# Lowering Entry Barriers to Developing Custom Simulators of Distributed Applications and Platforms with SimGrid*

Henri Casanova[1], Arnaud Giersch[2], Arnaud Legrand[3], Martin Quinson[4], Frédéric Suter[5]

[1]Information and Computer Sciences, University of Hawai'i at Manoa, Honolulu, HI, USA
[2]Université Marie et Louis Pasteur, CNRS, Institut FEMTO-ST, F-90000 Belfort, France
[3]Université Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France
[4]Rennes University, Inria, IRISA, Rennes, France
[5]Oak Ridge National Laboratory, Oak Ridge, TN, USA

## Abstract

Researchers in parallel and distributed computing (PDC) often resort to simulation because experiments conducted using a simulator can be for arbitrary experimental scenarios, are less resource-, labor-, and time-consuming than their real-world counterparts, and are perfectly repeatable and observable. Many frameworks have been developed to ease the development of PDC simulators, and these frameworks provide different levels of accuracy, scalability, versatility, extensibility, and usability. The SimGrid framework has been used by many PDC researchers to produce a wide range of simulators for over two decades. Its popularity is due to a large emphasis placed on accuracy, scalability, and versatility, and is in spite of shortcomings in terms of extensibility and usability. Although SimGrid provides sensible simulation models for the common case, it was difficult for users to extend these models to meet domain-specific needs. Furthermore, SimGrid only provided relatively low-level simulation abstractions, making the implementation of a simulator of a complex system a labor-intensive undertaking. In this work we describe developments in the last decade that have contributed to vastly improving extensibility and usability, thus lowering or removing entry barriers for users to develop custom SimGrid simulators.

## 1 Introduction

Many parallel and distributed computing (PDC) research results are obtained, at least in part, based on experiments conducted in simulation. Reasons for PDC researchers to use simulation are: the ability to explore arbitrary experimental scenarios; the fact that simulation experiments can be less labor-, time- and resource-intensive than their real-world counterparts; and the fact that simulation experiments can be precisely controlled and instrumented, making them perfectly observable and repeatable. Many discrete-event simulation frameworks have been developed to provide abstractions and models for compute, communication, and I/O resources and their usage, thus easing the implementation of PDC simulators. The main concerns for these framework are: (i) *accuracy* – do their simulation models make it possible for the simulated behavior to match that of the real-world system being simulated? (ii) *scalability* – do they allow the simulation of large and long-running scenarios with low computational complexity and memory footprint? (iii) *versatility* – can they be used to develop simulators for a diverse range of PDC scenarios and domains? (iv) *extensibility* – can their simulation models be configured, extended, or even replaced so that the simulator can serve specific purposes? and (v) *usability* – do they make it possible to implement simulators with low software engineering and development effort?

PDC simulation frameworks have been developed for decades, placing different levels of emphasis on and using different approaches for achieving compromises between these often conflicting concerns. For instance, a possible approach for achieving higher accuracy is to implement the simulation at a higher level of detail, but doing so typically increases computational complexity and memory footprint, thus reducing scalability. A way to increase scalability is to target a specific PDC domain, which makes it possible to eschew simulating features of the real-world system that are not relevant to that specific domain. For instance, when simulating an IoT system in which only small messages are exchanged, one can forgo the simulation of network contention effects and still produce meaningful results. But doing so reduces versatility by design.

The SimGrid framework [1] has demonstrated that it is

*This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doepublic-access-plan).

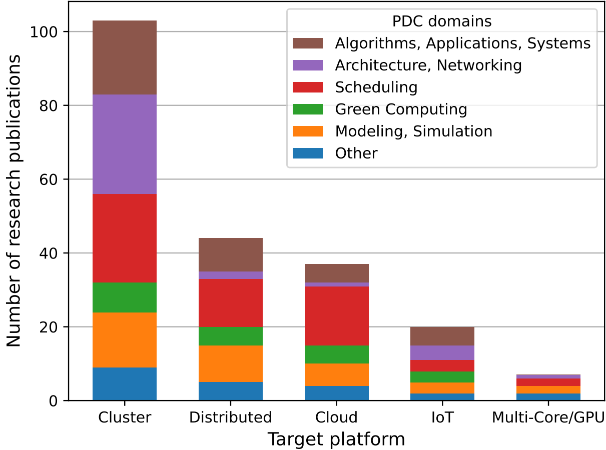



Figure 1: Publication counts by PDC domains and platforms for 176 research publications between 2016 and 2022 that include SimGrid-driven simulation results. The sum of the counts is larger than 176 because some publications target multiple domains and/or platform scenarios.

possible to design and implement a simulation framework that achieves high accuracy, scalability, and versatility, as shown in the reference SimGrid paper published in 2014 [2] and in many validation studies [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].

In [2] it is claimed that SimGrid could be used to develop simulators for a broad range of PDC domains, thus achieving versatility. We keep track of research publications that include simulation results obtained with the SimGrid software [16], which makes it possible to verify the validity of that claim. Figure 1 shows the counts of the 176 research publications between 2016 and 2022 that use SimGrid (out of 646 such publications to date since 2001). We have categorized each publication by target PDC domain and target platform scenario based on our own inspection of each publication's content. The "Other" category for the target PDC domains includes publications that focus on fault-tolerance, model-checking, security, and education. Although our categorization of some research publications may not be 100% accurate, the counts clearly show SimGrid's versatility.

Despite its popularity, SimGrid had severe shortcomings in terms of extensibility and usability. First, it was difficult for users to customize, extend, let alone replace, simulation models. While these simulation models are sensible for the common case, the range of research questions that users seek to answer using simulation is large and often requires going beyond SimGrid's extant models. Second, SimGrid only provided relatively low-level simulation abstractions and corresponding APIs. These shortcomings were seen plainly in user-provided feedback and through direct interactions with users during SimGrid-focused events and hackathons, and also mentioned in the literature [17]. Then, implementing a simulator of custom and/or complex systems often required large software engineering and development efforts.

In the last decade, since the publication of [2], which used SimGrid v3.10 (which we call v3 for simplicity), numerous efforts have focused on improving SimGrid's extensibility and usability. **The overall goal was to lower or remove entry barriers for users to develop custom and/or complex simulators, while retaining all of SimGrid's accuracy, scalability, and versatility advantages**. To achieve this goal, many features have been released over a 10-year development cycle up to v3.36, culminating in the SimGrid v4 release. While SimGrid has been the subject of several previous publications, this work describes technical developments and report on measures of success that have never been previously published. Specifically:

1. A description of the fundamental simulation abstractions in SimGrid v4 (Section 3);
2. A description of mechanisms for customizing or replacing simulation models underlying these abstractions (Section 4);
3. A description of how these abstractions can be combined into higher-level abstractions (Section 5);
4. A demonstration of how a single simulator can employ several programming models (Section 5);
5. A survey of how the above has enabled a wide range of use cases (Section 6); and
6. A quantitative comparison of simulation scalability of SimGrid v3 and SimGrid v4 (Section 7).

## 2 Related work

There is a large literature devoted to the simulation of PDC platforms and applications with many proposed PDC simulation frameworks. Some of these frameworks have garnered sizable user communities and are still actively maintained at the time of writing. They can be placed into two broad categories based on the level of details of their underlying simulation models, with different implications on accuracy, scalability, and versatility.

Perhaps the most natural approach is to implement simulation models at a high level of details, in an attempt to reproduce near-exact real-world behaviors so as to achieve high accuracy. Many simulation frameworks have followed this approach for simulating network resources (packet-level simulators [18, 19]), compute resources (cycle-accurate simulators [20, 21]), and I/O resources (block-accurate simulators [22]). These frameworks are not versatile from a PDC standpoint because they each focus on a particular class of hardware components, and as such are typically used by researchers who specialize in studying these components. While it is conceivable to combine multiple such frameworks to create a versatile PDC simulation framework, doing so is typically impractical due to scalability limitations as simulation time is typically orders of magnitude longer than simulated time. Some PDC simulation frameworks have been developed that simulate net-

work communications at the packet level for high accuracy, with typically a severe loss in scalability, but simulate other components at lower levels of details [23, 24].

One solution to address the scalability issue posed by simulation models that implement a high level of detail is to employ Parallel Discrete Event Simulation (PDES) [25]. This approach has been used successfully by PDC simulation frameworks developed for simulating high-performance computing (HPC) applications and platforms [26, 27, 28, 29, 30]. The use of PDES allows these frameworks to scale up the platform on which the simulation is executed, possibly requiring a platform of the same scale as that of the platform being simulated. By contrast, many other PDC simulation frameworks, as discussed hereafter, aim to achieve simulation scalability when executing the simulation on a single compute node (or even a single core).

An alternate approach for achieving scalable simulations, without scaling up compute resources, is to implement simulation models at a relatively low level of detail. That is, rather than simulating "microscopic" behaviors of the target system, these models instead rely on mathematical models that aim to capture "macroscopic" behaviors. For instance, rather than simulating the lifecycle of individual network packets, they compute instantaneous data transfer rates based on network path bandwidths and latencies and on current network usage. The main challenge is to come up with such coarse-grain simulation models that are reasonably accurate in spite of this low level of detail [4].

Several PDC frameworks have been developed using the above approach. The most notable two such frameworks are SimGrid and GridSim [31], both of which have garnered large user communities, but using different approaches for and achieving different levels of versatility, accuracy, scalability, usability, and extensibility. SimGrid aims to be directly versatile across a large range of PDC domains (see Figure 1). Instead, versatility has been achieved in GridSim by implementing domain-specific frameworks on top of it such as CloudSim [32] (which was later re-implemented standalone reusing a subset of the GridSim simulation abstractions) for cloud simulations, iFogSim [33] for IoT simulations, DISSECT-CF [34] for IoT and cloud simulations, GroudSim [35] for grid and cloud simulations, or OpenDC [36] and DCSim [37] for data center simulations. The main focus of SimGrid up to v3 was the development of accurate and scalable simulation models via extensive validation and invalidation studies [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].

GridSim and the frameworks built on top of it have provided high usability (due to providing higher-level simulation abstractions) and high extensibility (as seen in the number of works that have successfully extended CloudSim, e.g., [38, 39, 40, 41]). By contrast, SimGrid v3 has had usability and extensibility shortcomings, as already discussed in Section 1. This paper focuses on the technical developments that have addressed these shortcomings, while preserving simulation accuracy and scalability.

# 3 Fundamental simulation abstractions

SimGrid v4 provides convenient APIs, and efficient implementations thereof, for three fundamental simulation abstractions that are sufficient to describe and simulate virtually any PDC scenario: resource, activity, and actor. SimGrid v3 provided abstractions similar in intent, but suffered from a lack of separation of concern between the models that determine the behaviors of resources, the activities that consume these resources and the actors that launch these activities. The refactoring and re-implementation of SimGrid in C++ in the last 10 years, was the occasion to enforce a strong separation between the user and kernel spaces: each abstraction exposed to the user has its counterpart in the simulation kernel. This design has been instrumental for improving both extensibility and usability: the fundamental abstractions can be customized and/or replaced in various ways (see Section 4) and they can serve as the basis for higher-level abstractions (see Section 5).

## 3.1 Resources

The first step in most SimGrid simulations is to describe a hardware platform to simulate. This platform is composed of three types of basic hardware resources based on the following three abstractions: (i) **CPUs**, defined by a number of cores and a core compute speed; (ii) **Network links**, defined by a latency and a bandwidth; and (iii) **Disks**, defined by read and write bandwidths.

These basic resources are then aggregated and structured in higher-level abstractions to describe a hardware platform either programmatically in the simulator's code or in an external XML file. A physical *host* comprises a CPU, any number of disks, and a network endpoint. SimGrid provides also a *virtual machine* abstraction, which is a non-physical host that can be instantiated on and moved between physical hosts. A *route* is a vector of network links that defines a possibly multi-hop network path between two network endpoints (e.g., between two hosts). A route is either defined by the user or computed using some algorithm, and the set of routes defines the physical network topology. A *network zone* represents a set of hosts and links, has a network endpoint, and knows how to determine the routes, if they exist, between two hosts in the set and between a host and the zone's endpoint. For instance, a network zone can be defined to represent a compute cluster in which all nodes can communicate with each other and communicate with the outside world via a shared endpoint. Finally, a *platform* is a hierarchy of network zones. Determining the route between two hosts in differing network zones is done recursively [42].

## 3.2 Activities

### 3.2.1 Activity abstraction

Given a hardware platform description, the goal of any simulator is to simulate the execution of *activities* on the platform's hardware resources: computations on CPUs, communications on network links, and I/O operations on disks. SimGrid thus provides an activity abstraction, and its API allows the user to write the code to create and manage the execution of activities. All activities, regardless of on which resources they are meant to run, follow the same lifecycle depicted in Figure 2.

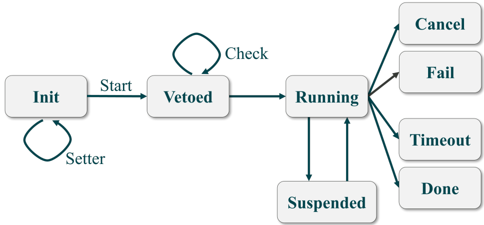



Figure 2: SimGrid activity lifecycle.

Once created (Init state), configuration parameters or properties of an activity can be set using setters, including setting dependencies on other activities. The activity is then started, at which point it may run immediately (Running state) or be held back (Vetoed state). The activity may be in the Vetoed state because it depends on the completion of other ongoing activities or because the resource on which the activity must execute is not yet available or identified. While running, an activity may be paused (Suspended state) and resumed any number of times. When an activity terminates it can end up in four different states depending on whether it terminates successfully (Done state), is canceled by the user code (Cancel state), terminates unsuccessfully due to a simulated resource failure or shutdown (Fail state), or reaches a user-specified timeout (Timeout state).

### 3.2.2 Activity simulation

At the core of SimGrid are simulation models that determine the completion date, in simulated time, of each activity. Each activity is defined by a total amount of work to do (e.g., bytes to read, compute operations to perform), an amount of work that remains to be done, and a set of resources that are used to perform this work. The objective is to compute the activity's completion date based on a notion of latency, in seconds, and of a maximum rate of progress, or speed, in units of work per time unit, both of which are computed based on the hardware specifications of these resources. The latency is computed as function (e.g., the sum) of the latencies of the resources. The speed is based on a determination of the bottleneck resource and on the specifications of that resource.

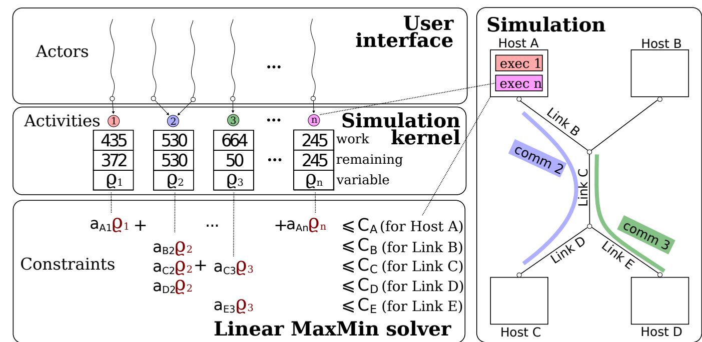
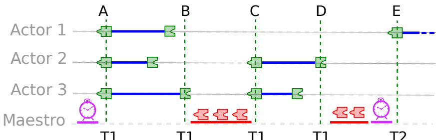


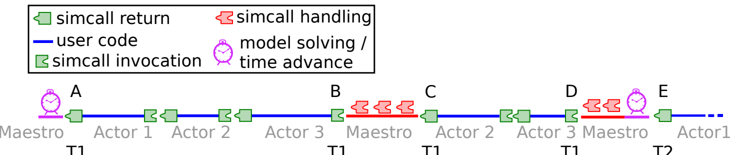

Figure 3: Overview of a SimGrid simulation.

At some point during the simulation multiple concurrent activities can contend for some resources. In this case, resource shares must be computed and allocated to each of these activities. SimGrid computes these resource shares using a linear max-min (LMM) solver, which is invoked at each simulation step. It solves a constrained optimization problem to compute the speed of each activity. The default objective function is to maximize the minimum speed across all activities. The constraints impose that linear combinations of one or more activity speeds are bounded by constants. These constants depend on the hardware characteristics of the resources and on user-imposed constraints on the rate of progress of activities. The coefficients in the linear combinations are used to model various effects (see Section 4.3). Once the speed of each activity has been computed, the time until the first activity termination is computed, the simulation clock is advanced accordingly, and the amounts of remaining work for all activities are updated based on their speed and time elapsed. Completed Activities are then removed from consideration for the next invocation of the solver.

Figure 3 depicts a 4-host platform with a 5-link "dogbone" network topology (right-hand side). 4 concurrent activities are simulated, each shown in a different color: two computation activities on Host A, a communication activity that uses links B, C, and D, and another communication that uses links C and E. The two computations contend for Host A's compute capacity, and the two communications contend for Link C's bandwidth. All these activities are defined in the simulation kernel by total and remaining amounts of work. Each activity is also associated to a variable $\varrho$, which is the activity's speed and whose value needs to be determined by the LMM solver (left-hand side). The constraints of the optimization problem are shown in the lower-left part of the figure. There is one constraint for each resource used by ongoing activities, where the right-hand side is the resource's capacity and the left-hand side is a linear combination of the activity speeds, each scaled by some factor (denoted as $a_{x,y}$ where $x$ is the resource and $y$ is the activity). For instance, the speed of the two communication activities ($\varrho_2$ and $\varrho_3$) both appear in the constraint for Link C since both activities use some of that link's bandwidth.



(a) Simulation execution timeline in simulated time.



(b) Simulation execution timeline in wallclock time.

Figure 4: Example simulation execution timeline for three actors between simulated times T1 and T2. **(a) Simulated time –** Initially, the maestro is involved in running the LMM solver to update all activities' remaining amounts of work. These activities correspond to simcall invocations by the three actors, which, in this example, all complete at the same simulated time (T1). At this point, a scheduling round begins and each actor executes its code until the next simcall is placed (shown as blue line segments). Once all actors are blocks on a simcall, the maestro regains control and handles all these simcalls (B). In this example, the simcalls placed by Actor 2 and Actor 3 take zero simulated time. A new scheduling rounds begins and the code of these two actors is resumed (C) until they place a new simcall and become blocked again (D). Since all actors are blocked on simcalls, the maestro regains control, processes these two new simcalls, which, in this example, take non-zero simulated time. The LMM solver is invoked to update all activities' remaining amounts of work, thus advancing the simulated time until the first simcall completion (T2). This occurs for Actor 1, whose code is resumed (E). This process continues until all actors have completed, or until no actor can make further progress (which denotes a deadlock bug in the simulator). **(b) Wallclock time –** The flow of control in the simulation's actual execution alternates between the maestro and the actors. Importantly, the code of two actors (in between the simcalls they place) never executes concurrently.

See [2] for all details regarding SimGrid's simulation models, and [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] for their experimental validations.

## 3.3 Actors

### 3.3.1 Actor abstraction

It is possible to write a SimGrid simulator that merely creates activities (typically with dependencies), assigns them to resources, and then launches the simulation, which ends when all activities have completed. Most users, however, wish to simulate complex systems with dynamic runtime behaviors using a more general Communicating Sequential Processes (CSP) model. To do so, a SimGrid simulator can create one or more *actors*. An actor is a (simulated) sequential process, defined by a main procedure written by the user. This procedure can create activities and manage their lifecycle.An actor can start an activity in three modes: (i) Blocking – the start call is blocking, and the corresponding actor can only proceed when the activity terminates; (ii) Asynchronous – the start call returns immediately with a handle for the activity, which provides an API to check for and to wait on the activity's termination; (iii) Detached – the start call returns immediately without a handle in a "fire and forget" fashion. A simulation typically comprises many actors, each of which is responsible for one or more activities.

Conceptually actors execute on hosts independently, just like processes would in a real-world platform. Actors can synchronize with each other using classical synchronization abstractions (i.e., mutex, condition variable, semaphore, barrier). These abstractions are implemented internally as zero-time activities. There is another synchronization abstraction called a *mailbox*. A mailbox is a logical rendez-vous point through which actors can exchange messages, similar to a URL on which one could post and retrieve data. Mailboxes are only used to match the *put* and *get* requests of actors. Each such matching results in a communication activity, either within a host when the communicating actors run on the same host, or over an end-to-end network path when the communicating actors run on different hosts.

### 3.3.2 Actor simulation

For the sake of deterministic execution, the actors cannot modify their environment directly: each modification of the activities or other interaction with the simulation kernel is serialized through a central component that processes them in a deterministic order. For the sake of modularity and correctness, SimGrid is designed as an operating system: the actors issue *simcalls* (akin to system calls) to a simulation kernel (akin to an OS kernel) called the *maestro*. The maestro decides which actors can proceed and which ones must wait. Simcalls can be either *immediate* if they take no time in the simulation (e.g., spawning another actor), or *blocking* if their completion must wait for future events (e.g., mutex locks require the mutex to be unlocked by its owner; communications wait for the network to have provided enough communication bandwidth for the data transfer to have completed).

A SimGrid simulation proceeds as a sequence of scheduling rounds. At each round, code in all actors that are not currently blocked on a simcall gets executed. This is done by the maestro, which passes the control flow to the code of each actor in sequence. The control flow is returned

to the maestro when the actor blocks on its next simcall. Once all actors have executed up until the point where they have placed a simcall, all simcalls are handled by the maestro. If some simcalls are immediate, another scheduling round starts immediately for those actors that are ready to execute. Once all actors have placed a blocking simcall, the maestro invokes the LMM solver, which computes the time at which the first pending simcall terminates. The simulation time is then advanced to that point, one or more actors are unblocked, and a new scheduling round begins. Figure 4a depicts scheduling rounds for an example with three actors. From the simulation's perspective the code written in between each simcall by the user for each actor takes zero time; only simcalls can take time. If the user wishes to simulate the time taken to execute this code, then a compute activity must be created and started (via a simcall).

By default, the maestro and the actors execute sequentially on a single core of the processor that runs the simulator (a multi-core execution mode is also available [43]). Figure 4b shows the same example as that shown in Figure 4a, but with the timeline drawn according to wallclock time instead of simulated time. The execution flow alternates between the maestro and the actors, and the code of two actors (in between the simcalls they place) never executes concurrently. As a result, although actors are implemented as threads of control within the same address space, they can share memory without any risk of race condition. This greatly simplifies the design and implementation of a SimGrid simulator: although the simulator conceptually implements a distributed system, this implementation can safely use globally visible data structures for actors to share information in zero simulated time. For instance, consider a simulator of a distributed storage system in which there is a central directory of file locations, but the overhead of accessing/updating this directory is deemed negligible by the user. In this case, instead of implementing the in-simulation management of this directory (i.e, implementing the directory as an actor with which other actors must communicate via messages), the directory can simply be implemented as, say, a dictionary data structure.

#### 3.3.3 Model checking ability

As stated in the previous section, the design by which all interactions of the actors with their environment are properly mediated via simcalls and the maestro ensures that the simulated execution is deterministic. As a result, simulated execution events are *observable* and the causality between these events is well-defined. This makes it possible to replace the classical performance-oriented simulation engine with another engine that only models the causal ordering of observed events for the purpose of software model checking (SMC) [44]. The goal is to efficiently explore all possible outcomes of the simulated application's execution so as to assess the correctness of the application's implementation (i.e., discover possible deadlocks, livelocks, message mismatches, and other logical bugs). SimGrid v4 now comes with a model checker that performs SMC and uses formal techniques to reduce the explored search space by exploiting symmetries and prune causally equivalent histories. This model checker can be used to verify MPI applications [45] and pthread applications [46]. More generally, the correctness of any application whose execution is simulated using SimGrid can be evaluated in this manner (which, incidentally, also helped ironing out bugs in SimGrid itself).

## 4 Extension mechanisms

A key concern for a simulation framework is its extensibility. This is because users often want to model (part of their) target systems in specific ways that do not correspond to the common case. In SimGrid v3 users had to understand and modify internals to extend simulation abstractions and/or models. We have since added various mechanisms so that the simulation framework can be extended conveniently.

### 4.1 Rich and unified resources

In SimGrid v3, to answer the needs of many users, it was possible to model Dynamic Voltage Frequency Scaling (DVFS) for CPU resources, by which each CPU resource naturally implements a notion of *pstate*, where a given state is defined by a numeric id and a compute speed, and can be associated to a wattage. It was also possible to model background load on CPU resources by attaching a *load profile* to a host that specifies dynamic changes in the nominal speed according to a trace file or to a random generator. Finally, it was also possible to attach to a host a *state profile* to specify whether the host is up or down, which makes it possible to simulate arbitrary host failures and churn patterns.

The above features were clearly useful but limited to CPUs and hosts. Since SimGrid v3.11 they have been extended to the other resources (network links and disks), thus unifying resource abstractions. As a result, users can now modify the behavior of any resource by using the pstate, load profile, and state profile features in creative ways to simulate a range of complex phenomena, the simulation of which would have been labor-intensive (i.e., requiring modifying internals) with SimGrid v3. For instance, the startup period of any resource can now be modeled easily with an extra pstate in which the resource performs no work for activities on that resource. An actor can be in charge of changing the resource's pstate after a given lapse of time that corresponds to the duration of the startup period. Another example is the simulation of Wi-Fi networks (introduced in SimGrid v3.26), where the bandwidth experienced by a station is a function of the signal-to-noise ratio between the station and the access point. It turns out that this phenomenon can be simulated by assigning particular pstate values to network links. Section 5.2 describes

how this technique, combined with other extension mechanisms, makes it possible to model the behavior of Wi-Fi networks.

## 4.2 Plugins

In SimGrid v4 we have developed a generic *plugin* feature, by which arbitrary *callbacks* can be attached to *signals* that are fired by specific simulation events (e.g., activity starts and completions, actor creations and terminations). Plugins can be configured through arbitrary properties attached to any resource when the simulated platform is instantiated. They can also extend any simulation abstraction with any arbitrary objects. These objects can be used to store persistent state for the callbacks to use throughout the simulation.

Plugins have allowed us to not only simplify SimGrid's internal implementation but also to make SimGrid more extensible. For instance, SimGrid v3 provided a model of the energy consumption of CPU resources. The implementation of this model was spread throughout SimGrid's core, so as to pass relevant parameters to the energy model and invoke it to update the energy consumption values at the correct simulated times. This implementation was difficult to maintain. Furthermore, a user wanting to modify (let alone extend or replace) the energy model had to become essentially a full-fledged SimGrid developer. The energy model was re-implemented as a self-contained plugin with its own configuration (which specifies the consumption of each pstate in Watts) and callbacks for the relevant events related to CPU resources and their use by activities. The implementation of the plugin is confined to a single source file which contains only 350 lines of C++ code, that is easy for users to copy and modify.

The SimGrid distribution now comes with several built-in plugins (e.g., network link load and energy modeling (v3.18), WiFi network load and energy modeling (v3.26), host energy modeling (v3.11), host load modeling (v3.22), file system (v3.18), batteries and solar panels (v3.34), computer room air handling (v3.35)). Users can also easily develop their own plugins from scratch. For instance, often users wish to collect and output information regarding specific events that happen throughout the simulated execution, typically for post-mortem analysis. Say that the user wishes to output to the terminal a time-stamped trace of actor termination events. They can develop such a tracing plugin in only a few lines of C++, as shown in Figure 5.

```
1  SIMGRID_REGISTER_PLUGIN(tracer, "Tracer", &tracer_init)
2
3  // Callback to place whenever an actor terminates
4  static void trace_exit(simgrid::s4u::Actor& actor) {
5    std::cout << simgrid::s4u::Engine::get_clock() << ",";
6    std::cout << actor->get_name() << std::endl;
7  }
8  // Plugin initialization function
9  void tracer_init() {
10   // Attaching the callback to the relevant event
11   simgrid::s4u::Actor::on_exit(&trace_exit);
12 }
```

Figure 5: A simple "tracing" plugin.

## 4.3 Advanced modeling mechanisms

As explained in Section 3.2, SimGrid's simulation core is implemented as a constrained optimization problem solver, the LMM solver. Using the SimGrid v4 API, there are three ways for users to modify the LMM so as to extend SimGrid's simulation models, as described hereafter.

First, it is possible to simulate changes of the speed at which resources perform activities, given the current set of simulated activities, by defining ad hoc correction factors applied to the resources' latencies and speeds. Such factors were already present in SimGrid v3, for the specific purpose of simulating various network protocol effects (e.g., for modeling TCP overhead [4] or adaptive protocols used by MPI implementations [3]). But these factors were global, static, only for network links and communication activities, and were all hard-coded in the LMM. In SimGrid v4 we have extended these correction factors to all resource kinds. A callback function can be attached to each resource, at creation time or later. This function takes the size of an activity as parameter and applies user-specified correction factors. This allows users to extend SimGrid's simulation core to, for instance, simulate CPU affinities to study classical unrelated-machines scheduling problems, account for protocol change depending on resources (e.g., Infiniband RDMA between distinct nodes, memory copy for intra-node communications, cudaMemcpy when GPUs are involved) or simulate the heteroscedastic performance of mechanical disks.

Second, it is now possible to modify the LMM to simulate the fact that the performance delivered by a resource can degrade with contention. In other words, although by default the LMM simulates a linear sharing of a resource among contending activities, the user can specify arbitrary, including non-linear, sharing policies. Specifically, a user can attach an arbitrary function to a resource, which computes the current resource's speed as a function of the number of concurrent activities using that resource. This function could implement any analytical model, or could reproduce specific discrete behaviors observed on particular real-world resources. This function is invoked by the LMM each time new resource shares must be computed for that resource.

Combining these two methods, introduced in SimGrid v3.29, correction factors and non-linear sharing behaviors, makes it possible to simulate resource behaviors that go beyond what the default LMM can do. For instance, it has allowed us to faithfully simulate the execution of I/O benchmarks on several types of mechanical and solid-state disk drives [7].

Third, users can specify arbitrary concurrency limits for any resource. When the number of activities that try to use this resource exceeds that limit, the extraneous activities are delayed and have to wait for currently running activities to

complete. Specifying different concurrency limits makes it possible to extend the use of SimGrid to many scenarios, such as, the simulation of network throttling on a multiprocessor system-on-chip, or to study scheduling algorithms under classical theoretical constraints (e.g., 1-port network model, strictly serial executions on a CPU).

The above mechanisms allow users to vastly modify and extend the default behavior of the LMM so as to simulate a wide range of real-world phenomena. But SimGrid also offers the capability to bypass the LMM altogether. For instance, SimGrid can be compiled to use the well-known ns-3 packet-level network simulator [18] as a network model. Packet-level simulation accounts for the movement of every network packet involved in every communication. This higher level of detail, compared to the LMM-based models that only recompute the respective instantaneous speeds of the currently ongoing communications when a communication starts or stops, expectedly comes with a much higher simulation time.

Finally, users can extend SimGrid with arbitrary models by overriding two methods: `next_occurring_event()`, which returns the date of the next event that will occur according to the model, and `update_state(delta)`, which updates the model's state by shifting the date forward by `delta` seconds. SimGrid v4 uses this plugin mechanism to simulate the behavior of resources that are not handled by SimGrid's extant models, e.g., batteries or solar panels that provide power to hosts. The battery plugin can thus interrupt the simulation when a battery becomes depleted, which requires to turn off the hosts that use that battery as their primary power source. An FMI (Functional Mock-up Interface) plugin for SimGrid has also been developed that makes it possible to run a SimGrid simulation while co-simulating any FMI model [47], such as ones built with OpenModelica. This plugin leverages the above extension mechanism to enable such co-simulation, interrupting the SimGrid simulation when an event occurs in the FMI model, and allowing SimGrid actors to modify the parameters of the FMI model during the simulation.

# 5 Better usability via composite abstractions and programming models

Using the abstractions in Section 3 and the extension mechanisms in Section 4, we have implemented several high-level, composite abstractions in SimGrid v4. We have also made it possible to combine multiple programming models within the same SimGrid simulator. The goal is to increase usability by making simulator implementation easier for a broad range of use cases.

## 5.1 Composite platforms

Although the abstractions described in Section 3.1 make it possible to specify any conceivable hardware platform, doing so for large platforms is time-consuming or labor-intensive, and almost always error-prone. One of the most common parallel computing platforms is a homogeneous cluster in which hosts are interconnected via some network topology. Describing a cluster using the XML format would entail specifying: (1) each host individually with its own name and speed; (2) a private link connecting a host to a single backbone switch define by its latency and bandwidth; (3) the backbone switch itself with its own latency and bandwidth values; and (4) all the routes between each pair of hosts. Figure 6 shows such an XML description of a 64-node homogeneous cluster.

```
<platform version="4.1">
 <zone id="cluster" routing="Full">
  <!-- Declare the 64 hosts -->
  <host id="node-0.cluster" speed="1Gf"/>
  <host id="node-1.cluster" speed="1Gf"/>
  [...]
  <host id="node-63.cluster" speed="1Gf"/>

  <!-- Declare the 64 private links -->
  <link id="link_0"  bandwidth="10Gbps" latency="10us"/>
  <link id="link_1"  bandwidth="10Gbps" latency="10us"/>
  [...]
  <link id="link_63"   bandwidth="10Gbps" latency="10us"/>

  <!-- Declare the backbone switch -->
  <link id="backbone" bandwidth="100Gbps" latency="100us">

  <!-- Declare the (64 x 63) / 2 routes -->
  <route src="node-0.cluster" dst="node-1.cluster">
    <link_ctn id="link_0"/>
    <link_ctn id="backbone"/>
    <link_ctn id="link_1"/>
  </route>
  [...]
  <route src="node-62.cluster" dst="node-63.cluster">
    <link_ctn id="link_62"/>
    <link_ctn id="backbone"/>
    <link_ctn id="link_63"/>
  </route>
 </zone>
</platform>
```

Figure 6: A partial verbose XML description of a homogeneous cluster.

Unsurprisingly, many SimGrid users target such cluster platforms (see Figure 1). To increase usability for these users SimGrid's XML format was extended (even before SimGrid v3) with built-in netzone definitions that can be re-used, customized, and combined at will.

Figure 7 describes the same platform as in Figure 6 but using the built-in `<cluster>` tag that drastically simplifies the code users have to write. The underlying network topology and routing of the clusters are computed automatically [42], in a way that is more efficient than when

```
<platform version="4.1">
  <!-- Declare a 64-node cluster -->
  <cluster id="cluster" prefix="node-" radical="0-63"
         suffix=".cluster" speed="1Gf" bw="10Gbps"
       lat="10us" bb_bw="100Gbps" bb_lat="100us"/>
</platform>
```

Figure 7: A compact XML description of a homogeneous cluster.

done as in Figure 6. Additional built-in descriptions of complex network topologies commonly found in HPC clusters, such as multi-dimensional torus, Fat-Tree, or Dragonfly topologies are made available to the users.

Describing platforms in XML is convenient as users can produce human- and machine-readable descriptions relatively easily. Unfortunately, it has proven difficult to evolve SimGrid's XML format to include new capabilities while maintaining backward compatibility. To illustrate these limitations let us consider the Summit leadership-class supercomputer from Oak Ridge National Laboratory. SimGrid's XML format makes it possible to declare a cluster with the desired number of nodes (i.e., approximately 4,600 compute nodes) and the desired topology (i.e., 3-level Fat-Tree network topology with 18 director switches) using the appropriate properties in the <cluster> tag. However, this tag only supports compute nodes defined as simple multicore processors. As a result, while the <cluster> tag is convenient for some users, it is too limiting for others who then have to revert to an overly complex XML platform description. For this reason we have introduced, in SimGrid v3.28, the capability to describe platforms programmatically directly in the code of the simulator. This provides users with a much greater flexibility than with XML, allowing them to combine the fundamental abstractions (CPU, network link, disk) in arbitrary fashion. For instance, in the Summit use case, it is straightforward to implement a programmatic description of each compute node as comprising 2 CPUs, 6 GPUs, 2 NICs, and 1 NVMe, all interconnected via a custom internal network topology. Note that although SimGrid does not provide a built-in GPU model, custom GPU models can be easily built by combining SimGrid's fundamental abstractions [48].

Figure 8 shows a programmatic C++ description of a more complex platform with two independent hosts, representing a coordinator and a database service, each connected to three homogeneous clusters of different sizes. While this way of describing platforms may seem more complicated at first, it is much easier to evolve than its XML counterpart. For instance, adding three more clusters to that platform to simulate a larger scale scenario would simply amount to adding three more sizes in the vector (line 42), but would require adding three more <cluster> tags and declare all the additional routes in the XML description, which is error-prone. Moreover, increasing the complexity of the internal structure of the cluster can be done programmatically, while it is highly constrained by the XML format, as explained earlier.

```cpp
NetZone* create_cluster(NetZone* root, std::string suffix,
                        int num_hosts) {
  auto* cluster =
     create_star_zone("cluster"+suffix)->set_parent(root);

  // create gateway
  cluster->set_gateway(cluster->create_router("cluster" +
     suffix + "-router"));

  // create the backbone link
  auto* backbone = cluster->create_link("backbone" + suffix,
     "100Gbps")->set_latency("100us");

  // create all hosts and connect them to outside world
  for (int i = 0; i < num_hosts; i++) {
    std::string name = "node-"+std::to_string(i)+suffix;
    // create host
    auto* host = cluster->create_host(name, "1Gf");
    // create link
    auto* link = cluster->create_link(name+"_link","10Gbps")
                   ->set_latency("10us");
    // add route between host and any other host
    cluster->add_route(host, nullptr, {link, backbone});
  }
  cluster->seal();
  return cluster;
}

int main(int argc, char** argv) {
  // Create the platform
  auto* root = create_full_zone("world");

  // Create a coordinator zone/host
  auto coordinator_zone =
     create_full_zone("Coordinator")->set_parent(root);
  coordinator_zone->create_host("coordinator.org", "1Gf");
  coordinator_zone->seal();

  // Create a database zone/host
  auto database_zone =
     create_full_zone("Database")->set_parent(root);
  database_zone->create_host("database.org", "1Gf")
           ->create_disk("db", "100MBps", "50MBps");
  database_zone->seal();

  // Create a single link as a simple abstraction of the
     whole wide-area network
  auto* internet = root->create_link("internet","200MBps")
                  ->set_latency("1ms");

  // Create three clusters and connect them to the
     coordinator and database
  std::vector<int> cluster_sizes = {16, 32, 40};
  int i = 0;
  for (auto size : cluster_sizes) {
    auto* cluster =
     create_cluster(root,".cluster"+std::to_string(i++)+
                 ".org", size);
    root->add_route(coordinator_zone, cluster, {internet});
    root->add_route(database_zone, cluster, {internet});
  }
  root->seal();
}
```

Figure 8: Programmatic C++ description a platform composed of two individual hosts and three homogeneous clusters of different sizes.

## 5.2 Composite network routes

By default, network topologies specified in simulated platforms assume classical TCP-based end-to-end network routes. But expressing composite routes where different behaviors are simulated for different portions of the routes can be compelling. This is the case of the simulation of Wi-

Fi networks, for instance, needed to study IoT platforms. This requires a new type of network zone that comprises the *access points* of the Wi-Fi network, the hosts (*stations* in the Wi-Fi terminology) connected to it, and a single network link declared with a specific bandwidth sharing policy adapted to Wi-Fi networks. This pseudo-link is then included into the network routes between any two stations within the Wi-Fi zone.

One fundamental difference between Wi-Fi and wired networks is that the performance of the former is not determined by the bandwidths and latencies of network links but by two characteristics of the access point, defined as properties of the zone. First, the Modulation and Coding Scheme (MCS) defines the speed at which the access point exchanges data with all the stations. This speed directly depends on the access point's model and configuration, and also on the distance between the involved station and access point. More precisely, the data rate depends on the signal-noise ratio (SNR) between the communicating entities. Second, the number of antennas defines the amount of Spatial Streams that the access point can simultaneously serve. In practice, a given access point will provide several levels of performance (called Data Rates) depending on its hardware characteristics. In SimGrid, the Wi-Fi pseudo-link is given a set of pstates representing each of these data rates (see Section 4.1). Each host connected to this access point may have a different pstate on that link, representing the MCS that would be used between the station and the access point corresponding to the SNR between these elements. Additionally, the LMM is extended to model non-linear resource sharing (see Section 4.3) so that the maximum throughput of a Wi-Fi network is not a constant but rather a function of the number of concurrent flows [49]. Implementing a fast but accurate model of Wi-Fi performance would have been much more difficult without these extension mechanisms provided since v3.26 (pstates for any resource, user-provided resource sharing function).

## 5.3 Composite activities

When writing a simulator for complex application execution scenarios, actors often need to start multiple activities of different types asynchronously and manage their concurrent execution. For instance, when writing a simulator of an iterative numerical simulation based on domain decomposition, each actor may have to exchange data with its neighbors, write intermediate results to the file system, and perform computations, all concurrently. To ease the management of concurrent asynchronous activities, SimGrid v3.35 introduced an *ActivitySet* data container, in which asynchronous activities of any types can be stored. This container allows users to simply test or wait for the completion of any or all the activities.

As seen in Figure 2, activities have a well-defined lifecycle that corresponds to the unique execution of a certain amount of work. However, in some execution models, such as Synchronous Data Flow (SDF) or Bulk Synchronous Parallel (BSP), the same activity, or activity sequence, must be repeated several times.

To support such execution models, SimGrid v3.34 introduced the concept of *tasks*. A task has an underlying activity that can be repeated multiple times. Tasks can be organized in graphs to simulate complex iterative execution patterns with inter-task control dependencies. SimGrid v4 also introduced the notion of *tokens* that circulate through a task graph and can carry any user-defined data to implement inter-task data dependencies. When all dependencies of a task become satisfied it fires a new instance of its underlying communication, computation, or I/O activity. It then blocks until all its dependencies become satisfied again. The parameters and successors of a task can be redefined at runtime using the same callback/signal mechanism as that used for plugins. This allows users to dynamically change the topology of the task graph, implement conditional branches, or introduce variability in the duration of the simulated activities.

The task and token abstractions can be used to simulate relatively simple, yet commonplace, designs which involve pipelined activities. As an example, Figure 9 illustrates the usage of tasks and tokens to implement an iterative "computation and data exchange" pattern between two hosts, which is automatically repeated $N$ times. That is, at each iteration, each host performs a computation task and then a communication task to simulate the exchange of data over the network. This example also includes the simulation of a periodic checkpoint that must occur every $n \ll N$ iterations. This is easily simulated via a token sent every $n$ iterations, which triggers the execution of an I/O task.

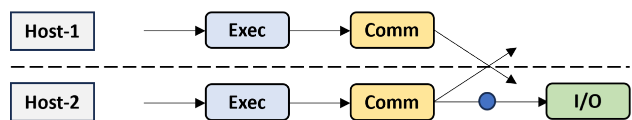



Figure 9: A simple BSP execution implemented with tasks and tokens. Two hosts perform a computation and then exchange their data. This pattern is automatically repeated $N$ times. A conditional checkpoint is triggered by the emission of a token sent every $n \ll N$ iterations.

Although the above task and token abstractions are powerful, they can become relatively costly in terms of simulation time due to each operation being simulated as an individual task. This would be the case, for instance, for simulating the buffered transfer of data stored on disk to a remote host, which involves fine-grain pipelining of I/O operations and communications. For this reason, SimGrid v4 provides simulation abstractions that implement these types of concurrent operations using a continuous fluid approximation. This approximation does not simulate the discrete execution of these operations, but instead computes the bottleneck operation and makes all operations proceed at the speed of that bottleneck. The simula-

tion is thus implemented at a coarser grain, trading off simulation details for faster simulation execution. Specifically, two such abstractions are provided: parallel task, or *ptask*, and I/O stream (introduced in v3.31). A *ptask* encapsulates computations and communications to be executed over a set of interconnected hosts (e.g., an iterative matrix multiply parallel computation). It is defined by volumes of computation to be executed at each host and volumes of data to transfer between each host pair. An I/O stream encapsulates I/O operations and communications for sending data between hosts and/or disks (e.g., a video stream application with repeating I/O reads on a host, each followed by network transfer to a remote host).

### 5.4 Composite programming models

The simulation abstractions presented in Section 3 allow users to implement a simulator using the CSP programming model, which is general and can be used to implement a simulator of virtually any distributed system. However, SimGrid also supports other, less general, programming models that are specialized for common use cases, so that simulators for these use cases can be implemented with minimal effort.

As mentioned at the beginning of Section 3.3, it is possible to implement a SimGrid simulator without creating any actor: the simulator simply creates activities and sets up dependencies between these activities. Since SimGrid v3.30, it is possible to describe a static Directed Acyclic Graph (DAG) of activities. Each activity in this DAG can be a computation, a network communication, an I/O operation, or any of the composite activities described in Section 5.3. As a result, it is possible to describe a static data-flow application, and the simulator merely specifies on which resource(s) each activity is to be executed. It can then launch the simulation until completion of all activities or until a particular lapse of (simulated) time has elapsed. This programming model is more restricted than the CSP model, but proves useful and convenient for many users. Figure 1 shows that many SimGrid users develop simulators for evaluating scheduling algorithms. Many of these simulators are used to study applications structured as static DAGs, for which different schedules are computed and must be evaluated in simulation. The above programming model makes it straightforward to implement such simulators.

A commonplace programming model for cluster platforms (the most frequently targeted class of platform according to Figure 1) is distributed-memory programming using message-passing. In practice, this programming model is implemented using the Message Passing Interface (MPI) standard [50], which is not a fully general CSP programming model but provides many convenient higher-level abstractions such as collective communications. For this reason, SimGrid provides an implementation of MPI, called SMPI (Simulated MPI). It makes it possible for users to write standard MPI programs, or use unmodified existing MPI programs, and execute them seamlessly in simulation. SMPI uses several techniques to ensure that the simulated executions can be executed on a single machine scalably [15]. It also implements accurate simulation models of both point-to-point and MPI collective communication operations, accounting for the specific schemes and algorithms implemented in particular implementations of the MPI standard (which the user can select at will). These models have been thoroughly validated, and we refer the reader to [3, 9, 51] for experimental validation results and all technical details.

A key usability enhancement in SimGrid v4 is that all the above programming models can now be combined at will within a single simulator: different components of a simulator can use different programming models.

To illustrate this capability, we present a full-fledged example. Consider a coordinator-worker application where each worker executes on a node of a compute cluster. When idle, each worker requests work from a coordinator that is running on some remote host. This coordinator-worker scheme can be implemented naturally using SimGrid's CSP programming model. Say that the work that each worker must perform consists in invoking an MPI program that executes on all compute nodes of its cluster. At each iteration of this MPI program the MPI ranks perform a computation and then synchronize via an all-to-all communication. This can be implemented easily, since v3.34, using SimGrid's MPI programming model. Say now that, as part of this computation, at each iteration the MPI process with rank 0 must upload 1MB of data to a remote database. This consists in sending 1MB to a remote host and then writing 1MB to a disk at that host. This can be easily implemented as a 2-activity sequence using SimGrid's data-flow programming model. While the general CSP programming model would be conceptually sufficient to implement the entire simulator, the ability to combine multiple programming models vastly reduces the overall development effort.

We have produced a working implementation of the above example using SimGrid's C++ API in less than 150 lines of code (code available on GitHub [52]). We show here relevant code fragments, all redacted for brevity. Most of the code of the `main()` function (Figure 10) defines the simulated hardware (as already shown in Figure 8). A few lines of code are used to state that one Coordinator actor is to be started on some host in the platform, and one Worker actor is to be started on a compute node of each of 3 clusters (lines 9-13).

Figure 11 shows the Coordinator code which first creates its mailbox (line 5), and then constructs a queue of workunits for workers to perform (lines 7-9). Each workunit is described by a number of iterations, an amount of data to communicate, and an amount of work to compute. The workunit queue ends with 3 "poison pills" through which workers will discover that there is no more work to be done. The coordinator then goes through a loop

```
int main(int argc, char **argv) {
  // Create a simulation engine
  auto engine = new Engine(&argc, argv);
  // Create a simulated platform
  std::vector<int> cluster_sizes = {16, 32, 40};
  [...]
  // Create 1 coordinator
  Actor::create("Coordinator",
     Host::by_name("coordinator.org"), Coordinator());
  // Create 3 worker actors
  Actor::create("Worker1",
     Host::by_name("host-0.cluster1.org"), Worker());
  Actor::create("Worker2",
     Host::by_name("host-0.cluster2.org"), Worker());
  Actor::create("Worker3",
     Host::by_name("host-0.cluster3.org"), Worker());
  // Launch the simulation
  engine->run();
}
```

Figure 10: `main()` code for the example in Section 5.4.

(lines 15-22) in which it waits for workers to send it a mailbox to which it replies with the next workunit to be performed (via a simulated 128-byte message), until all workunits have been performed.

```
class Coordinator {
  public:
  void operator()() {
  // Create my mailbox
  auto my_mailbox = Mailbox::by_name("coordinator_mb");
  // Create 10 workunits:  iter=100, size=10MB, work=2Gf
  std::deque<WorkUnit*> todo;
  for (int i = 0; i < 10; i++)
    todo.push_front(new WorkUnit(100, 10*MB, 2*Gf));
  // Add "poison pills" for worker terminations
  for (int i = 0; i < num_workers; i++)
    todo.push_front(new WorkUnit(0, 0, 0));

  // Main loop
  while (not todo.empty()) {
    // Wait for a worker to send me their mailbox
    auto worker_mailbox = my_mailbox->get<Mailbox>();
    // Reply with the next workunit (128-byte message)
    worker_mailbox->put(todo.back(), 128);
    // Remove workunit from queue
    todo.pop_back();
  }
}
```

Figure 11: Coordinator C++ functor for the example in Section 5.4.

Figure 12 shows the Worker code. First, the worker identifies the database's host and disk (lines 4-6) and the compute nodes in its cluster (line 8), after which it creates its own mailbox (line 10) and loops (line 13). At each iteration of the loop it requests work from the coordinator via a 32-byte message (line 15), receives a workunit (line 17), and aborts if the workunit is a poison pill (line 19). If the workunit is not a poison pill, then it asynchronously starts a simulated MPI program at line 22. The first argument to the `SMPI_app_instance_start()` function is a name that will be used to later wait for the completion of the MPI program. The second argument is a lambda expression whose code is standard MPI code (lines 23-53). The third argument is the list of compute nodes on which to execute the MPI program (line 54).

```
class Worker {
  public:
  void operator()() {
  // Get database host and disk
  auto dbhost = Host::by_name("database.org");
  auto dbdisk = dbhost->get_disks().front();
  // Get list of compute nodes in my cluster
  auto nodes = this_actor::get_host()
     ->get_englobing_zone()->get_all_hosts();
  // Create my mailbox
  auto mailbox = Mailbox::by_name(this_actor::get_name());

  // Main loop
  while (true) {
    // Ask the coordinator for work (32-byte message)
    Mailbox::by_name("coordinator_mb")->put(mailbox, 32);
    // Get a workunit back
    auto wu = my_mailbox->get<WorkUnit>();
    // If it's a poison pill, terminate
    if (wu->iterations == 0) break;

    // Start and MPI program to perform the work
    SMPI_app_instance_start(this_actor::get_cname(),
                      [wu, dbhost, dbdisk]() {
      MPI_Init();
      int numprocs, myrank;
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
      // Allocate a data buffer
      void *data = SMPI_SHARED_MALLOC(wu->size * numprocs);
      for (int it= 0; it< wu->iterations; it++) {
        // Perform BSP compute work
        this_actor::execute(wu->work);
        // Perform BSP all-to-all communication
        MPI_Alltoall(data, wu->size, MPI_CHAR, data,
              wu->size, MPI_CHAR, MPI_COMM_WORLD);
        // Rank 0 performs a database update
        if (myrank == 0) {
          // Create 1MB read activity on the database disk
          auto io_activity =
           dbdisk->io_init(1*MB, Io::OpType::READ);
          // Create 1MB communication activity
          // from my host to the database host
          auto comm_activity =
           Comm::sendto_init(this_actor::get_host(),
                    dbhost)
          ->set_payload_size(1*MB));
          // Create activity dependency
          comm_activity->add_successor(io_activity);
          // Start the I/O activity, but it won't start
          // until the communication activity completes
          io_activity->start();
          // Start the communication activity
          // and wait for its completion
          comm_activity->start()->wait();
        }
      }
      // Free the data buffer
      SMPI_SHARED_FREE(data);
      MPI_Finalize();
    }, nodes);

    // Wait for the MPI program to terminate
    SMPI_app_instance_join(this_actor::get_cname());
  }
}
```

Figure 12: Worker C++ functor for the example in Section 5.4.

At line 29, the program allocates sufficient memory for the buffer that will be used for all-to-all communication, based on the workunit's specification. While this could be done using a standard `malloc()`, the simulation's memory footprint would become too large when simulating the program's execution with many MPI processes. Instead, the data is allocated using the `SMPI_SHARED_MALLOC()` macro. This macro, as explained in details in [51], allocates memory that is shared by all simulated MPI processes. While using this macro would make a real MPI program incorrect, for a program that does not perform any actual computation but only simulates the execution of computation volumes, it makes it possible to simulate the execution of large-scale programs on a single computer. The MPI program then loops in BSP fashion (lines 30-51). At each iteration of the loop, each process performs some computation volume as specified by the workunit (line 32), followed by an MPI all-to-all communication for a data volume also specified by the workunit (line 34). The process with rank 0 performs extra work at each iteration (lines 36-50). It creates an I/O activity to be performed on the database disk (line 38) and a communication activity to be performed between the host this process runs on and the database host (line 41). Both these activities have a 1MB payload, and the I/O activity depends on the communication activity (line 43). The process starts both activities and waits for the completion of the communication activity (lines 46 and 49). Finally, once all BSP iterations have been performed, each simulated MPI process frees the data buffer and calls `MPI_Finalize()` (lines 53-54).

# 6 Impact on simulation practice

In this section we demonstrate the impact of the enhanced usability and extensibility of SimGrid v4 on simulation practice. The simulation abstractions and programming models that we have described in the previous sections allow users to implement simulators for many different use cases with reasonably low effort. This has led users to develop simulators for several domains, as seen in Figure 1. Hereafter we describe notable projects for which SimGrid provides a foundation and that form a large SimGrid ecosystem. Some of these projects have produced simulators or simulation frameworks that aim at avoiding duplication of effort in research communities. Others have produced simulators or simulation-based tools for various production uses. All these projects are a testimony to SimGrid's versatility and have all benefited from its capabilities in terms of accuracy and scalability. But, in all that follows, we specifically explain how they have benefited from the extensibility and usability enhancements described in Sections 4 and 5.

## 6.1 Distributed cyberinfrastructure simulation

Many researchers wish to simulate the execution of various application workloads on distributed platforms, and often end up re-implementing the same simulation abstractions and mechanisms. WRENCH [53] is a simulation framework that provides implementations of highly configurable *services* that users can re-use as building blocks in their simulators. It also removes the burden of implementing inter-process communication, which is labor-intensive and error-prone when developing a simulator of a complex distributed system. The user only writes the code of one kind of actors called execution controllers. These controllers interact with the services deployed on the simulated platform using simple APIs, so as to execute application workloads defined by data and compute volumes with arbitrary dependencies. As a result, it is possible to implement simulators of complex deployments and runtime systems with low software engineering effort. WRENCH achieves this objective because SimGrid's API is expressive enough to describe a wide range of interoperable services and sufficiently usable to render the implementation of these services tractable. In particular, WRENCH relies on plugins (Section 4.2), on the ability to mix different programming models (Section 5.4), and on composite activities (Section 5.3).

## 6.2 Resources and jobs management systems simulation

An active research area is Resources and Jobs Management Systems (RJMS), i.e., the systems in charge of the scheduling of user jobs on shared parallel computing platforms. In particular, RJMS must employ so-called batch scheduling algorithms, which have been the subject of active research for decades, and are typically evaluated in simulation. The Batsim project [54] is a SimGrid-based RJMS simulator. While most research in this area simulates batch jobs at a completely abstract level (e.g., rectangles in a Gantt chart), Batsim users can describe workloads in which each job is defined by a profile that encodes specific computation, communication, and I/O patterns. Job executions are then simulated so that they generate load, contention, and electrical power consumption on hardware resources. The ElastiSim project [55] shares Batsim's objectives but specifically targets the simulation of malleable batch jobs that can both adapt their resource demands and report on their progress at runtime. These two features dramatically increase the design space for batch scheduling algorithms as these algorithms can make decisions regarding a job during that job's execution. Batsim uses the pstate feature for rich resource descriptions (Section 4.1), and both Batsim and ElastiSim rely on SimGrid's composite *ptask* activities (Section 5.3).

### 6.3 HPC runtimes and applications

Many researchers have used simulation for research and development in the field of HPC, a field in which SimGrid has seen a lot of usage (as seen in the fraction of works that target Cluster platforms in Figure 1). The most common use of simulation if to evaluate the performance of application workloads when executed at different scales on candidate platforms. Fast simulations can provide a first performance approximation before proceeding with resource-intensive testing on or even purchasing of real compute infrastructures.

Several large projects have used SimGrid in this context due to its SMPI component being able to simulate the execution of (almost) unmodified MPI applications. Evaluating the performance of classical application workloads on particular platforms and/or with particular runtime systems is so important to the HPC community that several "proxy apps" have been developed and maintained over the years. These correspond to representative MPI applications for various application domains (ECP proxy applications [56], MeteoFrance proxy applications [57]) and to standard MPI benchmarks from various benchmark suites (HPL, CodeVault, Trinity-Nersc, CORAL). Many such applications have been executed directly with SMPI, and are now part of SimGrid's integration testing infrastructure [58], along with the test suite for the OpenMPI implementation of the MPI standard [59] and the Intel MPI benchmarks [60]. The S4BXI project [61] has developed a full-fledged simulator of the Portals 4 network API [62] using SimGrid, in the context of MPI applications. The goal is not only to enable in-simulation performance evaluation capabilities, but also to perform what-if analyses to decide promising areas for hardware platform design optimization. S4BXI uses a multimodel simulation approach, by which SimGrid's SMPI feature is used for fast simulation of some portions of the execution and much slower, but highly accurate, simulation techniques are used to simulate other portions. The BigDFT project [63, 64], which provides open-source software for simulating macromolecular systems at the nanoscale, uses SimGrid to include simulated executions in its continuous integration process and regression testing process. Finally, SimGrid has also been used by Intel and Bull/Atos to explore different solutions and configurations for hardware components of HPC platforms, such as network interconnects or node size [3].

Key to the success of the above projects is SimGrid's scalability and accuracy levels. But key to their feasibility in the first place is the use of some of the extensibility and usability features added to SimGrid v4. All these projects use the composite platform features (Section 5.1) for better usability of SimGrid when describing complex HPC platforms. The advanced modeling mechanisms (Section 4.3) are also used heavily to make the accurate simulation models of MPI point-to-point and collective communications. Some of these projects also rely on the extensibility plugin feature (Section 4.2), e.g., for energy consumption simulation.

A notable HPC project that uses SimGrid but does not target MPI is StarPU [65]. It provides a unified runtime system for programming heterogeneous multicore architectures (i.e., multicore processors with accelerators and/or coprocessors, such as GPUs). StarPU integrates with SimGrid so that StarPU applications can seamlessly execute in simulation mode, which is a powerful testimony to SimGrid's usability. Simulated executions are also used for StarPU's continuous integration and performance evaluation purposes, and have been instrumental in uncovering performance bugs in StarPU. StarPU builds on SimGrid's models and abstractions to develop its own models and abstractions for the simulation of GPUs [48], which was made possible by the extensibility capabilities of SimGrid v4, namely rich and unified resources (Section 4.1) and advanced modeling (Section 4.3).

## 7 Impact on scalability

The usability and extensibility features described in this work have required a full rewrite of SimGrid v3's monolithic simulation core as well as performing API overhauls, including moving from C to C++. Therefore, one may wonder whether these developments have had a negative impact on scalability. To answer this question we consider a benchmark simulator that implements a coordinator-worker application (available on GitHub [66]). Each worker actor executes on one core of a multi-core host and a single coordinator actor executes on one core of a separate host. All hosts are interconnected over a network topology that comprises 100 network links. The route between any two hosts consists of 10 randomly selected links. The core speed of each host is sampled uniformly between 100 and 200 Mflop/sec, and the bandwidth of each link is sampled uniformly between 100 and 200 MB/sec. The coordinator greedily assigns workunits to idle workers. Each workunit entails sending 100 MB of data from the coordinator to the worker, and performing 100 Mflops of computation at the worker. The simulation of energy consumption is enabled. We built this benchmark for both SimGrid v3 and SimGrid v4 in the same Debian 12 Docker image (using the same compiler), and executed it on one core of a dedicated 2.3GHz Intel Xeon Platinum 8380 CPU.

Figure 13 shows results when simulating the execution of between 4,000 and 60,000 workunits on 1,000 4-core hosts. As expected, simulation time increases with the number of workunits (since more discrete events need to be simulated) and memory footprint is roughly constant (since memory is freed each time a workunit completes). The most striking observation is the large reduction in memory footprint when going from SimGrid v3 to SimGrid v4. Specifically, SimGrid v4 leads to at least a 4.1x reduction in memory footprint. This is due to optimizations of data structures and to the move from C to C++, allowing the use of smart pointers with reference counting

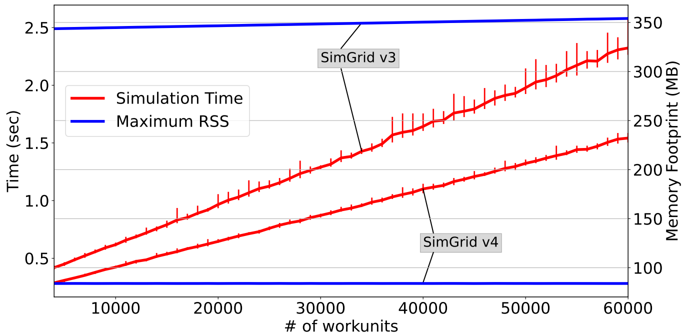



Figure 13: Simulation time and memory footprint vs. number of workunits on 1,000 4-core hosts for a benchmark coordinator-worker simulator, using SimGrid v3 and v4. Lines show average values over 10 repeats; error bars show minimum and maximum values.

for automated garbage collection to avoid memory leaks. The use of smart pointers and the extensibility and usability improvements described in this work do cause increases in simulation time. But these increases are largely offset by usage of the highly optimized data containers provided by the C++ standard library and by the rounds of optimizations that have been applied to the source code over the last decade. As a result, for this benchmark SimGrid v4 leads to at least a 1.45x reduction in simulation time when compared to SimGrid v3.

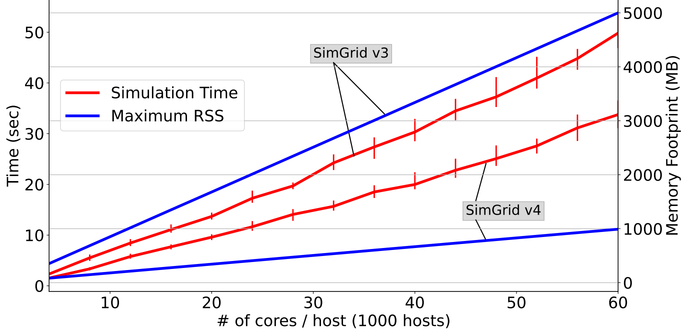



Figure 14: Simulation time and memory footprint vs. number of workers for executing 60,000 workunits for a benchmark coordinator-worker simulator, using SimGrid v3 and v4. Lines show average values over 10 repeats; error bars show minimum and maximum values.

Figure 14 shows results when simulating the execution of 60,000 workunits on 1,000 hosts where the number of cores per host varies from 4 to 60. As expected memory footprint increases due to the fact that each worker actor on each core of the simulated worker hosts has a memory footprint throughout the whole simulated execution. One may expect the execution time to be constant, since the number of simulated discrete events does not depend on the number of worker actors. But as the number of worker actors increases, so does the number of concurrent simulated activities, which in turn increases the computational complexity of the LMM solver (linearly). In these results, for the same reasons as for the results in Figure 13, SimGrid v4 leads to at least a 1.40x memory footprint reduction and a 4.21x simulation time reduction when compared to SimGrid v3.

Overall, while this work has focused on describing features that make SimGrid v4 significantly more extensible and more usable than SimGrid v3, the above results show that it is also significantly more scalable.

# 8 Conclusion

The primary research and development goals of the SimGrid simulation framework have been accuracy and scalability. A key accomplishment is that SimGrid can achieve high accuracy and scalability while remaining versatile, i.e., it has been used for conducting simulations for broad ranges of PDC domains and platforms. In spite of these achievements, a shortcoming of SimGrid has been its usability: the relatively low-level simulation abstractions it provided made the implementation of simulators of complex systems a labor-intensive process. Another shortcoming was extensibility. Although SimGrid provided simulation models that catered to the common case, it was difficult to users to extend these models for research- or domain-specific purposes. In this work we have described several usability and extensibility improvements implemented over the last decade. These improvements have enabled many research results and a led to a rich ecosystem of development and production tools.

Many simulation challenges have been addressed over the last couple of decades resulting by PDC simulation frameworks that provide various levels of accuracy, scalability, versatility, extensibility, and usability. Although future improvements along these five axes are still possible, the state of PDC simulation is now sufficiently mature for broader and overarching challenges to be tackled. One such challenge is simulation calibration: picking the simulation model parameter values in a way that maximizes simulation accuracy with respect to some ground-truth. A review of research articles that have used SimGrid-based simulators in recent years shows that calibration is often not performed, and that when it is performed is is mostly a labor-intensive and manual process [67]. A clear future research direction is the development of an automated simulation calibration tool that can be used by PDC simulator users to achieve desirable trade-offs between accuracy and scalability. Another broad challenge is that of the duplication of effort when it comes to simulator development. Many SimGrid simulator developers implement their own simulation abstractions using the mechanisms in Sections 4 and 5. The question is that of how these custom-developed abstractions, which would often be useful to others, can be contributed back to SimGrid. Relying on pull requests is too labor-intensive (code reviews, increasing amount of

code that must be maintained and documented using the SimGrid standards, regular release schedule constraints, etc.). An alternative approach, used successfully by the ns-3 network simulation framework [68], is to establish a SimGrid app store on which contributors can publish their simulation abstractions as standalone components (composite abstractions, programmatic platform descriptions, plugin implementations, implementations of reusable distributed system building blocks).

# References


[1] The SimGrid Project, http://simgrid.org (2024).

[2] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms, Journal of Parallel and Distributed Computing 74 (10) (2014) 2899–2917. doi:10.1016/j.jpdc.2014.06.008.

[3] P. Bédaride, A. Degomme, S. Genaud, A. Legrand, G. S. Markomanolis, M. Quinson, M. Stillwell, F. Suter, B. Videau, Toward Better Simulation of MPI Applications on Ethernet/TCP Networks, in: Proceedings of the 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2013. doi:10.1007/978-3-319-10214-6_8.

[4] P. Velho, L. M. Schnorr, H. Casanova, A. Legrand, On the Validity of Flow-Level Tcp Network Models for Grid and Cloud Simulations, ACM Transactions on Modeling and Computer Simulation 23 (4) (Dec. 2013). doi:10.1145/2517448.

[5] P. Velho, A. Legrand, Accuracy Study and Improvement of Network Simulation in the SimGrid Framework, in: Proceedings of the 2nd Intl. Conf. on Simulation Tools and Techniques, 2009. doi:10.4108/ICST.SIMUTOOLS2009.5592.

[6] K. Fujiwara, H. Casanova, Speed and Accuracy of Network Simulation in the SimGrid Framework, in: Proceedings of the 1st International Workshop on Network Simulation Tools, 2007. doi:10.4108/nstools.2007.2010.

[7] A. Lèbre, A. Legrand, F. Suter, P. Veyre, Adding Storage Simulation Capacities to the SimGrid Toolkit: Concepts, Models, and API, in: Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzen, China, 2015. doi:10.1109/CCGrid.2015.134.

[8] L. Pouilloux, T. Hirofuchi, A. Lebre, SimGrid VM: Virtual Machine Support for a Simulation Framework of Distributed Systems, IEEE Transactions on Cloud Computing (Sep. 2015). doi:10.1109/TCC.2015.2481422.

[9] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, F. Suter, Simulating MPI applications: the SMPI approach, IEEE Transactions on Parallel and Distributed Systems 18 (8) (2017) 2387–2400. doi:10.1109/TPDS.2017.2669305.

[10] A. Rizvi, T. Toha, M. Lunar, M. Adnan, A. Islam, Cooling Energy Integration in SimGrid, in: Proceedings of the 2017 International Conference on Networking, Systems and Security, 2017, pp. 132–137. doi:10.1109/NSysS.2017.7885814.

[11] F. C. Heinrich, T. Cornebize, A. Degomme, A. Legrand, A. Carpen-Amarie, S. Hunold, A. Orgerie, M. Quinson, Predicting the Energy-Consumption of MPI Applications at Scale Using Only a Single Node, in: Proceedings of the 2017 IEEE International Conference on Cluster Computing, 2017, pp. 92–102. doi:10.1109/CLUSTER.2017.66.

[12] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, B. Videau, Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers, in: Proceedings of the 2015 IEEE 21st International Conference on Parallel and Distributed Systems, 2015, pp. 481–490. doi:10.1109/ICPADS.2015.67.

[13] A. Fanfakh, Predicting the Performance of MPI Applications over Different Grid Architectures, Journal of University of Babylon for Pure and Applied Sciences 27 (1) (2019) 468–477. doi:10.29196/jubpas.v27i1.2232.

[14] L. Stanisic, A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic HPC Applications, Ph.D. thesis, Université Grenoble Alpes, France (2015).
URL https://theses.hal.science/tel-01248109v2/document

[15] T. Cornebize, A. Legrand, F. C. Heinrich, Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study, in: Proceedings of the 2019 IEEE International Conference on Cluster Computing, 2019, pp. 1–11. doi:10.1109/CLUSTER.2019.8891011.

[16] SimGrid's Use in Research Publications, https://simgrid.org/usages.html (2023).

[17] G. Kecskemeti, S. Ostermann, R. Prodan, Fostering Energy-Awareness in Simulations Behind Scientific Workflow Management Systems, in: Proceedings of the 7th IEEE/ACM Intl. Conf. on Utility and Cloud Computing, 2014, pp. 29–38. doi:10.1109/UCC.2014.11.

[18] G. F. Riley, T. R. Henderson, The ns-3 Network Simulator, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 15–34. doi:10.1007/978-3-642-12331-3_2.

[19] L. Mészáros, A. Varga, M. Kirsche, INET Framework, Springer International Publishing, Cham, 2019, pp. 55–106. doi:10.1007/978-3-030-12842-5_2.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, The gem5 simulator, SIGARCH Comput. Archit. News 39 (2) (2011) 1–7. doi:10.1145/2024716.2024718.

[21] T. E. Carlson, W. Heirman, L. Eeckhout, Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation, in: Proc. of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Association for Computing Machinery, New York, NY, USA, 2011. doi:10.1145/2063384.2063454.

[22] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, M. Jung, Amber: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018, pp. 469–481. doi:10.1109/MICRO.2018.00045.

[23] G. G. Castañé, A. Núñez, J. Carretero, iCanCloud: A Brief Architecture Overview, in: 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, 2012, pp. 853–854. doi:10.1109/ISPA.2012.131.

[24] D. Kliazovich, P. Bouvry, Y. Audzevich, S. U. Khan, GreenCloud: A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers, in: 2010 IEEE Global Telecommunications Conference GLOBECOM 2010, 2010, pp. 1–5. doi:10.1109/GLOCOM.2010.5683561.

[25] R. M. Fujimoto, Parallel discrete event simulation, Commun. ACM 33 (10) (1990) 30–53. doi:10.1145/84537.84545.

[26] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, R. Ross, CODES: Enabling Co-Design of Multilayer Exascale Storage Architectures, in: Proc. of the Workshop on Emerging Supercomputing Technologies, 2011.

[27] C. Carothers, D. Bauer, S. Pearce, ROSS: A High-Performance, Low Memory, Modular Time Warp System, in: Proc. of the 14th ACM/IEEE/SCS Workshop of Parallel on Distributed Simulation, 2000, pp. 53–60. doi:10.1109/PADS.2000.847144.

[28] S. Böhm, C. Engelmann, xSim: The extreme-scale simulator, in: Proc. of the International Conference on High Performance Computing & Simulation, 2011, pp. 280–286. doi:10.1109/HPCSim.2011.5999835.

[29] M.-Y. Hsieh, R. Riesen, K. Thompson, W. Song, A. Rodrigues, SST: A Scalable Parallel Framework for Architecture-Level Performance, Power, Area and Thermal Simulation, The Computer Journal 55 (2) (2012) 181–191. doi:10.1093/comjnl/bxr069.

[30] SST/macro 14.1: User's Manual, https://raw.githubusercontent.com/sstsimulator/sst-macro/refs/heads/master/manual-sstmacro-14.1.pdf (2024).

[31] R. Buyya, M. Murshed, GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing, Concurrency and Computation: Practice and Experience 14 (11 2002). doi:10.1002/cpe.710.

[32] T. Goyal, A. Singh, A. Agrawal, Cloudsim: Simulator for Cloud Computing Infrastructure and Modeling, Procedia Engineering 38 (2012) 3566–3572. doi:https://doi.org/10.1016/j.proeng.2012.06.412.

[33] E. U. Yousuf Khan, T. Rahim Soomro, M. Nawaz Brohi, iFogSim: A Tool for Simulating Cloud and Fog Applications, in: Proceedings of the International Conference on Cyber Resilience, 2022, pp. 01–05. doi:10.1109/ICCR56254.2022.9996018.

[34] G. Kecskemeti, DISSECT-CF: A Simulator to Foster Energy-Aware Scheduling in Infrastructure Clouds, Simulation Modelling Practice and Theory 58 (2015) 188–218. doi:https://doi.org/10.1016/j.simpat.2015.05.009.

[35] S. Ostermann, K. Plankensteiner, R. Prodan, T. Fahringer, GroudSim: An Event-Based Simulation Framework for Computational Grids and Clouds, in: Proceedings of the Euro-Par 2010 Parallel Processing Workshops, 2011, pp. 305–313. doi:10.1007/978-3-642-21878-1_38.

[36] F. Mastenbroek, G. Andreadis, S. Jounaid, W. Lai, J. Burley, J. Bosch, E. van Eyk, L. Versluis, V. van Beek, A. Iosup, Opendc 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters, in: Proc. of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2021, pp. 455–464. doi:10.1109/CCGrid51090.2021.00055.

[37] G. Keller, M. Tighe, H. Lutfiyya, M. Bauer, DCSim: A data centre simulation tool, in: Proc. of the IFIP/IEEE International Symposium on Integrated Network Management, 2013, pp. 1090–1091.

[38] X. Li, X. Jiang, P. Huang, K. Ye, DartCSim: An enhanced user-friendly cloud simulation system based on CloudSim with better performance, in: Proc. of the

2nd IEEE International Conference on Cloud Computing and Intelligence Systems, Vol. 01, 2012, pp. 392–396. doi:10.1109/CCIS.2012.6664434.

[39] S. Sotiriadis, N. Bessis, N. Antonopoulos, A. Anjum, SimIC: Designing a New Inter-cloud Simulation Platform for Integrating Large-Scale Resource Management, in: Proc. of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2013, pp. 90–97. doi:10.1109/AINA.2013.123.

[40] S. Sotiriadis, N. Bessis, N. Antonopoulos, Towards Inter-cloud Simulation Performance Analysis: Exploring Service-Oriented Benchmarks of Clouds in SimIC, in: Proc. of the 27th International Conference on Advanced Information Networking and Applications Workshops, 2013, pp. 765–771. doi:10.1109/WAINA.2013.196.

[41] Y. Shi, X. Jiang, K. Ye, An Energy-Efficient Scheme for Cloud Resource Provisioning Based on CloudSim, in: Proc. of the IEEE International Conference on Cluster Computing, 2011, pp. 595–599. doi:10.1109/CLUSTER.2011.63.

[42] L. Bobelin, A. Legrand, D. A. G. Márquez, P. Navarro, M. Quinson, F. Suter, C. Thiery, Scalable Multi-Purpose Network Representation for Large Scale Distributed System Simulation, in: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012, pp. 220–227. doi:10.1109/CCGrid.2012.31.

[43] M. Quinson, C. Rosa, C. Thiery, Parallel Simulation of Peer-to-Peer Systems, in: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2012, pp. 668—-675.

[44] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (4) (2009). doi:10.1145/1592434.1592438.

[45] M. Laurent, E. Saillard, M. Quinson, The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation, in: Proc. of the 5th IEEE/ACM International Workshop on Software Correctness for HPC Applications (Correctness), 2021, pp. 1–9. doi:10.1109/Correctness54621.2021.00008.

[46] G. Cooperman, M. Quinson, Sthread: In-Vivo Model Checking of Multithreaded Programs, The Art, Science, and Engineering of Programming 4 (3) (2020). doi:10.22152/programming-journal.org/2020/4/13.

[47] B. Camus, A.-C. Orgerie, M. Quinson, Co-simulation of FMUs and Distributed Applications with SimGrid, in: Proceefings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, 2018, pp. 145–156. doi:10.1145/3200921.3200932.

[48] L. Stanisic, S. Thibault, A. Legrand, B. Videau, J.-F. Méhaut, Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures, Concurrency and Computation: Practice and Experience 27 (16) (2015) 4075–4090. doi:https://doi.org/10.1002/cpe.3555.

[49] C. Courageux-Sudan, L. Guegan, A.-C. Orgerie, M. Quinson, A Flow-Level Wi-Fi Model for Large Scale Network Simulation, in: Proceedings of the International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, 2022. doi:10.1145/3551659.3559022.

[50] Message Passing Interface Forum, MPI: A message-passing interface standard version 4.0, https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (Jun. 2021).

[51] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, M. Quinson, Single Node On-Line Simulation of MPI Applications with SMPI, in: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, 2011. doi:10.1109/IPDPS.2011.69.

[52] SimGrid “Frankenstein" example simulator, https://github.com/henricasanova/simgrid_frankenstein (2024).

[53] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, S. Albrecht, J. Oeth, F. Suter, Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH, Future Generation Computer Systems 112 (2020) 162–175. doi:10.1016/j.future.2020.05.030.

[54] P.-F. Dutot, M. Mercier, M. Poquet, O. Richard, Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator, in: Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing, 2016. doi:10.1007/978-3-319-61756-5_10.

[55] T. Özden, T. Beringer, A. Mazaheri, H. M. Fard, F. Wolf, ElastiSim: A Batch-System Simulator for Malleable Workloads, in: Proceedings of the 51st International Conference on Parallel Processing, 2023. doi:10.1145/3545008.3545046.

[56] ECP Proxy Apps, https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/ (2023).

[57] Y. Zheng, P. Marguinaud, Simulation of the Performance and Scalability of Message Passing Interface (MPI) Communications of Atmospheric Models Running on Exascale Supercomputers, Geoscientific Model Development 11 (8) (2018) 3409–3426. doi:10.5194/gmd-11-3409-2018.

[58] SMPI Proxy Apps, https://framagit.org/simgrid/SMPI-proxy-apps (2023).

[59] OpenMPI Test Suite, https://github.com/open-mpi/mpi-test-suite (2023).

[60] Intel®MPI Benchmarks, https://github.com/intel/mpi-benchmarks (2023).

[61] J. Emmanuel, M. Moy, L. Henrio, G. Pichon, S4BXI: the MPI-ready Portals 4 Simulator, in: Proceedings of the 29th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2021, pp. 1–8. doi:10.1109/MASCOTS53633.2021.9614285.

[62] B. Barrett, R. B. Brightwell, R. Grant, K. Pedretti, K. Wheeler, K. D. Underwood, R. Riesen, A. B. Maccabe, T. Hudson, S. Hemmert, The Portals 4.1 Network Programming Interface, Tech. Rep. SAND2017-3825, Sandia National Laboratory, Albuquerque, NM (Apr. 2017). doi:10.2172/1365498.

[63] L. E. Ratcliff, W. Dawson, G. Fisicaro, D. Caliste, S. Mohr, A. Degomme, B. Videau, V. Cristiglio, M. Stella, M. D'Alessandro, S. Goedecker, T. Nakajima, T. Deutsch, L. Genovese, Flexibilities of Wavelets as a Computational Basis Set for Large-Scale Electronic Structure Calculations, The Journal of Chemical Physics 152 (19) (2020) 194110. doi:10.1063/5.0004792.

[64] F. Affinito, U. Alekseeva, C. Cavazzoni, A. Degomme, P. D. Delugas, A. Ferretti, A. Garcia, A. Kozhevnikov, P. Ordejón, N. Spallanzani, Second Report on Code Profiling and Bottleneck Identification, Deliverable d4.3, European Centre of Excellence in materials modelling, simulations and design (2018).
URL https://www.max-centre.eu/sites/default/files/D4.3%20Second%20report%20on%20code%20profiling%20and%20bottleneck%20identification.pdf

[65] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23 (2011) 187–198. doi:10.1002/cpe.1631.

[66] SimGrid "Coordinator-Worker flashback" benchmark simulator, https://github.com/simgrid/coordinator_worker_flashback (2024).

[67] J. McDonald, M. Horzela, F. Suter, H. Casanova, Automated Calibration of Parallel and Distributed Computing Simulators: A Case Study, in: Proc. of the 25th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC), 2024.

[68] The ns-3 App Store, https://www.nsnam.org/docs/contributing/html/external.html (2024).