Simulation of MPI applications with time-independent traces

Henri Casanova, Frédéric Desprez, George S. Markomanolis, Frédéric Suter

Simulation of MPI Applications with Time-Independent Traces

Henri Casanova
Dept. of Information and Computer Sciences, University of Hawai'i at Manoa, U.S.A.

Frédéric Desprez
INRIA, LIP, ENS Lyon, Lyon, France

George S. Markomanolis
INRIA, LIP, ENS Lyon, Lyon France

Frédéric Suter
INRIA, LIP, ENS Lyon, Lyon
IN2P3 Computing Center, CNRS, IN2P3, Lyon-Villeurbanne, France

November 24, 2015

Abstract

Analyzing and understanding the performance behavior of parallel applications on parallel computing platforms is a long-standing concern in the High Performance Computing community. When the targeted platforms are not available, simulation is a reasonable approach to obtain objective performance indicators and explore various hypothetical scenarios. In the context of applications implemented with the Message Passing Interface, two simulation methods have been proposed, on-line simulation and off-line simulation, both with their own drawbacks and advantages. In this work we present an off-line simulation framework, i.e., one that simulates the execution of an application based on event traces obtained from an actual execution. The main novelty of this work, when compared to previously proposed off-line simulators, is that traces that drive the simulation can be acquired on large, distributed, heterogeneous, and non-dedicated platforms. As a result the scalability of trace acquisition is increased, which is achieved by enforcing that traces contain no time-related information. Moreover, our framework is based on an state-of-the-art scalable, fast, and validated simulation kernel.

We introduce the notion of performing off-line simulation from time-independent traces, propose and evaluate several trace acquisition strategies, describe our simulation framework, and assess its quality in terms of trace acquisition scalability, simulation accuracy, and simulation time.

1 Introduction

Understanding the performance behavior of parallel applications implemented with the Message Passing Interface (MPI) on various compute infrastructures is a long-standing concern in the High Performance Computing community. Tools exist that enable high-resolution profiling of MPI applications and automated or semi-automated methods for performance bottleneck detection [1, 2, 3]. While understanding the performance behavior of an MPI application on an existing parallel platform is useful, in many cases one wishes to consider platforms that are not necessarily available. In this case, one must *simulate* application execution. When a platform is yet to be specified and purchased simulation can be used to determine a cost-effective hardware configuration appropriate for the expected application workload. Conversely, simulation can be used to study the performance behavior of an application by varying the hardware characteristics of a hypothetical platform. Simulating application execution may be useful even when the target platform is available. For instance, in simulation it is possible to bypass actual computations performed by the application and only simulate the corresponding delays of these computations. In this case the simulated application produces erroneous results, but its performance behavior may be preserved. It is then possible to conduct development activities for performance tuning in simulation only on a small-scale platform, thereby saving time when compared

to real executions on a large-scale platform. Finally, access to large-scale platforms is typically costly (possible access charges to the user, electrical power consumption). Simulation can thus not only save time but also money and resources.

Many frameworks for simulating the execution of MPI applications on parallel platforms have been proposed over the last decade. They fall into two categories: *on-line simulation* or "simulation via direct execution" [4, 5, 6, 7, 8, 9, 10], and *off-line simulation* or "post-mortem simulation" [11, 12, 13, 14, 15]. In on-line simulation the application code is executed but part of the execution takes place within a simulation component. Although it is powerful, the main drawback of on-line simulation is that the amount of hardware resources required to simulate application execution at a given scale is commensurate to that needed to run the application on a real-world platform at that scale. Even though a few solutions have been proposed to mitigate this drawback and allow execution on a single computer [16, 17], the simulation time can be prohibitively high when applications are not regular or have data-dependent behavior, thus leading to non-deterministic executions. In this work, instead, we focus on off-line simulation, by which a trace of a previous execution of the application is "replayed" on a simulated platform. The main advantage when compared to on-line simulation is that replaying the execution can be performed on a single computer. This is because the replay does not entail executing any application code, but merely simulating computation and communication delays. Computation delays are scaled to account to the performance differential between the compute nodes in the platform on which the trace was acquired and the simulated platform. Communication delays are typically computed based on communication volumes using a network simulator.

To the best of our knowledge, all off-line simulators of MPI applications rely on time-related information: some events in the traces (e.g., computations, communications) are associated with dates and/or durations. As a result, to simulate the execution of an application on a platform with a given number of CPUs, the trace must be acquired on a platform that is homogeneous and with the same number of CPUs. This is so that the simulated delays are coherent. In particular, homogeneity allows uniform scaling of compute delays when simulating a target platform with slower/faster processors. The same number of CPUs is required to avoid CPU and memory hierarchy sharing effects when "doubling up" MPI processes on the compute nodes of a smaller-scale platform. We conclude that trace acquisition is not easily scalable since a homogeneous platform may not be available at the required scale.

We propose an off-line simulation framework for MPI applications, the Time-Independent Trace Replay Framework, that solves the above trace acquisition scalability problem. This is achieved by eliminating all time-related information from application event traces, using what we call "time-independent traces". The only constraint is that all processors in the acquisition platform and on the target simulation platform must belong to the same processor family, e.g., i386, amd64. Removing all notions of time from event means that the trace-driven simulation cannot simply scale simulated delays but must instead rely on precise simulation models that determine delays solely based on computation and communication volumes. To address this challenge this work leverages the powerful simulation abstractions and models provided by the SIMGrid project [18]. In particular, this means that our work benefits from an accurate (i.e., validated) and yet scalable network simulation model that accounts for topology and network contention effects. By contrast, most state-of-the-art MPI off-line simulators employ either simplistic or unscalable network models. The main contributions of this work are:

1. A novel approach that relies on time-independent event traces to completely decouple trace acquisition from trace replay.
2. Several trace acquisition scenarios that allow for the acquisition of event traces for application executions at scales larger than that of the trace acquisition platform.
3. A trace replay tool on top of a fast, scalable, and validated simulation kernel;
4. An experimental evaluation of our off-line simulation framework, encompassing both trace acquisition (for overhead, accuracy, and scalability) and trace replay (for accuracy and speed).

This paper is organized as follows. Section 2 reviews related work and highlights the limitations of the off-line approach that are addressed by this work. Section 3 presents the format of a time-independent MPI application trace. Section 4 details the acquisition of such traces while Section 5 explains how time-independent traces are replayed in simulation using SIMGrid. These two sections also include quantitative evaluations of our framework. Finally, Section 6 concludes with a summary of results and perspectives on future research directions.

2 Related Work

One option for simulating the execution of an MPI application is *on-line simulation*, by which the actual application code is executed on a *host platform* that attempts to mimic the behavior of a *target platform* with different hardware characteristics. Part of the instruction stream is intercepted and passed to a simulator. LAPSE is a well-known on-line simulator developed in the early 90's [4] (see therein for references to precursor projects), which allows an application to execute normally but with communication delays generated according to a simple network model. A similar approach has been used in subsequent on-line MPI simulators [5, 8]. BigSim [7] goes further by allowing the simulation of computational delays. One problem with this approach is that the computational application code is not executed so that data-dependent application behavior is lost, which may not be acceptable for simulating irregular parallel application, such as branch-and-bound or sparse matrix computations, which lead to non-deterministic executions. Another problem is that an on-line simulation is inherently distributed, because done through a direct execution of the MPI application. Parallel discrete event simulation raises difficult correctness and synchronization issues that require sophisticated simulation protocols [19, 7]. One way to side-step these challenges is to run the simulation on a single processor, which mandates techniques to reduce the simulation's CPU time requirements [7] and memory footprint requirements [16]. Such techniques are implemented in the recently proposed SMPI on-line simulator [17, 20], which builds on the same simulation kernel as this work.

The alternate approach is *off-line* simulation in which a log, or trace, of MPI communication events (begin and end time-stamps, 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*. The application event trace is decomposed in time intervals delimited by the MPI communication operations. The application execution is thus seen as a succession of computation bursts and communication operations. An off-line simulator then simply computes simulated delays in a way that accounts for the performance differential between the platform used to obtain the trace and the platform to be simulated. Unlike an on-line simulation, an off-line simulation can easily be executed in a centralized manner (i.e., using a single process), and thus does not require the use of complex distributed parallel simulation techniques. Furthermore, off-line simulation partially addresses the loss of data-dependent behavior caused by on-line simulation. Some data-dependent behavior can be captured during the acquisition of the trace. However, this behavior is limited to one particular set of input parameters. Overall, the simulation of irregular parallel applications, that lead to non-deterministic executions, whether on-line or off-line, remains a challenging problem, which we do not address in this work.

Many simulators have been proposed that, like the simulator described in this work, follow the above off-line simulation approach. For instance, since 2009 five off-line simulators of MPI applications have been described in the literature [11, 12, 13, 14, 15]. These simulators differ by the simulation models they employ to compute simulated durations of CPU bursts and communication operations. Computation delays are typically computed by scaling the durations of the CPU bursts in the trace [12, 13, 15]. In this work we follow the same general off-line simulation approach as these simulators and use scaling of the durations of the CPU bursts in the trace. Note that in [14] the authors actually execute CPU bursts on a processor of the target platform to measure their durations. Communication operation delays are typically computed using simplistic network models, because such models are straightforward to implement and scalable. Possible simplifications include: not using a network model but simply replay original communication delays [15]; ignoring network contention because it is known to be difficult and costly to simulate [11, 14, 21]; using monolithic performance models of collective MPI communications rather than simulating them as sets of point-to-point communications [12, 15, 22]. Other simulators opt for accurate packet-level simulation, which is not scalable and leads to high simulation times [13]. In this work we address this limitation by leveraging the validated and scalable network flow-level network models in the SIMGrid framework.

The main objective of this work is to improve the scalability of trace acquisition in state-of-the-art off-line simulation of MPI applications. As explained in Section 1, to simulate the execution of an application on a platform of a given scale, i.e., a given number of CPUs, the trace must be acquired on a homogeneous platform of that same scale, so that time-stamps, or rather differences betwen the end and begin time-stamps of computation or communication events, are meaningful. In some cases, extrapolating a smaller trace to larger numbers of compute nodes is feasible [11], but not generally applicable. One is thus limited to the scale of the largest homogeneous platform available, which may not be sufficient, especially since simulation is typically used to explore scenarios beyond the physical platform configurations available. Conceivably, it may be possible to acquire a trace with time-related information on

a heterogeneous platform and replay such a trace with standard off-line simulation techniques. However, in this case the trace would need to be accompanied with a precise description of the platform on which the trace was obtained, the mapping of application processes to the resources, and benchmarks that give relative computation speeds for all processor types, so as to compute meaningful simulated delays. To the best of our knowledge no previously proposed off-line simulator allows for this approach. In this work, instead, we completely forego all time-related information so that a trace is independent from the platform on which it was collected, while allowing for the use of heterogeneous, distributed, and non-dedicated trace acquisition platforms.

Besides the lack of availability of a large homogeneous platform, another scalability limitation for off-line simulation is the large size of the traces, which limits both trace acquisition and trace replay. One solution to improve the scalability of trace replay is to replay only judiciously selected subsets of the traces [14]. A more general approach, which improves the scalability of both trace acquisition and trace replay, is to design compact/compressed trace representations [12, 23, 24]. ScalaTrace [23, 24] is particularly related to this work because the traces collected/compressed are less dependent on time. It relies on the identification and recording of repetitive communication patterns to compact the produced traces. Moreover, advanced aggregation techniques, e.g., histograms, are implemented that allow for trace replay with only a limited loss of accuracy with a large reduction in trace size. ScalaTrace is thus well-suited for studying the performance of application that exhibit regular pattern-based communication schemes. It might also make it possible to extrapolate the trace obtained for a given application instance in a view to simulating larger instances. The techniques proposed in [23] for compressing traces and for processing compressed traces could be implemented in our framework. We leave such implementation for future work and instead report on some results obtained when using standard compression techniques (i.e., gzip).

3 Time-Independent Event Traces

As explained earlier the event traces used by existing off-line simulators to simulate large-scale application executions must be acquired on large-scale homogeneous platforms, which may not be readily available. Our approach to address this issue is to remove all time-related information from the traces. For each event occurring during the execution of the traced application, e.g., a CPU burst or a communication operation, we log its volume (in number of executed instructions or in number of transferred bytes) instead of the time when it begins and ends or its duration. The main advantage is that our logged information does not depend on the hardware characteristics of the platform on which the trace is collected, with the exception of the processor family. The size of the messages sent by an application is not likely to change according to the specifics of the network interconnect. The number of instructions performed within a basic block does not increase with the processing speed of the CPU, provided all processors belong to the same family (i.e., instruction counts are comparable across processors). These two claims do not hold for adaptive MPI applications that modify their execution path according to the execution platform. Such applications, which represent only a small fraction of production MPI applications, are outside the scope of this work.

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

```
0 compute 1e6
0 send 1 1e6
0 recv 3 1e6
1 recv 0  1e6
1 compute 1e6
1 send 2 1e6
2 recv 1 1e6
2 compute 1e6
2 send 3 1e6
3 recv 2 1e6
3 compute 1e6
3 send 0 1e6
```

Figure 1: Simple MPI application (left) and corresponding time-independent trace (right).

A time-independent trace is a list of *actions* performed by each process of an MPI application. An action is described by the *rank* of the process that performs it, a *type* (compute, send, or receive), a *volume* (number of instructions

or number of bytes), and some action-specific parameters (e.g., the rank of the receiving process in a communication).

The left-hand side of Figure 1 shows a simple MPI application executed on a logical ring with four processes. Each process computes one million instructions and sends one million bytes to its neighbor. The right-hand side of the figure displays the corresponding time-independent trace. For large numbers of processes and/or numbers of actions, it may be preferable to split the trace so as to obtain one trace file per process.

Table 1 lists the MPI functions that can be replayed by our Time-Independent Trace Replay Framework in its current implementation. As explained in Section 5, our framework replays traces using an MPI application simulator provided as part of SIMGrid. This simulator allows us to simulate the MPI collective communication operations in Table 1 in a way that corresponds to popular MPI implementations (MPICH2 [25] and OpenMPI [26]). The reader will note that MPI communicators and communicator operations are not listed in the table. In this paper all simulated applications only use the default MPI_COMM_WORLD communicator, and thus do not require any specific communicator support. However, it would be straightforward to add tracing support for communicators (i.e., logging in the trace for each MPI operation the corresponding communicator, as well as logging those MPI functions that operate on communicators). Furthermore, as explained in Section 5.1, we reuse an (on-line) MPI simulation framework, SMPI [17, 20], which implements the entire MPI standard and thus includes support for communicators. Adding support for communicators is thus a likely and straightforward short-term development objective for this work. All experiments in this paper are conducted using OpenMPI version 1.6.

Table 1: Time-independent actions corresponding to supported MPI communication operations.

MPI actions	Trace entry
CPU burst	`<rank> compute <volume>`
MPI_Send	`<rank> send <dst_rank> <volume>`
MPI_Isend	`<rank> Isend <dst_rank> <volume>`
MPI_Recv	`<rank> recv <src_rank> <volume`
MPI_Irecv	`<rank> Irecv <src_rank> <volume`
MPI_Broadcast	`<rank> bcast <volume>`
MPI_Reduce	`<rank> reduce <vcomm> <vcomp>`
MPI_Reduce_scatter	`<rank> reduceScatter <recv_counts> <vcomp>`
MPI_Allreduce	`<rank> allReduce <vcomm> <vcomp>`
MPI_Alltoall	`<rank> allToAll <send_volume> <recv_volume>`
MPI_Alltoallv	`<rank> allToAllv <send_volume> <send_counts> <recv_volume> <recv_counts>`
MPI_Gather	`<rank> gather <send_volume> <recv_volume>`
MPI_Allgatherv	`<rank> allGatherV <send_count> <recv_counts>`
MPI_Barrier	`<rank> barrier`
MPI_Wait	`<rank> wait`
MPI_Waitall	`<rank> waitAll`

4 Trace Acquisition Procedure

In this section we detail the acquisition of a time-independent event trace. The acquisition procedure, depicted in Figure 2, comprises three steps: (i) the instrumentation of the target application; (ii) the execution of this instrumented version; and (iii) the gathering of the different traces to a single location.

4.1 Instrumentation

The first step of the acquisition procedure is to instrument the MPI application to be simulated, so that an execution of the instrumented application generates application event traces that can be used to replay the application in simulation.

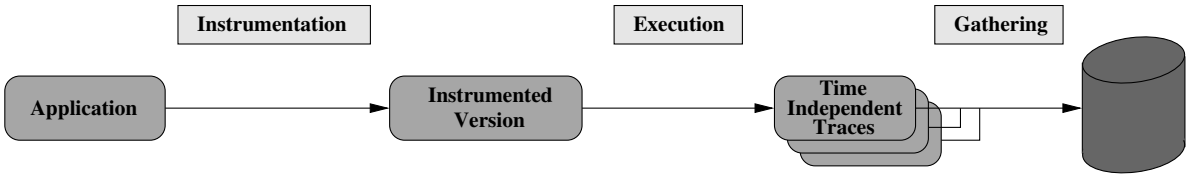


Figure 2: Time-Independent Traces acquisition procedure.

The only information required by our framework are: (i) the volume of computation in between two MPI calls at each MPI process, measured in number of instructions; (ii) the name of and parameter values passed to each MPI call; (iii) the volume of data transferred by each communication operation; and (iv) the exact sequence of computation bursts and communication operations executed by each process. For instance, here is a short example trace that describes the relevant application events for the process of rank 0:

```
0 compute 956140
0 send 1 1240
0 compute 2110
0 send 2 1240
0 compute 3821
```

Several tools are available that instrument MPI applications for various purposes including debugging, profiling, performance debugging, post-mortem analysis and visualization [1, 2, 3, 27, 28, 29, 30]. The most popular such tool is arguably TAU [1], which generates application event traces and reports on hardware performance counters through the PAPI [27] interface, and can thus be used for our purpose. In fact, like most of the aforementioned tools, TAU provides capabilities well beyond our needs in this work. Two undesirable side effects of instrumentation are *overhead*, i.e., the execution time increase due to the execution of the instrumentation code, and *skew*, i.e., the discrepancy (increase) in instruction count between a full run of the instrumented application and a full run of the uninstrumented application due to the instrumentation code. One may expect skew to be zero, since after all the instrumentation code should not count its own instructions. However, results presented hereafter show that skew is not only present but can be large, which is due to the resolution of the hardware counters. Generally, instrumentation that generates trace data beyond the information strictly needed for our replay framework, and thus unnecessary for our purpose, could then unnecessarily increase overhead and skew. It turns out that TAU enables *selective instrumentation*, by which parts of the application's source code can be ignored for the purpose of instrumentation. This feature can be used in an attempt to avoid unnecessary instrumentation, namely enabling only the tracing of MPI calls and the gathering of numbers of executed instructions in between these calls.

In early versions of our framework [31, 32] we used TAU since it is likely the most popular profiling and tracing tool for MPI applications. However, due to non-negligible overhead and skew, here we also present results with a lightweight instrumentation method. The MPI standard exposes two interfaces for each MPI function, one prefixed with `MPI_` and the other prefixed with `PMPI_`, the former calling the latter directly. This provides developers with the opportunity to insert their own code, e.g., for profiling purposes, in the implementation of all `MPI_` functions. This mechanism is used by several of the aforementioned tools, and we ourselves use it to insert code that is executed upon entry and exit for all MPI calls. This code retrieves hardware counters through PAPI, and generates event traces like the example trace above. This approach is guaranteed to perform the minimal amount of instrumentation needed for our purpose. It is thus expected that it should lead to lower overhead and skew than TAU.

To measure overhead and skew we perform experiments using one of the NAS Parallel Benchmarks (NPB) [33]. The NPB are a suite 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 example, a class D instance corresponds to approximately 20 times as much work and a data set almost 16 times as large as a class C problem.

We execute four different versions of the LU factorization benchmark from the NPB suite, all compiled with the highest level of compiler optimization. The first version is the original benchmark augmented with two calls to PAPI inserted at the beginning and the end of the LU computation to measure the total number of executed instructions.

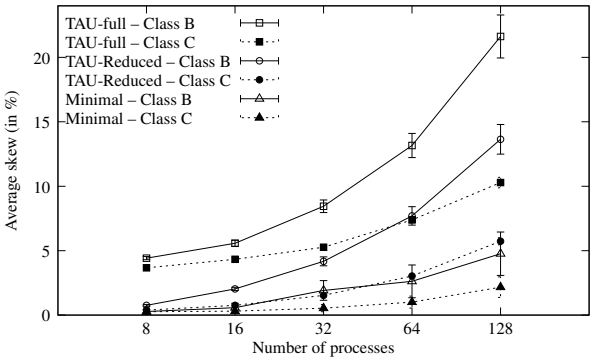


Figure 3: Instrumentation skew for the three instrumented LU benchmarks.

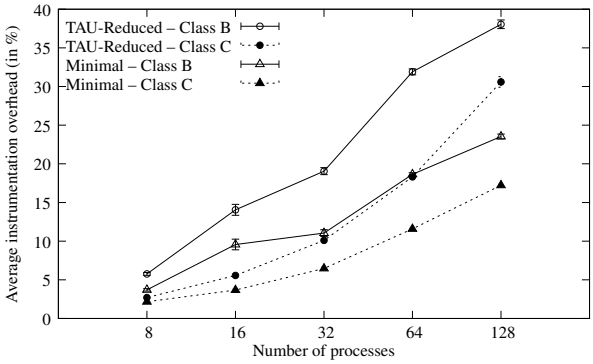


Figure 4: Instrumentation overhead for the TAU-Reduced and Minimal instrumented LU benchmarks.

Since the overhead and skew due to these two calls are negligible, we call this version "original". The second version is called "TAU-full" and corresponds to the benchmark instrumented using TAU with default configuration settings. The third version is called "TAU-reduced" and corresponds to the benchmark instrumented using TAU but enabling instrumentation exclusion to reduce overhead and skew. The fourth version is called "Minimal" and corresponds to the benchmark instrumented using our own lightweight method. We execute all versions on the same cluster, called *graphene*, that comprises 144 2.53GHz Quad-Core Intel Xeon x3440 nodes. Each core has a L2 cache of 2 MB. The nodes are spread across four cabinets, and interconnected by a hierarchy of 10 Gigabit Ethernet switches.

We compute the instrumentation overhead as the percentage difference in execution time between an instrumented version and the original version. The execution time for the instrumented version includes the time to access the hardware counters, to log the events, and to write this information to disk. For each MPI process, we compute the total number of instructions for the original application and an instrumented applications. We sum these numbers over all processes, and compute the instrumentation skew as the percentage difference in total number of instructions.

One design decision is how and when the trace data should be written to disk. One option is to buffer the whole event log in memory and then write it to disk in bulk at the end of the execution. However, this would preclude a large folding factor (see Section 4.2) when running multiple application processes on a CPU. In fact, popular profiling tools (such as TAU, Scalasca, or Score-P) all buffer and flush trace data to disk periodically, so as to overlap disk writes with computation. We conducted experiments to find empirically an appropriate buffer size that minimizes instrumentation overhead. Note that buffering/flushing may increase instrumentation skew because it impacts the content of hardware counters. We address this issue in our Minimal instrumentation method, accessing the hardware counters before and after each disk write so as to discount the involved instructions. A similar technique is likely implemented in TAU as well.

Figure 3 shows the skew for class B and C instances of the LU benchmark. Each data point is obtained as an average over ten executions and aggregates the skews measured on all the processes. We see that the skew can be non-zero and in fact quite large. In particular, the TAU-full instrumentation method leads to the highest skew, with values ranging from 3.66% to 21.62%. As the number of instructions is a fundamental component in our replay framework, the higher the skew, the less accurate the simulations will be. In other words, our framework will simulate the instrumented application rather than the original application. The TAU-reduced and Minimal instrumentation methods greatly reduce the skew (on average by a factor of 3.69 and 8.97, respectively). The lowest skew is achieved by the Minimal instrumentation method. It is always under 5% and in most of the cases under 2%. The highest skew caused by the Minimal instrumentation method (4.76%) is obtained for the B-128 instance. In this particular case a relatively small input data is distributed among 128 processes, so that each process holds only less than a hundred kilobytes of data. As a result each process performs a small volume of computation, meaning that the instrumentation instructions represent a non-negligible fraction of the executed instructions and contribute to a higher execution count.

Figure 4 shows the overhead in terms of execution time induced by the TAU-reduced and Minimal instrumentation methods for various instances of the LU benchmark, once again averaged over ten samples. The TAU-full instrumentation method is not displayed here, as its high skew removes it from consideration. Like for the skew results we see

Table 2: Execution time and slowdown relative to the regular acquisition mode for instrumented runs of the LU NAS benchmark (class B) executed with 64 processes, for several acquisition modes. Results obtained on the *graphene* and *paradent* clusters.

	Acquisition mode	R	F-2	F-4	F-8	F-16	F-32	C-2	CF-(2,2)	CF-(2,4)	CF-(2,8)	CF-(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.)	11.52	24.45	43.89	78.67	148.95	288.54	23.8	39.97	72.14	133.31	220.18
	Ratio to regular mode	1	2.12	3.80	6.82	12.92	25.03	2.09	3.47	6.26	11.57	19.17

that the Minimal method we implemented reduces the overhead, on average by a factor of 1.6 when compared to the TAU-reduced method. This is expected as TAU logs more information, which may be useful in other contexts but is unnecessary for our purpose. Using the Minimal method the results indicate that the overhead can become large (up to 23.5% for the B-128 instance). But in general the overhead remains low. It ranges from 1.55 to 2.01 seconds for class B instances and from 0.8 to 2.6 seconds for class C instances. As the original execution time of the application greatly decreases as more processes are used, the relative overhead increases. Since large number of processes are generally used to solve only large problem instances, unlike the class B LU benchmark, we conclude that the instrumentation overhead is well within acceptable limits provided the Minimal instrumentation method is used.

4.2 Execution

There are only three requirements for obtaining a valid time-independent trace in a view to simulating an MPI application with n processes on an arbitrary platform: (i) the instrumented application must be executed with n processes; and (ii) each process must fit in main memory; (iii) all processors must belong to the same processor family. This is in sharp contrast with time-dependent traces as used in previously proposed off-line MPI simulators, which must be obtained on homogeneous platforms with as many compute nodes as that in the simulated platform. In particular, our method makes it possible to execute the instrumented application in four ways:

Regular mode: execution on a single homogeneous cluster with one MPI process per CPU. This is the way in which one obtains traces for use with previously proposed off-line MPI simulators. As discussed earlier, this mode requires as many CPUs as that in the platform to be simulated, which limits its scalability. All three modes hereafter are only applicable to time-independent traces.

Folded mode: execution on a single homogeneous cluster with more than one MPI process per CPU. This allows for the acquisition of traces for instances of the application that are larger than what can be executed in regular mode on the same cluster. The folding factor is only limited by the available amount of memory on the CPUs.

Composite mode: execution on heterogeneous or multiple non-identical clusters. In this mode, the user can aggregate disparate CPUs together, such as those in multiple homogeneous clusters, so as to augment the scale of the trace without requiring a single (large) homogeneous cluster. The only constraint is to select CPUs of a same family so that the measured instruction counts are coherent.

Composite and folded mode: a combination of folded and composite mode. This mode further increases the scalability of trace acquisition by executing multiple MPI processes per CPU in a non-homogeneous platform.

Table 2 shows the execution time of the LU benchmark (class B) for the *Regular* (R), *Folded* (F-x), *Composite* (C-y), and *Composite and Folded* (CF-(y, x)) acquisition modes. The last row of the table shows the slowdown relative to the regular mode. 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 composite mode, y is the number of sites, i.e., of geographically distributed data-centers that host one or more clusters, used for the acquisition. Finally when both modes are combined, CF-(y, x) means that the execution is distributed over y sites and that x processes run on each CPU. The execution times are obtained on the *graphene* cluster described in the previous section and, when composite execution takes place, also on the *paradent* cluster that comprises 64 2.5GHz Dual Quad-Core Intel Xeon L5420 nodes. The nodes of these two clusters have very similar peak performances, around 8Gflop/s per core, as measured with the HPLinpack benchmark, and we use one core per node in this experiment.

We see that the time needed to execute the instrumented application increases with the folding factor (F-*), which is expected as several processes compete for a single CPU. However, we see that the execution time is increased by a factor smaller than the folding factor. A composite execution on two sites (C-2) requires roughly twice as much

time as that in the regular execution. It must be noted that these two clusters are separated by more than 400 miles and the LU factorization is a highly communicating application. The increase in execution time is thus not surprising and commensurate to the number of sites. Finally we can see that combining the composite and folded acquisition modes (CF-(2,*)) does not lead to a simple multiplication of their respective overheads. We conclude that folded, composite, or a combination of these two modes can be used to obtain traces with reasonable increases in trace acquisition times. While the experiments in this section are only for moderate scales (64 processes), in Section 4.5 we present trace acquisition results for scales up to 16K processes.

A key property of time-independent traces is highlighted by these experiments. A tracing tool such as TAU will produce traces with some erroneous timestamps for most scenarios, due to external load on the system or transient operating systems behaviors. An off-line simulator using these traces would then predict an execution time close to that of the corresponding acquisition scenario instead of the targeted *regular* mode execution time. With time-independent traces, the simulated time is totally independent of the acquisition scenario, and a dedicated acquisition platform is not even required. In our experiments, we observe variations of at most 1% between different acquisitions, caused by hardware counter inaccuracies.

4.3 Gathering

As mentioned in Section 2, one issue with off-line simulation is the large size of the traces. This size directly depends on the number of actions executed by the processes.Table 3 shows trace sizes for different instances of the LU benchmark as well as the number of actions in each trace.

Table 3: Trace sizes and number of actions for different instances of the LU benchmark.

#Processes	Trace size (in MiB)		#Actions (in millions)	
	class B	class C	class B	class C
8	25	48	1.73	3.24
16	72	117	4.89	7.78
32	159	255	10.60	16.87
64	339	550	22.83	36.33
128	711	1,200	47.3	75.25

As expected, trace size grows roughly linearly with the number of processes since in this benchmark each process performs roughly the same amount of computation regardless of the number of processes (i.e., the problem size is scaled with the number of processes). We also see that trace size grows by a constant factor of 1.6 when going from class B to class C, which is also directly related to the number of actions performed. These observations are confirmed by the numbers of actions shown in the last columns of the table. Dividing the trace size by the number of actions leads to a roughly constant ratio (between 14.45 and 15.94), which represents the average number of characters per action in the trace file.

One challenge for trace acquisition is the time needed to aggregate (potentially large) trace files generated at a large number of compute nodes. This corresponds to a standard "gather" collective communication operation, and it is known that using a K-nomial reduction tree allows for efficient gathering of the files in $\log_{(K+1)} N$ steps, where N is the number of files and K is the arity of the tree. We developed a simple script to perform this operation, which can be made efficient by picking an appropriate K value given to the number of trace files and the number of compute nodes.

Our trace format, as seen in the example trace in Section 3, is not optimized for size. As discussed in Section 2, other authors have proposed trace compression techniques [12, 23, 24] that could be applied to our framework case. Instead, for some of the results presented in the next section, we simply compress the traces after they are acquired using gzip, noting that custom compression techniques would lead to even better performance and scalability.

4.4 Analysis of the Acquisition Time

In this section we investigate the distribution of elapsed times for each step of our trace acquisition procedure, i.e., execution, instrumentation, and gathering. Figure 5 shows this distribution for the acquisition of time-independent traces for the most demanding NPB benchmark among those we used in our experiments, i.e., LU, for classes B and C and for different number of processes. These results were obtained on the *graphene* cluster in the *regular* acquisition mode, meaning that there was exactly one MPI process on each compute node. We performed trace acquisition 10 times and we show the average value for each step.

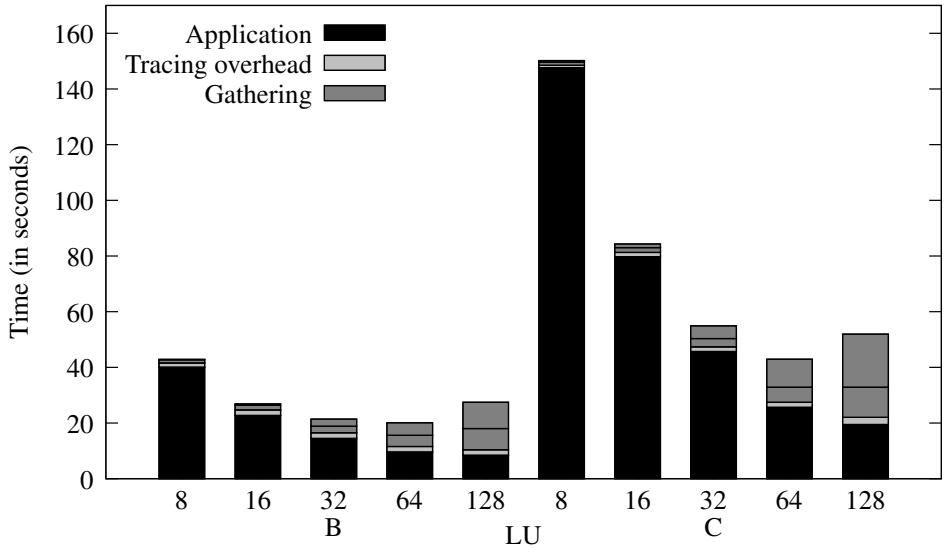


Figure 5: Distribution of the acquisition time for different instances of the LU benchmark in the *regular* acquisition mode.

An expected result in Figure 5 is that when more processes are used the acquisition overhead increases while the time to execute the application decreases. The time to gather traces on a single node becomes more and more prevalent, and even dominates the total trace acquisition time as more processes are involved, accounting for up to 62.02% of the acquisition time. This is expected since the size of each independent trace grows as more actions are logged and the depth of the reduction tree also increases. Data compression and/or a more compact trace format would help to reduce that trace gathering time. For instance, the horizontal line displayed in the gathering portions of Figure 5 indicates the time needed to gather the same traces but with a preliminary gzip compression step. For large numbers of processes, the trace gathering time can be reduced by up to a factor 3. Nevertheless, for realistic instances, i.e., for instances in which a small amount of work is not distributed over a large number of processes, the most time-consuming component of the trace acquisition procedure is the unavoidable application execution. The tracing overhead represents between 1.75% and 10.55% of the trace acquisition time. The higher values, as already explained for the results in Figure 4, are for cases in which each process performs little computation due to the distribution of a small dataset across many processes.

4.5 Scalability of Trace Acquisition

To evaluate the scalability of our trace acquisition procedure, and also to demonstrate how our trace acquisition framework can be used to acquire traces for simulating platforms that are not actually available, we perform larger-scale experiments. In the first experiment, we use folded execution (see Section 4.2) to execute the instrumented LU benchmark on a single node of a cluster called *StRemi*. This node comprises two twelve-core processors that share 48 GiB of memory. This experiment is intended to show that, thanks to folding, we can obtain traces for the largest possible number of processes for classes B and C and a large class D instance of the benchmark on that single node. Table 4

shows the obtained results in terms of time needed to acquire the trace and memory footprint in GiB. For comparison purposes, we also include results when obtaining traces with the TAU [1] (v2.23 with PDTool v3.2), Scalasca [2] (v1.4.2), and Score-P [3] (v1.2.2 with OTF v1.3) profiling tools. All these tools were used with reduced instrumentation enabled. Note that the latest available version of Scalasca at the time of writing is v2.0. As this release relies on Score-P for instrumentation, we did not include it and used a version of Scalasca that implements its own instrumentation method. While TAU, Scalasca, and Score-P have proved their capacity to profile and trace parallel applications executed with large numbers of processes, i.e., from hundreds of thousands and up to a million of processes, extreme process folding is a particularly unfavorable setting for these tools. The results presented hereafter thus do not cast doubts on these tools' scalability in the scenarios for which they have been designed, but instead motivates the use of our lightweight instrumentation method that can scale even when processes are folded on a single CPU.

We find that TAU cannot produce traces (i.e., it fails) for the larger benchmark instance (C-1024) due to a memory footprint exceeding the capacity of a single node. The results for our Minimal instrumentation are in line with those achieved by Scalasca, with some improvement over Scalasca in all cases. However, Scalasca does not include all information for all MPI events, and to use its traces in our framework it would be necessary to perform expensive trace post-processing. The Scalasca results in the table are thus optimistic. The memory footprint of Score-P is two to three times larger than that of our Minimal instrumentation. It also requires more time to acquire the traces. We conclude that although profiling tools exist, and in fact provide capabilities well beyond what is needed in this work, our Minimal instrumentation method is useful to achieve better scalability, especially when faced with extreme process folding, given our overall objective.

Table 4: Executing the instrumented LU benchmark using folding on a single node.

	Time in minutes / Memory footprint in GiB			
Instance	TAU Reduced	Scalasca	Score-P	Minimal Instrumentation
B - 256	2.58 / 11	2.1 / 2.8	1.75 / 4	1.9 / 1.65
C - 1024	N/A	16.3 / 12.9	26.2 / 31	12.9 / 7.95
D - 256	81.8 / 40	55.2 / 16.9	72.16 / 32	47.4 / 15.4

In a second experiment we executed larger application instances, instrumented with our Minimal instrumentation method, of the LU benchmark on 40 nodes of the *StRemi* cluster. Figure 6 shows the acquisition time and the memory footprint for the largest instances of classes C, D and E that can be executed. The applied folding factor grows from 25.6 for the smallest instance to 409.6 for the largest one. These results illustrate the scalability of our acquisition framework when the folded mode is used. An interesting detail highlighted by Figure 6 is that the acquisition time strongly depends on the total memory footprint. It appears to be roughly one order of magnitude larger. Estimating the time needed to acquire a trace for a given instance might then be possible from its memory requirements.

In a third experiment we acquired an execution trace of the largest available instance of the LU benchmark, i.e., class E executed with 16,384 processes. We compare two acquisition modes. The former applies an extreme folding factor, executing more than 400 process per CPU, while the latter distributes the execution in an extreme way on a heterogeneous platform, i.e., combining the folded and composite modes (see Section 4.2). More specifically, we distributed trace acquisition across 778 compute nodes in 18 clusters at 9 geographically distant sites of the Grid'5000 experimental testbed[1]. The set of clusters being heterogeneous, the MPI processes were not uniformly distributed across the clusters. Instead, they were distributed according to the memory capacity of each cluster, attempting to maximize usage of main memory without overcoming its capacity. As a result, we used a different folding factor on each cluster.

In both settings, our Minimal instrumentation method was successful in acquiring the trace, producing a 1.45 TiB time-independent trace. In the former configuration, the instrumented application was executed on 40 nodes of the *StRemi* cluster in three and a half hour. The memory footprint was around 1TiB. With the extremely distributed configuration, 53 minutes were needed to execute the instrumented application, and 16 minutes were needed to gather and aggregate individual trace files on a single node using a K-nomial tree (see Section 4.3). This time includes a gzip

[1] `http://www.grid5000.fr/`

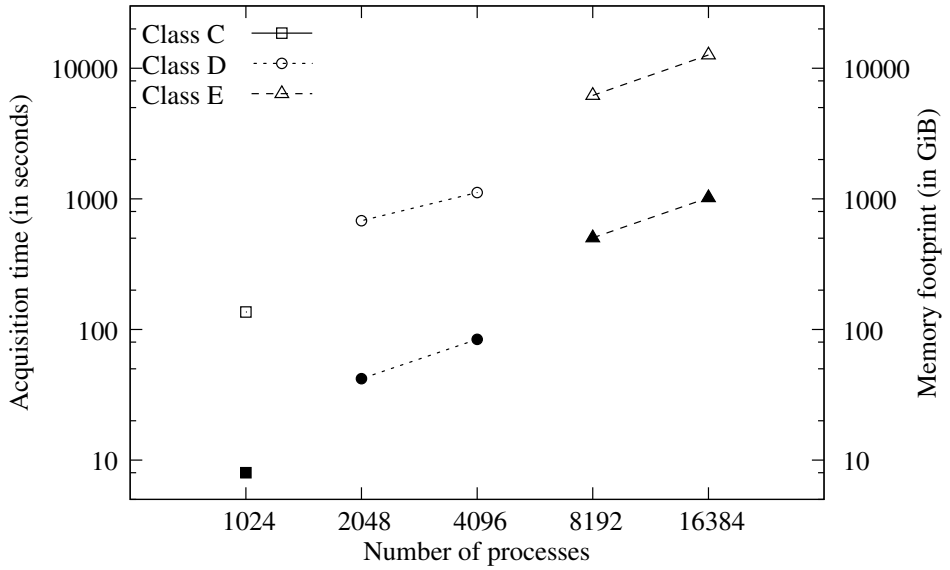


Figure 6: Acquisition time (white symbols) and memory footprint (black symbols) for larges instances of the LU benchmark on 40 nodes of the *StRemi* cluster.

compression step. Decompressing the files on a single node is a more expensive task that added around 40 minutes of computation. For such a large trace, using compression/decompression was worthwhile. Adaptive compression methods could be used to achieve a good trade-off between compression time and data transfer time [34]. In the end, the combination of folded and composite modes is sufficiently scalable to acquire a trace for a 16,384-process MPI application using 9 different clusters over a wide-area network in less than two hours. As seen in the first part of this experiment such a large trace can also be acquired on a single cluster thanks to the folded mode. In this case the acquisition time is 1.5 times longer, but this has to be put in perspective given the added complexity of reserving two or more acquisition platforms for a composite mode execution. These two experiments are extreme scenarios that represent both ends of the range of execution environments that can be used to acquire a time-independent trace. With the proposed framework, a user can exploit what computing resources are available at hand without being constrained by the "homogeneous and at scale" requirement imposed by other profiling tools.

5 Trace Replay

5.1 Approach

Our off-line MPI application simulator is integrated with the SIMGrid project [18]. SIMGrid provides core functionalities for simulating arbitrary distributed applications running in heterogeneous distributed environments. The main goal of SIMGrid is to facilitate research in the area of parallel and distributed computing in general, with most users studying more specifically cluster, grid, cloud, and peer-to-peer computing. SIMGrid provides a scalable and extensible simulation engine, and offers several user APIs. We have added in release 3.3.3 of SIMGrid the possibility for users to describe an application workload as a time-independent trace such as that described in Section 3. As shown in the upper part of Figure 7, three input files are needed to replay such traces with SIMGrid. Besides the *time-independent trace(s)*, the input to the simulator include descriptions of (i) the *simulated platform* (i.e., set of nodes with compute rates, set of network links with latency and bandwidth value, network topology graph) and of (ii) the *deployment* of the application (i.e., how simulated processes are mapped onto simulated processors). These input are passed to the *trace replay tool* which, in turn, is built on top of the *simulation kernel* in SIMGrid. Decoupling the simulation kernel, and then the simulator, from the simulation scenario offers flexibility. A wide range of simulation scenarios can be explored without modifying the simulator and instead simply changing the input files.

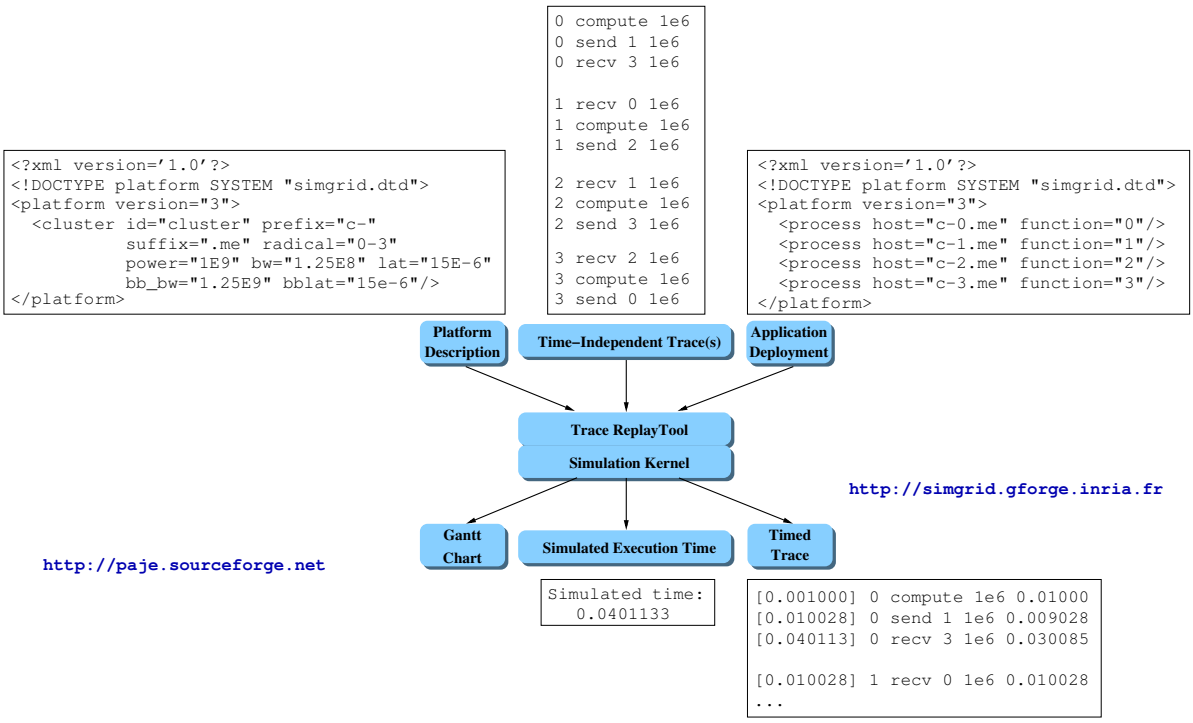


Figure 7: Overview of the Time-Independent Trace replay framework.

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 per second. 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 generic model is well suited to settings ranging from local-area to wide-area networks. Importantly for this work, SIMGrid also includes a network model for cluster interconnects that accounts for the specifics of MPI implementations that use TCP on GbE interconnects [20]. SIMGrid is thus ideally suited to serve as the simulation framework for implementing our Time-Independent Trace Replay Framework. Note that it would be interesting to experiment with other interconnects, e.g., InfiniBand, but at this time SIMGrid does not provide a validated model for such networks. Nevertheless, we note that over 40% of the systems on the Top500 list [35] at the time this article is being written use GbE interconnects, indicating that our results are representative for a significant fraction of High Performance Computing platforms.

As mentioned in Section 2, SIMGrid includes SMPI, an *on-line* simulator for MPI applications. To avoid code duplication and to benefit from the development effort in SMPI, we have opted for implementing our off-line simulator on top of SMPI. Although this may seem counter-intuitive, we can simply call an SMPI function for each action in the Time-Independent Traces. For instance, to replay a *send* action that occurs in a trace, we wrap an SMPI call in a short function, as shown in Figure 8. In this way, we reuse the simulation capabilities of SMPI, but following an off-line simulation approach.

From a user point of view, replaying a set of Time-Independent Traces simply amounts to running the following program called `smpi_replay`:

```
int main(int argc, char *argv[]){
 smpi_replay_init(&argc, &argv);
 smpi_action_trace_run();
```

```
static void action_send (const char *const *action){
 int to = atoi(action[2]);
 double size = parse_double(action[3]);
 smpi_mpi_send (NULL, size, MPI_BYTE, to, 0, MPI_COMM_WORLD);
}
```

Figure 8: Implementation of the *send* action using the SMPI internal API.

```
 smpi_replay_finalize();
 return 0;
}
```

This program first initializes some data structures. It then loads a Time-Independent Trace for each process and issues a SMPI call (via a simple wrapper like that shown in Figure 8) for each action in the trace. Finally it destroys all the data structures that were used for the simulation. This program is a regular SMPI program, but instead of being a hard-coded MPI application it issues MPI calls based on time-independent traces. As such, it is compiled with the `smpicc` wrapper and launched thanks to the `smpirun` command, both included with SMPI, as follows:

```
smpirun -np 8 -hostfile hostfile -platform platform.xml \
        ./smpi_replay trace_description
```

where `np` and `hostfile` are classical command-line switches of the standard `mpirun`. In addition, SMPI requires a description of the simulated platform, which is provided above in a file named `platform.xml` file. Finally, the `smpi_replay` binary launched by `smpi_run` takes a single file as argument, called `trace_description` in the example command-line above. This file contains a list of all the names of the trace files associated to the MPI processes. If this file contains a single entry, all the processes perform the same (simulated) actions in a single trace file. Otherwise, each process performs the actions in its own trace file. We refer the reader to [17, 20] for all details on SMPI.

The simulation can produce various types of output as indicated in the bottom part of Figure 7. In this work, we focus on obtaining a *simulated execution time*, which serves as a prediction 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 an event trace that corresponds to this particular scenario by adding timers (measuring simulated time) in the trace replay tool. Finally it is possible to visualize the trace as a Gantt chart, thanks to a built-in tracing mechanism provided by SIMGrid that produce traces in the Pajé format [30].

5.2 Simulation Accuracy

In this section we evaluate the accuracy of our simulation framework. Consequently, we simulate application executions on an existing platform so that we can compare actual and simulated execution times. For this comparison to be fair it is thus necessary to *calibrate* the simulation, i.e., to instantiate the platform description shown in the top-left corner of Figure 7. To determine the CPU's instruction processing rate we execute small application instances with only 4 processes on the compute nodes of the platform to be simulated. Since the instruction rate can be impacted by memory hierarchy effects we run several application instances with different problem sizes to determine several rates. The instantiation of the network parameters is done by running the `Pingpong_Send_Recv` benchmark of the SKaMPI [36] benchmark suite, executed between two nodes of the platform to be simulated. The latency corresponds to the time to send a zero-byte message, while the bandwidth corresponds to the peak bandwidth achieved when exchanging large messages. It is important to note that although this callibration is necessary in this section so as to validate our simulation framework, a user of our framework would not perform callibration but instead simply pick parameter values to describe an arbitrary, and likely hypothetical, platform.

5.2.1 Accuracy Results

Figure 9 shows actual and simulated execution times for various classes of the EP, DT, LU, and CG benchmarks on the *graphene* cluster. EP (Embarrassingly Parallel) consists of only computations, DT (Data Traffic) consists of only communications, while LU (LU Factorization) and CG (Conjugate Gradient) mix computation and communication. For DT, the class pertains not only to the data size but also to the number of communicating processes: classes A, B, C, and D involve 21, 43, 85, and 171 processes, respectively. Moreover three different communication graphs can be used for this benchmark. We chose the Black Hole (BH) communication graph that collects data from multiple sources to a single sink. For all trace replay experiments in this section we do not use the folded execution mode. This is because our goal is to compare simulation to real executions, and real executions are not folded so as to prevent contention for RAM and CPU resources among application processes. In fact, in these experiments we conservatively schedule one process per CPU, thus leaving some cores idle but ensuring that our results are easier to interpret because absolutely free of the memory bus and network interface contention issues that come with multi-core executions. For each benchmark, the mapping of processes to processors is chosen with respect to the benchmark's communication pattern.

The presented execution times are averages computed over ten "clean" runs. The *graphene* cluster is shared by multiple users and accessed through a resource management system. As a result, while jobs are well isolated in terms of compute resources they can interfere with each other because the network resources are shared. Out of a set of $n > 10$ runs, we pick a subset of 10 runs with minimal standard deviation of the execution time. In practice we had to use $n = 13$ or $n = 14$, and the non-clean runs appeared as clear outliers. Obtaining clean runs is needed for a fair comparison between simulated and real executions because simulations correspond to an ideal execution under perfect conditions. By contrast, real executions can be impacted by the load on the shared network as well as by hardware/software idiosyncrasies, as seen in the next section.

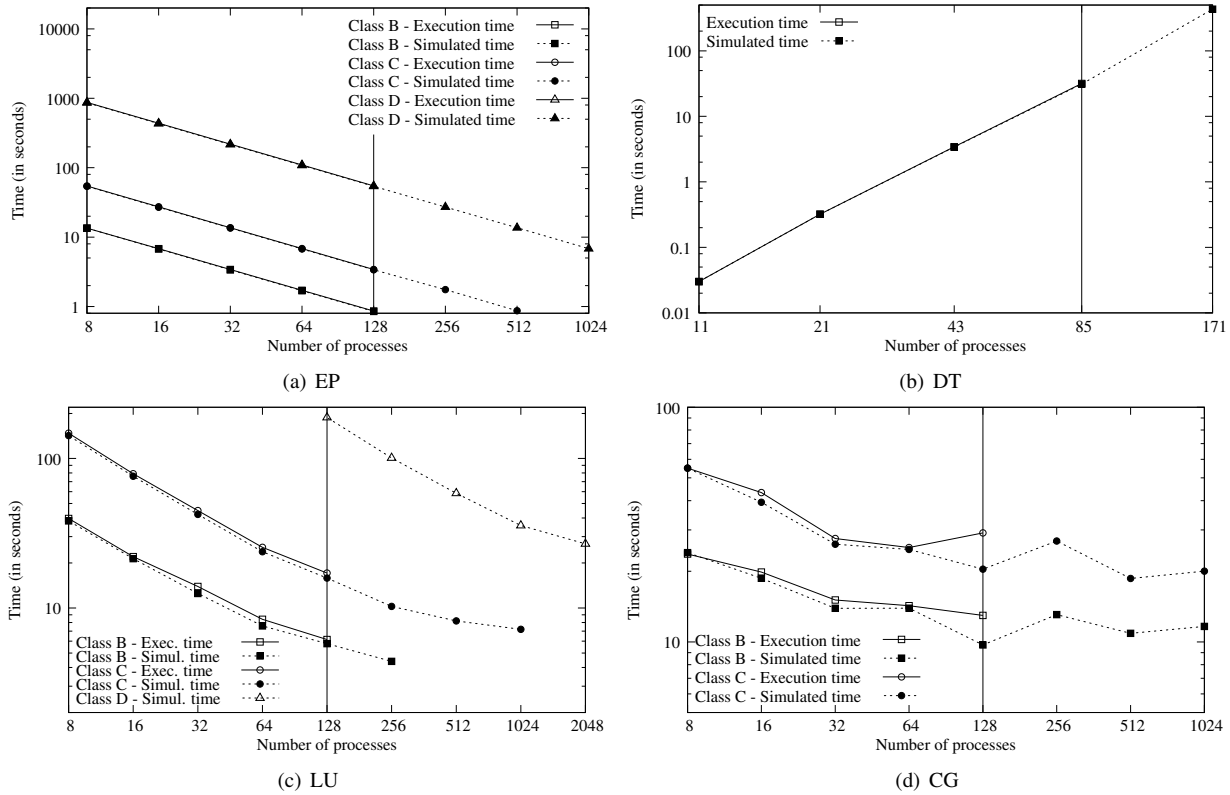


Figure 9: Comparison between simulated and actual execution times for four NAS parallel benchmarks.

Results for the EP and DT benchmarks show that the simulated execution times track the actual execution times

closely, with maximum absolute relative error of 1.35% and 6.33% (corresponding to an absolute error of only 2.2 ms), respectively. These two benchmarks are dominated either by computation or communication and are thus easier to simulate accurately. For the LU and CG benchmarks the simulation is less accurate with maximum absolute relative error of 10.18% and 29.95% respectively. The largest error observed for the CG benchmark is limited to instances executed with 128 processes, as seen in Figure 9(d). We explain the source of this particular inaccuracy in the next section. For other instances of the CG benchmark, i.e., up to 64 processes, the absolute relative error is at most of 9.10%. Our overall conclusion is that, at least based on these standard benchmarks, our approach achieves acceptable accuracy.

The vertical lines in Figure 9 denote the number of processes beyond which it is not possible to compare actual and simulated times due to the scale of the clusters available in our testbeds. Nevertheless, we added extra points on the right side of these vertical lines to illustrate the performance evolution trends that can be determined thanks to simulation on a larger, but not available cluster. For the EP, DT, and LU benchmarks, these trends are very coherent with the evolution of the execution time from 8 to 64 processes. The behavior of the CG benchmark is non-monotonic. This benchmark has a communication scheme that is very sensitive to the number of involved processes and their mapping on physical resources. Moreover, to simulate a cluster with up to 1,024 nodes, we had to modify the description of the simulated cluster to have eight cabinets instead of four. This implies more inter-cabinet communications that are more expensive. This could explain the increase of the simulated execution time for this particular benchmark that involve much more concurrent point-to-point communications than the three other benchmarks.

In Figure 9(c), we also show a more challenging scenario. We include simulation results for a class of larger instances, i.e., class D, executed with larger numbers of processes, i.e., up to 2,048 processes. The data processed for class D instances is sixteen times larger than for class C instances, which explains the great increase of simulated times. Moreover, the evolution of the execution time when the number of processes increases is coherent with what was measured for classes B and C.

We complete this study of the accuracy of our off-line simulation framework with experiments conducted with an actual HPC application. The BigDFT Density Functional Theory application [37] is the sole electronic structure code based on systematic basis sets that can use hybrid supercomputers and has good scaling. For this reason, BigDFT was selected as one of the eleven scientific applications in the Mont-Blanc project [38], whose goal is to assess the potential of low-power embedded components, such as commercially available ARM processing and network components, for building exascale clusters. We disabled the OpenMP and GPU extensions of BigDFT so as to focus on the behavior of the MPI operations. Figure 10 shows that despite the heavy use of various collective operations (i.e., `MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Allgather`, `MPI_Allgatherv`, and `MPI_Allreduce`) in this HPC application, our framework is able to replay it accurately. The relative error between simulated and execution times remains in a tight $[-5.65\%; -4.63\%]$ range as the number of processes increases.

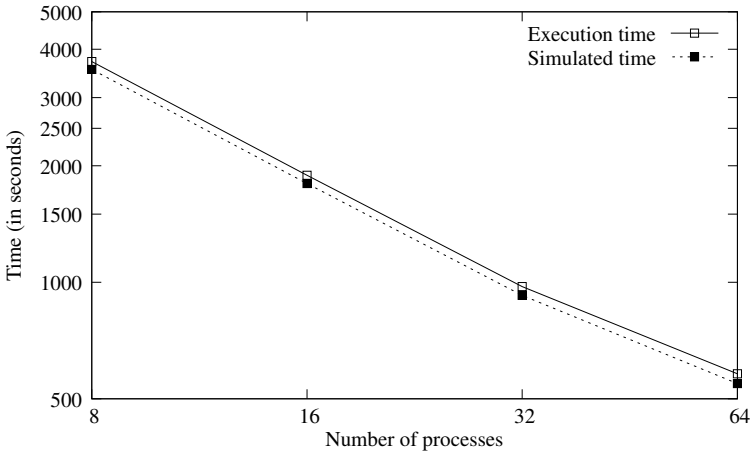


Figure 10: Comparison between simulated and actual execution times for the BigDFT application.

5.2.2 Sources of Inaccuracy

In this section we attempt to identify and explain the sources of the inaccuracies observed in the previous section. We begin by investigating the cause of the large simulation error observed for instances with the CG benchmark when executed with 128 processes. With 128 processes, the processes are scattered over the four cabinets of *graphene*, leading us to suspect a poor mapping of the processes to the compute nodes as the source of the inaccuracy. To confirm this suspicion we instrument the benchmark using TAU so as to visualize the communication patterns. The actual execution time of the instrumented benchmark is 14.41 seconds and the simulated execution time is only 9.90 seconds. The Gantt chart visualization for 2 seconds of a class B benchmark execution is shown in Figure 11. Our suspicion was in fact incorrect, but the source of the inaccuracy is glaring. We see two outstanding zones of `MPI_Send` and `MPI_Wait` operations. These operations typically take from a few microseconds up to less than a millisecond. In these zones, however, they take exactly 200 milliseconds. Our guess is that, due to high congestion, the switch drops packets and slows down at least one process to the point where it stops sending until a timeout of 200 milliseconds is reached. Because of the benchmark's communication pattern, blocking one process impacts all the other processes. This phenomenon occurs 24 times leading to a delay of 4.86 seconds. Without this behavior, the real execution would take 9.55 seconds, which is much closer to the 9.90 second actual execution time (3.54%). The same phenomenon was observed for class C executions of the benchmark. Our guess is that this behavior is due to a configuration bug/idiosyncrasy of this particular production platform.

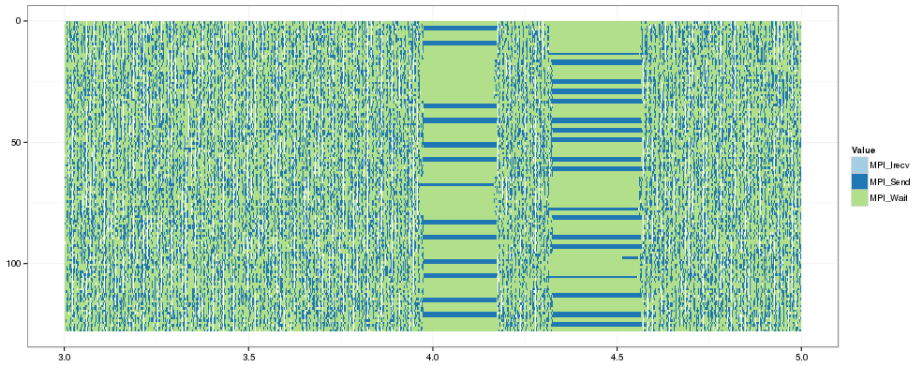


Figure 11: Two seconds Gantt-chart of the actual execution of a class B instance of CG for 128 processes.

We continue our analysis of the results with those obtained for the iterative LU benchmark, which alternates computations and communications at a high frequency. First we analyze the evolution of the relative error between execution and simulated times as the number of processes grows. Figure 12 shows that the relative error stays within a somewhat narrow $[-10.18\%, -3.07\%]$ interval for executions with between 8 and 128 processes. The results for class C show that the (absolute value of the) error increases as the number of processes grows. This trend is not visible for class B executions, which leads to larger error for 32 and 64 processes.

Close completion times seen in Figure 9(c) or small relative errors seen in Figure 12 may hide a poor, but "lucky," behavior. It is possible for the estimation errors of computation times and estimation errors of communication times to compensate each other. For instance, an aggregate -2% error could in fact be due to an overestimation of the communication that leads to a $+10\%$ error and to an underestimation of the computation time that leads to a -12% error. To decouple both errors we enforce *perfectly accurate* simulated computation times. To this end, we modify the trace acquisition step to record the duration of each compute action in addition to the volume of computed instructions, collecting the traces in regular mode. This correspond to the traditional way to collect time-dependent traces for use with previously proposed off-line MPI simulators. We can then replace the number of instructions with the *duration* multiplied by the *processing rate*. When replayed, the compute actions will thus last exactly the same (simulated) time as the time observed during the real execution. Figure 13 shows results obtained with this method.

The simulations with perfectly accurate computation times exhibit improved accuracy, with a maximum at 2.82% and all errors in the narrower $[-2.64\%, 2.82\%]$ interval. Executions of the LU benchmark are dominated by computation for class B up to 32 processes and for class C up to 64 processes. Even for 128 processes computation time still

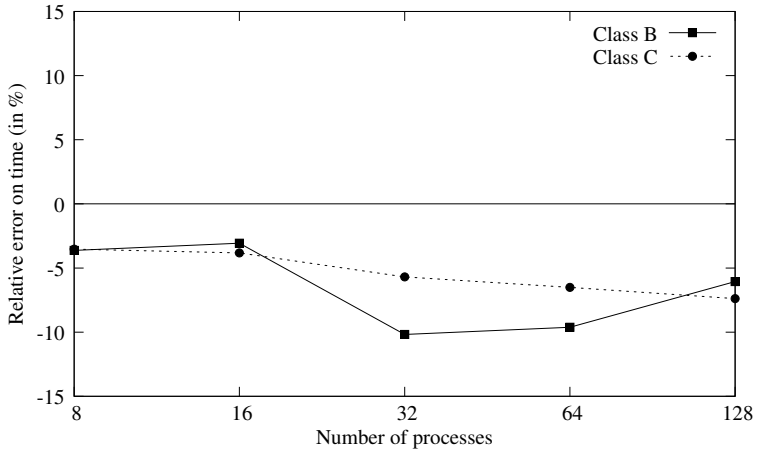
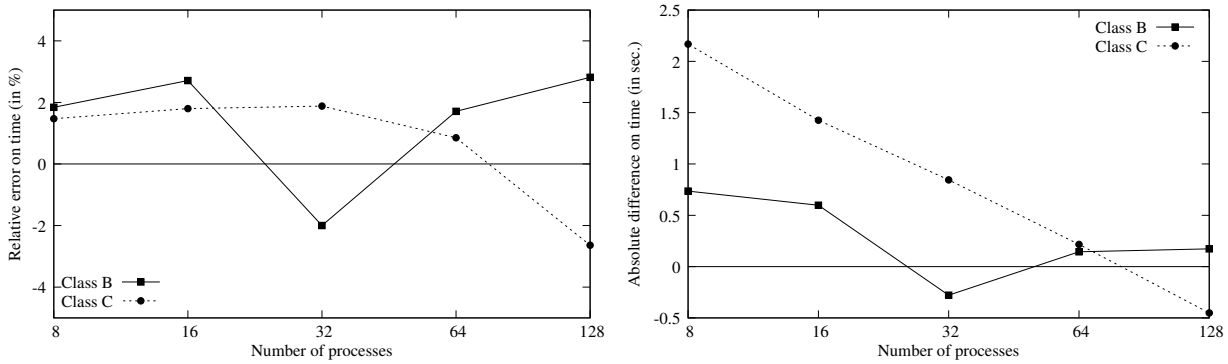


Figure 12: Relative error betwen actual and simulated execution time vs. number of processes, for class B and C executions of the LU benchmark.



(a) Relative error

(b) Absolute difference

Figure 13: Relative error (left) and absolute difference (right) between actual and simulated execution time vs. number of processes, for Class B and C executions of the LU benchmark. Simulated times are obtained with perfectly accurate computation times.

represents more than 32%, resp. 44%, of the execution time for class B, resp. class C. Accurate simulation of the computation times is thus essential. We see in Figure 13(a) that the relative error is more positive than negative, denoting that communication times are on average over-estimated in simulation. Since the relative errors in Figure 9(c) for the original simulations are negative, we can infer that computation times are underestimated and that this underestimation is marginally compensated by communication time overestimations, indeed a "lucky" behavior. We still observe a greater error for the class B instance executed with 32 processes, for which we were not able to determine a probable cause. Ignoring this data point and looking at the absolute difference between actual and simulated execution time for both class B and C, which is plotted in Figure 13(b), we observe that the absolute difference decreases roughly quadratically as the number of processes increases.

To explain this decrease we first study the distribution of messages sizes for class C executions, for which the decrease is the steepest of the two benchmark classes. We find that almost all the messages sent are smaller than 3 kiB (99.37%). This proportion remains constant as the number of processes grows while the number of exchanged messages grows linearly with the number of processes (from 808,298 with 8 processes to 18,752,372 with 128 processes). Importantly, the sizes of the large messages also decrease as the number of processes increases. The larger message sizes are between 100 kiB and 1 MiB. In order to determine sources of error, we designed a simple communication benchmark. Given a chain of four MPI processes of rank $i = 0, \ldots, 3$, the process of rank 0 sends a message

of size S to the process of rank 1, which forwards it to the process of rank 2, etc. When it receives the message, the process of rank 3 sends it back to the process of rank 2, all the way back to the process of rank 0. We measure the time for the message to perform its round-trip from and back to the process of rank 0. With this simple benchmark we find that in our simulation the communication times for small messages are underestimated while those for large messages are overestimated. It turns out that the overestimation error due to large messages thus decreases as the number of processes increases (more messages but of decreasing sizes). By contrast, the underestimation error due to small messages increases (more messages of similar sizes). Consequently, as the number of processes increases our simulation leads to more optimistic simulated execution times, explaining the downward trend seen in Figure 13(b).

The above analysis outlines the difficult of instantiating a reasonable network model of the target cluster for use in simulation. In our case, this difficulty mostly comes from the fact that the cluster consists of several cabinets connected through a *hierarchical* network. The network latency experienced by pairs of communicating processes depends on the number of switch hops. In our platform description, for the sake of simplicity, we use a single latency value that corresponds to the latency of a single-hop network path. As a result, for very small messages whose communication times roughly correspond to the network latency, our simulation leads to underestimations. Note that we could reduce this error by using a more complex target platform description that accounts for the network hierarchy, as possible with SIMGrid. However, we note that the accuracy of the simulation of communications is already very good, with simulated times within roughly 2% of actual times. In the end, we conclude from the results in Figures 12 and 13(a) that the largest source of error is the simulation of computation times.

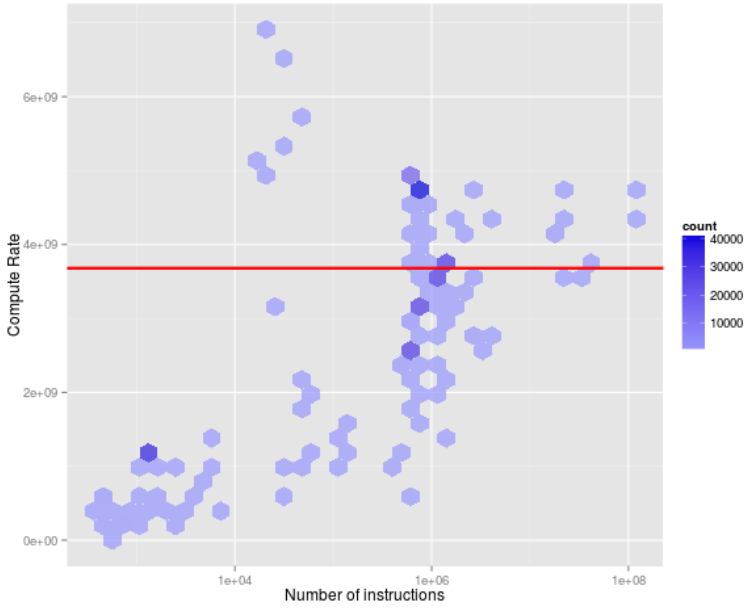


Figure 14: Distribution of compute rate (instruction/sec) vs. number of instructions for all compute bursts during a 4-process execution of the LU benchmark. Average compute rate shown as a horizontal line.

As mentioned in Section 5.1, the calibration procedure of the computation part leads to a single value, that corresponds to the processing rate of a CPU, for the *whole execution*. However, the compute rate is not constant in a real execution due to differences in instruction mixes between compute bursts. For instance, some parts of the applications have more cache affinities or may exploit the CPU better than others. The compute rate may change even within a single CPU burst or basic block. For instance, the first iteration of a loop may load data into cache while subsequent iterations experience high cache hit rate. As for the network latency issue explained in the previous paragraph, picking a single compute rate value contributes to simulation error. To illustrate this issue Figure 14 shows compute rates vs. number of instructions for all computation bursts as measured during small 4-process execution of the LU benchmark on the *graphene* cluster. The shade of each data point is based on the number of compute bursts. We observe that

compute rates vary by orders of magnitudes, and that the average value (shown as a horizontal line) corresponds to few actual compute rates. This explains the relatively large, but still within around 10%, simulation error due to compute activities. In general, we do not know the compute rate distribution of the compute bursts for an application execution on a hypothetical platform to be simulated. While it would be straightforward to allow the specification of such an arbitrary distribution in our framework, it is not clear how users would come up with a realistic such distribution. Most users would likely be content to use a single compute rate, which is the approach used by most off-line simulators reviewed in Section 2. However, some of the techniques used in ScalaTrace [24] could be leveraged to refine the instruction rate. To reduce the size of the produced traces, ScalaTrace identifies patterns in the application code, e.g., a send-receive within a loop, and replaces all the instances of a given pattern, e.g, coming from the iterations of the loop, by a single entry in the trace during the execution. Different methods to combine the execution delay of each instance into a single value or a distribution are also proposed. These two techniques, pattern identification and delay combination, could be applied to our framework to go further than using a simple average processing rate. We would need to augment our instrumentation tool with a pattern detection feature similar to that used in ScalaTrace. The only major difference is that, while executing the instrumented application, we would aggregate numbers of instructions instead of delays. An attractive side effect would be to reduce the trace size, as mentioned in Section 4.3.

5.3 Simulation Time

The time to replay a time-independent trace is directly related to the number of actions. The large number of actions in the LU traces implies a large simulation time. Figure 15 shows the simulation time as the number of processes increases for class B and C instances of the LU benchmark. These timings were obtained on a laptop with a 2.5GHz Intel i5-3210M processor, 8 GiB of RAM and a Samsung SSD 830 drive.

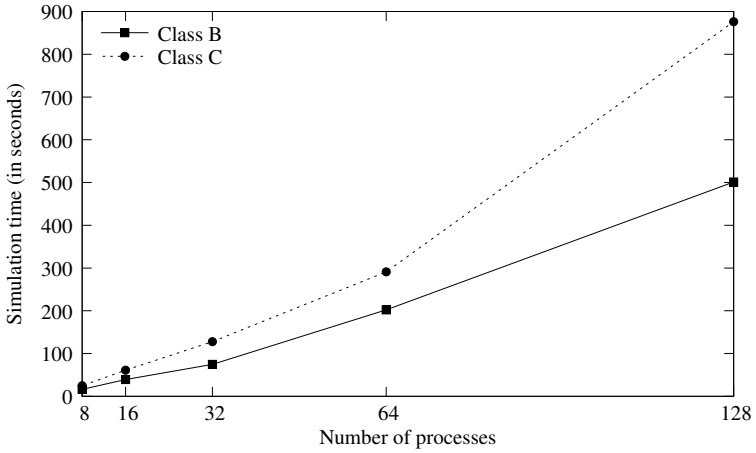


Figure 15: Evolution of the trace replay time with the number of processes executing a LU benchmark.

We see that the simulation time grows roughly linearly with the number of processes up to 128 processes for class B and up to 64 processes for class C. The simulation time is larger for the C-128 instance, whose trace is more than 1 GiB. This simulation time is directly related to the number of actions. The average time to process one million actions is 8 seconds with up to 64 processes. With 128 processes, however, this processing time is higher at around 11 seconds. This allows us to estimate the simulation time based on the size of the time-independent trace. For instance, replaying the 1.45 TiB trace of a class E instance of the LU benchmark executed with 16,384 processes (as obtained in Section 4.5) would take around 8 days on the above laptop (provided it has sufficient storage space). This large trace contains around 70 billion actions. This simulation time may seem prohibitive but, as mentioned earlier, neither the trace format nor the trace loading mechanism have been optimized for performance. Moreover, current efforts in the SIMGrid project aim at enabling the distribution of the simulation across several compute nodes. Our Time-Independent Trace Replay Framework would likely benefit from such a capability, especially because traces could be distributed across multiple drives as well, so as to allow for concurrent accesses.

6 Conclusion

In this paper we have proposed a new approach for the off-line simulation of MPI applications. Instead of relying on logs of execution that associate a begin and end time-stamp or a duration to each event, we use time-independent traces as input to our simulation framework. These traces contain only information about volumes of computation and communication. As a result, our proposed approach is scalable since a trace for simulating a large-scale homogeneous platform can instead be obtained on platform that is smaller, heterogeneous and/or multi-site. We have implemented our framework as part of the SIMGrid toolkit so as to benefit from its fast, scalable, and validated simulation kernel. We have evaluated both the trace acquisition and the trace replay features of our framework. We have found trace acquisition to be accurate (low skew when compared to the original application), fast (low overhead for obtaining the trace), and scalable (ability to acquire and retrieve a large 16K-process trace in under two hours). We have found trace replay to lead to acceptable low error (e.g., in the 10% range) and to tractable simulation time on a standard laptop. We have identified and explained several sources of error. We have found the most predominant source of error to be the assumption that each compute burst experiences the same compute rate.

Our Time-Independent Trace Replay framework is freely distributed as part the SIMGrid project under the LGPL license (`http://simgrid.org/download.html`). Moreover, the process to acquire a Time-Independent Trace is fully described in [39].

Acknowledgments

This work is partially supported by the SONGS ANR project (11-ANR-INFRA-13) 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`).

The authors would also like to thank all the members of the SIMGrid project working on the SMPI effort, and especially A. Degomme, A. Legrand, and L. Schnorr for their invaluable help.

References

[1] Shende S, Malony A. The Tau Parallel Performance System. *International Journal of High Performance Computing and Applications* 2006; **20**(2):287–311.

[2] Geimer M, Wolf F, Wylie B, Mohr B. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing* 2009; **35**(7):375–388.

[3] Knüpfer A, Rössel C, Mey D, Biersdorff S, Diethelm K, Eschweiler D, Geimer M, Gerndt M, Lorenz D, Malony A, *et al.*. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. *Tools for High Performance Computing 2011*, Brunst H, Müller MS, Nagel WE, Resch MM (eds.). Springer Berlin Heidelberg, 2012; 79–91.

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

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

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

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

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

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

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

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

[12] Tikir M, Laurenzano M, Carrington L, Snavely A. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. *Proc. of the 15th International EuroPar Conference*, *Lecture Notes in Computer Science*, vol. 5704, Delft, Netherlands, 2009; 135–148.

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

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

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

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

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

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

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

[20] Bédaride P, Degomme A, Genaud S, Legrand A, Markomanolis GS, Quinson M, Stillwell M, Suter F, Videau B. Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. *Proceedings of the 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Denver, CO, 2013.

[21] Zheng G, Wilmarth T, Jagadishprasad P, Kalé L. Simulation-Based Performance Prediction for Large Parallel Machines. *International Journal of Parallel Programming* 2005; **33**(2-3):183–207.

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

[23] Noeth M, Mueller F, Schulz M, de Supinski BR. Scalable compression and replay of communication traces in massively parallel environments. *Proc. of the IEEE International Parallel and Distributed Processing Symposium*, 2007; 1–11.

[24] Ratn P, Mueller F, de Supinski BR, Schulz M. Preserving Time in Large-scale Communication Traces. *Proc. of the 22nd Annual International Conference on Supercomputing*, 2008; 46–55.

[25] Gropp W. MPICH2: A new start for MPI implementations. *Proc. of the 9th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, vol. 2474, Springer: Linz, Austria, 2002; 7.

[26] Gabriel E, Fagg G, Bosilca G, Angskun T, Dongarra J, Squyres J, Sahay V, Kambadur P, Barrett B, Lumsdaine A, *et al.*. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Proc. of the 11th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, vol. 3241, Springer: Budapest, Hungary, 2004; 97–104.

[27] Browne S, Dongarra J, Garner N, Ho G, Mucci P. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Internation Journal of High Performance Computing and Applications* 2000; **14**(3):189–204.

[28] Kufrin R. Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux. *Proc. of the 6th International Conference on Linux Clusters: The HPC Revolution 2005 (LCI-05)*, Chapel Hill, NC, 2005.

[29] Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, Müller M, Nagel W. The Vampir Performance Analysis Tool-Set. *Proc. of the 2nd International Workshop on Parallel Tools for High Performance Computing (HLRS)*, Stuttgart, Germany, 2008; 139–155.

[30] Chassin de Kergommeaux J, de Oliveira Stein B, Bernard PÉ. Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications. *Parallel Computing* 2000; **26**(10):1253–1274.

[31] Desprez F, Markomanolis GS, Quinson M, Suter F. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. *Proc. of the 2nd International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, Taipei, Taiwan, 2011; 467–476.

[32] Desprez F, Markomanolis GS, Suter F. Improving the Accuracy and Efficiency of Time-Independent Trace Replay. *Proc. of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Salt Lake City, UT, 2012.

[33] Bailey D, Barszcz E, Barton J, Browning D, Carter R, Dagum L, Fatoohi R, Frederickson P, Lasinski T, Schreiber R, *et al.*. The nas parallel benchmarks - summary and preliminary results. *Proc. of Supercomputing '91*, Albuquerque, NM, 1991; 158–165.

[34] Jeannot E. Improving Middleware Performance with AdOC: An Adaptive Online Compression Library for Data Transfer. *Proc. of the 19th International Parallel and Distributed Processing Symposium*, Denver, CO, 2005.

[35] Top 500 Supercomputer Sites. `http://www.top500.org/`.

[36] Reussner R, Sanders P, Träff JL. SKaMPI: a Comprehensive Benchmark for Public Benchmarking of MPI. *Scientific Programming* 2002; **10**(1):55–65.

[37] Genovese L, Neelov A, Goedecker S, Deutsch T, Ghasemi SA, Willand A, Caliste D, Zilberberg O, Rayson M, Bergman A, *et al.*. Daubechies Wavelets as a Basis Set for Density Functional Pseudopotential Calculations. *Journal of Chemical Physics* 2008; **129**(014109).

[38] Mont-Blanc: European Approach Towards Energy Efficient High Performance. Montblanc. `http://www.montblanc-project.eu/`.

[39] Markomanolis GS, Suter F. Time-Independent Trace Acquisition Framework – A Grid'5000 How-to. *Technical Report RT-0407*, Institut National de Recherche en Informatique et en Automatique (INRIA) Apr 2011. URL `http://hal.inria.fr/inria-00586052`.