Assessing the Performance of MPI Applications Through Time-Independent Trace Replay

Frédéric Desprez*, George S. Markomanolis*, Martin Quinson†, Frédéric Suter‡

*INRIA, LIP, ENS Lyon, Lyon, France
†Nancy University, LORIA, INRIA, Nancy, France
‡IN2P3 Computing Center, CNRS, IN2P3, Lyon-Villeurbanne, France

***Abstract*—Simulation is a popular approach to obtain objective performance indicators platforms that are not at one's disposal. It may help the dimensioning of compute clusters in large computing centers. In this work we present a framework for the off-line simulation of MPI applications. Its main originality with regard to the literature is to rely on time-independent execution traces. This allows us to completely decouple the acquisition process from the actual replay of the traces in a simulation context. Then we are able to acquire traces for large application instances without being limited to an execution on a single compute cluster. Finally our framework is built on top of a scalable, fast, and validated simulation kernel.**

In this paper, we present the used time-independent trace format, investigate several acquisition strategies, detail the developed trace replay tool, and assess the quality of our simulation framework in terms of accuracy, acquisition time, simulation time, and trace size.

***Index Terms*—Message Passing Interface; Off-line simulation; Performance prediction.**

I. Introduction

Computational Science is the third scientific way to study problems arising in various domains such as physics, biology, or chemistry. It is complementary to theory and actual experiments and consists in conducting studies *in silico*. This approach makes an heavy use of resources located in computing centers. As the number of scientific domains producing results from *in silico* studies increases, the computing centers then have to upgrade their infrastructures in a continuous way. The resources impacted by such upgrades are of different kinds, e.g., computing, network and storage. Moreover each kind of resource grows more complex with each generation. Processors have more and more cores, low latency and high bandwidth network solutions become mainstream and some disks now have access time close to that of memory.

The complexity of the decision process leading to the evolution of a computing center is then increasing. This process often relies on years of experience of system administrators and users. The former knows how complex systems work while the latter have expertise on the behavior of their applications. Nevertheless this process lacks of objective data about the performance of a given candidate infrastructure. Such information can only be obtained once the resources have been bought and the applications can be tested. Any unforeseen behavior can then lead to tremendous but vain expenses.

Many simulation frameworks have been proposed over the last decade to obtain objective indicators beforehand. Most of them focus on parallel applications relying on the Message Passing Interface (MPI) [1]. The simulation approaches used in these frameworks fall into two categories: *off-line simulation*, also called trace-based simulation or post-mortem simulation, and *on-line simulation*, also called simulation via direct execution. In off-line simulation a log of a previous execution of the application is "replayed" on a simulated platform. In on-line simulation the application is executed but part of the execution takes place within a simulation component.

In this work we present a framework to simulate MPI applications following the off-line approach. This effort is part of the SimGrid project [2]. It leverages the simulation existing techniques and models in SimGrid. More specifically, this work makes these main contributions with respect to the MPI simulation literature:

1) Propose a new execution log format that is independent of time; This format includes volumes of computation (in number of instructions) and communications (in bytes) for each event instead of classical time-stamps;
2) An original approach that totally decouples the acquisition of the trace from its replay; Several scenarios can then be proposed that allow for the acquisition of large execution traces;
3) A trace replay tool on top of fast, scalable and validated simulation kernel; This ensures the efficiency and quality of our off-line simulation framework;
4) Experimental results that show the simulation accuracy, the acquisition time, the simulation time, and the trace size, of our off-line simulation framework.

This paper is organized as follows. Section II reviews related work. Section III introduces the format of a time-independent trace of an MPI application while Section IV details the acquisition process of such traces. Section V explain how time-independent traces can be replayed within the SimGrid simulation framework. Our prototype is evaluated in Section VI with regard to different metrics. Section VII concludes with a summary of results and perspectives on future research directions.

II. Related Work

One option for simulating the execution of an MPI application is *on-line simulation*. In this approach, the actual code, with no or only marginal modification, is executed on a *host platform* that attempts to mimic the behavior of a *target platform*, i.e., a platform with hardware different characteristics. Part of the instruction stream is then intercepted and passed to a simulator. LAPSE is a well-known on-line simulator developed in the early 90's [3] (see therein for references to precursor projects). In LAPSE, the parallel application executes normally but when a communication operation is performed a corresponding communication delay is simulated on the target platform using a simple network model (affine point-to-point communication delay based on link latency and bandwidth). MPI-SIM [4] builds on the same general principles, with the addition of I/O subsystem simulation. A difference with LAPSE is that MPI processes run as threads, a feature which is enabled by a source code preprocessor. Another project similar in intent and approach is the simulator described in [5]. The BigSim project [6], unlike MPI-SIM, allows the simulation of computational delays on the target platform. This makes it possible to simulate "what if?" scenarios not only for the network but also for the compute nodes of the target platform. These computational delays are based either on user-supplied projections for the execution time of each block of code, or on scaling measured execution times by a factor that accounts for the performance differential between the host and the target platforms, or based on sophisticated execution time prediction techniques such as those developed in [7]. The weakness of this approach is that since the computational application code is not executed, the computed application data is erroneous. Data-dependent behavior is then lost. This is acceptable for many regular parallel applications, but is more questionable for irregular applications (e.g., branch-and-bound or sparse matrices computations). Going further, the work in [8] uses a cycle-accurate hardware simulator of the target platform to simulate computation delays, which leads to a high ratio of simulation time to simulated time.

One difficulty faced by all above MPI-specific on-line simulators is that the simulation, because done through a direct execution of the MPI application, is inherently distributed. Parallel discrete event simulation raises difficult correctness issues pertaining to process synchronization. For the simulation of parallel applications, techniques have been developed to speed up the simulation while preserving correctness (e.g., the asynchronous conservative simulation algorithms in [9], the optimistic simulation protocol in [6]). A solution could be to run the simulation on a single node but it requires large amounts of CPU and RAM resources. For most aforementioned on-line approaches, the resources required to run a simulation of an MPI application are commensurate to those of that application. In some cases, those needs can even be higher [8], [10]. One way to reduce the CPU needs of the simulation is to avoid executing computational portions of the application and simulate only expected delays on the target platform [6]. Reducing the need for RAM resources is more difficult and if the target platform is a large cluster, then the host platform must then be a large cluster. SMPI [11], which builds on the same simulation kernel as this work, implement all the above techniques to allow for efficient single-node simulation of MPI applications.

One approach that avoids these particular challenges, but that comes with challenges of its own, is *off-line* simulation. In this approach, which we use in this work, a log, or trace, of MPI communication events (time-stamp, source, destination, data size) is first obtained by running the application on a real-world platform. A simulator then replays the execution of the application as if it were running on a *target platform*. This approach has been used extensively, as shown by the number of trace-based simulators described in the literature since 2009 [12], [13], [14], [15], [16]. The typical approach is to compute the durations of the time intervals between MPI communication operations, or "CPU bursts." Upon replaying the application, the CPU bursts are modified to account for the performance differential between the platform used to obtain the trace and the target platform, either using simple scaling [13], [14], [16] or using a more sophisticated convolution between the application computational signature and the target platform's hardware signature [7]. Network communications are simulated based on the communication events recorded in the trace and on a simulation model of the network. This tight link between *host* and *target* platforms, caused by the time-stamps in the traces, is a potential drawback of the off-line approach. Indeed it limits the usage of the traces to similar platform for which simple scaling is possible.

A challenge for off-line simulation is the large size of the traces, which can prevent running the simulation on a single node. Mechanisms have been proposed to improve scalability, including compact trace representations [13] and replay of a judiciously selected subset of the traces [15]. Another challenge is that the it is typically necessary to obtain the trace on a platform that has the same scale as the target platform. However, trace extrapolation to larger numbers of nodes than that of the platform used to obtain the trace is feasible in some cases [12], [14].

For both approaches, the complexity of the network simulation model has a high impact on speed and scalability, thus compelling many authors to adopt simplistic network models. One simplification, for instance, is to use monolithic performance models of collective communications rather than simulating them as sets of point-to-point communications [13], [17]. Another simplification used in most aforementioned simulators, whether off-line or on-line, is to ignore network contention because it is known to be costly to simulate [18]. The work in [13] proposes the use of simple analytical models of network contention for off-line simulation. Exceptions are the MPI-NetSim on-line simulator [10], which relies on a slow packet-level discrete even network simulator, and the SMPI [11] project that benefit of the advanced contention models of the SimGrid simulation kernel [2]. The present work also builds on this simulation kernel.

III. Time-Independent Trace Format

All the off-line MPI simulators reviewed in the previous section relies on timed traces, i.e., each occuring event is associated to a time-stamp. Consequently a simulator has to apply a correction factor to these time-stamps when it simulates the execution on the target platform. This implies to know precisely what are the respective performance of the host and target platforms. In other words, each execution trace must come with an accurate description of how it has been acquired. Finally determining the right scaling factor can be tedious depending on the degree of similarity of both platforms.

To free ourselves of these constraints related to time-stamps, we propose in this work to produce *time-independent* traces. For each event occurring during the execution of the application, e.g., a CPU burst or a communication operation, we log the volume of the operation (in number of instructions or bytes) instead of the time spent to execute it. Indeed this type of information does not vary with the characteristics of the host platform. For instance, the size of the messages sent by an application is not likely to change according to the specifics of the network interconnect, while the computation amount performed within a `for` loop does not increase with the processing speed of a CPU. This claim is not valid for adaptive MPI applications that modify their execution path according to the execution platform. This type of applications, that represents only a small fraction of all MPI applications, is not covered by our work.

A time-independent trace can then been seen as a list *actions*, e.g., computations and communications, performed by each process of an MPI application. An action is described by the *id* of the process that does this action, a *type*, e.g., a computation or a communication operation, a *volume*, i.e., a number of instructions or bytes, and some action specific parameters, e.g., the id of the receiving process for a one-way communication.

```
for (i=0; i<4; i++){
 if (myId == 0){
  /* Compute 1M instructions */
  MPI_Send(1MB,..., (myId+1));
  MPI_Recv(...);
 } else {
  MPI_Recv(...);
  /* Compute 1M instructions */
  MPI_Send(1MB,...,
           (myId+1)% nproc);
 }
}
```

```
p0 compute 1e6
p0 send p1 1e6
p0 recv p3

p1 recv p0
p1 compute 1e6
p1 send p2 1e6

p2 recv p1
p2 compute 1e6
p2 send p3 1e6

p3 recv p2
p3 compute 1e6
p3 send p0 1e6
```

Figure 1. MPI code sample of some computation on a ring of processes (left) and its equivalent time-independent trace (right).

The left hand side of Figure 1 shows a simple computation on a ring of four processes. Each process computes one million instructions q and send one million bytes to its neighbor. The right hand side of this figure displays the corresponding time-independent trace. Note that, depending on the number of processes and the number of actions, it may be preferable to split the time-independent trace in several files, e.g., one file per process.

MPI actions	Trace entry
CPU burst	`<id> compute <volume>`
`MPI_Send`	`<id> send <dst_id> <volume>`
`MPI_Isend`	`<id> Isend <dst_id> <volume>`
`MPI_Recv`	`<id> recv <src_id>`
`MPI_Irecv`	`<id> Irecv <src_id>`
`MPI_Broadcast`	`<id> bcast <volume>`
`MPI_Reduce`	`<id> reduce <vcomm> <vcomp>`
`MPI_Allreduce`	`<id> allReduce <vcomm> <vcomp>`
`MPI_Barrier`	`<id> barrier`
`MPI_Init`	`<id> init`
`MPI_Finalize`	`<id> finalize`
`MPI_Wait`	`<id> wait`

Table I
Time-independent counterparts of the actions performed by each process involved in a MPI application.

Table I lists all the MPI functions for which there is a corresponding action implemented in our first prototype. For the collective operations, we consider that all the processes are involved as the `MPI_Comm_split` function is not implemented. Another design choice is to root these collective operations on process 0. Finally the `init` and `finalize` actions have to appear in the trace file associated to each process to respectively create and destroy the needed internal data structures.

IV. Trace Acquisition Process

In this section we detail the acquisition process of a time-independent execution trace that can be replayed within a simulation environment. This process, depicted in Figure 2 comprises four steps: (i) the instrumentation of the application; (ii) the execution of the instrumented version of the application; (iii) the extraction of the action list for each process; and (iv) the gathering of the different traces into a single node. In what follows we give some details on each of these steps.

A. Instrumentation

The first step of the acquisition process is to instrument the application. We base our prototype on TAU [19] for its non-intrusive instrumentation method. TAU is actually a profiling tool that offers a tracing features. To enable them the `-TRACE` flag has to be used and some environment variables, e.g., `TAU_TRACK_MESSAGE` to track message sender and receiver ids, have to be set. The semi-automatic instrumentation described in what follows is activated by the `-pdt` flag, while the access to hardware counters through the PAPI [20] interface is allowed by adding the `-papi` flag to the command line.

An interesting feature of TAU is selective instrumentation. It can be done in different ways. One consists in listing in a separate file which functions have (or have not) to be traced. All the functions in the call path of the listed functions will also be traced. However, this technique may

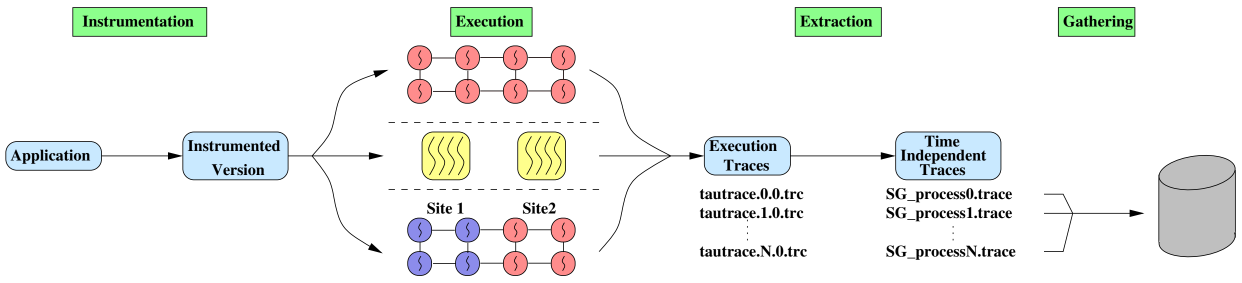

Figure 2. Time-independent traces acquisition process.

not be enough to isolate a given call, e.g., if a function is called twice but only one call has to be traced. A solution is then to insert two macros provided by the tau API, namely `TAU_ENABLE_INSTRUMENTATION` and `TAU_ENABLE_INSTRUMENTATION` in the source code.

The following example illustrates the instrumentation of the `SSOR(itmax)` function call in a LU factorization.

```
      call TAU_ENABLE_INSTRUMENTATION()
      call ssor(itmax)
      call TAU_DISABLE_INSTRUMENTATION()
```

One additional call to each of these macros is required to define neat disable/enable sections. Such slight modifications of the source code can be handled by the pre-processor. Indeed the initial program has to be compiled using one of the scripts provided by TAU, i.e., `tau_cc.sh` and `tau_f77.sh` for C and Fortran codes respectively.

B. Execution

As shown in Figure 2, a time-independent trace can be acquired in many different ways. The only mandatory parameter is the number of participating processes, as the trace only comprises information about computation and communication volumes. We remind that the off-line approach is not suited for irregular and adaptive applications and that regular applications represent a large part of current MPI codes used in production.

Figure 2 shows three possible acquisition methods that consist in running the MPI application in a:

a) Regular mode: with one process per CPU. This is the way the other off-line simulators obtain execution traces. Indeed, such an acquisition prevents abnormal timings due to resource access concurrency. The main drawback of this acquisition mode is to require as many processors as comprised in the target platform to get a trace. Its scalability is thus limited and the acquisition of traces for larger is not possible.

b) Folding mode: with more than one process per CPU. This allows for the acquisition of traces for larger instances of the application or to use less resources. The folding factor is obviously limited by the available amount of memory on the involved computing nodes.

c) Scattering mode: where the CPUs do not necessarily belong to the same commodity cluster. More nodes than what is available on the target platform can thus be used to acquire traces of large instance. This mode thus tackles the issues of the regular mode.

A fourth acquisition can also be envisioned. The *Scattering and Folding* mode is, as the name says, the combination of the last two modes. It further increases the scalability of the acquisition process. Apart from the *regular* mode, all the proposed acquisition methods are possible only because the independence to time of the produced traces. Consequently, other off-line simulators can not take benefits of these alternate acquisition modes.

C. Post-processing of the Execution Traces

When the execution of a program instrumented with TAU completes, many files are produced. They fall in two categories: *trace* files and *event* files. The generated trace files are named:

`tautrace.<node>.<context>.<thread>.trc`,

where `<node>` is the rank of the MPI process whose execution is logged in the file. The two other fields, i.e., `<context>` and `<thread>`, are only used for multi-threaded applications. In this case, TAU distinguishes each thread and groups the threads according to the virtual address space they share.

A *trace file* is a binary file that includes all the events that occur during the execution of the application for a given process. For each event, this file indicates when this event (e.g., a function call or an instrumented block) starts and finishes. The time spent and the number of computed instructions between these begin/end tags are also stored. For MPI events all the parameters of the MPI call, including source, destination, and message size, are stored.

To reduce the size of the trace files, TAU stores a unique id for each traced event instead of its complete signature. The matching between the ids and the the functions descriptions can be found in the *event files*. These files are named:

`events.<node>.edf`.

There is only one event file per MPI process. Each event file contains information about each traced function. For any function, an event file stores its numerical id, the group it belongs to, e.g., `MPI` for all MPI functions, a tag to distinguish TAU events from those defined by the user, and the *name type* which is the actual name of the traced function. Some extra parameters required by TAU can also be stored into an event file. For instance, the keyword `EntryExit` is used to declare a function that occurs between two separate events, i.e., entry and exit. Conversely the `TriggerValue` keyword typically corresponds to a counter that increases monotonically from the beginning of the execution. Such a trigger has to be activated twice to determine the evolution of the counter value during the corresponding period of time.

The following example shows two entries of an event file generated by TAU that corresponds respectively to the `MPI_Send` function and to the access to an hardware counter that measures the number of instructions.

```
49 MPI 0 "MPI_Send()  " EntryExit
1 TAUEVENT 1 "PAPI_TOT_INS" TriggerValue
```

To perform an off-line simulation of a MPI application with SimGrid, two steps are mandatory. First we have to *extract* a time-independent trace from the trace and event files produced by TAU. Second we have to *gather*, and sometimes *merge*, the extracted traces on a single node where the replay takes place.

As the trace files generated by TAU are binary files, there is a need for an interface to extract information. Such an API is provided by the TAU Trace Format Reader library (TFR) [21]. This tool provides the necessary functions to handle a trace file, including a function to read events. It also defines a set of eleven callback methods, that correspond to the different kinds of events that appear in a TAU trace file. For instance there are callbacks for entering or exiting a function and triggering a counter. The implementation of these callback methods is let to the developer.

We thus developed a C/MPI parallel application, called `tau2simgrid`, that implements the different callback methods of the TFR library. This program basically opens, in parallel, all the TAU trace files and read them line by line. For each event, the corresponding callback function is called. To illustrate how `tau2simgrid` extracts the necessary data to produce a time-independent trace, we detail the case of a call to the `MPI_Send` function. Figure 3 presents the parameters of the different callbacks related to this function call on process 1 in a readable format. Each line starts by the process id, the thread id, the time at which the event occurred and the name of the event. The remaining fields are event dependent.

```
1  1 0 1.42947e+06 EnterState     49
2  1 0 1.42947e+06 EventTrigger    1  164035532
3  1 0 1.4295e+06  EventTrigger   46  163840
4  1 0 1.4295e+06  SendMessage   0 0  163840      1 0
5  1 0 1.4299e+06  EventTrigger    1  164035624
6  1 0 1.4299e+06  LeaveState     49
```

Figure 3. List of callbacks related to a call to the `MPI_Send` function.

As mentioned earlier, the event that corresponds to a `MPI_Send` is tagged as `EntryExit` in the event file with the event id 49. The first occurring callback will then be on the `EnterState` function (line 1). The matching `LeaveState` event (line 6) define the scope of events related to the function call. Four events are enclosed between these boundaries. Two of them (lines 2 and 5) correspond to the hardware counter measuring the number of instructions, as identified in the event file and indicated by the event id 1 in the trace file. These two events are used to respectively ends the CPU burst preceding the MPI call and starts the next one. The number of instructions computed within a MPI call, mainly due to buffer allocation costs, are ignored as they are accounted for by the network model. The last two events are related to the sent message. The `EventTrigger` on line 3 only provides the size of the message (163,840 bytes), which is not enough to build an entry in the time-independent trace. The `SendMessage` event (line 4) gives more information, namely the process and thread ids of the receiver, the size of the message, and the MPI tag and communicator for this communication.

Thanks to all these information extracted from both TAU trace and event files, our `tau2simgrid` application can generate the following entry of a time-independent trace:

```
p1 send p0 163840
```

Note that for asynchronous and collective communications, the extraction process is more complex. For instance, the mandatory information to write the entry corresponding to a `MPI_Irecv`, e.g., the receiver id, are given by the `RecvMessage` event which generally occurs within the `MPI_wait` function. This implies to implement some lookup techniques in `tau2simgrid` to retrieve all the necessary parameters.

After the execution of `tau2simgrid`, the produced traces can be injected into the simulation framework. However, they first have to be gathered on a single node onto which the simulation will take place. File gathering is a problem that has been studied for a long time. A common and efficient approach is to rely on a K-nomial tree reduction allowing for $\log_{(K+1)} N$ steps, where N is the total number of files, and K is the arity of the tree. We developed a simple script to perform such a gathering. This script can be configured to adapt the arity to the total number of traces and the number of computed nodes involved in the trace acquisition.

The cost, in terms of execution time, of `tau2simgrid` and the gathering script will be assessed in Section VI.

V. Trace Replay with SimGrid

Our framework to simulate MPI applications following the off-line approach is tightly connected to the SimGrid project [2]. SimGrid provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments. The main goal of SimGrid is to facilitate the research in the area of parallel and distributed large scale systems such as Grids, P2P systems and clouds. SimGrid relies on a scalable and extensible simulation engine and offers several user APIs.

Since release 3.3.3, SimGrid allows users to describe an applicative workload as a time-independent trace such as that described in Section III. Different components are needed to replay such traces with SimGrid. Apart from the *time-independent trace(s)*, a description of the *target platform* and the *deployment* of the application, i.e., how processes are mapped onto processors, are also passed to the *trace replay tool* which, in turn, is built on top of the *simulation kernel*. Decoupling the simulation kernel, and then the simulator, from the simulation scenario offers a greater flexibility. This way a wide range of "what if?" scenarios can be explored without any modification of the simulator. Changing the input files of the *trace replay tool* is enough.

The *simulation kernel* of SIMGrid relies on macroscopic models for computation resources. Tasks costs are expressed in number of instructions. The processing rate of a CPU is then in instructions/s. For network resources, SIMGrid uses an analytical network contention model. This model was developed for arbitrary network topologies with end-points that use standard network protocols, such as TCP/IP, and are connected via multi-hop paths. Instead of being packet-based, the model is flow-based, meaning that at each instant the bandwidth allocated to an active flow (i.e., a data transfer occurring between two end-points) is computed analytically given the topology of the network and all currently active flows. This model is described and validated via comparison to the GTNetS packet-level simulator in [22]. While this generic model is applicable to networks ranging from local-area to wide-area networks, it can be specialized for cluster interconnects. An original model has been recently added to the SIMGrid simulation kernel to take the specifics of MPI implementations on compute cluster interconnects using TCP into account [11]. For instance, a message under 1 KiB fits within an IP frame, in which case the achieved data transfer rate is higher than for larger messages. Also, MPI implementations for `MPI_Send()` typically switch from buffered to synchronous mode above a certain message size. Consequently, instead of being an affine function of message size, communication time is *piece-wise linear*. This model is instantiated for 3 segments, leading to 8 parameters defining the model (2 for defining the boundaries of the 3 segments, and one latency and bandwidth parameter for each segment).

Figure 4 present a *platform* and *deployment* file that correspond to the scenario of Figure 1. The platform described here is a compute cluster that comprises four homogeneous machines interconnected through a switched network. The deployment file indicates on which node of the cluster each process will run. For instance, the MPI process of rank 0 will be executed on the node named `cluster-0.site.fr`

An off-line simulation can produce various types of outputs. In this work, we focus on obtaining a *simulated execution trace*. This kind of output provides an estimation of the execution time of the target application in the particular experimental scenario described by the platform and deployment files. It is also possible to generate a timed trace that corresponds to this particular scenario by adding timers (measuring simulated time) in the trace replay tool. Finally it would also be interesting to derive a profile of the application from this timed trace. But this last kind of output requires complex analysis tools such as those develop in the TAU and Scalasca [23] projects.

To replay a time-independent trace with SIMGrid, a simulator, the *trace replay tool* has to be written on top of the

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="3">
 <AS id="AS_site" routing="Full">
  <cluster id="AS_cluster" prefix="cluster-"
           suffix=".site.fr" radical="0-3"
           power="1.17E9" bw="1.25E8" lat="16.67E-6"
           bb_bw="1.25E9" bb_lat="16.67E-6"/>
 </AS>

 <process host="cluster-0.site.fr" function="p0"/>
 <process host="cluster-1.site.fr" function="p1"/>
 <process host="cluster-2.site.fr" function="p2"/>
 <process host="cluster-3.site.fr" function="p3"/>
</platform>
```

Figure 4. Example of SIMGrid platform and deployment file corresponding to the MPI application described in Figure 1.

simulation kernel using one of the APIs provided by SIMGrid. Our prototype uses the MSG API. This simulator has to:

1) Include a function that corresponds to the expected behavior of a given action. This has to be done for each action that occurs in the trace. Generally such function just calls one or several MSG functions.
2) Register this function with `MSG_action_register`. This call made in the `main` function of the simulator links the action keyword (as defined in Table I) to the function defined in the previous step.
3) Call the function `MSG_action_trace_run` that takes either a trace file name or `NULL` as input. When no file name is given, this mean that there exists one trace file per process. In this case, the names of these trace files are given in the platform file, as shown below.

```
<process host="cluster-1.site.fr" function="p1">
  <argument value="SG_process1.trace"/>
</process>
```

An essential step to make accurate performance predictions through trace replay is the calibration of the simulation framework. In the case of SIMGrid, it consists in instantiating the platform file with pertinent values. In other words the number of instructions a CPU can compute in one second and the latency and bandwidth of communication links have to be set. Such a calibration strongly depends on both application and execution environment. Indeed different types of computation may lead to different processing rates on a given CPU. This is mainly due to how efficiently the computation can use the different levels of cache. Moreover the performance of a given computation may differ with regard to the processor brand.

We instantiate the processing rate of each host in the platforms as follows. A small instrumented instance of the target application is run on the platform to describe. This allows us to determine the number of instructions of each event as long as the time spent to computed them. Then we can determine a processing rate of each single action, compute a weighted average on each process, and get an average processing rate for all the process set. Finally we repeat this this procedure five times and compute an average over these five runs to smoothe the runtime variations. We use this final value to instantiate the SIMGrid platform file.

The instantiation of the network parameters of the platform file is done in two steps. To set the bandwidth, we use the nominal value of the links, e.g., 1 GiB for GigaEthernet links. For the latency of a communication link, we rely on the `Pingpong_Send_Recv` experiment of the SKaMPI [24] benchmark suite. We take the value obtained for a 1-byte message and divided it by six. This factor of six comes from two sources. We have to divide the ping-pong time by two to obtain the latency of a one-way message. Then we divide it by three to take the topology of a cluster into account. Indeed, two nodes in a compute cluster are generally connected through two links and one switch. In case of hierarchical network, we account for this hierarchy in the determination of the latency.

The second step consists in instantiating the piece-wise linear model used by SIMGrid dedicated to MPI communications on compute clusters. SIMGrid provides a Python script that takes as input the latency and bandwidth determined as above, the output of the SKaMPI run, and the number of links connecting the two nodes used for the ping-pong. Then this script determines the latency and bandwidth correction factors that lead to a best-fit of the experimental data for each segment of this piece-wise linear model.

VI. EXPERIMENTAL EVALUATION

A. Experimental Setup

To conduct this evaluation, we selected three kernels with different characteristics from the NAS Parallel Benchmarks (NPB) suite. The NPB are a set of programs commonly used to assess the performance of parallel platforms. Each benchmark can be executed for 7 different *classes*, denoting different problem sizes: S (the smallest), W, A, B, C, D, and E (the largest). For instance, a class D instance corresponds to approximately 20 times as much work and a data set almost 16 as large as a class C problem. We selected the Embarrassing Parallel (EP) because it is only composed of computations, the Data Traffic (DT) benchmark which iscommunication-intensive benchmark and the LU factorization (LU) that mixes computations and communications. Note that the classes have not the same meaning for the DT benchmark. The class refers not only to the data size but also the number of communicating processes. Classes A, B and C respectively involve 21, 43 and 85 processes. Moreover two of communication graphs can be tested. We focus on th Black Hole graph (BH) that collects data from multiple sources in a single sink.

To acquire trace-independent traces we used two clusters of the Grid'5000 experimental platform: *bordereau* and *gdx*. The *bordereau* cluster comprises 93 2.6GHz Dual-Proc, Dual-Core AMD Opteron 2218 nodes. All these nodes are connected to a single 10 Gigabit switch. The *gdx* cluster comprises 186 2.0 GHz Dual-Proc AMD Opteron 246 scattered across 18 cabinets. Two cabinets share a common switch and all these switches are connected to a single second level switch through Ethernet 1 Gigabit links. Consequently a communication between two nodes located in two distant cabinets goes through three different switches. These two clusters are interconnected through a dedicated 10 Gigabit network.

One of the key concept of the Grid'5000 experimental platform is to offer its users the capacity to deploy their own system image at will. For our experiments we built a Debian Lenny image. The kernel (v2.6.25.9) was patched with the `perfctr` driver (v2.6.38) to enable hardware counters. We also installed TAU (v2.18.3) and its software dependencies on Program Database Toolkit (v3.14.1), which is used for automatic instrumentation, and PAPI (v3.7.0), that provides access to the hardware counters. We built the NAS Parallel Benchmarks (v3.3) on top of OpenMPI (v1.3.3). Finally the traces are replayed in SIMGrid (v3.6-r9613).

B. Evaluation of the Acquisition Modes

The main characteristic of the proposed acquisition process is to be totally decoupled from the target platform. As mentioned in Section IV, several modes can be used to acquire a time-independent trace. The first experiment of this section investigates the distribution of the acquisition time among the different steps, i.e., execution, instrumentation, extraction, and gathering. Figure 5 shows such a distribution for the acquisition of time-independent traces of the three considered benchmarks for classes B and C and for different number of processes. These results were obtained on the *bordereau* cluster in the *Regular* acquisition mode and deploying only one process per node. We run the complete acquisition process ten times and show the average value for each part of it.

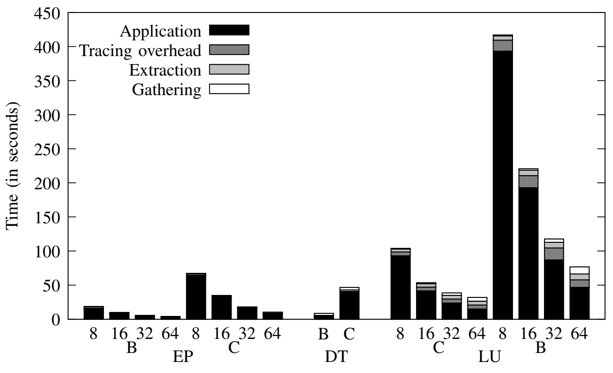

Figure 5. Distribution of the acquisition time for different instances of the EP, DT, and LU benchmarks in the *regular* acquisition mode.

As more processes are involved in the computation of the EP and DT benchmarks, the time needed to run the application, get a TAU trace, and extract the time-independent information decreases linearly with the number of processes. This direct benefit of parallelism exploitation shows its limits when the sequential part of the execution becomes too small, as for Class B on 64 processes, for instance. Conversely, the time needed to gather all the generated traces on a single node, increases with the depth of the reduction tree. We see an opposite behavior for the DT benchmark as going from class B to class C implies more communications to execute.

The acquisition time strictly related to the production of time-independent traces, i.e., the extraction and gathering steps, represents at most 49.31% of the total acquisition time.

	Acquisition mode	R	F-2	F-4	F-8	F-16	F-32	S-2	SF-(2,2)	SF-(2,4)	SF-(2,8)	SF-(2,16)
	Number of nodes	64	32	16	8	4	2	(32,32)	(16,16)	(8,8)	(4,4)	(2,2)
LU	Execution Time (in sec.)	20.73	52.96	88.66	179.07	347.27	689.18	37.54	79.19	134.05	277.25	505.64
	Ratio to regular mode	1	2.55	4.28	8.64	16.75	33.25	1.81	3.82	6.47	13.37	24.39
EP	Execution Time (in sec.)	1.99	4.06	8.33	16.29	36.42	69.05	2.67	5.38	10.77	21.52	43.09
	Ratio to regular mode	1	2.04	4.18	8.18	18.3	34.7	1.34	2.7	5.41	10.81	21.65

	Acquisition mode	R	F-2.625	F-5.25	F-10.5	F-21	S-2	SF-(2,2.625)	SF-(2,5.25)	SF-(2,10.5)	SF-(2,21)
	Number of nodes	43	16	8	4	2	(22,21)	(8,8)	(4,4)	(2,2)	(1,1)
DT	Execution Time (in sec.)	4.56	12.62	18.67	32.2	58.8	10.66	31.277	41.06	47.49	66.93
	Ratio to regular mode	1	2.76	4.09	7.06	12.89	2.33	6.86	9	10.4	14.67

Table II
EVOLUTION OF THE EXECUTION TIME OF INSTRUMENTED CLASS B INSTANCE OF EP, DT AND LU EXECUTED BY 64 PROCESSES (43 FOR DT) WITH REGARD TO THE ACQUISITION MODE. RESULTS OBTAINED ON THE *bordereau* AND *gdx* CLUSTERS USING ONE CORE PER NODE.

The worst value is obtained for EP Class B with 64 processes instance for which the execution time of the benchmark is the smallest. However, large number of processes are generally used to solve large problem instances. Then we can conclude that the extra overhead required to get a time-independent trace can be afforded. Moreover such traces need to be acquired only once and can then be used to explore much more “what if?” scenarios than with timed traces.

Now we estimate the impact of the *Folding* (F-x), *Scattering* (S-y), and *Scattering and Folding* (SF-(u, v)) acquisition modes on the execution time when compared to the *Regular* mode (R). When processes are folded on a smaller number of nodes, x denotes the folding factor. For instance F-4 means that four processes are executed on a single CPU. In the scattering mode, y is the number of sites used during the acquisition. Finally when both modes are combined, SF-(u, v) means that the execution is scattered over u sites and that v processes run on each node. Table II presents the results of this comparison in terms of execution time and performance degradation. Again, we use only one core per node.

We see that the time needed to execute the instrumented application increases linearly with the folding factor (F-*). This was expected as the codes of several processes have to be executed concurrently on a single CPU. However, there is no extra overhead induced by folding. When the execution is scattered across two clusters (S-2), the overhead comes from two factors. First, some communications are made on a wide area network and then take more time to complete. Second, the progression of the execution is limited by the slowest cluster (*gdx*). While this overhead remains lower than the number of sites, further experiments showed that it increases with the number of sites and is also greater for smaller problem classes. Indeed, a lower amount of computations leads to a greater impact of wide area communications. Finally the combination of process folding and scattering (SF-(2,*)), the overhead are cumulated. If we divide the ratios to regular mode by the value obtained for S-2, we still observe that the execution time increases with the folding factor in a roughly linear way.

For DT we used different folding factors to ensure that the load is well balanced among processes. Compared to the other benchmarks, *folding* has a smaller impact on execution time as this benchmark is dominated by communications, while *Scattering* is more detrimental for the same reason. Large buffer allocations can prevent folding for big DT instances.

An interesting property of time-independent traces is examplified by these experiments. A tracing tool such as TAU will produce traces full of erroneous timestamps for most scenarios. An off-line simulator using such traces will predict an execution time close to that of the corresponding acquisition scenario instead of the targeted *Regular* mode execution time. Preventing such a behavior would require an accurate description of the acquisition platform along with the trace. With time-independent traces, the simulated time is independent of the acquisition scenario. Only slight variations (under 1%) are observed caused by hardware counter accuracy issues.

C. *Analysis of Trace Sizes*

As mentioned in Section II, the main challenge for off-line simulation is the large size of the traces. This size directly depends on the number of actions executed by the processes. For EP, each process only executes a big block of instructions. Then the trace of a given process has only one line. For DT, the number of actions is also relatively small. The processes that execute the most actions are *comparators* that receive data from four other processes, merge them, and forward the aggregated data. Consequently the total trace size for both benchmarks is at most a few hundreds of kilobytes for class C instances. With LU, computations and communications are interleaved and each process executes a lot of actions. Table III presents trace sizes for different instances of the LU benchmark and the correlation with the number of actions.

#Processes	Trace size (in MiB)		#Actions (in millions)	
	Class B	Class C	Class B	Class C
8	29.9	48.4	2.03	3.23
16	72.6	117	4.87	7.75
32	161.3	256.8	10.55	16.79
64	344.9	552.5	22.73	36.17

Table III
SIZES OF TIME-INDEPENDENT TRACES AND NUMBER OF ACTIONS FOR DIFFERENT INSTANCES OF THE LU BENCHMARK.

The size of traces grows linearly with the number of processes which is explained by the evolution of the number of traced events. We also see that the size of the time-independent traces grows from a constant factor of 1.6 from class B to class C which is also directly related to the number of actions. Indeed, the ratio of size to the number of actions, that denotes the average number of characters per action, is roughly constant (from 14.72 to 15.29).

D. Accuracy of Time-Independent Trace Replay

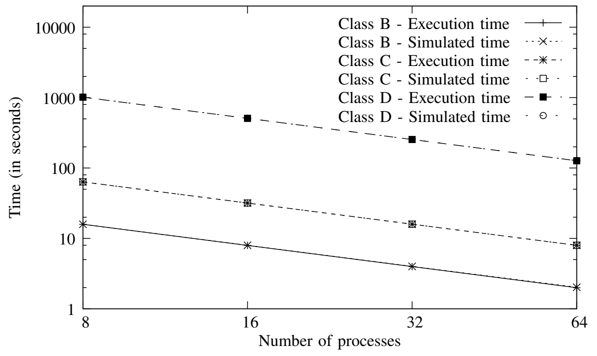

Figure 6. Comparison of simulated and actual execution time for the EP benchmark on the *bordereau* cluster.

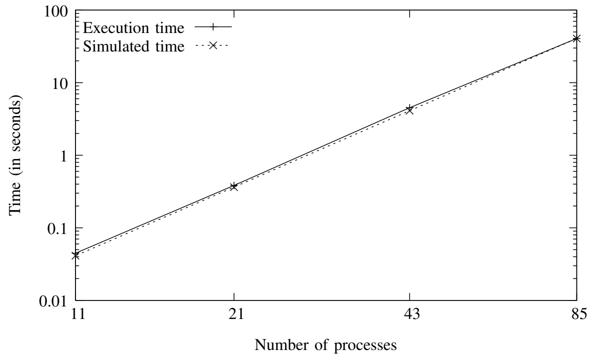

Figure 7. Comparison of simulated and actual execution time for the DT benchmark on the *bordereau* cluster.

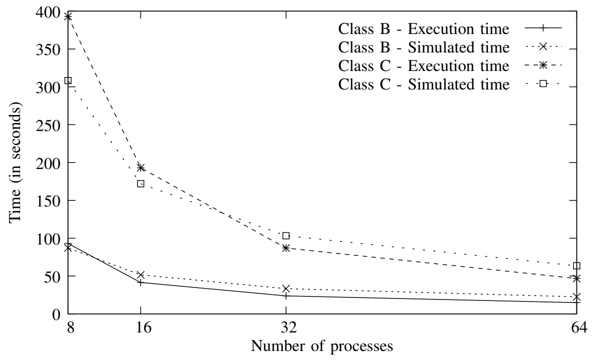

Figure 8. Comparison of simulated and actual execution time for the LU benchmark on the *bordereau* cluster.

Figures 6, 7, and 8 show the accuracy of the time-independent trace replay by comparing the actual execution time respectively of the EP, DT and LU benchmarks on the *bordereau* cluster for various classes with the simulated time obtained with SIMGrid. We see that the simulated execution time is very accurate for the EP and DT benchmarks that are dominated either by computation or communication. For LU, results are more ambivalent. Here the trace replay is able to predict the correct evolution trend, but the local relative error may be quite high (up to 47.1% for Class B on 64 processes) and not constant. This prevent our tool to provide predictions with a fixed interval of confidence.

These difficulties to match the experimental data are clearly related to the calibration of the simulator. More precisely it principally comes from the calibration of the processing rate that is not necessary with other off-line simulators that rely on timed traces. Indeed, the processing rate is not constant over the computation of a LU benchmark. Moreover, this processing rate does not even depend on the size of the computation. Improving the accuracy of the trace replay would imply to acquire more information on each computation during the calibration step to adapt the processing rate accordingly.

E. Acquiring Large Traces

The analysis detailed the previous sections were made on traces corresponding to small instances (up to 64 and 85 processes). Moreover this number of processes in smaller than the number of nodes in the compute clusters used for these experiments (respectively 93 and 186 nodes). To demonstrate that the proposed approach can be used to assess the performance of MPI applications on clusters that are not available, we study the acquisition of traces for large instances of the EP and LU benchmarks. We ran our acquisition process on the *bordereau* cluster for Class D instances executed on 1,024 processes. Such instances are almost three times bigger than the number of cores ($93 \times 2 \times 2 = 372$) that this cluster comprises. To obtain such traces, we only used 32 nodes (128 cores, and a $8\times$ folding), that is only about one third of the total amount of available resources.

For a simple code such as EP, acquiring a large trace is quite fast. Indeed it took less than 86 seconds to get the 4.1MiB time-independent trace. For a more complex code such as LU, with much more actions, the acquisition time takes more time. Indeed around 25 minutes were needed to acquire the time-independent trace whose size is 32.5 GiB. When compressed with `gzip`, the time-independent trace takes 1.2 GiB.

As stated earlier, memory constraints prevent the folding of the DT benchmark for big instances. Then getting a large trace requires a large number of nodes, or a large amount of memory on each node used for the acquisition.

F. Simulation Time

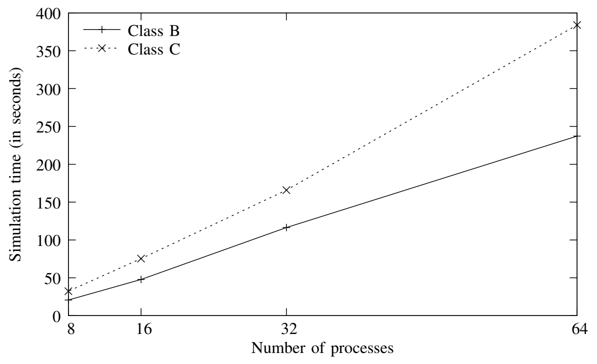

Figure 9. Evolution of the trace replay time with the number of processes executing a LU benchmark.

The time to replay a time-independent trace is directly related to the number of actions. Consequently EP and DT traces for 64 processes are replayed in less than 0.1 seconds. The large number of actions in the LU traces then implies a larger simulation time. Figure 9 presents the evolution of the simulation time as the number of processes increases for classes B and C instances of the LU benchmark. These timings were obtained on one node of the *bordereau* cluster.

We see that the simulation time grows linearly with the number of processes. More interestingly, the simulation time is directly related to the number of actions. Indeed, the average time to proceed with one million of actions is around 10 seconds. This allows us to estimate the simulation time from the size of the time-independent trace.

VII. Conclusion and Future Work

In this paper, we proposed a new approach for the off-line simulation of MPI applications. Instead of relying on logs of execution that associate an event to a time-stamp, we use time-independent traces as an input of our simulator. These traces contain only information about volumes of computation and communications. This allows to decouple the acquisition process from the replay of the trace. Heterogeneous and distributed platforms can then be used to get traces without impacting the quality of the simulation. Our simulator is built on top of the SimGrid toolkit and thus benefits of its fast, scalable, and validated simulation kernel. We also rely on a well established tracing tool, TAU, in the acquisition process. In our experiments we have estimated the overhead required to produce a time-independent trace and showed the trace size reduction traces when compared to TAU. We also discussed the accuracy of the trace replay and he time needed to perform such simulations. The conclusion is that decoupling acquisition and replay is a sound approach and deserves further investigation. As future work, we plan to improve the accuracy and simulation time. We also aim at exploring techniques to reduce the size of the traces, e.g., using a binary format. Finally we plan to compare off-line simulations results with those produced by on-line simulators.

Acknowledgments

This work is partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022) and the CNRS PICS N° 5473. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

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

[2] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *Proc. of the 10th IEEE International Conference on Computer Modeling and Simulation*, Cambridge, UK, Mar. 2008.

[3] P. Dickens, P. Heidelberger, and D. Nicol, "Parallelized Direct Execution Simulation of Message-Passing Parallel Programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1090–1105, 1996.

[4] R. Bagrodia, E. Deelman, and T. Phan, "Parallel Simulation of Large-Scale Parallel Applications," *IJHPCA*, vol. 15, no. 1, pp. 3–12, 2001.

[5] R. Riesen, "A Hybrid MPI Simulator," in *Proc. of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sep. 2006.

[6] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *Proc. of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, Apr. 2004.

[7] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Application Performance Modeling and Prediction," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC'02)*, Baltimore, MA, Nov. 2002.

[8] E. León, R. Riesen, and A. Maccabe, "Instruction-Level Simulation of a Cluster at Scale," in *Proc. of the International Conference for High Performance Computing and Communications*, Portland, OR, Nov. 2009.

[9] S. Prakash, E. Deelman, and R. Bagrodia, "Asynchronous Parallel Simulation of Parallel Programs," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 385–400, 2000.

[10] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler, "MPI-NetSim: A network simulation module for MPI," in *Proc. of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzen, China, Dec. 2009.

[11] P.-N. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson, "Single Node On-Line Simulation of MPI Applications with SMPI," in *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Anchorage, AK, May 2011.

[12] T. Hoefler, C. Siebert, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, Chicago, IL, Jun. 2010, pp. 597–604.

[13] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications," in *Proc. of the 15th International EuroPar Conference*, ser. LNCS, vol. 5704, Delft, Aug. 2009, pp. 135–148.

[14] A. Núñez, J. Fernández, J.-D. Garcia, F. Garcia, and J. Carretero, "New Techniques for Simulating High Performance MPI Applications on Large Storage Networks," *Journal of Supercomputing*, vol. 51, no. 1, pp. 40–57, 2010.

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

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

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

[18] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. Kalé, "Simulation-Based Performance Prediction for Large Parallel Machines," *Int. Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 183–207, 2005.

[19] S. Shende and A. Malony, "The Tau Parallel Performance System," *IJHPCA*, vol. 20, no. 2, pp. 287–311, 2006.

[20] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *IJHPCA*, vol. 14, no. 3, pp. 189–204, 2000.

[21] Performance Research Lab, *TAU User Guide, chapter Tracing, TAU Trace Format Reader Library*, University of Oregon, http://www.cs.uoregon.edu/research/tau/docs/newguide/ch06s02.html.

[22] P. Velho and A. Legrand, "Accuracy Study and Improvement of Network Simulation in the SimGrid Framework," in *Proc. of the 2nd International Conference on Simulation Tools and Technique (SIMUTools)*, Roma, Italy, Mar. 2009.

[23] M. Geimer, F. Wolf, B. Wylie, and B. Mohr, "A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, 2009.

[24] R. Reussner, P. Sanders, and J. L. Träff, "SKaMPI: a Comprehensive Benchmark for Public Benchmarking of MPI," *Scientific Programming*, vol. 10, no. 1, pp. 55–65, 2002.