply` where the caller unblocks all worker threads so that the data is processed in parallel. During the parmap execution, the caller thread also processes data with the worker threads to reduce the amount of synchronization. When all jobs are executed, the last terminating thread resumes the calling execution context.

In theory, resuming the caller could be done without any synchronization since the caller is implemented as an execution context, just like the user processes. This would allow the last terminating worker (*i.e.*, the last worker to notice that there is no more jobs available in this execution) to simply switch its execution context to the calling one. This situation is depicted in Figure 3.11: the calling context (in red) is carried by T_1 when starting the parmap execution and by T_n upon completion. In practice, this refinement is still to be implemented in SimGrid. For now, T_1 waits for a specific signal when it is not

the last terminating thread, leading to two more synchronization steps.

The parmap mechanism can easily be implemented using a standard POSIX condition variable that is broadcasted by the calling context to awake the worker threads. This approach reveals portable across the operating systems differences (our implementation were tested on Linux, BSD and Mac OS X). However, maximizing the performance requires to diverge from the standard. 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.

Thanks to this design, a parmap execution only requires N synchronizations (with N being the number of threads involved in the parmap execution), which seems optimal. Under Linux, these synchronizations are achieved using futexes to ensure that at most one syscall is done per synchronization. On other systems, a less efficient Posix implementation is used as a fallback. Within the parmap execution, our *wait-free* approach ensures a load balancing that is as efficient as possible without any prediction of the job durations.

3.2.3 Experimental Evaluation

Material and Methodology. 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 [Sto03] 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 [BCC$^+$06], 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.

The used experimental scenario is the one proposed in [BHK07]: 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

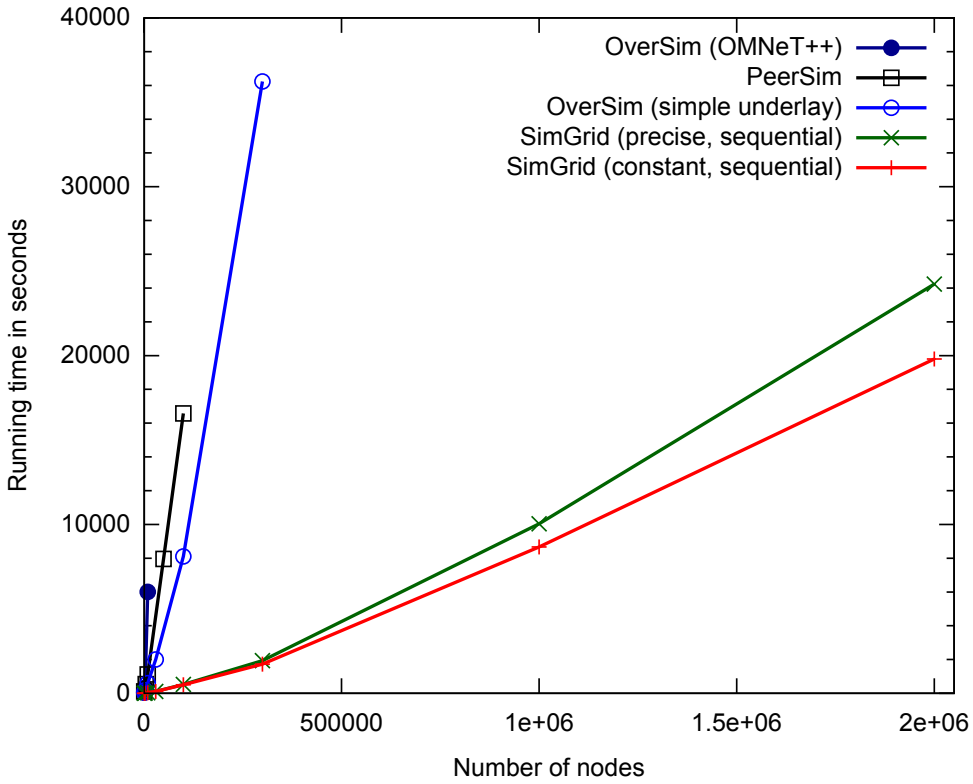


Figure 3.12: Running times of the Chord simulation on SimGrid (with constant and precise network models), on OverSim (with a simple underlay and using the OMNeT++ bindings) and on PeerSim.

messages are exchanged). What we call a message here is a communication between two processes (which may or may not succeed due to timeouts).

The whole experimental settings and data is available at https://github.com/mquinson/simgrid-scalability-XPs/.

Microbenchmarks of Parallelization Costs. The first set of experiments is a microbenchmark aiming at assessing the efficiency of the parmap synchronizations. 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.

Sequential SimGrid Scalability in the State of the Art. Figure 3.12 reports the simulation timing of the Chord scenario as a function of the node amount. It compares the results of the main P2P simulators of the literature: OverSim when using a simple and scalable network underlay, OverSim using its complex but precise network overly based

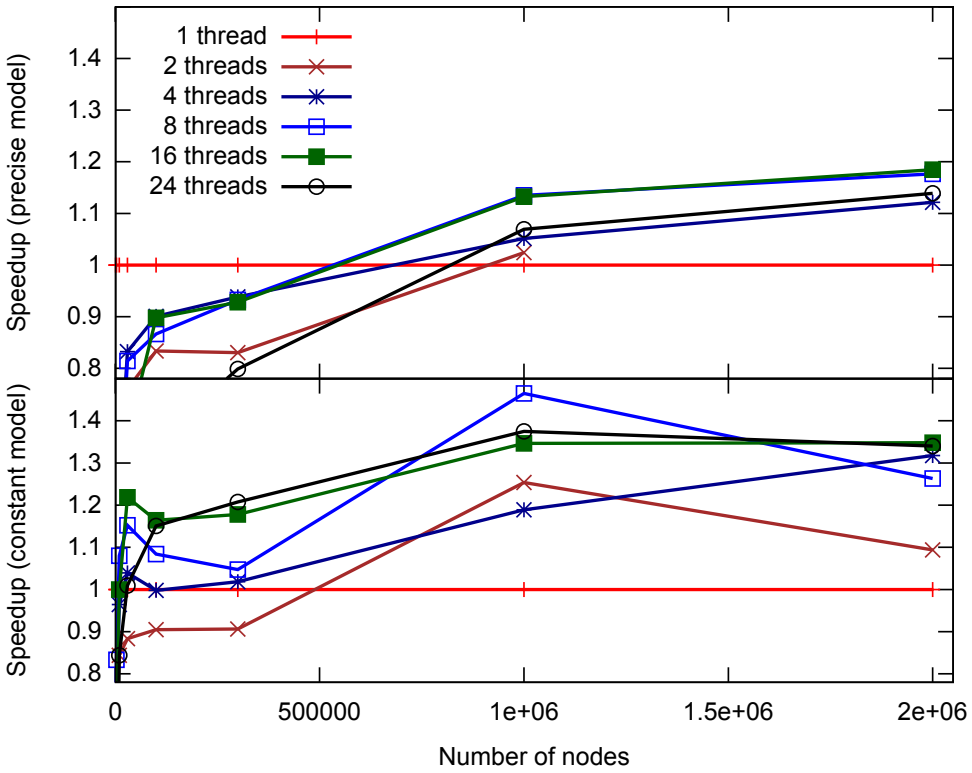


Figure 3.13: Parallel speedups observed for the precise (above) and constant (below) models of SimGrid, as a function of both the system size and the amount of worker threads.

on OMNet++, PeerSim, SimGrid using the precise network model (that accounts for contention, TCP congestion avoidance mechanism and cross traffic – [VL09b]), 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. 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 §3.2.1). 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.

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 3.1: Parallel efficiencies achieved for 2 million nodes.

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 §3.2.1, 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 3.13 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 with 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 performance drop, as the execution lasts 6h. Conversely, increasing the amount of threads beyond 16 threads 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 around 16 workers. This difference in the optimal between models is due to the Amhdal's law.

Table 3.1 presents the parallel efficiency achieved in different settings for 2 million nodes. The parallel efficiency is the ratio $\frac{S}{p}$ 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 mem-

ory 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.

3.2.4 Conclusion and future works

Overall, this work 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 frameworks as well by applying the new parallelization approach that we propose.

The short term future work will focus on the automatic tuning of the working thread amount to reduce the synchronization costs, and on testing our approach on other P2P protocols, possibly involving more SimGrid features (such as the churn). On the longer term, this work could be extended to allow distributed simulations, leveraging the memory of several machines. This was not necessary for P2P simulation since typical P2P processes exhibit very small memory footprints, but this could reveal mandatory for HPC simulations such as the ones presented in §4.1.2. Another execution mode inspired from this work would be to spread the simulated processes in separate Unix processes on the same machine. This could reveal useful in cases such as the ones presented in §4.1.3.

3.3 Scalable Representation of Large-Scale Platforms

This section presents a work aiming at allowing the simulation of very large heterogeneous platforms. The main quality metric is here the size of the platforms that can be simulated, even if we also try to limit the impact on simulation speed. This presentation summarizes [BLD+12].

This work was initiated in 2007 during the internship of Marc Frincu that I co-advised with F. Suter. It continued in collaboration with F. Suter and also A. Legrand, L. Bobelin and P. Navarro. This work culminated in 2010 with the internship of David Marquez that I supervised. This effort is partially supported by the ANR project USS-SimGrid (08-SEGI-022), and by Inria (that co-founded the internships of M. Frincu and D. Marquez).

3.3.1 Motivation and Problem Statement

This work aims at questioning the common belief stating that realistic network models cannot be made scalable. Indeed, it is commonly admitted that when the platform size is at stake, the only applicable models are the delay-based ones while the models accounting for the network contention are limited to much smaller platforms.

	Community	Input	Memory	Network model	Routing
NS-2/3	Network	API	flat	packet-level	static
Omnet++	Network	text	hierarchical	packet-level	static, dynamic
PeerSim	P2P	API	cloud	delay-based	static (direct)
OverSim	P2P	API	cloud	delay-based	static (direct)
SimBA	VC	text	none	delay-based	none
LogGOPSim	HPC	text	cloud	delay-based	static (direct)
GridSim	Grid	API	flat	delay-based	none
OptorSim	Grid	Text file	flat	contention-based	short. path
GroudSim	Grid, Cloud	API	cloud	contention-based	static (direct)
CloudSim	Cloud	Brite, API	flat	delay-based	dynamic (short. path)
SimGrid	LSDC	XML, API	flat	Fluid	static (indirect)

Table 3.2: Simulation tools' answers to Network Concerns.

For example, P2P simulators generally rely on delay-based models and forget about the underlying physical topology. Using this approach, tools such as PeerSim [JMJV] can simulate platforms that scale up to millions of nodes. Yet, such models do not account for network contention, whereas most peers generally sit behind asymmetric DSL lines with very limited bandwidth. Although this kind of assumption may not be harmful when studying simple overlays and investigating the efficiency of look-up operations in a Distributed Hash Table (DHT), the use of such simulators for streaming operations and file sharing protocols is much more controversial. However, the only tools of the P2P literature that do capture the network contention effects (such as narses [GB02]) can only simulate a few hundreds of nodes using flow-based models. The scalability issue comes from the platform representation, as these models compute the completion time of each flow using the complete list of traversed links as an input.

The need for a scalable but accurate network representation is not specific to P2P studies. Many simulators in the High Performance Computing (HPC) community assume that bandwidth has been over-provisioned and that contention can be ignored. Although this assumption may hold true for supercomputers, it may not for commodity clusters or future exa-scale platforms where energy consumption is at stake, which precludes resource over-provisioning.

Problem Statement. When handling larger platforms in a simulator, two related issues arise, that both involve space and time considerations. Table 3.2 summarizes how these issues are dealt with by the state of the art simulators, that were presented in §2.4.

First, any simulator takes a **platform description** as an input. This can be given as a separate file (either XML or not), or only provide a programmatic API to the users. The size of this description depends on the expressiveness of the chosen description format. *Compact descriptions* can leverage any regularity that happen in the platform. For instance, describing a homogeneous cluster or a set of peers whose speed is uniformly distributed does not require to detail each single entity. Such an approach greatly reduces both platform description size and parsing time. On the other hand, *flat descriptions* of all components may be needed to represent complex scenarios breaking these regularity

	Time		Space	
Representation	Parsing	Lookup	Input	Footprint
Flat	N^2	1	N^2	N^2
Dijsktra	$N+E$	$E+N\log N$	$N+E$	$E+N\log N$
Floyd	N^3	1	$N+E$	N^2
Clique	N^2	1	N	N^2
Star	N	1	1	N
Cloud	N	1	N	N

Table 3.3: Θ Complexity of network routing representations.

assumptions. This naturally increases platform description size and parsing time.

Then, the simulator builds a **memory representation** of the platform during the parsing. Flat memory representation are naturally easier to build, either from flat description or by expending compact descriptions. Indeed, even if a set of machines can be described compactly (either using their homogeneity or through a statistical distribution), the simulator still needs to keep track of the activity on every single machine during the simulation. The size of the description would then be $\Theta(1)$ while its memory representation would be $\Omega(N)$. In some cases, the regularity can be preserved leading to a *compact memory representation*. In particular, since the network topology is often hierarchical, our proposal is to leverage this regularity to build modular and a scalable platform memory representation without hindering the expressiveness. It should allow the researchers to adjust the level of details at will while limiting the impact on performance.

Several classical graph representations can be used to encode the simulated platform, each kind of representation presenting a different trade-off between information retrieval time and representation size. Table 3.3 summarizes the time and space costs for most common network representations, considering a set of N nodes interconnected by a general graph with E edges. For all of them, the lookup time is actually in $\Theta(\text{route size})$ and we assume a static routing (although some of these representations support dynamic routing).

In the **flat representation**, each route is completely defined by the set of links belonging to it. It can represent arbitrarily complex platforms and routing at the price of a poor scalability [DDMVB08]. The **Dijkstra graph representation** proposed in [DMVB09] only stores information on shortest paths and enables a better scalability. Shortest path routing is only slightly restrictive since most Internet protocols implement such a routing. Furthermore, this representation trivially allows to model dynamic routing. However, this memory scalability comes at the cost of a lookup time several orders of magnitude larger [DMVB09]. It can be reduced by adding caches, which are completely ineffective in scenarios that have poor locality or involve a very large number of entities. The **Floyd graph representation**, also proposed in [DMVB09], is another way to store information only on shortest paths with different time and space requirements. While this approach reduces the description size and has a very good lookup time, its parsing time and memory footprint are prohibitive. Finally, network graphs, such as clique, star graphs or "clouds" (where each peer is connected to the core of the network where contention is

ignored), exhibit a regularity that can be exploited thanks to an *ad hoc* **local routing table** and a specific routing management.

Each of these network routing representations has its own pros and cons, none is perfectly adapted to every situation. We thus have to provide users the ability to simply and efficiently adapt the representation to their needs. This expressiveness has to match community requirements from fine grain (*e.g.*, router backplane) to coarse grain (*e.g.*, cloud networks). In addition, none of these exploits the hierarchy and the regularity of the platform, calling for an efficient way to combine them efficiently. It should be possible to have differing autonomous systems (AS) using differing routing protocol. Finally, the path lookup and routing computation times should depend as less as possible on the size of the network to ensure a good scalability.

3.3.2 Hierarchical Representation of Heterogeneous Platforms

This section details our proposal for a satisfying network representation, that is scalable, efficient, modular and expressive.

We take advantage of the *hierarchical* structure of current large scale network infrastructures, thanks to the concept of autonomous systems (AS), be they local networks or conform to the classical Internet definition. In addition, we allow users to (recursively) specify platform representation within each AS, which allows us to take advantage of the regular structure of an AS when possible. We propose stock implementations of well known platform routing representations, such as Dijkstra, Dijkstra with cache, Floyd, Flat, and rule-based. This last routing model relies on regular expressions to exploit regular structures. Figure 3.14 shows an example of such a hierarchical network representation.

We assume that the routing remains static over the time for scalability reasons, although we plan to allow dynamic route changes as future work. This assumption is based on studies that have shown that less than 20% of the paths change in a 24 hour period [BMA06]. Moreover, such changes may especially affect load balancing on backbone links, that are usually not communication bottlenecks. Then they can be ignored without any significant impact on simulation accuracy.

Each AS declares one or more gateways, which are used to compute routes between ASes included in an AS of higher level. This mechanism is used to determine routes between hosts belonging to different ASes by looking for the first common ancestor in the hierarchy, and then resolving the path hierarchically (as depicted in Figure 3.15). This thus constitutes a compact and effective representation of hierarchical platforms. In addition, as real platforms are not strictly hierarchical, we also define bypassing rules to manually declare alternate routes between ASes.

In addition to these semantic principles, we also define some syntactical principles. We define the network representation as an XML file, or through a dedicated API. The users can chose to use standard XML editors (and advance features such as autocompletion, validation, and well-formed checking), or programatically declare the platform. We also define a set of tags (resp. functions) that simplify the definition of regular ASes, such as homogeneous compute clusters, or peers. The cluster tag (resp. function) creates a set of homogeneous hosts interconnected through a backbone and sharing a common gateway. The peer tag (resp. function) allows users to easily create P2P overlays

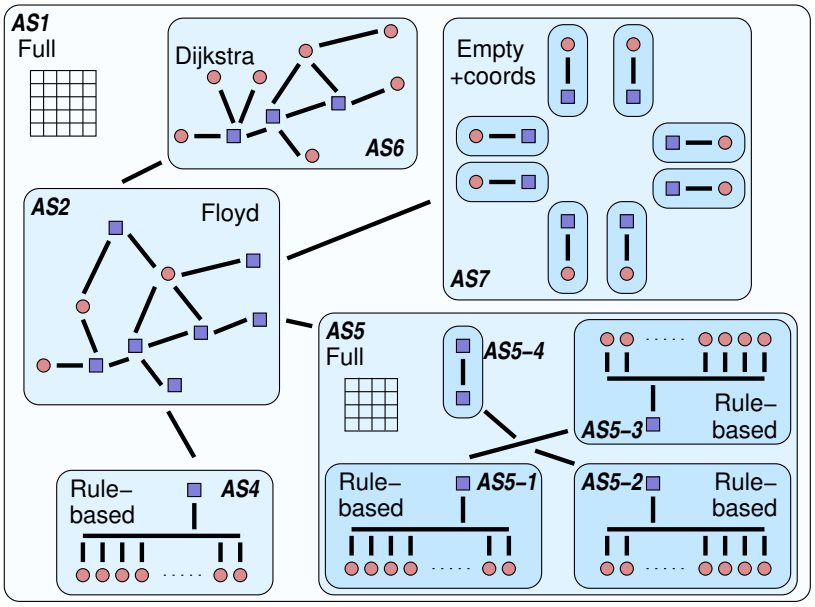


Figure 3.14: Illustration of hierarchical network representation. Circles represent processing units and squares represent network routers. Bold lines represent communication links. AS2 models the core of a national network interconnecting a small flat cluster (AS4) and a larger hierarchical cluster (AS5), a subset of a LAN (AS6), and a set of peers scattered around the world (AS7).

by defining at the same time a host and a connection to the rest of the world (with different upload and download characteristics and network coordinates). This brings both the compactness of coordinate-based models that account for delay heterogeneity and correlation, and the accuracy of fluid models for contention. As such, the Vivaldi [DCKM04] and last-mile [BEDW11] models can be unified.

3.3.3 Experimental Evaluation

We claim that our proposal drastically reduces memory footprint without implying any prohibitive computational overhead. We show that this approach competes in terms of speed and memory usage with state-of-the-art simulators while relying on much more accurate models that are generally considered as prohibitive. For sake of concision, we present throughout results only for HPC and Grid simulations. For both domain, we first define a classical simulation scenario. Then, we simulate this scenario with SimGrid and the corresponding state-of-the-art simulator to evaluate their respective scalability. Elsewhere, SimGrid were demonstrated orders of magnitude faster than state-of-the-art simulators in Volunteer Computing ([DCLV10]) and Peer-to-Peer systems (§3.2.3).

SimGrid v3.7-beta, in which our proposal is integrated, was used. Source code, logs, platform files and analysis R scripts related to the experiments are freely available[2].

[2] https://github.com/mquinson/simgrid-scalability-XPs/

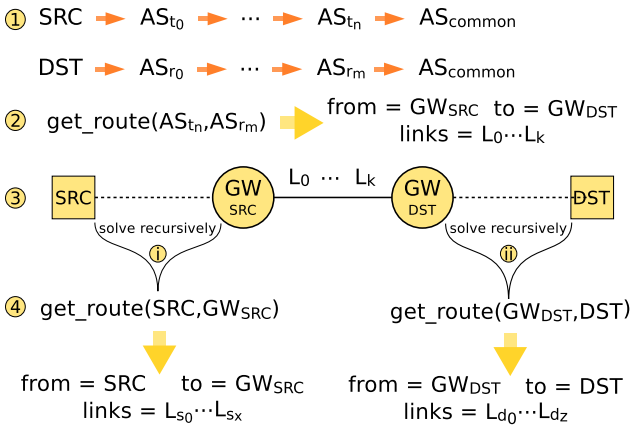


Figure 3.15: Main steps of the hierarchical routing mechanism.

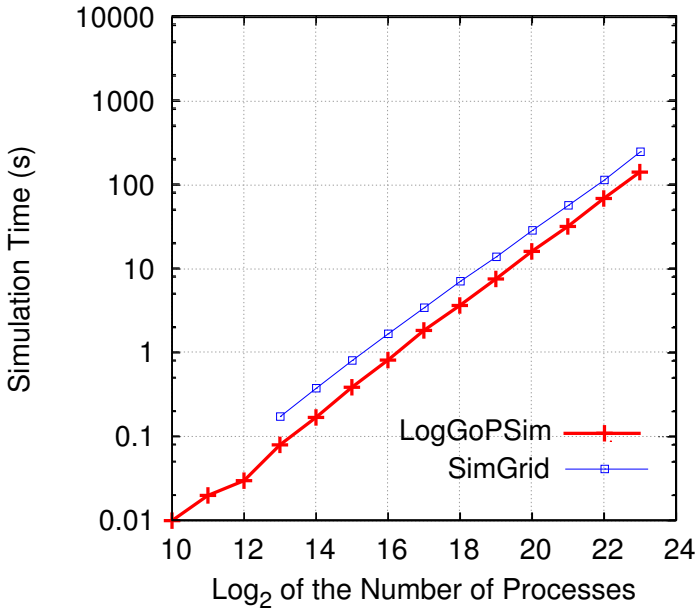


Figure 3.16: LogGOPSim and SimGrid simulation performance for a binomial broadcast.

Input Platform Representation Compactness. As a first evaluation, we note that our formalism allows to represent the Grid'5000 platform [BCC+06] (10 sites, 40 clusters, 1500 nodes) in 22 kiB only, while the flat representation that we had to use in 2007 with SimGrid v3.2 lasted 520 MiB [FQS08]. It took about 20 minutes to parse while the current version parses under the second. We could reel off a whole string of platforms exhibiting a comparable gain of compactness.

Comparing Simulation Performance for HPC scenarios. We now compare our proposal to the results published in [HSL10a]. LogGOPSim is a recent simulator specifically tailored for the study of MPI programs on large-scale HPC systems. It leverages a detailed delay-based model, specifically designed for HPC scenarios. Nevertheless, we show in §4.3.3 the importance of accounting for network contention for the timing accuracy of MPI collective operations involving large amounts of data.

We aimed at comparing our proposal with LogGOPSim in the very same experimental setting used in section 4.1.2 of [HSL10a], i.e., the execution of a binomial broadcast on various the number of processes. Unfortunately, the authors of this work were unable to provide us with the input traces that they used, so we had to compare ourselves to the published results instead of reproducing the experiments with LogGOPSim. The evaluation of LogGOPSim was done on a 1.15 GHz Opteron workstation with 13 GB of memory. We used a single core of a node with two AMD Opteron 6164 HE 12-core CPUs at 1.7 GHz with 48 GB of memory, that we scaled down to 1Ghz for a fair comparison.

Figure 3.16 shows the results. While using significantly more elaborate platform and communication models, and thus producing more meaningful results, SimGrid is only roughly 75% slower than LogGOPSim. This demonstrates that scalability does not necessarily comes at the price of realism (*e.g.*, ignoring contention on the interconnect). The price of SimGrid's versatility is a slight memory overhead since our memory usage for 2^{23} processors is 15 GiB, which is larger than what is achieved in [HSL10a] (as they tested on a 13 GiB host). Yet, we think this loss is reasonable and amply offset by the gain in flexibility.

Comparing Simulation Performance for Grid Computing scenarios. We now compare our proposal to the widely used GridSim toolkit [SCV+08] (version 5.2 released on Nov. 25, 2010). The experimental scenario consists in a simple master worker setting where the master distributes N fixed size jobs to P workers in a round-robin way. In GridSim, we did not define any network topology, hence only the output and input baud rate are used to determine transfer speed. For SimGrid, we use a model of the Grid'5000 mentioned earlier. It models each cabinet of the clusters as well as the core of the wide area network interconnecting the different sites. The experiments were conducted using an Intel Xeon Quadcore 2.40GHz with 8GiB of RAM.

The number of tasks n_{task} to distribute was uniformly sampled in the $[1; 500,000]$ interval and the number of workers W was uniformly sampled in the $[100; 2,000]$ (GridSim cannot run for more than 10,000 hosts as already mentioned in [DDMVB08]). We performed 139 such experiments for GridSim and 1000 for SimGrid, and measured the wallclock time T (in seconds) and the memory consumption M (Maximum Resident Set Size in kiB). As expected, the size (input, output and amount of computation) of the tasks have no influence. These experiments prove that $T_{GridSim}$ is quadratic in n_{task} and linear in W ($R^2 = 0.9871$):

$$T_{GridSim} \approx 5.599\,10^{-2}W + 1.405\,10^{-8}n_{task}^2$$

Surprisingly, the memory footprint is not a simple polynom in W and n_{task}. It seems to be piecewise linear in both W and n_{task} (with a very steep slope at first and then a less steep one). Furthermore there are a few outstanding points exhibiting particularly low or high memory usage. This can be probably explained by the Java garbage collection. Hence, we only report the behavior for the situation where the number of tasks is larger than 200,000 and the slope is not steep, taking care of removing a few outliers (R^2=0.9972):

$$M_{GridSim} \approx 2.457\,10^{6} + 226.6W + 3.11n_{task}$$

Conducting the same analysis for SimGrid shows that it is much more stable and requires a much smaller time and memory footprint (R^2 equal to 0.9984 and 1 respectively):

$$T_{SimGrid} \approx 1.021\,10^{-4}W + 2.588\,10^{-5}n_{task}$$
$$M_{SimGrid} \approx 5188 + 79.9W$$

This means that 5.2Mb are required to represent the Grid 5000 platform and the internals of SimGrid while we have a payload of 80K per worker (we used the default settings for the processes' stack size). Last, there is no visible dependency on n_{task}.

This clearly indicate that SimGrid (with a flow-based network model and a very detailed network topology) is several orders of magnitude faster and smaller in memory than GridSim (with a delay-based model and no network topology). To illustrate the time and memory scalability, the previous regressions indicate that GridSim requires more than an hour and 4.4 GB to dispatch 500,000 tasks between 2,000 processes while SimGrid requires less than 14 seconds and 165MB for the same scenario.

3.3.4 Conclusion and future works

This demonstrates that the widespread beliefs that scalable simulations necessarily imply simplistic network models are erroneous. Our implementation within SimGrid does not trade accuracy and meaning for scalability and also allows its users to simulate complex applications through a hierarchical representation of the platform that proves to be expressive, modular, scalable and efficient. It also allows us to combine different kinds of platforms, *e.g.*, a computing grid and a P2P system, to broaden the range of simulation possibilities. Our experiments showed that our approach is far more scalable than what is done by state-of-the-art simulators from any of the targeted research communities.

Within the SONGS ANR project, we are currently working on at considering the specifics of emerging systems such as IaaS Clouds. In addition, the student internship of J.-B. Hervé that I currently advise explores on further reductions of platform description size (and hence parsing time) and memory footprint by exploiting stochastic regularity when available and by improving the programmable description approach. We also plan to allow dynamic changes to the routing in future work.

3.4 Dynamic Verification of Distributed Applications

This section presents a work aiming at adding formal methods to the SimGrid framework. The goal is to allow the dynamic verification of unmodified SimGrid applications through model checking. This summarizes and extends [RMQ10, MQR11].

This work started in 2007 with the internship of C. Rosa that I co-advised with S. Merz. It continued during the PhD of C. Rosa, that I also co-advised with S. Merz. It is now followed by the PhD of M. Guthmuller that I co-advise with S. Contassot-Vivier. This effort is partially supported by the ANR project USS-SimGrid (08-SEGI-022), and by Inria (that co-founded the internship of C. Rosa).

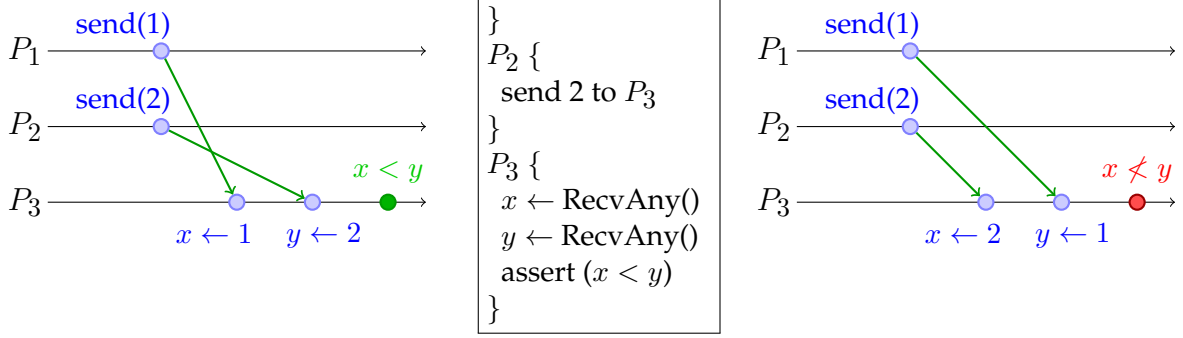


Figure 3.17: Example of Bugged Distributed Program. The execution depicted on the left is valid, while the assertion is violated in the right one.

3.4.1 Formal Verification of SimGrid Applications

The motivation to add a model checker to the SimGrid framework is twofold. First, it seems perfectly *sensible* to evaluate the correctness of the algorithms in the same way than their performance, since fast and faulty algorithms are only marginally better than slow and faulty ones. Then, it is *possible*, as the correctness-oriented model checkers and performance-oriented simulators share many constituent functionalities. The research communities are however completely distinct, and I am very thankful to S. Merz, who accepted to guide me in this new thematic.

SimGrid is intended to simulate distributed programs in a controlled environment. In particular, it manages the control state, memory, and communication requests of simulated processes, which can be run selectively and interrupted at visible (communication) actions. We used these building blocks to implement verification capabilities. Informally, this is a sort of exhaustive simulation assessing the application behavior on any possible platform. This *demonstrates* the tested properties, provided that the state space is exhaustively explored. Most of the time however, this sound exploration is particularly hard to achieve because of the state space explosion (see §2.5). Even if the state space is not exhaustively explored, model checkers can turn into powerful bug finding tools, thanks to their ability to efficiently scout the realms of possibility for faulty executions.

Problem Statement. Our goal is first to integrate an explicit-state model checker within SimGrid, allowing a user to verify safety properties over instances of distributed systems without requiring any modifications to the program code. This will be extended in subsequent sections to introduce reduction techniques to mitigate the state space explosion, and to tentatively verify liveness properties.

Figure 3.17 presents an example of system that we want to verify with SimGrid. Two processes P_1 and P_2 send an information to a third process P_3. We use a bogus assertion stating that the ordering of messages is fixed, with the message of P_1 arriving before the one of P_2. This assertion is naturally wrong.

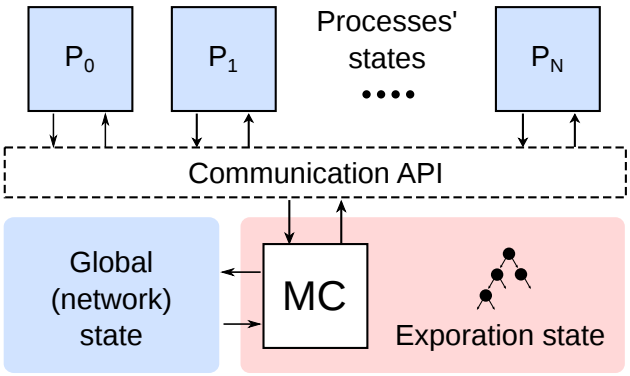


Figure 3.18: SimGrid MC Architecture.

The current implementation is limited to the C API of SimGrid. The model checker (MC) replaces Surf and some parts of Simix (see §3.1) with a state exploration algorithm that exhaustively explores the outcomes of all possible non-deterministic choices of the application. In practice, on each decision point where we intercepted a non-deterministic action of the application, we need to checkpoint the system state, explore a first possible outcome, and rollback the system state to explore the other possible outcomes.

Capturing the Indeterminism. As explained in §3.1.2, a distributed system in SimGrid consists in a set of processes that execute asynchronously in separate address spaces, and that interact only by exchanging messages. Although it is possible to partially represent them in SimGrid, we do not consider here the case of multithreaded applications that interact through shared memory. Any non-determinism thus comes from the network, acting upon the message receive order. In addition, a specific `rand()` function is introduced for the applications that introduce randomness at the application level.

The process isolation introduced in §3.2 for parallel simulation reveal particularly useful here: the process can only interact with the environment through *simcalls*, that constitute perfect interception points for the model checker. This adequacy of an OS-inspired design to both the parallel simulation and the model checking is very fortunate, but it is not a complete surprise, as we introduced it specifically with these two use cases in mind.

The only transition visible to the model checker are the process executions between subsequent simcalls. It comprises the modification of the shared state achieved by the simcall handler in Simix, followed by all the internal computations of the process until the next simcall. The state space is then generated by the different interleavings of these transitions; it is generally infinite even for a bounded number of processes due to the unconstrained effects on the memory and the operations processes perform.

Manipulating the System State. The system global state consists of the state of each process (determined by its CPU registers, stack, and allocated heap memory) plus the network state (given by the messages in transit). Because the global state contains unstructured heaps, and the transition relation is determined by the execution of C program

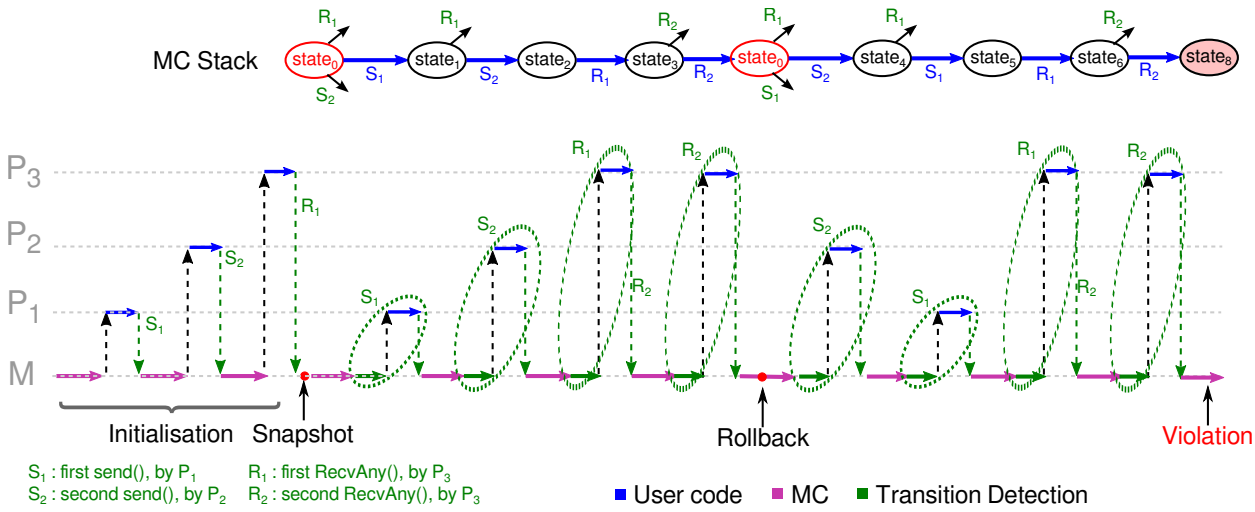


Figure 3.19: State Exploration of the code presented in Fig. 3.17.

code, it is impractical to represent the state space or the transition relation symbolically. Instead, we explicitly explore the system transitions by actually scheduling the processes.

Rollbacking the application mandates to separate its state from the model checker state, even if both modules live in the same UNIX process. For that, we use a specific malloc implementation allowing the model checker to have a separate heap, located in a specific memory segment.

There is a well known trade-off between the memory requirements of the checkpoints and the time to rollback [God97]. Our tool is state less by default, taking a unique checkpoint at the beginning of the simulation. To rollback to a subsequent execution point, it first rollbacks to the time origin and replays the schedule leading to the wanted point. The user can configure to take a checkpoint every N steps if the situation mandates it.

Figure 3.18 illustrates the resulting architecture. Each solid box labeled P_i represents a thread executing the code of a process in the distributed system being verified. The exploration algorithm is executed by a particular thread labeled MC that intercepts the calls to the communication API (dashed box) and updates the state of the (simulated) communication network. The areas colored blue represent the system being explored, the area colored red corresponds to the state of the model checker, which holds the snapshot of the initial state and the exploration stack. When a backtracking point is reached, the blue area is reset as described above, but the exploration history is preserved intact.

Exploration Algorithm. SimGrid MC is an explicit-state model checker that explores the state space by systematically interleaving process executions in depth-first order, storing a stack that represents the schedule history. As the state space may be infinite, the exploration is cut off when a user-specified execution depth is reached. That bound naturally compromises the exploration soundness if reached in practice, but this limitation is inherent to explicit-state model checking, as explained in §2.5, page 15. The tool can still be used as a bug finding tool in such situations, ensuring a complete exploration of the state space up to the search bound. This bound was not reached in the experiments of §3.4.3.

Figure 3.19 illustrates our exploration technique the example presented by Figure 3.17. During its initialization, the model checker executes the code of all processes up to, but excluding, their first simcall ($send$ for P_1 and P_2 and $recv$ for P_3 in this example). The resulting global state $state_0$ is pushed on the exploration stack (depicted on top of Figure 3.19); A snapshot of this initial state is also taken. Then the exploration begins. The model checker selects one of the enabled transition (S_1, S_2, R_1 – it picks S_1). The corresponding user process is scheduled and performs the communication action and all following local computations up to, but excluding, the next simcall (here, P_1 terminates). This corresponds to one transition as considered by the model checker. Afterward, it continues with another enabled transitions (say, S_2), proceeding until the exploration reaches the depth bound or no more actions are enabled; depending on the processes' state, the latter situation corresponds either to a deadlock or to program termination. Afterward, the model checker backtracks to initial snapshot and replays the previously considered execution until it reaches the global state that must be further explored. The continues until the complete exploration of the state space, or until an invalid state is discovered.

3.4.2 Partial Order Reduction for Multiple Communication APIs

(Dynamic) Partial-Order Reduction [FG05] has proved to be efficient mitigate the state explosion problem by avoiding the exploration of equivalent interleavings. This is particularly adapted to distributed systems that lack any global state, as any event only directly affect a limited amount of sites. We thus leverage this approach in SimGrid MC. Our algorithm, detailed in [MQR11], is somewhat simpler than the original presentation of DPOR [FG05] because the transitions remain enabled until they execute in SimGrid.

The quality of the DPOR reduction is directly given by the precision of the dependency relation. False positive lead to a conservative exploration of redundant executions while false negative endanger the exploration soundness as it may miss some possible executions. Unfortunately, precisely determining the (in)dependence of two given transitions can be costly, as it involves evaluating their exact effects in either order *for any given reachable state*. In practice, this relation is approximated in practice, assuming that two transitions are dependent unless they can be proved independent.

Most of the actual communication APIs used to specify distributed systems (such as the user interfaces of SimGrid) were not designed for formal reasoning. As a consequence, they lack any formal specification of their semantic that could be used to formally assess the actions' (in)dependence Palmer et al. [PGK07] have given a formal semantics of a substantial part of MPI for use with DPOR, consisting in is more than 100 pages of TLA$^+$ specification [Lam02]. In addition, this tedious and daunting task would have to be repeated to cover other APIs such as MSG in SimGrid.

Instead, we specified the Simix networking interface in TLA$^+$, and computed sufficient condition of independence for each pair of primitives. Thanks to the extreme concision of this interface, our complete specification is only a few pages long. All user interfaces of SimGrid being implemented on top of this interface[3], our model checker operates

[3]SimDag is not implemented on top of Simix and thus cannot be model checked currently. On the other hand, this interface leaves any message ordering issues to the user,

```
1 if (rank == 0){
2   for (i=0; i < N-1; i++)
3     MPI_Recv(&val, MPI_ANY_SOURCE);
4   MC_assert(val == N);
5 } else {
6   MPI_Send(&rank, 0);
7 }
```

Figure 3.20: MPI implementation of the test code depicted in Figure 3.17.

at this elementary level with no loss of generality.

As presented in §3.1.2 (page 32), processes willing to communicate queue their requests in rendez-vous points called *mailboxes*. The actual communication takes place when a matching pair is found. The API provides just the four operations $Send$, $Recv$, $WaitAny$ and $TestAny$. The first two post a send or receive request into a mailbox, returning a communication identifier. A $Send$ matches any $Recv$ for the same mailbox, and vice versa. The operation $WaitAny$ takes as argument a set of communication identifiers and blocks until one of them has been completed. $TestAny$ checks whether any of the provided communications has completed, and returns a Boolean result without blocking.

We found these conditions to be sufficient for independence (see [RMQ10] for the proofs):

- *Any two* Send *and* Recv *transitions are independent.* They are trivially independent if they concern separate mailboxes. In the other case, the order in which they are declared in the mailbox is irrelevant, as depicted in Figure 3.6.
- *Two* Send *or two* Recv *posted to different mailboxes by different processes are independent.*
- Wait *or* Test *operations for the same communication request are independent.*
- *Local operations of different processes are independent.*
- *Any two* Local *and* Send *or* Recv *transitions are independent.*
- *Any two* Local *and* Wait *or* Test *transitions are independent.*

3.4.3 Experimental Evaluation

We now present some experiments using two of the APIs supported by SimGrid. We thus illustrate the ability of our approach to use a generic DPOR exploration algorithm for different communication APIs through an intermediate communication layer. Each experiment aims to evaluate the effectiveness of the DPOR exploration at this lower level of abstraction compared to a simple DFS exploration. We use a depth bound fixed at 1000 transitions (which was never reached in these experiments), and run SimGrid SVN revision 9888 on a CPU Intel Core2 Duo T7200 2.0GHz with 1GB of RAM under Linux.

as it does not provide any process concept *per se*.

#P	Finding the assertion violation						Complete state space coverage					
	DFS			DPOR			DFS			DPOR		
	states	time	RSS	states	time	RSS	states	time	RSS	states	time	RSS
3	119	0.10 s	24 MB	43	0.06 s	24 MB	520	0.2 s	23 MB	72	0.07 s	23 MB
4	123	0.11 s	25 MB	47	0.06 s	25 MB	61k	19 s	24 MB	3.4k	0.9 s	24 MB
5	127	0.11 s	26 MB	51	0.07 s	26 MB	-	-	-	300k	84 s	25 MB

Table 3.4: Number of expanded states, timing and peak memory usage to find the assertion violation (left) and for complete state space coverage (right).

SMPI Experiments. The first experiment aims at assessing the performance of our DPOR algorithm. It relies on the same example as before, implemented with the MPI interface and generalized to run with a variable amount of processes (see Figure 3.20). Any process but rank 0 send their rank to the root, that receive all incoming messages and asserts that the last received message was sent by the last process.

Table 3.4 some performance results of standard the depth-first search exploration without any reduction (noted DFS) and of our implementation of DPOR. The performance metrics are the number of states, the timing and the peak memory occupation (RSS). On the left, the exploration is conducted only up to the point where the violation is found. In this case, the number of processes does not have a significant impact on the number of visited states because the error state appears early in the visiting order of the DFS. Still, DPOR reduces the number of visited states by more than 50%. On the right, the performance for a complete state space exploration of the same program (without the assertion). Here, the use of DPOR reduces the number of visited states by an order of magnitude, allowing to verify the program for $N = 5$ in 1.5 minute while we had to interrupt DFS after one hour of computation.

MSG Experiment: CHORD. Our second case study is based on an implementation of the Chord protocol [Sto03] using the MSG communication API. In contrary to the previous example, this is no artificial example in the sense that we did not intentionally write the bug that we found through model checking. Instead, we wrote this example for performance experiments presented in §3.2, in particular for Figure 3.12.

Running our initial implementation in the simulator, we occasionally spotted a memory error leading to a segmentation fault. Due to the scheduling produced by the simulator, the problem only appeared when running simulations with more than 90 nodes. Although we thus knew that the problem came from our code, we were unable to identify the cause of the error because of the size of the instances where it appeared and the amount of debugging information that they generated.

We decided to leverage the model checking to investigate the issue, exploring a scenario with just two nodes. We checked a simple property that we believed to be true and that would have made the memory error impossible. In a matter of seconds we were able to trigger the bug and could understand the source of the problem by examining the counter-example trace. Our error was to reuse a variable in a given code branch, incorrectly assuming this to be safe because of the guard of that branch, but in fact the

condition may change after the guard is evaluated.

The verified code has 563 lines, and the model checker found the bug after visiting just 478 states (in 0.280 s) using DPOR; without DPOR it had to compute 15600 states (requiring 24 s) for that. Both runs had an approximate peak memory usage of 72 MB.

3.4.4 Conclusions and Future Work

We successfully integrated a model checker for distributed C programs into the SimGrid framework. This was eased by the fact that simulation and model checking share core functionality such as the virtualization of the execution environment and the ability to execute and interrupt user processes. This integration allows to use the same code and the same framework for verification and for performance evaluation.

The resulting model checker is stateless by default, but allows the user to manually configure the trade-off between the memory occupation of the checkpoints and the time to recompute the state on rollback. Another specificity is the support for multiple communication APIs, as it is usable from both the MSG and MPI SimGrid interfaces. For that, the model checker works directly in the Simix module, which bases the user interfaces. The concision of Simix's API eased the formal specification of its semantic, without compromising the performance of our reduction technique.

An unexpected outcome of this work is that it led us to better organize the simulator internal architecture, following a classical OS design (*c.f.* §3.1). It also improved our understanding of the semantics, benefiting to the work on parallel simulation presented in §3.2. It is notable that applying OS design ideas to a distributed system simulator enabled us to both integrate a model checker and to run efficient parallel simulation.

In the PhD of M. Guthmuller, we currently extend this work to permit the verification of liveness properties in addition to safety ones currently possible. This mandates to detect cycles in the exploration, which in turn requires to store some of the visited states and to detect state equality denoting that the execution cycles to a previous state. To allow the introspection on the state of arbitrary C applications, we leverage the meta-data of a specifically tailored malloc reimplementation. After this OS difficulty, we will face a major theoretical challenge as basic DPOR techniques may break or add cycles to the state space, preventing their use for liveness properties. We plan to leverage the classical ideas of transition invisibility and stuttering equivalence; To the best of our knowledge, this was never implemented for real software, only for models, but we hope to gather the mandated information about the current state through the Dwarf meta-data, that allow the debugger to link memory addresses to symbol names from the source code at runtime. We could also use static information, for example gathered through the LLVM framework.

The expressiveness of the communication API could also be further improved. SMPI presents a communication semantic bug, as a receive is matched to the send that were posted first in the Simix mailbox. In real implementation however, it is matched with the message that arrives first on the destination site. Modeling this will require to add new primitives to communication synchronization object. Similarly, one can currently only wait for the communication completion, while one could want to wait to until the other

peer connects to the communication, or until the communication starts, or until a given completion ratio. The dependence predicate will need to be adapted to these refinements.

Another similar lead would be to build upon [VAG+10], where the authors complete their independence detection with a simple vector clock algorithm to test only mandatory ordering of indeterministic RecvAny() calls with deterministic RecvFrom().

Finally, we would like to explore the possibility to express other interfaces on top of Simix, to open new vistas to our model checker. A first candidate could be the ∅MQ [Hin07] interface for modern, loosely coupled distributed applications. A much more generic solution would be to leverage the work presented in §4.1.3 to model check the applications directly at the system level using the send/recv API.

3.5 Conclusion

In this chapter, I presented several works on the design and implementation of a simulator of distributed platforms. I discussed how to make this framework fast, scalable, usable for both performance and correction studies. The result of these works were implemented in the SimGrid framework, that became one of the world leading tool for this kind of studies. A lot of engineering work were necessary to make these work possible. I see these implementations as a validation mean rather than the primary goal of the work. I thus believe that some sizable parts of the work presented in this section will partially interesting in a few years, even when the SimGrid technical realization will be obsolete.

All these works open many research leads. All sections conclude with ideas that would be interesting to investigate. A specific concern is that all existing SimGrid models were written by us. Instead, the users should appropriate this element too to fit the models to their studies. This mandates specific kernel adaptations to make this happen.

As a concluding remark on the engineering aspects of the SimGrid framework, it is interesting to note that SimGrid is now very close to an operating system: we provide task containers and control their scheduling (according to the models). We reimplemented the malloc memory management system to adapt the provided meta-data to our needs. We naturally provide the inter-process communication means as well as intra-process synchronization mechanisms. All together, this makes me wonder of what benefits we could get by executing SimGrid simulation as a real operating system, thus replacing Linux by our code completely. Given the already implemented mechanisms, this would not require an insane amount of development to work. And this would certainly improve the simulation scalability even further. By mediating the memory through MMU, we could relocate the processes in memory on need, allowing for growable stacks.

Actually, I am not sure that we need to push the limits further and remove Linux, provided that SimGrid is already orders of magnitude faster and more scalable than comparable simulators. Nevertheless, this still occurs to me as a funny and enlightening idea for the future. Operating Systems' Design is an endless source of inspiration for the architecture of SimGrid's internals.

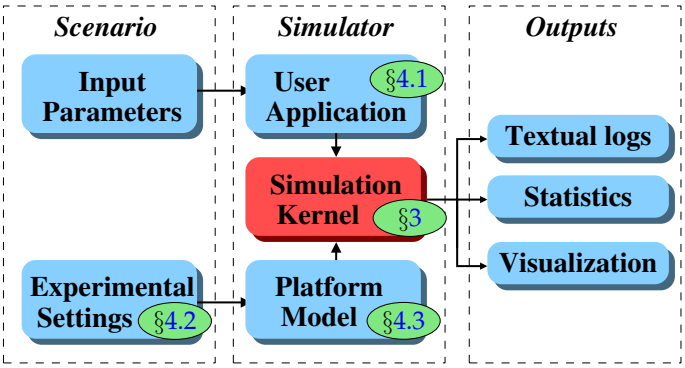
More importantly, this work on the simulation kernel actually highlights that this element is not sufficient for effective scientific studies. Many other tools must be associated to the simulation kernel to that extend, as we will investigate in the next chapter.

Chapter 4

Beyond Simulation

Essentially, all models are wrong,
but some are useful.
– George E. P. Box.

TYPICAL simulation studies involve the animation of the models of both the platform and the application through the simulation kernel in order to get statistics and other outputs. This chapter presents my work around the simulation kernel.



A first concern of the experimenters is to model and describe the algorithms or applications that they want to study within the simulator. The scope of simulation studies is thus limited to the applications that can be expressed to the simulation framework. For that, §4.1 contrasts several attempts at extending the application field of the simulator, be it by allowing to take simulation prototypes out to real platforms, or by allowing the simulation of unchanged real applications. Another major concern of the experimenters is to come up with the right experimental settings for their studies. §4.2 introduces a methodology model real platforms by to gathering performance information on them so that they can be simulated. Finally, the model's accuracy directly impact the domain of application and the *trust* that scientists can put into their experimental tools. §4.3 presents an ongoing effort to model the performance of MPI middleware to improve the simulation accuracy.

This coherent vision of how simulation studies should be organized is the result of fruitful collaborations and interactions with numerous colleagues, in particular F. Suter and A. Legrand and the other contributors to the SimGrid framework.

4.1 New Vistas for the Simulation

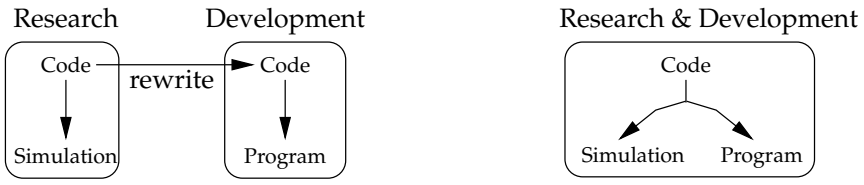
This section presents several efforts aiming at extending the scope of the simulation studies. Since day 1 of my implication in SimGrid, I try to study real applications within the simulator. §4.1.1 presents my first attempt doing so: GRAS (Grid Reality And Simulation) aims at allowing the development of real applications within the simulator before their deployment on real platforms. The main issue with this approach is that it locks the users into the GRAS API. In order to increase the potential user base, we started the SMPI project (simulated MPI). It enables the study of legacy MPI applications through simulation. This project raises both engineering questions (*e.g.*, to allow the folding onto a single node of programs intended to be executed in distributed settings), and modeling issues (because these applications are typically highly optimized, mandating a relatively high accuracy on relatively highly detailed simulations). The questions arising to *enable* the simulation of MPI applications are tackled in §4.1.2 with the modeling issues are postponed to §4.3. Finally, §4.1.3 presents a recent work aiming at virtualizing any application to allow its execution within the simulator.

4.1.1 Taking Prototypes Out of the Simulator

Ten years ago, I came to SimGrid to study the algorithms underlying a complex distributed infrastructure. This was part of my doctoral work, that aimed at developing a distributed infrastructure able to automatically discover both the qualitative characteristics and the quantitative availability of the underlying computing platform. My solution for the availability were extending the Network Weather Service [WSH99] to some extends, but the discovery of platform characteristics were more challenging. In particular, I had to develop a novel network mapping algorithm (presented in §4.2) not only aiming at gathering topological information about the network, but also application-level performance metrics.

One of my main concern in this work was methodological: I wanted to develop a generic algorithm and not only something limited to the discovery of the platforms that I had access to. That is why I decided to conduct large evaluation studies within the simulator, to maximize my control over the experimental settings and to ensure that the solution remains generic. But on the other hand, the expected outcome of my work was a usable distributed infrastructure so studying the algorithms within the simulator was not sufficient. These thoughts gave birth to the GRAS project (Grid Reality And Simulation). This section presents this work, that took place between 2003 and 2006, and that was first published in [Qui06].

The GRAS project. This framework aims at easing the development of distributed event-oriented applications. As depicted in Figure 4.1, the **main big idea of GRAS** is to allow the same unmodified code to run both on top of a simulator and on real distributed platforms, using two specific implementations of its API. This approach (dubbed *develop once, execute twice*) let the developers benefit from the ease-of-use and control of the simulator during most stages of development cycle while seamlessly producing efficient real-life-enabled code. The same approach was used in subsequent work such as [Bri08]



(a) Classical approach. (b) GRAS approach.

Figure 4.1: GRAS big idea.

for the rapid and easy development of dedicated infrastructures. To make this idea real, GRAS provides both an *easy to use interface* and an *efficient execution framework*.

The **provided interface** was designed build distributed infrastructures offering a specific service to large-scale distributed applications and middlewares. For instance, GRAS was intended to build grid computational servers comparable to NETSOLVE [CD97], platform monitoring sensors like the NWS ones [WSH99] or Distributed Hash Tables like Pastry [RD01]. Such infrastructures are constituted of several entities dispatched on the various hosts of the platform and collaborating with each other using some specific application-level protocol. The primarily targeted applications were thus loosely coupled collections of communicating processes using an application-level protocol.

These applications are naturally described in an *event-driven* fashion instead of the *SPMD* model provided *e.g.*, by MPI. The GRAS framework provides a high level message passing interface allowing *agents* to exchange *active messages* over BSD-like sockets. The application semantic was carried by the message type, with agents either declaring automatic callbacks to these messages, or explicitly waiting for them. GRAS also provided RPC-like messages. The payload of these message is automatically marshaled and transferred over the wire.

Since the main goal of the GRAS framework is to allow the development of efficient real applications, the provided **execution framework** was optimized for efficiency. Since GRAS does not interfere with the computation and storage facilities and because of the distributed settings of the targeted applications, the communication layer deserves a lot of attention. The *Native Data Representation* (NDR) constitutes an efficient data representation first demonstrated by PBIO [EBS02] and used in GRAS. Data structures are sent as they are represented in memory on the sender side. If the receiver architecture matches the sender one, the data can be placed in memory without any analysis, completely avoiding the encoding costs. When architectures do not match, the receiver converts the remote data representation to the local one. Any valid C type can be used as payload, including structures and pointers. The datatype format can be parsed directly from the C structure definition automatically in most cases.

To run the same code both on top of a real platform and in simulation mode, GRAS acts as a **virtualization layer** of the operating system and provides explicit system call wrappers. Indeed, `time` calls should return the current *simulated* time rather than the current real time on the machine running the simulation, which is meaningless within the simulation. This also constitute an elegant solution to the portability issues of deployed platform, as this system call virtualization mechanism is used as a portability layer over

```
1  typedef struct { /* message payload */
2    int id, row_count;
3    double time_sent;
4    row_t *rows; /* array, size=8 */
5    int leaves[MAX_LEAFSET];
6  } welcome_msg_t;
7
8  typedef struct { /* helper structure */
9    int which_row;
10   int row[COLS][MAX_ROUTESET];
11 } row_t;
```

Figure 4.2: C definition of the exchanged message.

the different operating systems, ensuring that any user code built on top of GRAS remains portable. The framework itself is ported to Linux (X86, AMD64, IA64, ALPHA, SPARC, HPPA and PPC); Mac OS X; Solaris (SPARC and X86); IRIX and AIX and Windows. GRAS have no external dependency to ensure its usability everywhere. GRAS offers several additional features such as classical data containers (dynamic arrays, hash tables), a distributed logging service, a unit testing framework and an exception mechanism, all in C ANSI.

Finally, the computation durations have to be reported into the simulator. When the user code needs W Mflop, the corresponding simulated process has to be blocked for W/ρ virtual seconds if its virtual host delivers ρ Mflop/s. GRAS provides a mechanism to automatically benchmark W.

Experimental Evaluation. Since the simulation aspect of GRAS is naturally carried by SimGrid, which performance is well studied elsewhere in this document, this section evaluates the other aspects of the GRAS framework: the code complexity and the communication performance. To that extend, we implemented a simple example using several communication libraries. The code simplicity was then measured using classical metrics and the performance was compared in different settings. The chosen message is involved in the Pastry application protocol [RD01]. Figure 4.2 presents the C definition of this data type, which is 5236 bytes long.

In this experiments, we compare GRAS to the following solutions: the MPICH implementation (version 1.2.5.3) of the MPI standard; the OmniORB implementation (version 4.0.5) of the CORBA standard; PBIO (that first introduced the NDR data transmission technique used in GRAS) and a hand-rolled solution using the expat XML parser. To our knowledge each of these implementations were amongst the best solutions in their categories when the experiment were conducted in 2005.

User code complexity. We first compare the complexity of the code that the user has to write to exchange this message. This comparison, presented in Table 4.1, uses two classic code complexity metrics: The McCabe Cyclomatic Complexity metric is the amount of functions plus the occurrence count of `for`, `if`, `while`, `switch`, `&&`, `||`, and `?` statements and constructs. This metric assesses the code complexity and its maintenance difficulty [Hen92]. The second line reports the number of lines of code (not counting blank lines nor comments).

	GRAS	MPI	PBIO	CORBA	XML
McCabe	8	10	10	12	35
Lines	48	65	84	92	150

Table 4.1: Comparison of code complexities and sizes.

The XML solution is by far the most complicated solution. It may be an artifact of the used parser, but expat was selected for its alleged performance. MPI is quite simple, the main difficulty being that it requires manual marshaling and unmarshaling of data. PBIO exempts the user of these error-prone tasks, but requires the declaration of data type description meta-data. OmniORB requires the user to override several methods of classes automatically generated from an IDL file containing the data type description. GRAS automatically marshals the data according to the type description, which is automatically parsed from the C structure declaration. This allows GRAS to be the least demanding solution from the developer perspective, according to both metrics.

Communication performance. The main performance concern of GRAS is to reduce the message delivery latency, as only communications are mediated by the framework, leaving computations unchanged. We now present a set of experiments involving computers of different architectures (PPC, SPARC and X86), and at different scales. On Figure 4.3, the first part presents the timings measured when data are exchanged between processes placed on the same host. The second part presents the timings measured on a LAN. The sending architecture is indicated on the row while the receiving architecture is shown by the column (for instance, the most down left graphic was obtained by exchanging data from a PPC machine to a X86 one). The last part presents the timings measured in an intercontinental setting: data is exchanged from the previously used hosts located in California, to an X86 host placed in France. The X86 machines are 2GHz XEONs, the SPARC are UltraSparc II and the PPC are PowerMac G4. The SPARC machines are notably slower than the other ones while X86 and PPC machines are comparable. All hosts run Linux. The LAN is connected by a 100Mb ethernet network, and both sites are connected to a T1 link. Each experiment was run at least 100 times, for a total of more than 130,000 runs. Moreover the different settings were interleaved to be fair and equally distribute the external condition changes over all the tested settings.

The first result of these experiments is the relative portability of communication libraries. PBIO does not seem to work on the PPC architecture while the version of MPICH that was available in 2005 failed to exchange data between little-endian Linux architectures and big-endian ones. We were also unable to use MPICH on the WAN.

The performance of the XML based solution is worse than any other by one order of magnitude. The systematic data conversions from and to a textual representation induce an extra computation load while the verbosity of this representation stresses the network. When MPICH is usable (half of the settings), it is about twice as fast as the other solutions. Our performance loss comes from the extra analysis performed: we interpret the data description at runtime to perform the data exchange automatically while MPICH requires to write the exchange code manually. This is a trade-off between code simplicity *vs.* speed. Instead, the best solution would be to automatically generate the marshaling

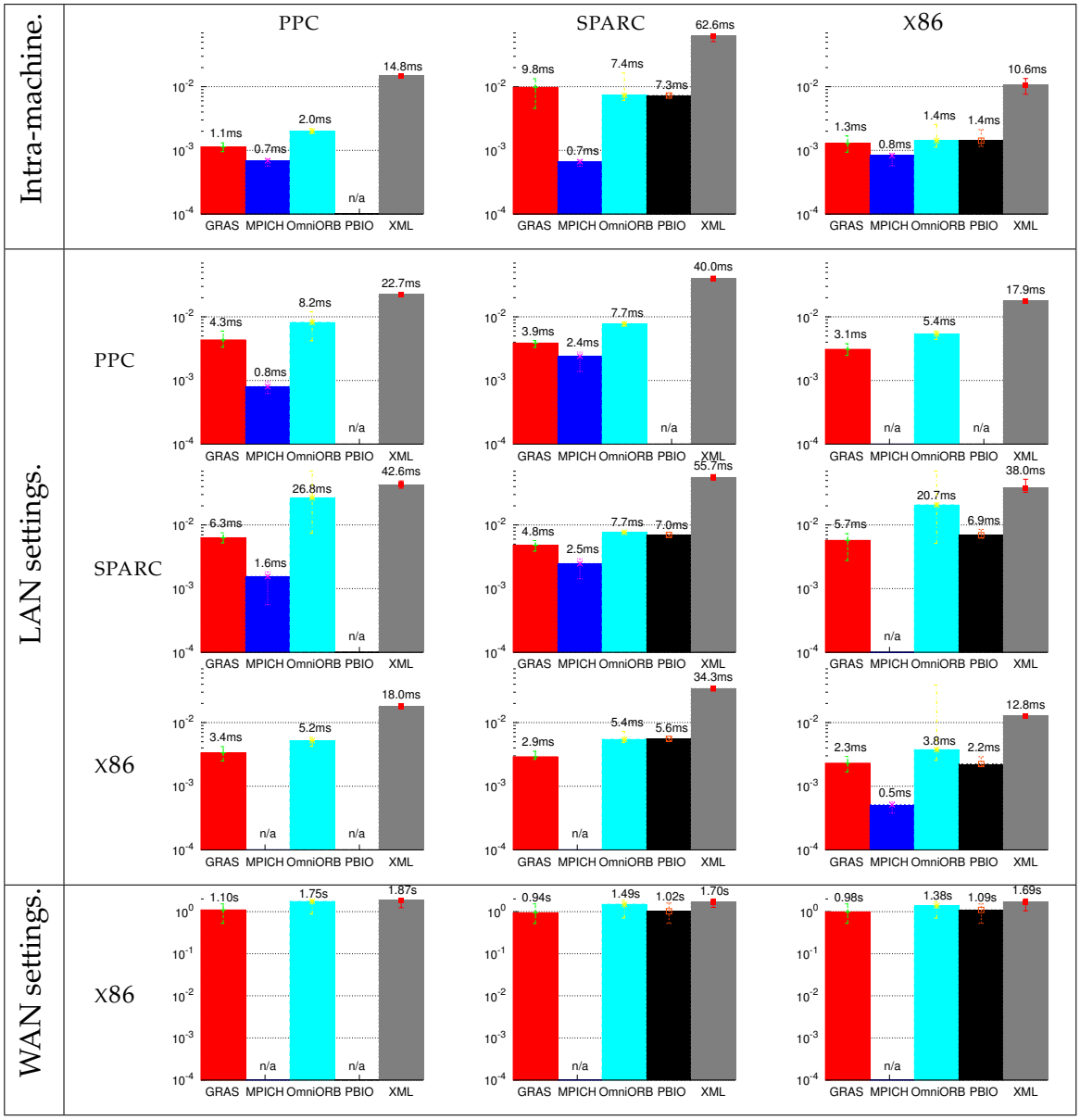


Figure 4.3: Performance comparison.

code at compilation time, providing maximal performance without inducing any extra complexity on the user. This was demonstrated by several frameworks introduced after GRAS (such as Thrift [SAK07] or ProtoBuffs [Buf08]), and could possibly be implemented in GRAS.

One can also remark that the results are not symmetric: it is twice as long to exchange the data with GRAS from PPC to SPARC than the reverse. Indeed, the NDR data representation mandates that the conversion is done on receiver side while the SPARC hosts are much slower than PPC ones. The same effect can be observed with OmniORB and PBIO.

Finally, the differences between solutions tend to be attenuated on WAN since the latency masks any optimization.

These results are quite satisfying for us: beside of MPICH, GRAS is the fastest solution in all settings, but the LAN X86/X86 setting (where PBIO is faster by 0.1ms – 4%) and the SPARC intra-machine setting (where both OmniORB and PBIO are faster by 2.5ms – 25%). This performance, added to the portability of our solution and its simplicity of use shown above constitute strong arguments for the quality of the GRAS framework.

Conclusion. GRAS was an ambitious project. It constituted an elegant solution to the development of distributed applications by conveniently allowing the development to take place in the simulator before the deployment of the resulting applications on real platforms. It was served by both an API allowing to easily specify such applications, and a portable and efficient execution runtime which performance were comparable to the most effective solutions at this time.

In addition, this runtime was though as a portability layer: with only 200kib in ANSI C code without any dependency, it made possible to enroll a given host in the deployed infrastructure as long as a shell and a C compiler were available. I planned to have versions of GRAS able to inject themselves over ssh connections to spread the infrastructure. The purpose is not very different of the technologies that enable the botnets on the dark side of the Internet: constituting a light, efficient and overly portable *underware* allowing to build any dedicated middleware wanted by the users.

Unfortunately, **GRAS did not stood the test of time**. This project revealed very demanding on engineering resources to remain on par with the state of the art. Other frameworks with very competitive arguments emerged, and it was impossible for me to compete with large companies such as Google (with ProtoBuffs [Buf08]) or Facebook (with Thrift [SAK07] – now part of the Apache foundation). These solutions do not allow to test applications through simulations, but their runtime is very probably better than the one I devised for GRAS. Note that Probably Buffers only marshals the data, but does not take the network communication in charge. It is usually coupled to ∅MQ [Hin07] for that.

In addition to these engineering issues, I now think that the major drawback of the GRAS framework was the event-oriented interface. As shown experimentally, it allows to efficiently write distributed infrastructures matching the targeted application architecture. It however revealed too rigid in practice, forcing this architecture on the user. This was ways too inflexible, as the users had to learn the new API provided by GRAS and then learn how to design their applications in a way that could be expressed in GRAS. I guess that this is the main reason why maybe only three applications were developed in this framework, despite the performance of the provided execution runtime, very competitive at this time.

That being said, none of the currently existing solution covers the whole scope of GRAS. The best solution with today's solutions could be to combine protobuf's marshaling abilities to ∅MQ's high performance and low latency communications to get an efficient runtime environment. Then, the resulting interface could be reimplemented on top of the simulator to ease the development and testing of the applications before their deployment on real platforms. I may implement this framework in the future, but the recent experience shows that other SimGrid considerations tend to completely fill my agenda, leaving no space to GRAS anymore.

4.1.2 Study of MPI Applications through Simulation

After my experience with GRAS (presented in previous section), I realized that taking the prototypes out of the simulator is not the right approach. It is much more interesting to take existing applications within the simulator instead. The user are can use the API and runtime they are used to, and it enables the study of legacy applications. Among all existing communication APIs, MPI [GLS99] is probably the most commonly used one.

This grounds the work presented in this section, aiming at allowing the seamless execution of MPI applications within the simulator. This work started in 2007 as a side product of the doctoral work of M. Stillwell, advised by H. Casanova, and continues since then. I joined the project in the following years, along with other colleagues: S. Genaud, F. Suter and A. Legrand. It was the topic of the post-doctoral work that P.-N. Clauss did with me in 2010. This project is thus the result of a large collaborative work, partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022) and the CNRS PICS N° 5473. This section extends [CSG+11].

Motivations. There are many reasons why it is interesting to enable the simulation of MPI applications, as demonstrated by the abundance of such projects in the literature (see §2.4.3). Simulation allows to predict the performance of an application for a platform that is not available, for example because it is yet to be specified and purchased. Simulation can be used to determine a cost-effective hardware configuration appropriate for the expected application workload. Conversely, simulations can also be used to study the performance behavior of an application by varying the hardware characteristics of an hypothetical platform.

Simulation of an application on a platform may also be useful even when the platform is available. For instance, the simulation may bypass actual computations performed by the application, and only simulate the corresponding delays of these computations. In this case the simulated application produces erroneous results, but its performance behavior may be preserved. It is then possible to conduct development activities for performance tuning in simulation only on a small-scale platform, thereby saving time when compared to real executions on a large-scale platform. Furthermore, access to large-scale platforms is typically costly (possible access charges to the user, electrical power consumption). The use of simulation can thus not only save time but also resources.

Even if the simulation realism is limited, it remains interesting for example in classroom settings. It allows students without access to a parallel platform to execute applications in simulation on a single node as a way to learn the principles of parallel programming and high-performance computing.

Problem Statement & Methodology. The challenges for simulating MPI codes include:

- **Virtualization:** Is the simulation possible at all? Do my application run within the simulator as if it were a real platform?
- **Accuracy:** Does the simulation match the real execution?

- **Scalability:** Is it possible to simulate large applications executing on large-scale platforms? Is it possible to have a low ratio of simulation time to simulated time?

These challenges mandate very different methodologies to be addressed. *Enabling* the simulation in a *scalable* way pose severe engineering questions while the *accuracy* issue require a modeling process similar to the one often found in natural science. Because of these methodological differences, I decided to handle these questions separately in this document (as discussed in §2.6.2). This section only deal with the engineering aspects mandated to *enable* a *scalable* execution while the modeling issues involved to ensure the *accuracy* of this simulation are postponed to §4.3.

Enabling the Simulation of MPI Applications with SMPI. We now discuss the main design decisions that make the SMPI project possible.

Online vs. *Offline simulation.* Simulation studies fall into two categories: *off-line simulation*, also called trace-based simulation or post-mortem simulation, and *on-line simulation*, also called simulation via direct execution. In off-line simulation a log of a previous execution of the application is "replayed" on a simulated platform. In on-line simulation the application is executed but part of the execution takes place within a simulation component. While both approaches have merit, **on-line simulation is more general** because the simulation is not tied to a log obtained for particular application and platform configurations. As such, it allows the study of network-aware applications, that is applications that adapt their behavior to the network conditions that they experience.

Which MPI implementations? Most MPI implementations (such as MPICH2 and OpenMPI, among many others) offer a layered design that could be leveraged to implement such an online simulation mode. For instance, in [PWTR09], the simulator is implemented as a peculiar network device for the MPICH2. It would certainly be possible to follow this approach to implement a MPICH2 or OpenMPI device on top of Simix (that is described in §3.1.2). The advantage is that the simulator should be easily evolvable to accommodate future releases of the MPI implementation, but the drawback is that it become highly tied to a particular implementation. But since different MPI implementations employ different algorithms, with notable differences including various data distribution schemes for collective communications and different algorithmic choices depending on message length, it may well happen that the behavior of an application on a given MPI implementation differs from what could be observed with another implementation.

Our goal in the long term is to allow the study of applications on each major implementations of MPI, so we decided to not tie our tool with one given implementation. Instead, we copied the implementation of each MPI primitives and modify it to integrate it in the simulator. It is worthwhile to note that there is no unique algorithm for any collectives operations, even for a given implementation. Depending on particular settings (such as message size or amount of processes), the implementation picks the most adapted variant. SMPI currently only implements one variant for each operation (either coming from OpenMPI or MPICH2 depending on the operation). Future versions will provide multiple variants, letting users choose which ones to use in the simulation.

Distributed vs. *Single-Node Simulation.* Since MPI applications are inherently distributed, it is very tempting to go for distributed simulation. But parallel discrete events simulation (PDES) raises difficult correctness issues pertaining to process synchronization (*cf.* §3.2). The classical techniques of PDES have been leveraged to speed up the simulation or MPI applications while preserving correctness (*e.g.*, the asynchronous conservative simulation algorithms in [PDB00], the optimistic simulation protocol in [ZKK04]). Since the current SimGrid kernel cannot be used for distributed simulation yet, we decided to side-step this difficulty by running the simulation on a single node.

To that extend, the MPI application is folded into a unique UNIX process, with each MPI process running in its own thread. In addition, these threads usually run sequentially, under the control of the SimGrid simulation kernel, that is fully sequential. The potential drawback of such sequential simulations is that simulation time may increase drastically with the scale of the simulation. However, the analytical simulation models implemented in SURF that can be computed quickly, leading to scalable simulation capabilities. Techniques to improve the scalability of the single-node online simulations are discussed in the next section.

Global variables pose the major technical difficulty to the single-node online simulation of MPI applications. These variables are usually private to each MPI rank, that usually live in a separate system processes. When folding the MPI ranks into threads of the same system process, the global variables become shared between MPI ranks. This naturally breaks the application semantic in most case, mandating a source code modification step to *privatize all global variables*. We could ask the users to do so manually, but this would defeat our goal to enable the study of unmodified legacy MPI applications.

Instead, we leverage existing source-to-source modification techniques to automatize this variable privatization, similarly to [BDP01]. In our technical framework, applications written in Fortran 77 are automatically translated to C using the `f2c` tool [Fel90] to be linked against our C implementation of the MPI standard. The extremely regular format of the C code produced by f2c allows us to privatize the global variables using a simple Perl script. This approach falls however short to privatize globals of arbitrary C code, so we leverage the coccinelle [MPLH06] transformation engine for that. Using so-called semantic patches that are applied during our `smpicc` compilation script, we use this tool to replace all global variables and local static variables by arrays that are indexed by the MPI rank number. Although some technical limitations in the coccinelle tool (*e.g.*, concerning multiple declarations on the same source code line), this approach allows to fully automatize the global privatization step of most MPI applications we tested.

Benchmarking the Applications. For the simulator to predict the execution time, it must naturally know all actions done by the application (be they communications or computations) and their sizes. The size of communications is trivially be obtained by SMPI as they are parameters of the MPI communication functions. As for the computation, SMPI benchmarks the host node computational power at initialization (in flops per seconds), and then benchmarks the time between any two subsequent MPI calls (in seconds). This trivially indicates the amount of flops to report within the simulator for each CPU burst. The first assumption here is that the processes are CPU-bound between MPI calls. It is often the case since communications occur within MPI calls, not between, and since MPI

applications are very rarely interactive. Accesses to the disk storage would mandate a specific mediation, and such applications are not covered yet. Another assumption backing our approach is that the simulated MPI processes do not compete for the CPU when being benchmarked. This is trivially guaranteed as SimGrid simulations are completely sequential by default. When run in parallel (as described in §3.2), the amount of user processes executing at a given time cannot exceed the amount of worker threads, that is configured to the amount of core by default.

Scalable Single-Node On-line Simulation. As discussed earlier, MPI Single-Node Simulation can pose severe scalability issues since MPI is typically used for HPC applications intended to leverage and saturate the resource of a whole cluster. Folding such an application onto a unique node may reveal challenging because of the CPU and the RAM requirements of the simulated application. We address both challenges as follows, considering that the simulation is hosted on a single host node.

Reducing CPU requirements. A first approach to speed up the simulation is to leverage all available cores during parallel simulations, as presented in §3.2. This is interesting because typical MPI programs exhibit coarse computational grain, efficiently hiding the synchronization costs that made fine-grained parallel simulation so difficult in §3.2.

It is possible to further reduce the CPU requirements by sampling the execution benchmarks of each CPU burst. Instead of benchmarking every occurrence of a given CPU burst, it is actually executed only the first n times, and then the average delay computed over these n samples is used as the delay in the simulation for future occurrences of this CPU burst. Using $n > 1$ is useful for CPU bursts that exhibit execution time variations, e.g., due to application data. In such case, the user can specify a measurement threshold t. The benchmark is then executed at least n times, and then rerun until the standard deviation becomes smaller than t. These benchmarks can be either global (common to all processes) or local, which allows to deal with situation where a given CPU burst presents different timings for each MPI rank. We also allow for $n = 0$, in which case the user must manually supply the number of flops that gets used. This technique makes the simulation time independent on the number of nodes in the target platform, and thus scalable.

Such execution sampling presents some limitations. It is impossible when the execution time variations become too important (*e.g.*, for data-dependent applications), and the CPU bursts that are sampled must be carefully selected to ensure that the application behavior is not modified by the sampling. In many parallel applications however, computations are regular, meaning that the MPI processes execute identical or similar CPU bursts. This is the case, for instance, for most applications using the SPMD (Single Program Multiple Data) paradigm. For applications that are irregular or data-dependent, replaying previously measured CPU burst durations may not lead to accurate results. In the worst case, all CPU bursts would need to be executed. In this case, single-node simulation would suffer from severe scalability issues. Should these issues endanger the applicability of the approach, one would have to face the challenges of developing a parallel discrete event simulator that can be executed on a cluster, as done for instance in MPI-SIM or MPI-NetSim.

Reducing RAM requirements. The memory footprint of the application cannot be accommodated on the host node unless the number of nodes in the target platform is small and/or the application's footprint is small. In an SMPI simulation, all MPI processes run as threads that share the same address space. In this case, two techniques are proposed in [ABDS02] for removing large array references.

1. Because MPI processes run as threads, references to local arrays can be replaced by references to a single shared array. If the MPI application has m processes that each use an array of size s, then the RAM requirement is reduced from $m \times s$ to s.

2. Because a CPU burst is simulated by replaying a delay rather than by executing its code, memory references in that code can be removed, which can lead to the removal of potentially large, now unreferenced, arrays.

The first technique is fully implemented in SMPI, and the CPU burst execution sampling presented earlier can trivially be used to implement the second technique. In this case, as in [ABDS02], CPU burst durations are user-provided and the CPU burst code is effectively removed.

Application Source Code Modifications. The presented solutions to reduce the CPU and RAM requirements of the simulated application mandate modifications to the applications' execution path. For the time being, we requires the user to manually insert relevant preprocessor directives into the source code for CPU and RAM requirement reductions. In the future, we plan to build upon [ABDS02, BDP01] to propose compiler-based approaches similar to the one proposed for the variable privatization on page 69.

Figure 4.4 shows a source code sketch that utilizes these macros. RAM factorization is simply achieved using the `SMPI_SHARED_MALLOC` and `SMPI_FREE` macros, that are used similarly to malloc and free, but ensure that the data is shared by all simulated MPI processes.

The execution sampling is achieved by the `SMPI_SAMPLE_*` macros. At line 3, the `SMPI_SAMPLE_LOCAL` macro is used to indicate that the following CPU burst (in between curly braces) should be executed and timed at most 10 times by *each* MPI process, and subsequently bypassed and replaced by a simulation of a delay equal to the average of the 10 measured execution times. Note that it does not mean that the block is executed 10 times in a row. Instead, it means that if this block is included in an external loop, any occurrence after the 10th one will be bypassed. The use of `SMPI_SAMPLE_GLOBAL` macro at line 6 is similar but the CPU burst is measured only 10 times in total (possibly when executed by 10 different MPI processes), before its execution is bypassed. At line 9, only the second parameter of `SMPI_SAMPLE_GLOBAL` is non-null. The marked block will thus not be bypassed until the standard deviation on the benchmarked value goes below 0.01. The block at line 13 will not be bypassed until at least 10 runs are executed *and* the standard deviation becomes smaller than 0.01, with the statistics being local to each process. At line 15, the block following `SMPI_SAMPLE_FLOPS` will never get executed but instead replaced in the simulation by the given amount of flops. At line 18, the block following `SMPI_SAMPLE_DELAY` is never executed either. Instead, the amount of flops that the host machine could compute in 1.3 seconds is simulated. All these macros are expanded into

```
1  double *data = (double*)SMPI_SHARED_MALLOC(...);
2  MPI_Init(...);
3  SMPI_SAMPLE_LOCAL(10,0) {
4    // Executed at most 10 times per process, and then bypassed
5  }
6  SMPI_SAMPLE_GLOBAL(10,0) {
7    // Executed at most 10 times in total
8  }
9  SMPI_SAMPLE_GLOBAL(0,0.01) {
10   // Executed as long as the standard deviation is over 0.01
11 }
12 SMPI_SAMPLE_LOCAL(10,0.01) {
13   // Executed at least 10 times, and until the stddev < 0.01
14 }
15 SMPI_SAMPLE_FLOPS(1000*1000*1000) {
16   // Never executed, 1 Gflop is simulated instead
17 }
18 SMPI_SAMPLE_DELAY(1.3) {
19   // Only simulates the amount of flops doable by host in 1.3 second
20 }
21 MPI_Finalize();
22 SMPI_FREE(a);
```

Figure 4.4: SMPI macro usage example.

one or more calls to various functions that look up and update hash tables where each entry contains a unique identifier (based on source file name and line number), execution counters, reference counters, and/or pointers to user arrays.

Evaluation. In this section we evaluate the scalability and speed of SMPI simulations for simple scenarios. Experiments were conducted on the *griffon* cluster of the Grid'5000 platform, that comprises 92 2.5 GHz Dual-Proc, Quad-Core, Intel Xeon L5420 nodes, using SMPI as implemented within SimGrid v3.5-r8210.

Qualitative Evaluation. SMPI supports MPI applications written in C or Fortran 77. In its current implementation SMPI implements the following subset of the MPI standard: error codes, predefined datatypes, and predefined and user-defined operators; process groups, communicators, and their operations (except Comm_split); The following point-to-point communication primitives are handled: Send_Init, Recv_Init, Start, Startall, Isend, Irecv, Send, Recv, Sendrecv, Test, Testany, Wait, Waitany, Waitall, and Waitsome. In addition, the following collective communication primitives are provided: Broadcast, Barrier, Gather, Gatherv, Allgather, Allgatherv, Scatter, Scatterv, Reduce, Allreduce, Scan, Reduce_scatter, Alltoall, and Alltoallv.

Single-Node Online Simulation Effectiveness. One of the attractive aspects of on-line simulation is that simulation time can be shorter than simulated time. This time reduction can come from the simulation of communication operations. Figure 4.5 presents results from an experiment in which we measure the time needed for performing a scatter

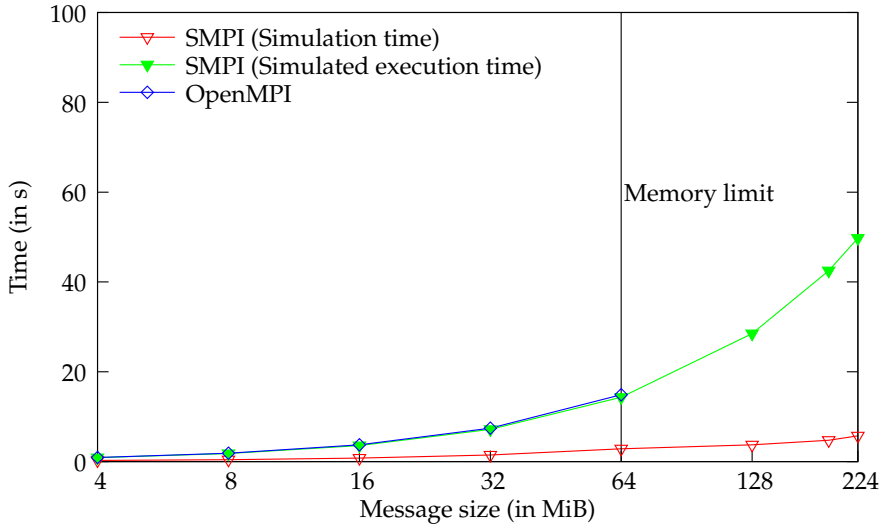


Figure 4.5: Simulation time versus experimental time for a binomial tree scatter operation with 16 processes with messages of increasing sizes.

of messages of increasing sizes on a real-world platform with OpenMPI and in simulation with SMPI. The gain offered by SMPI in terms of execution time increases with the message size. For medium-sized messages of 4 MiB, SMPI leads to a good estimation (maximum error of 4%) of the OpenMPI execution time while running 3.58 faster. For larger messages, this factor reaches 5.25.

Note that OpenMPI is only tested up to messages of 64MiB. This is because earlier versions of SMPI were limited to that value by their RAM requirements. RAM folding techniques allowed us to break this limit and obtain values up to 224 MiB, but we forgot to update the values of OpenMPI before the publication, wrongly assuming that OpenMPI shared this limitation at 64MiB. Since this section postpones the realism considerations to focus on SMPI's scalability, these results are still interesting, even if slightly incomplete.

Pushing the Scalability Limits. We now show the effectiveness of the RAM footprint reduction techniques to push the simulation scalability limits. Figure 4.6 shows how these techniques allow us to scale the simulations up to the 448 processors needed for class C of the SH variant of the DT benchmark by reducing the maximum Resident Set Size (RSS). Memory consumption is drastically reduced when using such techniques, and it becomes possible to simulate applications that would otherwise overcome memory (these are represented by "OM" — Out-of-Memory — labels in the figure). In these experiments, SMPI's memory consumption has been reduced by a factor 11.9 on average, and up to 40.5 for the WH graph in class B.

A side effect of RAM footprint reduction is that less CPU time is devoted to memory allocations, which is not captured by SMPI at the moment. The simulated execution times obtained are thus slightly lower than those obtained with the full memory footprint. When comparing SMPI with RAM footprint reduction techniques to OpenMPI, for those DT benchmarks that we were able to run on our cluster (WH and BH in classes A and B), the average error of the simulated execution time is increased to 18%, with a worst case

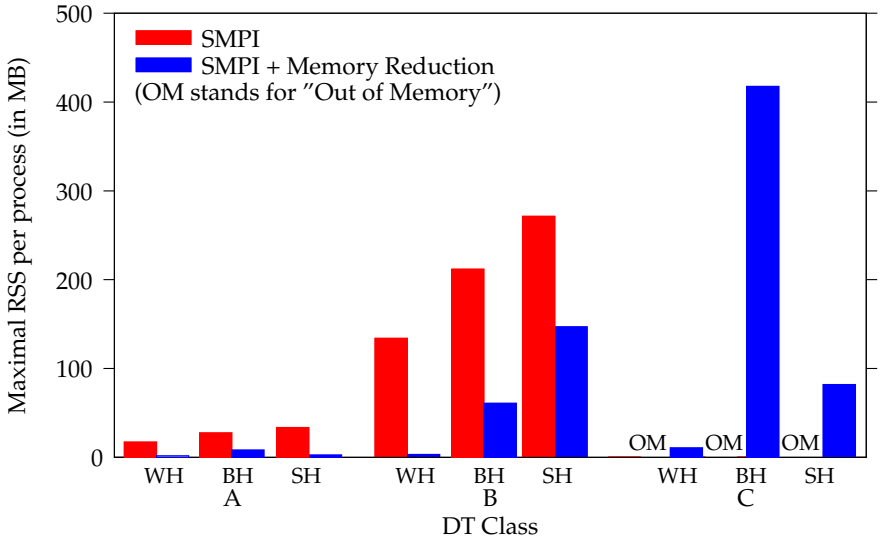


Figure 4.6: Memory consumption of DT with and without RAM footprint reduction.

at 42%. As shown in §4.3.3, the error without RAM folding on the DT benchmark of 8.1% on average, with a worst case at 24%.

Improving the Simulation Speed. The speed advantage over real execution that comes from the simulation of communications may be hindered by the execution of the computational part of an application (which is done on a single node). We now demonstrate the effectiveness of the execution sampling techniques introduced page 72 to alleviate this.

Figure 4.7 presents the effects of using the `SMPI_SAMPLE_LOCAL` macro on the Embarrassingly Parallel (EP) application from the NAS Parallel Benchmark suite. This application simply distributes a large computation among the processes. Each process computes its share of the computation without any further communication. The results shown in Figure 4.7 are for a class B with four processes; we observed similar results for other classes and numbers of processes. The x-axis is the sampling ratio, i.e., which fraction of the iteration space was actually executed. For instance, a sampling ratio of 25% means that only the first 25% of the iterations are executed, while the remaining 75% are replaced by the average computation time of the first iterations. The left y-axis shows the simulation time and the right y-axis shows the simulated execution time of the benchmark.

As expected, the simulation time decreases linearly as the sampling ratio decreases. When only one fourth of the iterations are actually executed (1024 instead of 4096 in this case), the simulation time is also divided by a factor four. More interestingly this reduction of the simulation time is not at the expense of accuracy, as the sampling factor has almost no impact on the simulated execution time and that accuracy is constant with respect to the OpenMPI implementation. This phenomenon is application dependent. The impact on accuracy would be zero for perfectly regular data-independent applications while it could be large (and thus unreasonable) for irregular data-dependent applications.

Conclusion and Future Work. The SMPI project presented in this section effectively enables legacy MPI applications to be simulated within the SimGrid framework. Thanks to

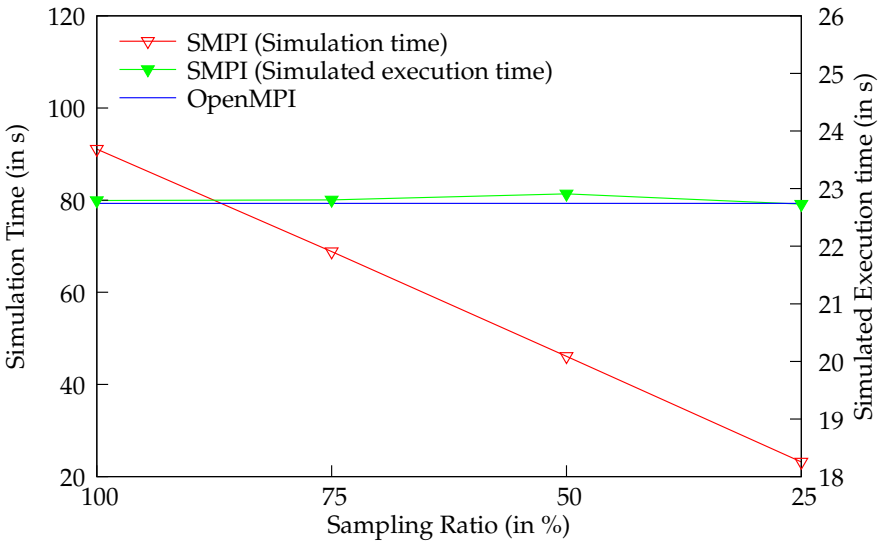


Figure 4.7: Impact of CPU sampling on the simulation time and accuracy.

automatic source-to-source transformations, it is possible to study unmodified applications within the simulator, on a single node. To push further the scalability limits of what can be studied, some support is given to the user wishing to reduce the CPU requirements (by sampling the execution of certain CPU burst) and the memory requirements (by factorizing memory areas between simulated processes). These techniques are particularly well adapted to regular SPMD kernels that are relatively common in MPI applications.

Although already usable, SMPI could benefit of several improvements. We are working on the technical details that mandate C or Fortran 77 to use SMPI. In particular, we plan to allow the use of modern dialects of Fortran through the development of Fortran bindings, just like the Java bindings that we developed for P2P users and others. Better source-to-source transformations should also be investigated, to automatize the placement of the macros increasing scalability and speed on a single node (similarly to [ABDS02, BDP01]), and also to improve the robustness of our variable privatization tools. It would also be interesting to assess our compliance to the MPI standard, and implement the missing primitives.

A central question raised by SMPI naturally pertain to SimGrid's prediction realism. MPI applications are typically tested directly on real platforms, and from my experience, most MPI developers are suspicious of a simulation tool such as SMPI, since they distrust the realism of the predictions obtained this way. This naturally justify the modeling work presented in §4.3. I however strongly believe that SMPI is interesting regardless of the provided realism.

Since simulations are completely reproducible, SMPI is a perfect debugging framework for MPI applications, free of any *heisenbug*. These issues that disappear when activating the profiling, tracing or debugging tools are impossible within the simulator, since these tools only impact the timing on the host machine. SimGrid is designed to ensure that the host execution conditions have no impact on the simulated world. It gives very interesting possibilities, such as the work of Lucas Schnorr in the following of [SHN10].

A slightly different but related idea is to combine SMPI with the work presented in §3.4. It results in a competitive solution for the formal verification of MPI applications. I think that the timing of this work is perfect given that the MPI 3.0 standardization effort is converging. This new version of the standard contains asynchronous group communications, that are known to pose very challenging semantic issues [CKV01]. It would thus be interesting to formally assess the correctness of the MPI applications using these primitives.

A long-term goal for SMPI on application semantic aspects is to allow users to specify the real-world MPI implementations that they want to simulate (*e.g.*, OpenMPI or MPICH2). This is particularly important for collective operations, since implementations provide differing algorithms for each collective, depending on the parameters (data size, amount of processes), and possibly on the network conditions. This feature would either require a careful (and automated) analysis of these implementations, opening the gate for the verification of these framework themselves.

4.1.3 Study of Arbitrary Applications through Simulation

When considering my goal of studying real applications using the simulator, SMPI (presented in §4.1.2) constitutes a clear improvement over GRAS (presented in §4.1.1): the MPI interface is already known to some potential users. MPI is however a difficult communication system that pose troublesome constraints on the programmer to improve the potential performance. In some sense, MPI is a sort of communication assembly language, where the programmer has to specify every details, because some of them are sometimes mandatory to maximize the performance.

This leads to the question behind the work presented in this section: **how to simulate *arbitrary* legacy applications, even the ones for which the source code is not accessible?**

I conducted this work in collaboration with Lucas Nussbaum, often through the advising of student work: K. Lin and A. Seng on the feasibility of Java interception, and M. Guthmuller on the feasibility of system-level interception, both in 2010. This work is still rather preliminary at this point, and only the feasibility of the later approach was published so far, as [GNQ11]; another publication is in preparation to account for the internship of G. Serrière in 2012.

Simulation *vs.* Emulation. The question's wording blurs the boundary between these experimental methodologies. Classically, simulation aims at studying prototypes of application onto virtual platforms while emulation is the set of techniques aiming at allowing the study of real applications onto virtual platforms. The difference is not only on the intended studies, but also on their *modus operandi*: simulation computes the platform behavior using *models* while emulation degrades the capacity of the host platform (through network delays, application containment or even CPU burners) to imitate the target platform. Several such *emulation through worsening* solutions exist in the literature (such as Modelnet [VYW+02], DieCast [GVV08], Emulab [WLS+02] or Wrekavoc [CDGJ10]), but they induce complex technical frameworks and are by design unable to provide a target platform that is faster and larger than the host platform.

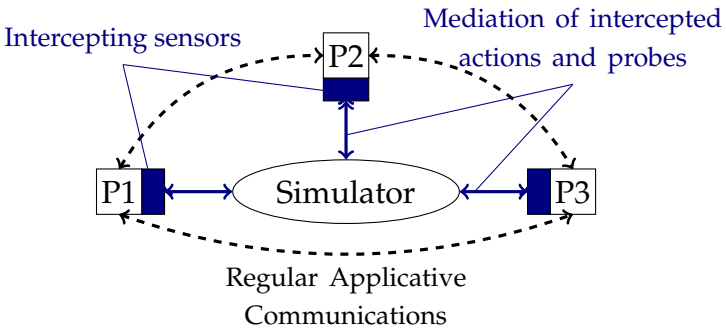


Figure 4.8: Emulation through Interception principle.

Our goal is to leverage the models of SimGrid for the study of real applications. Instead of worsening the real platform to imitate the target, this approach modifies the application perception of the platform (see Figure 4.8). For that, the application's actions are **intercepted** (computations and communications), and injected into the simulator. The application's probe are also intercepted and mediated through the simulator. For example, when the applications retrieves the current time, it is provided with the simulated clock instead of the host system's physical clock; when probing for messages, their arrival order is given by the simulation results; the DNS name resolution service is also usually mediated to control how processes connect to each other. As in the SMPI project (cf. §4.1.2), this can lead to situations where the simulation is faster than the real execution, for example when intercontinental communication-bound applications are folded onto a single node.

This approach of *emulation through interception* is rather uncommon to study the application performance as it mandates both a working simulator and an interception framework. To the best of my knowledge, MicroGrid [XDCC04] is the only related project in the literature. Other interception frameworks serve other goals. Fuzzing frameworks such as Spike[1] or Autodafé [Vua05] try to discover bugs and security issues by intercepting the file and network operations and changing random bits in the program's input. Profiling and tracing tools (such as Tau [SM06]) allow to inspect the application's execution.

Numerous techniques can be leveraged to intercept the application's actions and probes, differing in their scope of application, effectiveness and performance. I now quickly present a first attempt to leverage Java specific features before going into more details about a lower-lever approach.

Interception Methods to Simulate Java Applications. The Java Platform (Enterprise Edition) is probably as pervasive in the commercial internet than MPI in the scientific computing community. It would thus be interesting to provide a way to study the performance and correction of Java EE applications within SimGrid. Java provides several ways to implement advanced monitoring facilities in this context. The `java.lang.instrument` API allows so-called *agents* to start even before the execution of the application's main function, and to modify the loaded bytecode on the fly.

[1]Spike: `http://www.immunitysec.com/resources-freesoftware.shtml`

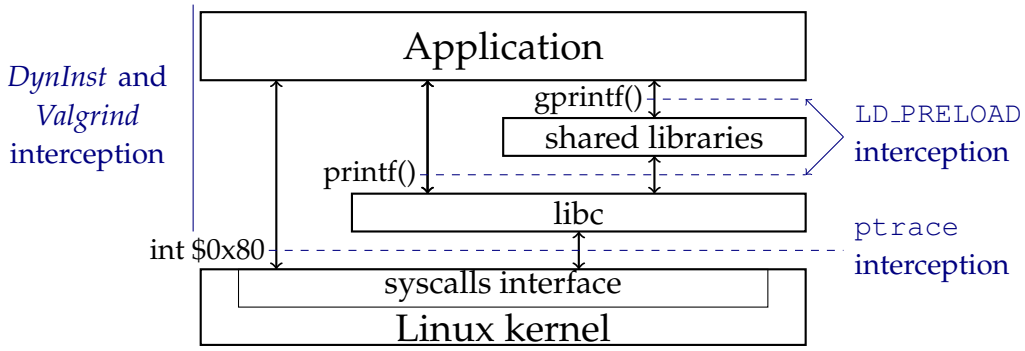


Figure 4.9: Possible approaches for a system-level interception.

We explored through a student work how such interception techniques could be used to simulate Java EE applications within SimGrid. We found that these techniques are applicable, but require an important development effort. Intercepting the communications would mandate to reimplement any low-level networking code used in the Java EE platform. This is feasible, but would be time- and labor-intensive. Likewise, intercepting system probes on time or DNS would probably be extremely challenging. The main problem comes from the richness of the provided APIs. The amount of functions that must be intercepted to ensure that the application cannot use the feature without being mediated becomes intractable.

System-Level Interception. We found much more effective to to mediate the applications directly at the system level instead of at the JVM level. The deeper in the system, this interception is done, the lesser functions need to be intercepted. In SMPI for example, we decided to intercept every call provided by the interface. If it presents other advantages, it increases the burden to fully cover the API. Likewise, reimplementing all APIs of the Java EE platform seem very difficult to do. Finally, this approach is not generic as it should the work should be duplicated for each high-level API to intercept.

Virtual machines could be used to mediate the application actions and probes. This solution would however lack scalability as the amount of virtual machines that can be started per physical machine is extremely limited due to their resource requirements. It is preferable to search for lighter yet efficient ways of intercepting the application actions and probes.

Figure 4.9 presents several techniques that could be leveraged to intercept the application at system level, that we now detail.

Interception through Binary Rewriting. *Valgrind* and *DynInst* are two solutions allowing to instrument binary programs through on-the-fly binary rewriting. Valgrind provides several tools to track invalid memory usage, or to profile the execution timings. DynInst is used in the Tau profiling tool targeting the HPC community. Both tool work by desassembling the application binary, adding some instrumentation code, and recompiling the resulting assembly code. This approach would permits to intercept the application actions and probe by simply modifying any call to the intercepted functions.

The main issue of Valgrind consists in the performance of the modified binary. Because the recompilation phase lacks basic optimization steps, an application that is modified by Valgrind runs almost 10 times slower than before, even if absolutely no modification were introduced in the source code. This makes this approach acceptable for a debugging tool, but not in our context. DynInst does not suffer from these performance issues, but it provides only a very low-level API, making the interceptors very tedious and complex to implement. The DynInst framework itself would also constitute a difficult dependency, because of the framework's size and complexity to install for the users.

Interception through dynamic linking mechanisms. Another way to intercept the function calls is to leverage the dynamic linking mechanisms. One can for example use the `LD_PRELOAD` environment variable to inject a specially crafted library into the binary. The functions of this library are then used by the dynamic linker in preference to the usual functions of the same name. These interceptor functions are also provided a way to invoke the masked function, constituting an effective way to simply write wrappers functions. This approach is leveraged in the MicroGrid project to intercept every network-related functions. This technique is often used by crackers attempting to compromise the system security. Several refinements are for example presented in [VGA+07].

The performance of this approach is very good, as the only overhead occurs during the setup. Its main drawback is that it can only be applied to library calls and not to system calls, resulting in a very large amount of function calls that must be intercepted. For example, one would have to intercept `printf`, `fprintf` and all their variants because the corresponding system call `write` is inaccessible this way. This makes this approach repetitive and error prone to implement. Moreover, binaries that are statically compiled cannot be handled this way.

Interception at the kernel boundary. The *ptrace interface* allows a debugger-like process to control a debugee process. The debugee is interrupted on each system call, so that the debugger inspects it before restarting it. This method is very interesting in our context because only a dozen of system calls need to be intercepted to fully mediate the network communications. But since it can only intercept syscalls, some probes (such as DNS name resolution) are invisible to this technique. The performance of ptrace is acceptable in our context even if it is often cited as a major drawback. Another interface called *Uprobes* [KD10] is under development to solve these issues of applicability (it will allow the interception of library calls too) and performance, but it is not usable yet.

The simterpose Project. We implemented a prototype based on the `ptrace` approach, that the study shows as a solid (although not perfect) candidate in our case. In [GNQ11], we only presented an offline tool, able to capture an application trace that could be replayed in SimGrid afterward, but we are currently developing an online version. The main limitation of this online version over the previous one is that it cannot deal with multithreaded applications yet because the process and thread creation system calls are not intercepted. This is however a limitation of our current code and not of the approach.

Mediating Data Exchanges. Several ptrace primitives (such as `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`) allow the debugger to read and write in the debugee memory. This

mechanism is used to identify the ongoing system calls and retrieve their parameters. The communications can also be intercepted with way, but the performance are deceiving. It is better to let the actual data exchange between debugee occur on regular sockets. Note that our approach is not applicable to applications communicating through shared memory pages, as no system call is involved for the communication.

Mediating Rendez-Vous Mechanisms. The classical system mechanisms to establishing the connection between the processes must be handled in the interceptor so that the outcome matches the simulated scenario. This is relatively easy to do by intercepting the relevant system calls such as `connect()` or `accept()`. It can then be matched to the Simix rendez-vous mailboxes to decide on the correct communication matching.

Several techniques are then applicable to get the processes connected in the way computed by the simulator. First, it is possible to not interconnect the debugees at all, and get the debugger PEEK the data out of the sender and POKE it into the receiver directly, using the `ptrace` interface. As explained before, performance considerations advise however to not use this approach when possible. Another approach is to rewrite the socket connection calls to use local sockets. An array then associates the {host×port} in the simulation to a port on the host machine. The main drawback is the high consumption of port numbers on the host machine. Even if about 60,000 ports are usable on a typical UNIX system, this will probably soon reveal limiting. If each process connects to a dozen of other processes, the system-wide limitation on the port numbers will limit the amount of simulated processes to a few thousands, while SimGrid could certainly handle ten times this amount of processes: Provided that the process code is loaded only once in memory by the system, only the user data is duplicated in memory. This seems to induce that the memory load of simulating real programs may not be very different of simulating prototypes of the same algorithms. A third approach to control the matching of communications would be to change any network connection through a named pipe. This would remove the system-wide limitation induced by the port availability. The only hard limits would be the amount of file descriptor per debugee, but this limit would be unchanged with regard to the regular execution settings, as each network socket also take such file descriptor.

Mediating Computation Phases. In SMPI, we benchmarked the computation phases assuming that the application is CPU-bound between the intercepted calls. This assumption cannot be taken for arbitrary applications, and other techniques must be leveraged. Fortunately, Linux allows to accurately retrieve the time that each thread spent on the CPU through the *NETLINK TASKSTATS* interface. There is thus no need to manually benchmark the application, and we simply report this value into the simulation.

Mediating Probes. The main difficulty to mediate the time-related system calls is to come up with a comprehensive list: `time()`, `gettimeofday()`, `sleep()` must be intercepted when available, but they are sometimes implemented at the library level (for example, sleep is implemented using `alarm()` on some architecture).

Simterpose provides no mediation of the name resolution mechanisms (DNS) so far. As no system call is involved, we would have to either use dynamic linking interception methods to intercept the relevant library calls, or apply a specific mediation on the communications to the name servers, that usually listen on the port 42 of any system.

Conclusion. It is possible to trick most distributed applications in order to mediate all their interactions with their environment. This mandates rather advanced technical mechanisms, but I believe that these mechanisms are lighter that the ones typically used to emulate distributed applications. It is also the only way to run the experiment faster than on real platforms as emulation techniques work by worsening the performance of the host platform.

This approach were already partially demonstrated by the MicroGrid project. Unfortunately, some technical decisions made this project very difficult to install and use. The application calls were intercepted through dynamic linking methods, making the tool fragile as applications could evade the mediation by using unusual calls. The chosen simulation tool was a packet-level simulator which performance was problematic at the envisioned scale of study. As an answer, a distributed version of the simulator were introduced, at the expense of the framework's complexity. Finally, time probes were not mediated. Instead, the whole simulation was slowed down to ensure that the simulated time passed by at a fixed ratio over the real time. This mechanism too made the framework complex and tedious to use correctly. As a result, the project died and the tool is not usable anymore on modern systems.

I believe that the design decisions of the simterpose project presented here will allow us to avoid these pitfalls. This would result in the first usable tool of emulation through interception, effectively concluding my quest to study arbitrary applications within the simulator.

4.2 Automated Network Mapping and Simulation

The work presented in this section had a major impact on my scientific orientation. The objective is to propose automatic methods able to scout performance information out of a real network in order to instanciate the models of a simulator. This work actually started during my PhD thesis, that aimed at automatically gathering quantitative and qualitative information on grid platforms. It continued during master work of H. Harbaoui that I advised in 2006, and then with the postdoc that L. Eyraud-Dubois did with F. Vivien, A. Legrand and me the year after. This section revisits [EDLQV07].

§4.2.1 presents the motivations rooting this work, and situates it with regard to the relevant literature. Identifying some weaknesses of previous approaches leads to proposes new network tomography algorithms in §4.2.2. But proposing a generic algorithm for that, based on application-level measurements poses a methodological challenge, as it is usually not trivial to compare the quality of such algorithms on real platforms. We thus introduced a specific workbench called ALNeM (Application Level Network Mapper), presented in §4.2.3 and then used to evaluate our algorithms.

4.2.1 Motivation and Problem Statement

Unlike classical parallel machines, modern large-scale distributed systems are heterogeneous and non-dedicated. Gathering accurate and relevant information about them is then a challenging issue, but it is also a necessity to efficiently use their resources. In par-

ticular the performance impact of the network topology is crucial to achieve tasks such as running network-aware applications [LRRV04], efficiently placing servers [CDCV06], or predicting and optimizing collective communications performance [KHB+99]. However, the description of the network structure and characteristics of these systems is usually not available to users, as the providers do not want to disclose the bottlenecks and limitations of their infrastructures. Hence a need for tools which automatically construct performance models of modern large-scale computational platforms.

Many tools and projects provide some network information. The first difficulty in our context comes from the methodology used for the measurements. The classical network protocols aiming at gathering performance information such as SNMP and ICMP are usually restricted or disabled on large-scale computing platforms, as they can be used to conduct DoS attacks on the infrastructures. This makes a lot of approaches from the literature unappliable in our context. For example, Remos [DGK+01] requires access to SNMP information while TopoMon [dBKB02], Lumeta [BCW], IDmaps [FJJ+01] or Global Network Positioning [NZ02] depend on ICMP. Likewise, pathchar [Dow99] require specific privileges on the machines on which it runs, making it difficult to use in our context. It is mandatory to rely on tools that only use *application-level measurements*, *i.e.*, measurements that can be done by any application without any specific privilege. This comprises the common end-to-end measurements, like bandwidth and latency, but also interference measurements (*i.e.*, whether a communication between two machines A et B has non negligible impact on the communications between two machines C et D).

Several projects rely on this type of measurements, such as the NWS (Network Weather Service) [WSH99]. It gathers information about the current state of a platform (end-to-end bandwidth, latency, and connection time) and predicts its evolution. However, this tool focuses on quantitative information and does not provide any kind of topological information. This issue is usually addressed by aggregating all NWS information in a single clique graph and use this labeled graph as a network model. In another example, interference measurements have been used in ENV [SBW99] and enabled to detect, to some extent, whether some machines are connected by a switch or a hub. A last example is ECO [LB99], a collective communication library, that uses plain bandwidth and latency measurements to propose optimized collective communications (*e.g.*, broadcast, reduce, etc.). These approaches have proved to be very effective in practice, but they are generally very specific to a single problem and we are looking for a general approach.

4.2.2 Proposed Algorithms

In most previous works, the reconstructed network topology is either a clique [WSH99, LB99] or a tree [BBH05, SBW99]. Our reference reconstruction algorithms are thus CLIQUE, TREELAT (minimal spanning tree on latencies), and TREEBW (maximal spanning tree on bandwidths). In addition, we designed two new reconstruction algorithms. The first algorithm aims at improving an already built topology, e.g. an existing spanning tree, while the second one builds a platform model from scratch, by growing a set of connected nodes. All these algorithms keep track of the routing while building their model, as this information is part of the simulation model that we want to instantiate.

Algorithm IMPROVING. This algorithm is based on the observation that if the latency between two nodes is badly over-predicted by the current route connecting them, an extra edge should be inserted to connect them through an alternate and more accurate route. Among all pairs of "badly connected" nodes, we pick the two nodes with the smallest possible measured latency, and we add a direct edge between them. For each pair of nodes which the latency was over-predicted, we then attempt to use that link to improve the prediction accuracy, and update the routing if needed. This edge addition procedure is repeated until all predictions are considered sufficiently accurate. In our implementation, we arbitrarily decided to continue until the deviation between predictions and actual measurements become smaller than 10%.

This algorithm comes with two variants: IMPTREEBW that takes maximal spanning tree on bandwidths as an input, and IMPTREELAT that improves over the minimal spanning tree on latencies. Note that IMPTREEBW uses both latency and bandwidth information and is thus expected to perform better than the other algorithms using only partial information. It is tempting to develop a symmetric algorithm using bandwidth information to improve a spanning tree built using latencies, but it is not possible since the bandwidth of links do not sum up nicely on the path, as latencies do.

Algorithm AGGREGATE. This algorithm uses a more local view of the platform. It expands a set of already connected nodes, starting with the two closest nodes in terms of latency. At each step, *Aggregate* connects a new *selected* node to the already connected ones. The selected node is the one closest to the connected set in terms of latency. *Aggregate* iteratively adds edges so that each route from the selected node to a connected node is sufficiently accurate. Added edges are greedily chosen starting from the edge yielding a sufficiently accurate prediction for the largest number of routes from the selected node to a connected node. We slightly modified this scheme to avoid adding edges that will later become redundant. A new edge is added only if its latency is not significantly larger (meaning less than 50% larger) than that of the first edge added to connect the selected node. Because of this change, we may move to a new selected node while not all the routes of the previous one are considered accurate enough. We thus keep a list of *inaccurate* routes. For each edge addition we check whether the new edge defines a route rendering accurate an inaccurate route. When all nodes are connected, we add edges to correct all remaining inaccurate routes, starting with the route of lowest latency.

4.2.3 Evaluation

To fairly compare these topology mapping algorithms, we developed a specific workbench called ALNeM (Application Level Network Mapper). As depicted in Figure 4.10, it is composed of three main parts: (1) A collection of **distributed sensors** performing bandwidth, latency, and interference measurements. Being based on the GRAS interface (see §4.1.1), the sensors can work seamlessly on real platforms or on simulated platforms. (2) A **measurement repository** centralizing the data coming from the sensors; (3) A set of **candidate algorithms** that are to be evaluated. They use the data from the repository.

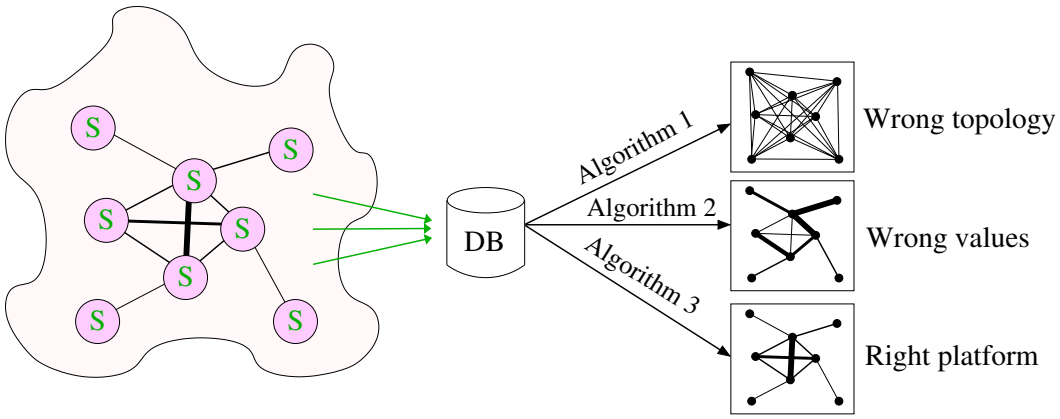


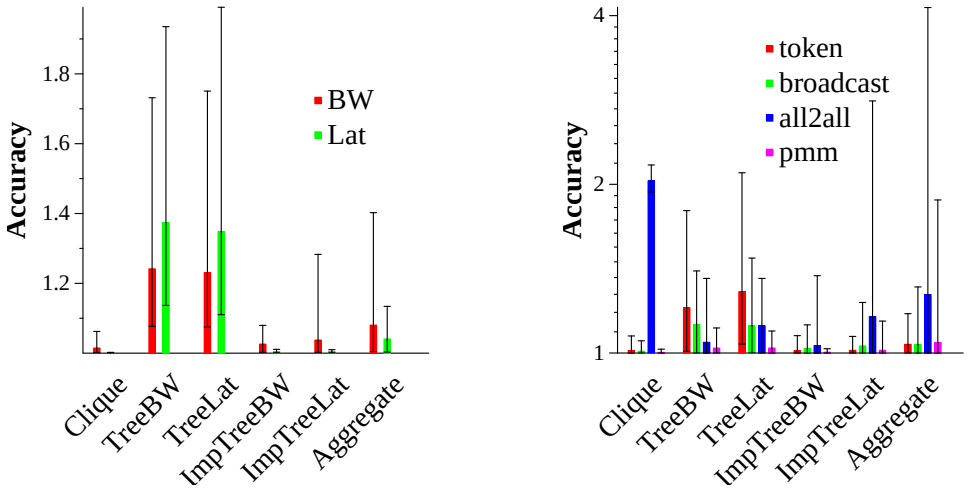
Figure 4.10: Evaluation Workbench for Network Tomography Algorithms.

Evaluation Methodology. We evaluate the quality of each tomography algorithm through both network-level and application-level measurements. We compare the results obtained on the original platform on which the sensors are deployed with the results obtained on the reconstructed platform. The reconstructed platform must obviously be simulated as building a real platform corresponding to the reconstructed platform is near to impossible. Then, the original platform should also be simulated. Otherwise, the comparison between original and reconstructed measurements would suffer of a bias due to the differences between the actual world and the simulator. Thanks to GRAS, it is expected that once the tomography algorithms are evaluated in the simulator, they will be usable on real platforms.

End-to-End Metrics. We consider the three following characteristics: First, the **bandwidth** is clearly mandatory when the messages' size differ in the application. Then, the **latency** cannot be neglected given their impact on communication performance, *e.g.*, through their interactions with the TCP contention mechanisms [Cas04]. Finally, the performance of collective communications (*e.g.*, broadcasts or all-to-all) or independent communications between disjoint pairs of processors cannot be predicted if the **interference** between concurrent flows remains unknown. This depends on the kind of shared resources in the underlying topology and can be evaluated by measuring the performance impact of one flow (AB) onto another flow (CD), for each 4-tuple of hosts {A,B,C,D}.

Application-Level Measurements. To evaluate the predictive power of the reconstruction algorithms for applications with more complex but realistic communication patterns, we study the following simple distributed algorithms: In a **token ring**, a token circulates three times along a randomly built ring (the ring structure is not correlated to that of the interconnection network); In a **broadcast**, a randomly picked node sends a message to all the other nodes; **All-to-all** involves that all the nodes simultaneously perform a broadcast; a **parallel matrix multiplication** implements the classical outer product algorithm, typical of some numerical applications.

Evaluation Results. We ran two sets of experiments. In the first set, all of the hosts are known to the measurement procedure, which means that a sensor process was deployed



(a) End-to-end metrics

(b) Applicative metrics

Figure 4.11: Simulated tests on the GridG platforms, with sensors on every host.

on all nodes of the platform, even on the internal routers. In the second group, only the external hosts are known to the algorithms, meaning that the sensors were only deployed on the leafs of the graph. For each set of experiments, we generated 40 different platforms, of about 60 hosts each, using the GridG [LD03] platform generator.

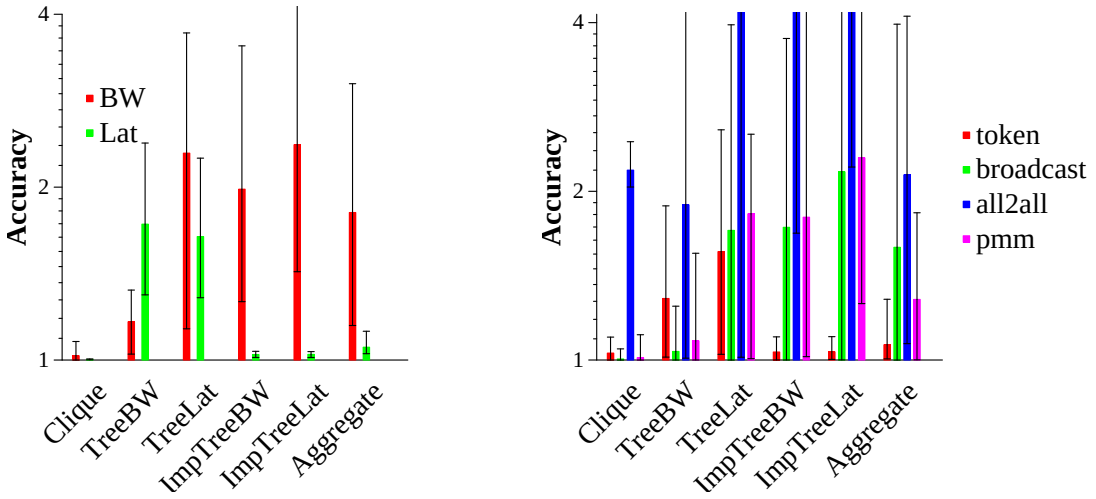
For each metric (both network- and application-level), we build an accuracy index for each reconstruction algorithm, each graph. Following [TEF07], we define the accuracy as $\max\left(\frac{x_R}{x_M}, \frac{x_M}{x_R}\right)$ where x_R is the reconstructed value and x_M is the original measured one. The best possible result is thus 1, meaning that $x_R = x_M$. We compute the accuracy of each pair of nodes, and then the geometric mean of all accuracies over a given platform.

The results are shown on Figures 4.11 and 4.12 for each set of experiments. For each metric, we plotted the average accuracy index over all test platforms, as well as the minimum and maximum accuracy indexes measured for any given platform.

Figure 4.11 presents the results when all hosts are known to the measurement infrastructure. Unsurprisingly, CLIQUE has excellent end-to-end performances whereas TREELAT and TREEBW have poor ones (the fact that CLIQUE over-estimates some bandwidths is due to routing asymmetry in the original platform). IMPTREELAT have very good end-to-end performances, better than AGGREGATE that over-estimates the bandwidth for a few couples, but IMPTREEBW behaves even better.

Regarding applicative performance, CLIQUE is unsurprisingly good for TOKEN and BROADCAST where there is always at most one communication at a time and very bad for ALL2ALL and PMM since it fails to capture the contention. Basic spanning trees presents rather good results, but this may be due to the fact that GridG platforms contain parts that are very tree-like, which are easy to reconstruct for these algorithms. The improved trees have very good predictive power, especially IMPTREEBW, with an average error of 3% on its worst case, the ALL2ALL application.

However, Figure 4.12 shows that platforms with hidden routers are much more diffi-



(a) End-to-end metrics

(b) Applicative metrics

Figure 4.12: Simulated tests on the GridG platforms, with hidden routers.

cult to reconstruct. The performance of the clique platform remains the same as before, but all other algorithms suffer from a severe degradation. Future work is clearly mandated to address this limitation.

4.2.4 Conclusion and Future Works

Retrospectively, the most interesting outcome of this work is not the proposed algorithms, that remain hardly applicable in practice, but the general approach. The workbench intended to evaluate our contributions was maybe more advanced than the reconstruction algorithms themselves, allowing to assess on simulator the use of measurements taken on real platforms. This was the my first practical tool developed to solve methodological issues. As such, it influenced my whole subsequent work, as attested by this document.

I am still convinced of the potential interest of a network tomography tool, but I must admit that I did not grant enough time to this problem recently. Fortunately, others built upon this work in between [Bob08, DFRL12]. As in [BBH05], I would like to explore the addition to new measurement metrics such as back-to-back packets. I also feel that machine learning and data mining theories constitute promising leads in this context.

4.3 Characterizing the Communication Performance of MPI Runtimes

4.3.1 Motivation and Problem Statement

Typical MPI applications being highly optimized, their performance is of major importance to their developers. As such, enabling the simulation of MPI applications (as shown in §4.1.2) is necessary, but not sufficient for the potential users to use a tool such as SMPI.

Assessing and improving the level of realism reached by the models is also mandatory to this end. This section presents a work pursuing this goal, mainly done during the post-doctoral stay of Pierre-Nicolas Clauss with me. It was summarized in [CSG+11].

This work is my first personnal implication in the 10 years long effort on the model validity in SimGrid. As detailed in §2.3.2, SimGrid relies on fluid models that represents the data streams as fluids in pipes. The first version introduced in SimGrid were accurate for the network steady state only [FC07]. TCP slow start mechanisms hindered the accuracy for transfers of less than a few megabytes. This was improved mainly by A. Legrand and P. Velho to allow the simulation of a few dozen of kilobytes [VL09b], but this was still not sufficient in practice in HPC scenarios. The usual communication patterns include very large messages conveying the data to handle as well as numerous very small messages, such as the control messages implementing the synchronizations.

Our goal here was thus to extend the SimGrid model to improve its accuracy for very small messages, while preserving its accuracy for larger messages end its scalability properties. This scalability concern rules out the use of packet-level discrete event network simulator, that are notably slowler and less extensible that our approach [FC07].

4.3.2 Proposed Model

As detailed in §3.1.1, our models are split in two parts: we first model the dynamic of each flow separately and then the contention between flows is computed separately.

Point-to-Point Model. All on-line MPI simulators reviewed in §2.4.3 use the standard affine model defining the transfer time for a message of s bytes as $\alpha + s/\beta$ where α is the network latency and β the bandwidth. Unfortunately, this model fails to capture the behavior of real-world conditions. For instance, when the message to exchange is smaller than 1 KiB, the exchanged IP frame is filled with zeros. The time to exchange small messages is thus approximately constant, regardless of their size. Also, MPI runtimes typically switch from buffered to synchronous mode above a certain message size.

SMPI captures these effects by modeling point-to-point communication times with a *piece-wise linear* model with an arbitrary number of linear segments. Each segment is obtained using linear regression on a set of real measurements. The number of segments and the segments boundaries are chosen such that the product of the correlation coefficients is maximized. In practice, we find that the model should be instantiated for 3 segments, leading to 8 parameters defining the model (2 for defining the boundaries of the 3 segments, and one latency and bandwidth parameter for each segment). This can be reduced to only 6 parameters, as some of them are actually dependent on others.

Instanciating this new model with 6 parameters is much more challenging to the user than it is with the simplistic affine model. As an answer, we provide scripts to automatically instantiate these parameters based on point-to-point experiments executed on real-world clusters. To that extend, we use the freely available SkaMPI [RST02] benchmarking framework. We gather data transfer times achieved for a wide range of message sizes using a simple ping-pong benchmark. We can then automatically fit the experimental data to a piece-wise linear model, thereby obtaining an instantiation of the required parameters. A user can easily perform such instantiation when wanting to simulate a particular

cluster deployment. Alternatively, this instantiation can be conducted by a third party, for a range of typical cluster deployments, and made publicly available. SMPI users can then reuse these instantiations, or modify them to explore reasonable "what if?" scenarios (e.g., simulate a network that achieves 30% higher data transfer rate for large messages).

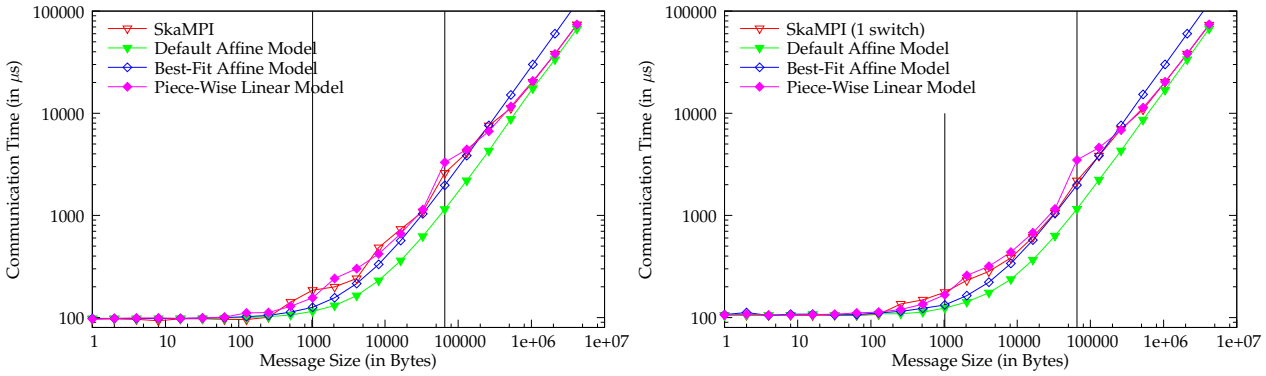
Contention Model. Among the works reviewed in §2.4.3, only [TLCS09] mentions an analytical model for network contention. However, few details are given and all results presented in this article are for a simple model that does not account for contention. In contrast, SimGrid provide an analytical network contention model accounting for standard multi-hop TCP networks. Combining this with the piece-wise linear point-to-point model described earlier preserves the advantages of each model. This leads to an immediate simulation model for collective communication operations. Just like in any MPI implementation, collective communications are implemented in SMPI as sets of point-to-point communications that may experience network contention among themselves. This is to be contrasted with monolithic modeling of collective communications [TLCS09, BLGE03].

4.3.3 Evaluation

This section presents experimental evidences of the accuracy improvement brought by the linear piece wise model of SMPI. We use several benchmarks for that, ranging from simple point-to-point communication to more complex communication benchmarks. All experiments were conducted using SMPI implemented within SimGrid v3.5-r8210. To compare simulation results to real-world measurements we ran all the MPI applications described hereafter on the following clusters of the Grid'5000 platform: *griffon* and *gdx*. The *griffon* cluster comprises 92 2.5 GHz Dual-Proc, Quad-Core, Intel Xeon L5420 nodes while the *gdx* cluster comprises 312 2.0 GHz Dual-Proc AMD Opteron 246. These cluster present similar hierarchical network interconnects by grouping nodes into cabinets and interconnecting these cabinets through a hierarchy of switches.

We did not compare our experiments by computing the relative error that is given by $Err = \frac{X-R}{R}$ (where R is the reference value and X the experimental value). Indeed, this metric is not symmetric: having X twice as large as R yields a relative error of 100%, while having X half as small as R yields a relative error of -50%. Instead, we computed the logarithmic error that is given as $Log_{Err} = |\ln X - \ln R| = |\ln R - \ln X|$. This metric is symmetric, fixing the previously observed bias. It can also be used with additive aggregation operation (e.g., maximum, mean, variance). Finally, the logarithmic error value can be interpreted as a regular percentage by taking it of the log-space: $Err = \left(e^{Log_{Err}}\right) - 1$.

Despite our intention to compare our work to the other simulators of the literature, we do not any such comparison experiment here. This is because these tools are either not publicly available (such as xSim) or not updated since years and thus probably obsolete (such as MPI-SIM). We failed to use PSINS in our context, even with the help of its authors. Its trace replay mechanism relies on a specificity of a certain version of MPICH, and the tool required to merge the traces was missing from the distribution. This sorry state seriously hinders the reproducibility of the results presented in these papers. In contrast, SimGrid (and thus SMPI) is now integrated in major Linux distributions.



(a) Run on calibration cluster *griffon*.

(b) Run on cluster *gdx*.

Figure 4.13: Comparing a SkaMPI run to SMPI predictions for a ping-pong operation, both for *on site* and *off site* usage of the calibration measurements obtained on the *griffon*.

Point-to-Point Communications This first set of experiments aims at assessing the validity of the piece-wise linear network model on a microbenchmark. We automatically calibrate the simulation as described earlier, using machines of *griffon*, and then use these values to predict a simple ping-pong. Figure 4.13 compares the predictions using this calibration in several settings to the results obtained with SkaMPI and OpenMPI.

Each sub-figure shows three sets of SMPI results. The results "Default Affine" are obtained for an affine model calibrated on the cluster using the time to send a 1-byte message for the latency and the maximum achievable bandwidth using the TCP/IP protocol (i.e., approximately 92% of the peak bandwidth). This standard method corresponds to the approach taken by many of the MPI simulators reviewed in §2.4.3. The "Best-Fit Affine" results are for an affine model instantiated using the latency and bandwidth values that minimize the average logarithmic error with respect to the SkaMPI results. We include these results to see whether a linear model could be inherently inaccurate. Finally, the "Piece-Wise Linear" results are for the model introduced in §4.3.2.

Figure 4.13a shows that the piece-wise linear model matches the real-world results very well (average error: 8.6%; worst case: 27%). By contrast, both affine models fail to capture the entire real-world behavior. The Default Affine model is accurate for small and big messages, but inaccurate in between (average error: 32%; worst case: 127%). The Best-Fit Affine model performs better for medium-sized messages, but overestimates other communications (average error: 19%; worst case: 63%). The next experiment uses the calibration obtained on *griffon* to predict timings on *gdx*. Figure 4.13b shows very similar results, with the piece-wise linear model being the most realistic (average error: 7.9%; worst case: 59%). Both Default and Best-Fit Affine models still exhibits the same drawback (average errors: 28% and 16%; worst cases: 90% and 64%).

Overall, these results induce that a piece-wise linear model is necessary for accurate simulation of MPI communication on a cluster. They also shows that *off site simulation* constitute a viable approach, relieving the burden of performing a calibration step on each target platform to obtain accurate results.

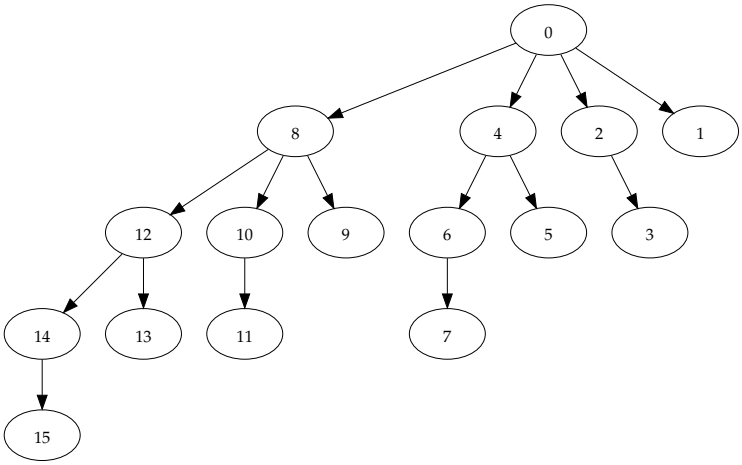
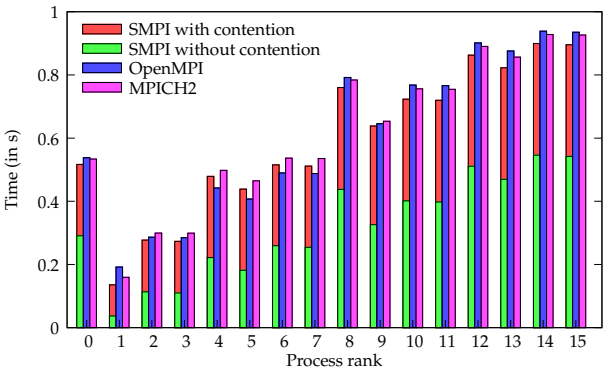


Figure 4.14: Communication scheme of a binomial tree scatter with 16 processes.

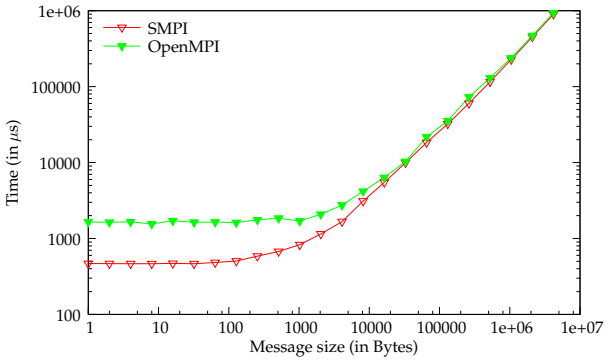
One-to-Many and Many-to-One Communications. We now assess the accuracy of SMPI using more complex communication operations. We focus on the `MPI_Scatter` function, and more precisely on the binomial tree scatter operation that is used by the major MPI implementations for that function in most circumstances. The communication pattern of this algorithm is depicted in Figure 4.14 for 16 processes. The volumes of data sent along each edge are different. For instance, P_0 sends 8 chunks of data to P_8 (then scattered in the subtree rooted in 8) while P_1 receives only one chunk of data.

In Figure 4.15 we compare the execution times of a binomial tree based scatter operation respectively achieved by classical MPI runtimes and by SMPI. To ensure that OpenMPI and MPICH2 use the binomial tree algorithm, we do not call directly `MPI_Scatter`, but use a manual implementation of this algorithm. This is realistic since this algorithm is used in most cases by the implementations. In a future version of SMPI, we plan to implement other existing algorithms and detect which algorithm to use based on the message size and number of processes, just as real implementations do.

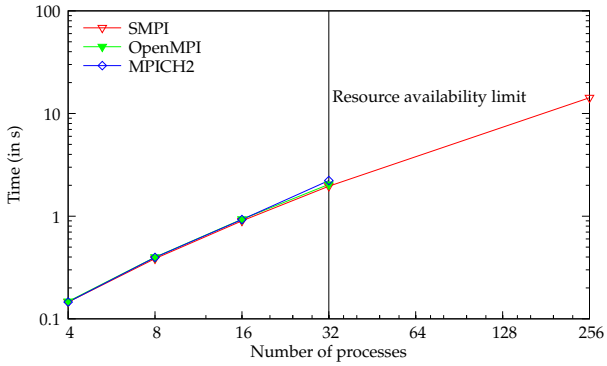
Figure 4.15a details the results per process when a 64 MiB buffer is scattered across 16 processes. The size of the receive buffers at the leaves of the binomial tree is thus 4 MiB. For the SMPI version, two execution times are displayed. The red bar shows the times to complete the scatter operation when network links suffer from contention. The green bar shows the results obtained on an equivalent platform where each communication going through one of these links will get the nominal bandwidth, whatever the number of concurrent communications. This no-contention scenario mimics for comparison purposes the behavior of most MPI simulators reviewed in §2.4.3, which do not take contention into account. The depicted result show that the network model without contention always underestimates the completion time of a scatter operation. Consequently, we claim that most of the MPI simulators previously reviewed would lead to similar underestimations. Conversely our piece-wise linear model with contention leads to simulated execution times that are very close to the performance of MPI implementations. On average, the difference between SMPI and MPICH2 is almost the same as the difference between OpenMPI and MPICH2 (average error: 5%; worst case: 20%).



(a) Per-process results for 4 MiB messages.



(b) Impact of message size for 16 processes.



(c) Impact of the processes number (4 MiB messages).

Figure 4.15: Timing results for a binomial tree based scatter operation.

Figure 4.15b shows the impact of the message size on the accuracy of SMPI for such a binomial tree based scatter operation. We see that the scatter simulation with messages over 10 KiB is reasonably accurate (under 10% error). However, with smaller messages, the simulation underestimates the real-world execution time. We hypothesize that the root cause of this error is a by-product of the fluid model used in SimGrid. This model, that computes the bandwidth allocated to each competing flow along a network path, is continuous. Conceptually, it means that all flows make progress simultaneously during each infinitesimal time unit. This is a continuous approximation of a discrete phenomenon in which the transmission of individual physical packets is serialized and packets are sent out in an interleaved manner. While the approximation error is amortized over large number of packets, i.e., for large messages, the approximation is optimistic in the case of small messages. Further work remains mandatory to assess this hypothesis.

Figure 4.15c shows the evolution of the execution time of a scatter operation with regard to the number of involved processes. Here the size of the receive buffer is constant and of size 4 MiB while the size of the data to scatter increases linearly with the number of processes. The performance of SMPI is very consistent with both MPI implementations for this message size. While the size of the actual cluster and its heavy load prevented us to make experiments with OpenMPI and MPICH for more than 32 processors, we pushed the scalability further with SMPI (up to 256 processes). We can see that the predicted execution times evolves in the same trend. Similar behavior is observed for receive buffers

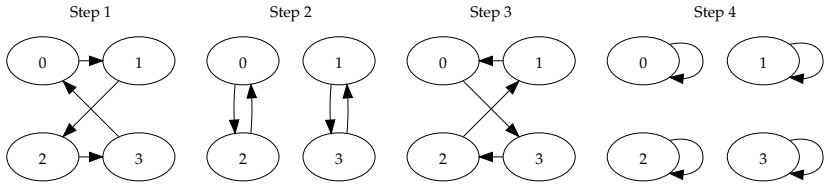
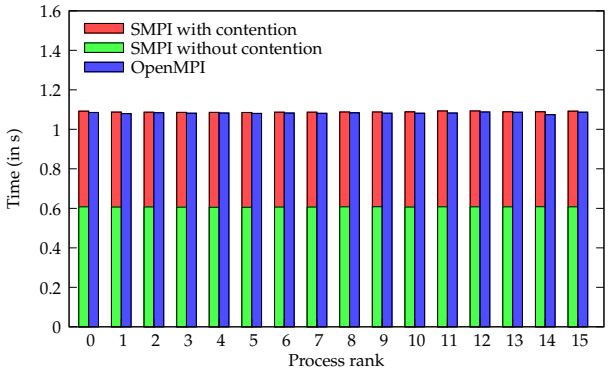


Figure 4.16: Communication scheme of a pairwise all-to-all with 4 processes.

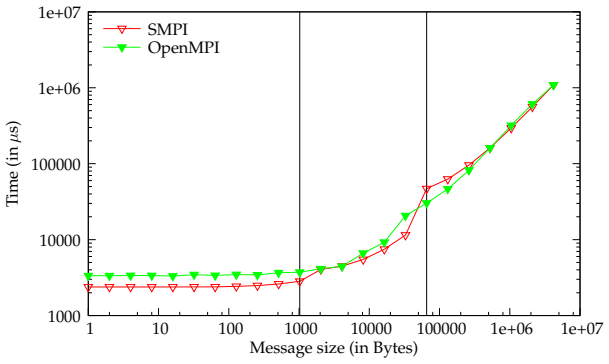
as small as 100 KiB (results not included). For smaller messages, SMPI underestimates the communication time as seen in Figure 4.15b and probably for the same reasons.

Many-to-Many Communications. The `MPI_Alltoall` function implements a collective operation in which each process sends distinct data to each of the receiver. As for scatter before, several algorithms exist for this operation, with different performance profiles depending on the conditions. The pairwise algorithm is used by OpenMPI and MPICH2 under some conditions on the message sizes and number of processes. This algorithm can be decomposed in as many steps as there are processes. At each step, each process exchanges data with a unique remote process, as depicted by Figure 4.16 for 4 processes.

We compare the accuracy of SMPI to that of a manual implementation of the pairwise algorithm with OpenMPI. As in Figure 4.15a, we show the execution times achieved with a network model that ignores contention. Figure 4.17a shows that this simple model, depicted by the green bars, induces an error of 78% that is consistent for all the 16 processes, for an all-to-all with 4 MiB messages. By contrast, the SMPI version that relies on the piece-wise linear model is accurate (less than 1% error) when accounting for contention. Figure 4.17b shows the impact of message size on the simulation accuracy (for 16 processes). The results are very similar to the Many-to-One communication schema: SMPI underestimates the transfer time for small messages (average error: 29%; worst case: 80%). We believe that the explanation presented for Figure 4.15b holds here as well.



(a) Per-process results for 4 MiB messages



(b) Impact of message size for 16 processes.

Figure 4.17: Timing results for a pairwise all-to-all operation.

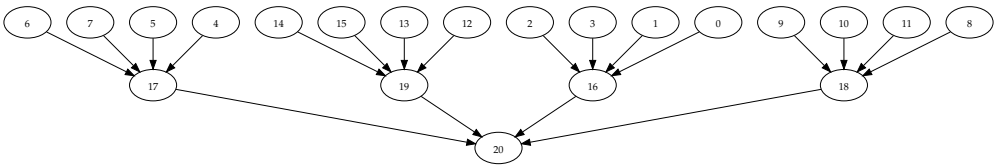


Figure 4.18: Communication scheme for the BH graph in DT Class A problem.

Data Traffic (DT) Benchmark. This application is part of the classical NAS Parallel Benchmarks (NPB) suite. As every NPB kernel, DT can be executed for 7 different *classes*, denoting different problem sizes: S (the smallest), W, A, B, C, D, and E (the largest). For instance, a class D instance corresponds to approximately 20 times as much work and a data set almost 16 as large as a class C problem. In this case, the classes also denote the number of communicating processes in addition to the data size. Moreover three communication schemes can be tested. The Black Hole variant (BH) collects data from multiple sources in a single sink, as shown in Figure 4.18. Conversely the White Hole variant (WH) distributes data from a single source to multiple consuming nodes, with a dual communication scheme than the one of Figure 4.18. The last communication variant is Shuffle (SH), that arranges the processes in different layers and shuffles data from the top layer down to the bottom layer. Classes A, B and C respectively involve 21, 43, and 85 processes for WH and BH, and 80, 192 and 448 processes for SH.

Figure 4.19 shows the comparison between SMPI and an OpenMPI implementation for the WH and BH variants of the DT benchmarks for classes A and B. Due to the access policy on the *griffon* cluster, we could not run real-world experiments with more than 43 nodes (SH variant and class C instances of WH and BH). The behavior of this complete benchmark is correctly predicted by SMPI (average error: 8.1%; worst case: 24%). SMPI is sufficiently accurate to predict the correct trend (i.e., that BH takes more time than WH) with strong confidence. Recall that although obtaining this kind of performance information with OpenMPI requires access to up to 43 nodes, SMPI provides it using a single node.

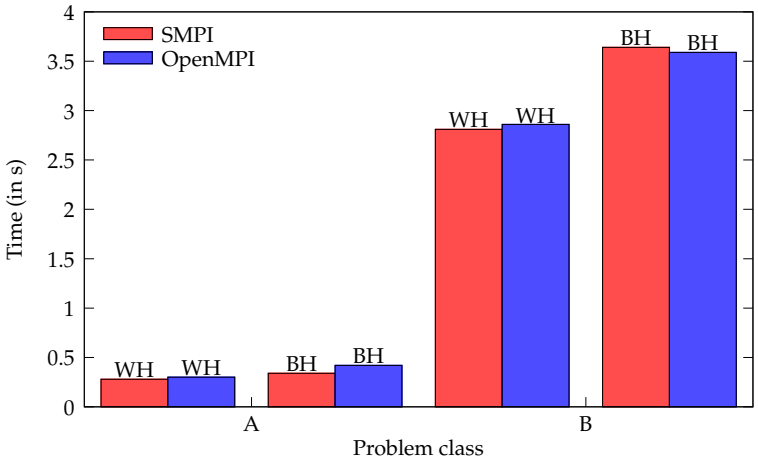


Figure 4.19: Execution time of the DT benchmark for classes A and B.

4.3.4 Conclusion and Future Work

The model accuracy improvement is an endless task. The work presented here constitutes a strong enhancement over the previous state. In particular, the timings of small messages are better predicted (as shown with the point to point experiments), but their interactions on the network contention is not correctly captured (as shown by other experiments). Since the synchronization messages (such as the one involved in a barrier) are typically very short, this lack of accuracy on very small messages is of prime importance on the overall accuracy of SMPI. From our preliminary results on the ongoing work in that direction, it seems very difficult to accurately simulate larger applications without solving this issue. We are currently pursuing a lead consisting in modeling some elements of the buffering mechanism that occur in MPI runtimes.

It must be however emphasized that even if the models proposed in SimGrid remain largely perfectible, they are much more accurate than the models used in the other tools of the literature. Most of them fail to capture the network contention (see §2.4.3), although we show experimentally that this effect is mandatory to capture in communication patterns as simple as one-to-many or many-to-one (see §4.3.3 and *e.g.*, Figure 4.15a). As explained §4.3.3, it unfortunately revealed impossible to run direct comparisons with these tools. This sorry state constitutes yet another argument for Open Science in our domain.

Another long-term goal is to simulate I/O resources and I/O operations, such as those implemented in MPI-IO. While the MPI-SIM project [BDP01] has already provided I/O simulation capabilities, recent work has also tackled I/O simulation for MPI application [NnFG+10]. Similar techniques, or models implemented as part of general-purpose I/O system simulators such as that in [BBS01], could be integrated in SimGrid for SMPI. Furthermore, the current network model is developed and validated only for TCP-based cluster interconnects, such as Gigabit Ethernet switches. Other interconnects including Myrinet or InfiniBand are currently not supported, and corresponding models need to be developed. Finally, the CPU modeling remains very prototypical in SimGrid. Given that the memory behavior of most HPC applications is highly optimized to avoid bad cache effects, a specific modeling must be undertaken to improve the simulation accuracy in SimGrid. These goals to improve the accuracy of the simulation of MPI applications are indeed very ambitious. I am nevertheless confident in the capacity of the SimGrid community to tackle these points as they constitute major goals for the ongoing SONGS ANR project. I am delighted to be the coordinator of this project and thus to be part of this great scientific adventure.

4.4 Conclusion

In this chapter, I presented several works aiming at changing the simulation kernel into a complete simulation framework. They can all be seen as modeling efforts targeting the reality. Concerning the modeling of applications being simulated, my goal was to remove the use of prototypes to allow the study of real applications directly within the simulator. Concerning the modeling of the platform, I worked on an automatic network mapping project that would be able to scout out the information of real platforms to instanciate

them within the simulator. Concerning the modeling of the network dynamics, I worked to push the accuracy limits of the network models, to target the HPC research community where the timing accuracy is a major concern.

It is notable that the works presented in this chapter bring more open questions than definitive answers. This opens the door to more research opportunities, as demonstrated with the numerous research leads concluding each sections. In particular, the network mapper is not functional yet, but the workbench we developed in this context had a great influence on my methodology and on my work on experimental methodologies. These methodologies are severely put to the test when validating the model accuracy. The most interesting difficulty in this context is that this requires to combine natural science and computer science methodologies: On one side, one has to hypothesize facts about the network behavior and then actually testing these facts through experiments; On the other side, one has to build an algorithmic model mimicking these facts without compromising the scalability of the simulation tool. This combination of methodologies is probably a commonplace of Computational Science, but when applied to Computer Science as I do, it leads to a staggering perspective where an emerging branch of Computer Science is mandated to understand the productions of another branch of Computer Science.

Chapter 5

Conclusions and Perspectives

Prediction is very difficult,
especially about the future.
– Niels Bohr

THIS CHAPTER sheds an historical light on the research activities that I presented in this document. §5.1 recaps my research trajectory through the problems that I was brought to work on, and their implications on the subsequent works. Even if *prediction is very difficult*, the rest of this chapter presents the main research directions that I plan to work on in the future. §5.2 calls for the constitution of a coherent workbench for the making of distributed systems, combining all major experimentation methodologies and fully implementing the Open Science approach in this domain.

5.1 Historical Perspectives

This document present the research activities that I conducted over the last decade. There is naturally several approaches to present this body of work. In §2.6.2, I propose a categorization depending on the **scientific methodology** used: either the theoretical tradition of mathematics (*e.g.*, with model-checking in §3.4), the empirical tradition of natural sciences (*e.g.*, to model the network dynamics in §4.3), or the engineering tradition of techniques (*e.g.*, to propose a parallel and scalable version of SimGrid in §3.2 and §3.3).

As detailed in §1.3, the overall organization of this document categorizes my work depending on the **object of study**, starting with the narrow scope of the simulation kernel in chapter 3. Chapter 4 considers the elements constituting a simulation beside of the simulation kernel. Later in this chapter, the scope widens further with perspectives on how the simulation could become part of a coherent ecosystem of experimental methodologies.

This section puts another light on my work, using an **historical perspective**. A first observation is that my trajectory was often rather *bottom-up*. I often worked on the tools that I needed myself for other studies. It explains my pragmatic orientation toward ready to use tools, since I actually needed them myself to continue previous activities.

This is best exemplified by the ALNeM project: During my PhD, I worked on a tool gathering information about the platform for network-aware applications. This led me to

the elaboration of network tomography algorithms. This eventually led to the ALNeM project as it is presented in §4.2. This work presented two main difficulties.

First, it was of uttermost importance to me to test and evaluate my contributions on a wide variety of platforms to ensure their genericity. The simpler way to do so was to test my algorithms on simulators. But since I wanted to get a pragmatic solution that would result in a tool usable in practice at the end, this was not sufficient to me. I was needing a solution to write my own network sensors, able to report bandwidth and latency measurements both on the simulator and on the real platform. That was the starting motivation of the GRAS project, presented in §4.1.1. This eventually evolved into an effort aiming at allow the study of real applications through simulation, as presented in §4.1. This also led me to the SimGrid project, that eventually became prevalent in my research.

Another challenge of the ALNeM project was the difficulty to quantify the quality of tomography algorithms, as we define the *realism* of the platform representation through the impact on the application performance. We want a platform representation on which the applications behave similarly to what can be observed on the "real" platform. This considerations led us to the development of a complete experimental workbench dedicated to the study of such tomography algorithms. This was clearly seminal of the work I did later on experimentation methodologies.

Ten years after my first attempts at the ALNeM project, I have the feeling that a cycle of my research is ending: the SimGrid simulator morphed from an scientific object into a scientific instrument, with a large community of users. This is an incredibly asset that I plan to leverage in the future. I would like to extend the SimGrid framework in two orthogonal directions, as detailed in the rest of this chapter.

5.2 Coherent Workbench for Distributed Applications

I am now more convinced than ever that experimentation is the royal way to evaluate any improvement to large-scale distributed systems, be them ideas, algorithms, prototypes or production-grade systems and applications. As depicted in Figure 5.1, the *essential complexity* of these constructs (that is, their irreducible complexity due to their very nature) tend to create a large gap between our understanding of the system and the reality, between our hypothesis and the facts. Experimentation occurs to me as the only way to close this gap, and thus to ensure that distributed systems match our expectations.

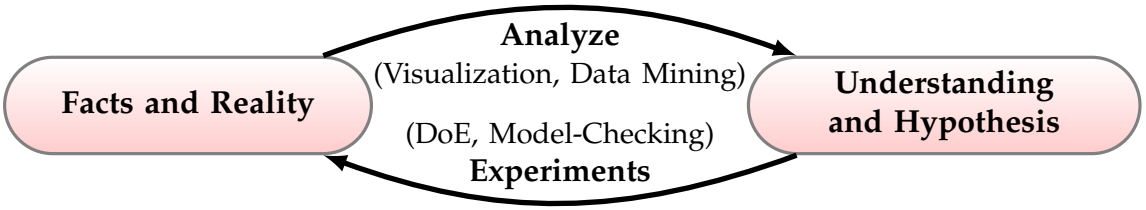


Figure 5.1: Main Motivations for Experimentation.

There is two ways to close this gap: either by building new hypotheses matching the system through *analysis*; or by assessing some existing hypotheses through *experiments*. This analysis can be done through the manual exploration of logs but it is much easier to rely on visualization; It could even be partially automatized through Data Mining. Even if it is possible to conduct experiments without any firm plan, it remains much more efficient to rely on Design of Experiments to statistically assess whether an hypothesis holds. Hypotheses can also be formally assessed through model-checking.

Regardless of the motivation to run an experiment, it can be conducted using either simulation, direct execution or emulation. In [Bro87], the author states that there is *no silver bullet* for Software Engineering, meaning that "no single development, in either technology or management technique" could magically solve all the issues faced by software development. Similarly, after a decade of research toward the experimentation of large-scale distributed systems, I get the feeling that none of the experimentation methodologies at hand is sufficient *per see*.

Instead, my personal experience argues for a combination of the differing methodologies during the making process of large-scale distributed systems. At each step of this process, some experimental methodologies are more adequate than others (cf. Figure 5.2). The boundaries are however very blurred here, let alone because the definition of each methodologies and steps of the making may differ between authors. It remains that the design of a complex distributed system is a complex process mandating to use the adapted methodologies and tools.

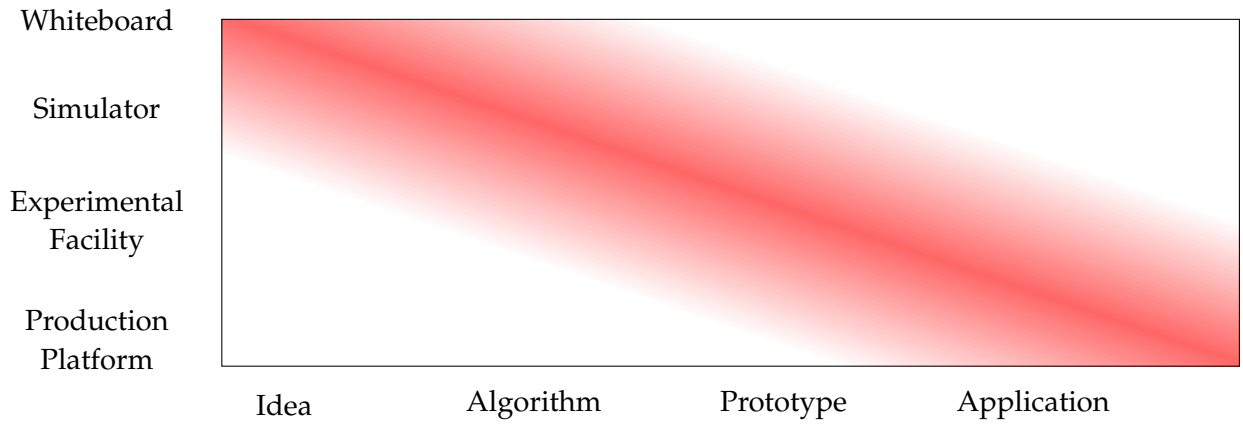


Figure 5.2: Matching between Experimentation Methodologies and Steps of the Making.

From an Ecosystem of Tools to a Coherent Workbench. I will work on the convergence of experimental methodologies such as simulation, emulation and experimental facilities. I hope to lower the barriers between these approaches, and to contribute this way to the emergence of the Computational Science in my research community.

This effort can be seen as an extension of the integration between performance simulation and dynamic formal verification (cf. §3.4). For that, I will build upon my assets with SimGrid and the Grid'5000 platform. Even if my own research never targeted this scientific instrument so far, I use it since its inception ten years ago and actively participated to the scientific animation of its community as a steering committee member for

years. I am thus well-versed on the diversity the scientific instruments for large-scale distributed systems. Some of these tools are proposed by very close colleagues: F. Desprez acts as a scientific leader for the Grid'5000 and advised my PhD; L. Nussbaum is my coworker in Nancy. He is heavily involved in the Grid'5000 instrument and proposed the Distem [BNG11] emulator.

Actually, this convergence of tools is already ongoing. For example, SimGrid's platform files can already be used to describe an equivalent experiment in Distem. I will further push this interoperability in the next years up to blur the boundaries between the simulator, the emulator and the Grid'5000 tooling up to the point where they constitute a coherent workbench. When doing so, I will pursue on my general trend toward pragmatic and ready to use tools that address the user's methodological difficulties seamlessly.

For an Open Science of Large-Scale Distributed Systems. Large-Scale Distributed Systems are very complex by essence, making any experimentation about them accordingly complex. This mandates a rigorous experimental discipline to control the scientific process, that is unfortunately still lacking in our domain (cf. §2.4.4). I recently got convinced that Open Science is the only way to tackle the difficulties that we are facing. Indeed, the models we are working on are difficult to assess and our technical constructs are difficult to get right (and easy to break through apparently unrelated change to the source code). The only valid answer to these difficulties is to provide access to all technical elements underlying any scientific affirmations. These elements are comparable to the formal proof for a theorem: of uttermost importance in a scientific work. This may sound as a pet theme to the researchers relying on Computational Science on a daily basis, but I think that this constitutes both an important mean and goal to ensure that the experimental habits of my community improves in the future.

There is still a long way to go to make this happen, and I will certainly not pretend to solve this problem all by myself. I however hope to leverage my assets about SimGrid (and to a lesser extend, about Grid'5000) to contribute to the solutions.

Acknowledgments

Most of the experiments presented in this document were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr)

Bibliography

[ABDS02] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou. Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures. *Journal of Parallel and Distributed Computing*, 62(3):393–426, 2002.

[BB09] Martin Barisits and Will Boyd. MartinWilSim Grid Simulator. Summer internship, Vienna UT and Georgia Tech, CERN, Switzerland, 2009. Available at http://www.slideshare.net/wbinventor/slides-1884876.

[BBH05] J. Byers, A. Bestavros, and K. Harfoush. Inference and labeling of metric-induced network topologies. *IEEE TPDS*, 16(11):1053 – 1065, 2005.

[BBS01] P. Berenbrink, A. Brinkmann, and C. Scheideler. SIMLAB: A Simulation Environment for Storage Area Networks. In *Proc. of the 9th Euromicro Workshop on Parallel and Distributed Processing*, pages 227–234, Feb. 2001.

[BCC+03] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger, and Floriano Zini. OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies. *International J. of High Performance Computing Applications*, 17(4), 2003.

[BCC+06] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E. Talbi, and I. Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *Int. Journal of High Performance Computing Applications*, 20(4), 2006.

[BCW] H. Burch, B. Cheswick, and A. Wool. Internet mapping project. http://www.lumeta.com/mapping.html.

[BDMT10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[BDP01] R. Bagrodia, E. Deelman, and T. Phan. Parallel Simulation of Large-Scale Parallel Applications. *International Journal of High Performance Computing Applications*, 15(1):3–12, 2001.

[BDU10] Rupali Bhardwaj, V.S. Dixit, and Anil Kr. Upadhyay. An Overview on Tools for Peer to Peer Network Simulation. *International Journal of Computer Applications*, 1(1):70–76, Feb. 2010.

[BEDW11] O. Beaumont, L. Eyraud-Dubois, and Y.-J. Won. Using the Last-mile Model as a Distributed Scheme for Available Bandwidth Prediction. In *EuroPar*, volume 6852 of *LNCS*. Springer, 2011.

[BHFC08] F. Bouabache, T. Herault, G. Fedak, and F. Cappello. Hierarchical replication techniques to ensure checkpoint storage reliability in grid environment. In *8th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2008.

[BHK07] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[BLD+12] Laurent Bobelin, Arnaud Legrand, Marquez David, Pierre Navarro, Martin Quinson, Frédéric Suter, and Christophe Thiéry. Scalable multi-purpose network representation for large scale distributed system simulation. In *12th ACM/IEEE Intl Symposium on Cluster Computing and the Grid (CCGrid'12)*, Canada, 2012.

[BLGE03] R.M. Badia, J. Labarta, J. Giménez, and F. Escalé. Dimemas: Predicting MPI applications behavior in Grid environments. In *Proc. of the Workshop on Grid Applications and Programming Tools*, 2003.

[BMA06] K. Butler, P. McDaniel, and W. Aiello. Optimizing BGP security by exploiting path stability. In *Proc. of the 13th ACM Conference on Computer and Communications Security*, 2006.

[BMM07] Jean-Yves Le Boudec, David McDonald, and Jochen Mundinger. A generic mean field convergence result for systems of interacting objects. In *Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 3–18, Washington, DC, USA, 2007. IEEE Computer Society.

[BMR02] F. Baccelli, D. R. McDonald, and J. Reynier. A mean-field model for multiple TCP connections through a buffer implementing RED. *Perform. Eval.*, 49:77–97, September 2002.

[BNG11] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. Methods for Emulation of Multi-Core CPU Performance. In *13th IEEE International Conference on High Performance Computing and Communications (HPCC-2011)*, Banff, Canada, September 2011.

[Bob08] L. Bobelin. *Tomographie depuis plusieurs sources vers de multiples destinations dans les réseaux de grilles informatiques hautes performances*. PhD thesis, Université de la Méditerranée – Aix-Marseille II, 2008.

[Box87] George E. P. Box. *Empirical Model-Building and Response Surfaces*. 1987. ISBN 0471810339.

[Bri08] Cyril Briquet. *Systematic Cooperation in P2P Grids*. PhD thesis, University of Liège, Belgium, Oct 2008.

[Bro87] F Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), 1987.

[BRR+01] Vijay Balakrishnan, Radharamanan Radhakrishnan, Dhananjai Madhava Rao, Nael Abu-Ghazaleh, and Philip Wilsey. A performance and scalability analysis framework for parallel discrete event simulators. *Simulation Practice and Theory*, 8(8), 2001.

[Buf08] Protocol Buffers. Google's data interchange format. `http://code.google.com/p/protobuf`, 2008.

[Cas01] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the First IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, Brisbane, Australia, 2001.

[Cas04] Henri Casanova. Modeling large-scale platforms for the analysis and the simulation of scheduling strategies. In *IPDPS*, April 2004.

[CBNEB11] Bogdan Cornea, Julien Bourgeois, The Tung Nguyen, and Didier El-Baz. Performance prediction in a decentralized environment for peer-to-peer computing. In *IPDPS Workshops - HotP2P'11: International Workshop on Hot Topics in Peer-to-Peer Systems*. IEEE, 2011.

[CD97] H. Casanova and J. Dongarra. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *The Int. Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[CD06] Eddy Caron and Frédéric Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[CDCV06] Pushpinder-Kaur Chouhan, Holly Dail, Eddy Caron, and Frédéric Vivien. Automatic middleware deployment planning on clusters. *IJHPCA*, 20(4):517–530, November 2006.

[CDGJ10] Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt, and Emmanuel Jeannot. Defining and controlling the heterogeneity of a cluster: The Wrekavoc tool. *J. Syst. Softw.*, 83:786–802, May 2010.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50:752–794, September 2003.

[CGM+89] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[CLM03] Henri Casanova, Arnaud Legrand, and Loris Marchal. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2003.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar 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, January 2011.

[CSG+11] Pierre-Nicolas Clauss, Mark Stillwell, Stéphane Genaud, Frédéric Suter, Henri Casanova, and Martin Quinson. Single node on-line simulation of mpi applications with smpi. In *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, Alaska, USA, 2011.

[dBKB02] Mathijs den Burger, Thilo Kielmann, and Henri E. Bal. TOPOMON: A monitoring tool for grid network topology. In *ICCS 2002*, volume 2330 of *LNCS*, pages 558–567, 2002.

[DCKM04] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *ACM SIGCOMM*, 2004.

[DCLV10] Bruno Donassolo, Henri Casanova, Arnaud Legrand, and Pedro Velho. Fast and scalable simulation of volunteer computing systems using simgrid. In *Proceedings of the Workshop on Large-Scale System and Application Performance (LSAP)*, Chicago, IL, 2010.

[DDMVB08] W. Depoorter, N. De Moor, K. Vanmechelen, and J. Broeckhove. Scalability of Grid Simulators : An Evaluation. In *Proc. of the 14th EuroPar Conference*, number 5168 in LNCS. Springer, 2008.

[Des34] René Descartes. *Discourse on the Method*. 1634.

[DFRL12] K. Dichev, F. Fergal Reid, and A. Lastovetsky. Efficient and reliable network tomography in heterogeneous networks using bittorrent broadcasts and clustering algorithms. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, 2012.

[DG06] Robert A. Day and Barbara Gastel. *How to write and publish a scientific paper*. Cambridge University Press, 2006. ISBN: 978-0-52167-167-5.

[DGK+01] P. Dinda, T. Gross, R. Karrer, B Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland. The architecture of the remos system. In *HPDC-10*, 2001.

[DHN96] P. Dickens, P. Heidelberger, and D. Nicol. Parallelized Direct Execution Simulation of Message-Passing Parallel Programs. *IEEE Trans. on Parallel and Distributed Systems*, 7(10):1090–1105, 1996.

[DLTM08] Tien Tuan Anh Dinh, Michael Lees, Georgios Theodoropoulos, and Rob 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.

[DMVB08] Wim Depoorter, Nils Moor, Kurt Vanmechelen, and Jan Broeckhove. Scalability of grid simulators: An evaluation. In *Proc. of the 14th Intl. Euro-Par Conf. on Parallel Processing*, 2008.

[DMVB09] S. De Munck, K. Vanmechelen, and J. Broeckhove. Improving The Scalability of SimGrid Using Dynamic Routing. In *Proc. of the 9th Int. Conference on Computational Science (ICCS)*, 2009.

[Dow99] Allen B. Downey. Using pathchar to estimate internet link characteristics. In *Measurement and Modeling of Computer Systems*, pages 222–223, 1999.

[Dow09] Gilles Dowek. *Ces préjugés qui nous encombrent*. Le pommier, 2009. ISBN: 978-2-7465-0448-6.

[Dow11] Gilles Dowek. Epistémologie de l'informatique. Grenoble, 3 february 2011. Oral communication at Congrès SPECIF 2011.

[DS10] Frédéric Desprez and Frédéric Suter. A bi-criteria algorithm for scheduling parallel task graphs on clusters. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, may 2010.

[dt04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[EBS02] G. Eisenhauer, F. Bustamante, and K. Schwan. Native Data Representation: An efficient wire format for high-performance distributed computing. *IEEE TPDS*, 13(12):1234–1246, 2002.

[EDLQV07] Lionel Eyraud-Dubois, Arnaud Legrand, Martin Quinson, and Frédéric Vivien. A first step towards automatically building network representations. In LNCS, editor, *13th International EuroPar Conference*, number 4641, 2007.

[EG11] Nathan Evans and Christian Grothoff. Beyond simulation: Large-scale distributed emulation of p2p protocols. In *4th Workshop on Cyber Security Experimentation and Test (CSET 2011)*. USENIX Association, 2011.

[exa] International exascale software project. http://www.exascale.org.

[FC07] Kayo Fujiwara and Henri Casanova. Speed and accuracy of network simulation in the simgrid framework. In *Proceedings of the First International Workshop on Network Simulation Tools (NSTools)*, Nantes, France, 2007.

[Fel90] S. I. Feldman. A fortran to c converter. *SIGPLAN Fortran Forum*, 9(2):21–22, October 1990.

[Fer95] Alois Ferscha. *Parallel and Distributed Simulation of Discrete Event Systems*. McGraw-Hill, 1995.

[Fet88] James H. Fetzer. Program verification: the very idea. *Communication of the ACM*, 31(9), Sept 1988.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[FJ91] Sally Floyd and Van Jacobson. Traffic Phase Effects in Packet-Switched Gateways. *SIGCOMM Comput. Commun. Rev.*, 21:26–42, April 1991.

[FJJ+01] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. Idmaps: A global internet host distance estimation service. *IEEE/ACM Transactions on Networking*, October 2001.

[FM07] Thomas Ferrandiz and Vania Marangozova. Managing scheduling and replication in the lhc grid. In *CoreGrid Workshop on middleware*, 2007.

[FQS08] Marc-Eduar Frincu, Martin Quinson, and Frédéric Suter. Handling very large platforms with the new simgrid platform description formalism. Technical Report 348, INRIA, Feb. 2008.

[FS09] W. Feng and T. Scogland. The Green500 list: Year one. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–7, Washington, DC, USA, 2009. IEEE Computer Society.

[GB02] Thomas J. Giuli and Mary Baker. Narses: A Scalable Flow-Based Network Simulator. *Computing Research Repository*, cs.PF/0211, 2002.

[GC05] D. A. Grove and P. D. Coddington. Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers. *Journal of Supercomputing*, 34(2):201–217, 2005.

[GKL+] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. P2PSim. http://pdos.csail.mit.edu/p2psim/.

[GKOH00] Jeff Gibson, Robert Kunz, David Ofelt, and Mark Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Architectural Support for Programming Languages and Operating Systems*, pages 49–58, 2000.

[GLS99] W. Gropp, E. Lusk, and A Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Scientific And Engineering Computation Series. MIT Press, 2nd edition, 1999.

[GNQ11] Marion Guthmuller, Lucas Nussbaum, and Martin Quinson. émulation d'applications distribuées sur des plates-formes virtuelles simulées. In *Rencontres francophones du Parallélisme (RenPar'20)*, Saint Malo, France, May 2011.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 176–185, London, UK, 1991. Springer-Verlag.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of programming languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM.

[GR10] Abdou Guermouche and Hélène Renard. A First Step to the Evaluation of SimGrid in the Context of a Real Application. In *19th International Heterogeneity in Computing Workshop(HCW 2010) AR=58%*, , pages 1 – 10, ATLANTA (Georgia) USA, April 2010. U.S. Office of Naval Research and by the IEEE Computer Society, IEEE Computer Society Press.

[gre] The Green500 List: Environmentally Responsible Supercomputing. http://www.green500.org.

[GVV08] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. DieCast: testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[GWZ+11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[Hen92] B. Henderson. Modularization and McCabe's Cyclomatic Complexity. *Communications of the ACM*, 37(12):17–19, 1992.

[HFH08] Eric Heien, Noriyuki Fujimoto, and Kenichi Hagihara. Computing low latency batches with unreliable workers in volunteer computing environments. In *Proceedings of the Workshop on Volunteer Computing and Desktop Grids (PCGrid 2008)*, Miami, FL, 2008.

[HGWW09] M.-A. Hermanns, M. Geimer, F. Wolf, and B. J. N. Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 78–84, 2009.

[Hin07] Pieter Hintjens. ∅mq: The guide. http://zguide.zeromq.org/, 2007.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[HSL09] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 574–581, Washington, DC, USA, 2009. IEEE Computer Society.

[HSL10a] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proc. of the 2nd Workshop on Large-Scale System and Application Performance*, 2010.

[HSL10b] T. Hoefler, C. Siebert, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, 2010.

[Ios01] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. 16th IEEE Intl. Conf. Automated software engineering (ASE 2001)*, pages 254–261, Washington, DC, USA, 2001. IEEE Computer Society.

[jag] The Jaguar XT5 system. http://www.nccs.gov/jaguar.

[JKVA11] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, and David P. Anderson. Discovering Statistical Models of Availability in Large Distributed Systems: An Empirical Study of SETI@home. *IEEE Transactions on Parallel and Distributed Systems*, 22(11):1896–1903, Nov 2011.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), October 2009.

[JMJV] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. PeerSim. http://peersim.sourceforge.net/.

[KAB+07] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Not.*, 42:179–188, June 2007.

[KAJV07] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association.

[KD10] Jim Keniston and Srikar Dronamraju. Uprobes: User-space probes. In *Linux Foundation Collaboration Summit*, 2010. http://events.linuxfoundation.org/slides/lfcs2010_keniston.pdf.

[KHB+99] Thilo Kielmann, Rutger Hofman, Henri Bal, Aske Plaat, and Raoul Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[Koo08] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3), 2008.

[Kun11] Vivek Kundra. Federal Cloud Computing Strategy. Technical report, The White House, United State of America, February 2011. Available at http://ctovision.com/wp-content/uploads/2011/02/Federal-Cloud-Computing-Strategy1.pdf.

[LA04] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–86, march 2004.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.

[Lam07] Leslie Lamport. A +CAL user's manual. http://research.microsoft.com/en-us/um/people/lamport/tla/pluscal.html, 2007.

[LB99] B. Lowekamp and A. Beguelin. ECO: Efficient collective operations for communication on heterogeneous networks. In *IPDPS'96*, 1999.

[LD03] D. Lu and P. Dinda. Synthesizing realistic computational grids. In *Proceedings of ACM/IEEE Supercomputing 2003 (SC 2003)*, November 2003.

[Lea10] Dawn Leaf. NIST Cloud Computing Program Overview. Available at http://www.nist.gov/itl/cloud/upload/Leaf-CCW-II-2.pdf, November 2010.

[LFHKM05] F. Le Fessant, S. Handurukande, A. Kermarrec, and L. Massoulié. Clustering in peer-to-peer file sharing workloads. *Peer-to-Peer Systems III*, pages 217–226, 2005.

[Liu09] Jason Liu. *Wiley Encyclopedia of Operations Research and Management Science*, chapter Parallel discrete-event simulation. 2009.

[LK00] Averill M. Law and David W. Kelton. *Simulation Modelling and Analysis*. McGraw-Hill Education - Europe, 2000.

[LM90] J.-L. Le Moigne. La science informatique va-t-elle construire sa propre épistémologie ? *Culture Technique*, (21):16–31, Juil. 1990.

[LRM09] E. León, R. Riesen, and A. Maccabe. Instruction-Level Simulation of a Cluster at Scale. In *Proc. of the International Conference for High Performance Computing and Communications (SC)*, November 2009.

[LRRV04] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Mapping and load-balancing iterative computations on heterogeneous clusters with shared links. *IEEE Trans. Parallel Distributed Systems*, 15(6):546–558, 2004.

[Mil] Rich Miller. Microsoft's 198 megawatts of motivation. http://www.datacenterknowledge.com/archives/2008/04/04/microsofts-198-megawatts-of-motivation/.

[Mit01] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12:1094–1104, October 2001.

[MPC+02] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[MPLH06] Gilles Muller, Yoann Padioleau, Julia Lawall, and Rydhof Hansen. Semantic patches considered helpful. *SIGOPS Oper. Syst. Rev.*, 40(3):90–92, July 2006.

[MQ08] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 362–371, New York, NY, USA, 2008. ACM.

[MQR11] Stephan Merz, Martin Quinson, and Cristian Rosa. Simgrid mc: Verification support for a multi-api simulation platform. In Lecture Notes in Computer Science 6722, editor, *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FMOODS/FORTE 2011)*, 2011.

[MR99] Laurent Massoulié and James Roberts. Bandwidth Sharing: Objectives and Algorithms. In *INFOCOM*, volume 3, pages 1395–1403, 1999.

[MSMO97] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27:67–82, July 1997.

[NBLR06] 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.

[NnFG+10] A. Núñez, J. Fernández, J. Garcia, F. Garcia, and J. Carretero. New Techniques for Simulating High Performance MPI Applications on Large Storage Networks. *Journal of Supercomputing*, 51(1):40–57, 2010.

[ns2] The Network Simulator (ns2). http://nsnam.isi.edu/nsnam/.

[NVPC+11] A. Núñez, J. Vázquez-Poletti, A. Caminero, J. Carretero, and I. M. Llorente. Design of a New Cloud Computing Simulation Platform. In *Proc of the 11th Intl Conf. on Computational Science and its Applications*, 2011.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[NZ02] T.S.E. Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM*, volume 1, pages 170– 179, 2002.

[OKM97] T. Ott, J. Kemperman, and M. Mathis. Window Size Behavior in TCP/IP with Constant Loss Probabillity. In *4th IEEE Workshop on High-Performance Communication Systems*, June 1997.

[OPF10] Simon Ostermann, Radu Prodan, and Thomas Fahringer. Dynamic cloud provisioning for scientific grid workflows. In *11th ACM/IEEE International Conference on Grid Computing*, Brussels, Belgium, October 2010.

[PDB00] S. Prakash, E. Deelman, and R. Bagrodia. Asynchronous Parallel Simulation of Parallel Programs. *IEEE Trans. on Software Engineering*, 26(5):385–400, 2000.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '98, pages 303–314, New York, NY, USA, 1998. ACM.

[PGK07] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *Proc. ACM Wsh. Parallel and distributed systems: testing and debugging (PADTAD 2007)*, pages 43–53, London, UK, 2007. ACM.

[PGSA09] Jordi Pujol, Pedro Garcia, Marc Sanchez, and Marcel Arrufat. An extensible simulation tool for overlay networks and services. In *Proceedings of 24th Annual ACM Symposium on Applied Computing (SAC' 09)*, Hawaii, USA, March 2009.

[PWTR09] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler. MPI-NetSim: A network simulation module for MPI. In *Proc. of the 15th International Conference on Parallel and Distributed Systems*, 2009.

[QRT12] Martin Quinson, Cristian Rosa, and Christophe Thiéry. Parallel simulation of peer-to-peer systems. In *12th ACM/IEEE Intl Symposium on Cluster Computing and the Grid (CCGrid'12)*, Canada, 2012.

[Qui03] Martin Quinson. *Découverte automatique des caractéristiques et capacités d'une plate-forme de calcul distribué.* PhD thesis, École normale supérieure de Lyon, Dec. 2003.

[Qui06] Martin Quinson. Gras: a research and development framework for grid services. In *18th IASTED Intl Conf. on Parallel and Distributed Computing and Systems (PDCS06)*, 2006. Best paper award in "software" track.

[QWB+09] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for internet-scale systems. *SIGCOMM Comput. Commun. Rev.*, 39(4):123–134, 2009.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *LNCS*, 2218, 2001.

[RF02] Kavitha Ranganathan and Ian 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)*, Washington, DC, USA, 2002. IEEE Computer Society.

[Rie06] R. Riesen. A Hybrid MPI Simulator. In *Proc. of the IEEE International Conference on Cluster Computing*, pages 1–9, September 2006.

[RMQ10] Cristian Rosa, Stephan Merz, and Martin Quinson. A simple model of communication apis – application to dynamic partial-order reduction. In Lecture Notes in Computer Science 6722, editor, *10th International Workshop on Automated Verification of Critical Systems (AVOCS'10)*, 2010.

[RST02] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: a Comprehensive Benchmark for Public Benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[SAK07] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.

[SBW99] Gary Shao, Francine Berman, and Rich Wolski. Using effective network views to promote distributed application performance. In *PDPTA*, June 1999.

[SCV+08] Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic, and Rajkumar Buyya. A Toolkit for Modelling and Simulating Data Grids: An Extension to GridSim, . *Concurrency and Computation: Practice and Experience (CCPE)*, 20(13):1591–1609, September 2008.

[SCW+02] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Proc. of the International Conference for High Performance Computing and Communications (SC)*, November 2002.

[SHN10] Lucas Schnorr, Guillaume Huard, and Philippe Navaux. Triva: Interactive 3d visualization for performance analysis of parallel applications. *Future Generation Computer Systems*, 26(3):348 – 358, 2010.

[Sim71] Herbert Simon. *Computers, Communications and the Public Interest*. The Johns Hopkins Press, 1971.

[SM06] S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[Sto03] Stoica, I. et Al. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[STP+09] Spyros Sioutas, Kostas Tsichlas, George Papaloukopoulos, Yannis Manolopoulos, and Evangelos Sakkopoulos. A novel Distributed P2P Simulator Architecture: D-P2P-Sim, 2009.

[Ted07] Matti Tedre. Know your discipline: Teaching the philosophy of computer science. *Journal of Information Technology Education*, 6, 2007.

[TEF07] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. In *IEEE TPDS*, 2007.

[TKVRB91] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert Van Renesse, and Henri E. Bal. The amoeba distributed operating system—a status report. *Comput. Commun.*, 14(6):324–335, 1991.

[TLCS09] M.M. Tikir, M.A. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proc. of the 15th International Euro-Par Conference on Parallel Processing*, pages 135–148, 2009.

[top] The top 500 ranking. http://www.top500.org.

[VAG+10] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B.R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, nov. 2010.

[Var] A. Varga. Omnet++ community site. http://www.omnetpp.org/.

[Var09] Franck Varenne. *Qu'est ce que l'informatique*. collection "Chemins Philosophiques". Vrin, Paris, 2009. ISBN : 978-2-7116-2178-1.

[Var10] Franck Varenne. *Formaliser le vivant : Lois, théories, modèles ?* Visions des Sciences. Hermann, Paris, 2010. ISBN: 9782705670894.

[VGA+07] J. Vanegue, T. Garnier, J. Auto, S. Roy, and R. Lesiank. Next generation debuggers for reverse engineering. In *4th Annual Hackers To Hackers Conference (BlackHat Europe)*, 2007.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 3–12, Washington, DC, USA, 2000. IEEE Computer Society.

[VL09a] 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.

[VL09b] Pedro Velho and Arnaud Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SIMUTools'09)*, Rome, Italy, March 2009.

[Vua05] Martin Vuagnoux. Autodafé: an act of software torture. In *22nd Chaos Communications Congress*, Berlin, 2005.

[VVD+09] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. *SIGPLAN Not.*, 44(4):261–270, 2009.

[VVGK09] Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. Reduced Execution Semantics of MPI: From Theory to Practice. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 724–740, Berlin, Heidelberg, 2009. Springer-Verlag.

[VYW+02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[Win06] Jannette Wing. Computational thinking. *Communication of the ACM*, 49(3), March 2006.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI'02*, Boston, MA, 2002.

[WSH99] R. Wolski, N. Spring, and J. Hayes. The NWS: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6):757–768, 1999.

[XDCC04] H. Xia, H. Dail, H. Casanova, and A. Chien. The MicroGrid: Using Emulation to Predict Application Performance in Diverse Grid Network Environments. In *Workshop on Challenges of Large Applications in Distributed Environments*, Honolulu, June 2004.

[XWWT11] He XU, Suo-ping WANG, Ru-chuan WANG, and Ping TAN. A survey of peer-to-peer simulators and simulation technology. *JCIT: Journal of Convergence Information Technology*, 6(5):260–272, May 2011.

[YCW+09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[YKKK09] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[ZCZ10] J. Zhai, W. Chen, and W. Zheng. PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, January 2010.

[ZKK04] G. Zheng, G. Kakulapati, and L. V. Kale. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[ZPK00] Bernard Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation : integrating discrete event and continuous complex dynamic systems*. San Diego : Academic Press, 2000.