

Toward More Scalable Off-Line Simulations of MPI Applications

Henri Casanova, Anshul Gupta, Frédéric Suter

▶ **To cite this version:**

Henri Casanova, Anshul Gupta, Frédéric Suter. Toward More Scalable Off-Line Simulations of MPI Applications. Parallel Processing Letters, 2015, 25 (3), 10.1142/S0129626415410029 . hal-01232787

HAL Id: hal-01232787

https://inria.hal.science/hal-01232787

Submitted on 24 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Toward More Scalable Off-Line Simulations of MPI Applications

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

Anshul Gupta
INRIA, LIP, ENS Lyon
Lyon, France

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

Abstract

The off-line (or post-mortem) analysis of execution event traces is a popular approach to understand the performance of HPC applications that use the message passing paradigm. Combining this analysis with simulation makes it possible to "replay" the application execution to explore "what if?" scenarios, e.g., assessing application performance in a range of (hypothetical) execution environments. However, such off-line analysis faces scalability issues for acquiring, storing, or replaying large event traces.

We first present two previously proposed and complementary frameworks for off-line replaying of MPI application event traces, each with its own objectives and limitations. We then describe how these frameworks can be combined so as to capitalize on their respective strengths while alleviating several of their limitations. We claim that the combined framework affords levels of scalability that are beyond that achievable by either one of the two individual frameworks. We evaluate this framework to illustrate the benefits of the proposed combination for a more scalable off-line analysis of MPI applications.

1 Introduction

Analyzing and understanding the performance behavior of parallel applications implemented with the Message Passing Interface (MPI) on various compute platforms is a long-standing concern in the High Performance Computing (HPC) community. A popular approach is to conduct *post-mortem* analysis, i.e., based on event traces recorded during an instrumented execution of the application. As the scale of computing platform increases (into the exascale era), it is crucial that post-mortem analysis be scalable. We identify below three main scalability challenges faced by state-of-the-art approaches.

Scalability challenge #1 – The scale of the performance analysis is limited to the largest, dedicated, homogeneous platform available. This is because most tools for the performance analysis of message passing applications require traces with precise and uniform measured elapsed times for all events (computation, communication) included in the trace.

Scalability challenge #2 – The sheer number of recorded events leads to traces so large that transferring, storing, and processing these traces are themselves analysis bottleneck. One possible solution to this challenge is to forego recording full traces but instead to record only a subset of the events and to record only aggregated statistical information [20, 11, 15]. Using such lossy information has proved sufficient for the automated or semi-automated detection of performance bottlenecks, load imbalance, or undesired behaviors such as late senders/receivers or wait states. However, they prevent more in-depth analysis such as Gantt-chart visualization, or even debugging. Those analysis tools [14] that focus on providing such advanced capabilities must thus trade off scalability for a higher trace resolution.

Scalability challenge #3 – One advanced message passing application performance analysis approach is *off-line simulation*. While understanding the performance behavior of an MPI application on an existing parallel platform is useful, in many cases one wishes to consider other platforms than the one on which the event trace has been obtained, perhaps hypothetical platforms that are not even available. Off-line simulation consists in simulating the application execution by "replaying" event traces. A common use case is when a platform is yet to be specified and purchased. In this case, simulation can be used to determine a cost-effective hardware configuration appropriate for the expected application workload. More generally, simulation can be used to understand the performance behavior of an application by varying the hardware and software characteristics of hypothetical platforms, e.g., compute and network speeds, network topologies, collective communication algorithms. Finally, 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 replay produces erroneous application results, but the application's performance behavior may be preserved. It is then possible to conduct development activities for performance tuning in a way that saves time and resources when compared to real executions. This is important due to the costs associated to using large-scale platforms (monetary charges, electrical power consumption). While off-line simulation for replaying event traces is a powerful and attractive proposition, it faces its own scalability challenges due to the high computational complexity of discrete-event simulation. For instance, many off-line simulators require the simulation to be executed using an amount of compute resources commensurate to that in the platform to be simulated.

In summary, the three scalability challenges are: (i) trace acquisition is limited to the largest dedicated, homogeneous platform available; (ii) trace size may be so large that transferring, storing, and processing traces are analysis bottleneck; and (iii) off-line simulation of a large-scale application execution for advanced performance analysis may itself require the use of a large-scale platform. Two recent and complementary works, ScalaTrace [17] and Time-Independent Trace Replay [7], have each partially addressed these scalability challenges.

The main objective of ScalaTrace [18] is to produce compact event traces of near-constant size while preserving structural information and temporal event order. This is achieved via a number of techniques to identify and encode patterns in event sequences so as to avoid recording all individual events, and alleviates the scalability challenge #2. ScalaTrace comes with a tool, ScalaReplay [23], to replay application execution directly from the compact traces. Traces generated by ScalaTrace *contain time-related information*. As a result, traces must be acquired on a dedicated, homogeneous cluster (scalability challenge #1). Furthermore, trace replay with ScalaReplay must be executed at that same scale (scalability challenge #3). Note that ScalaReplay does not allow for the straightforward exploration of "what if?" scenarios because it performs a "live replay" of the application. In fact, in [22] the authors mention that it would be useful to use ScalaTrace traces with an off-line simulation tool.

Time-Independent Trace Replay project [7] focuses on scalable trace acquisition and scalable trace replay. These objectives are achieved by eliminating all time-related information from the traces. As a result, trace acquisition can be performed on a heterogeneous, non-dedicated platform (e.g., a set of diverse clusters over a wide-area network), with multiple application processes possibly collocated on each compute node, thereby alleviating scalability challenge #1. The use of time-independent traces complexifies trace replay since a simple scaling of event durations is no longer possible. Instead, one must perform discrete-event simulation based on computation and communication volumes. To this end, Time-Independent Trace Replay leverages the state-of-the-art simulation abstractions and models in SimGrid [8], which make it possible to run the simulation quickly and on a single computer (e.g., a laptop), thus alleviating the scalability challenge #3. The main scalability drawback of this approach is the trace size (scalability challenge #2).

Given the above considerations, in this work we propose to combine the approaches in ScalaTrace and Time-Independent Trace Replay, namely, to use ScalaTrace's compact traces but in a time-independent manner. The resulting combined approach should allow for the off-line simulation of MPI applications with scalable trace acquisition, scalable trace sizes, and scalable trace replay. To the best of our knowledge, no previously proposed framework has achieved these three objectives simultaneously. More specifically, this

work makes the following contributions:

- We extend ScalaTrace's tracing mechanism to record volumes of instructions and produce compact time-independent traces;
- We enable application simulation based on ScalaTrace traces with the SimGrid simulation framework;
- We validate our combined approach through a series of experiments with representative benchmarks.

This paper is organized as follows. In Section 2 we provide necessary background information on ScalaTrace and Time-Independent Trace Replay. In Section 3, resp. Section 4, we describe and evaluate the combination of the approaches in these two projects in terms of trace collection, resp. off-line simulation. In Section 5 we discuss related work. We conclude in Section 6 with a brief summary of our results and with a discussion of future directions.

2 Background

2.1 ScalaTrace

ScalaTrace [17] is a tracing tool initially designed to characterize the communication behavior of large-scale applications. Its salient capability is that it produces and processes compact traces of near-constant size, while preserving structural information and temporal event order. This is achieved by taking full advantage of the regularity of the traced applications and implementing several techniques such as calling sequence identification, recursion-folding signatures, location-independent encoding of ranks, event aggregation, or causal cross-node reordering. The building blocks of a ScalaTrace trace are *Regular Section Descriptors* (RSDs) that capture MPI events in a single loop and *Power-RSDs* (PRSDs) that specify recursive RSDs nested in multiple loops. Time-related information is also encoded in a scalable way by ScalaTrace [18]. Instead of logging absolute timestamps, as done in other tracing tools, ScalaTrace keeps track of *relative delta times*, i.e., the time spent either between two consecutive MPI calls (*compute delta*) or within an MPI call (*communicate delta*). In order to preserve meaningful time information while keeping traces compact, ScalaTrace stores statistical information (i.e., average, minimum, and maximum delta times) for each event. This aggregate information is completed with *histograms* of delta times that better capture outliers and distribution properties. Finally, ScalaTrace distinguishes CPU bursts, and their corresponding histograms, not only by call stack but also by path sequence. Hence, the entry path to and exit path from a loop can be distinguished from the iteration path within the loop.

ScalaTrace comes with a tool, ScalaReplay, that replays traces without having to decompress them. ScalaReplay is itself an MPI program that must be executed with as many processes as that used to obtain the trace. It traverses the trace and triggers all the MPI calls with their original payload sizes. For instance, a RSD such as `RSD1: <100, MPI_Send1, MPI_Recv1>` leads to issuing a hundred pairs of `MPI_Send` and `MPI_Recv` calls and the *emulation* of the interleaving CPU bursts upon replay. This emulation is either done using sleeps or busy waits of appropriate delays that are stored in the trace. These delays are selected to reflect the distribution across the bins of the corresponding histograms. Significant improvements in ScalaReplay II [23] include an *elastic data element representation* that allows ScalaTrace to produce compact traces for applications that show inconsistent behavior across time steps.

2.2 Time-Independent Trace Replay

Unlike ScalaTrace, the Time-Independent Trace Replay project does not attempt to reduce trace sizes. One of its objective is to improve trace acquisition scalability. This is achieved by eliminating all time-related information from the collected trace, i.e., by collecting only computation volumes (numbers of instructions) and communication volumes (numbers of bytes). This simple idea makes it possible to remove several constraints regarding the nature and scale of the trace acquisition platform. First, it is no longer necessary to acquire a trace on a homogeneous platform. The compute nodes may have different compute speeds,

although processors must belong to the same Instruction Set Architecture family (so that computation volumes are comparable). The network topology can be arbitrary as well since network delays are not recorded. Consequently, the trace acquisition platform can be composite, e.g., a set of different clusters interconnected via a wide-area network. Furthermore, since no timing information is recorded, it is possible to "fold" the application execution by running multiple MPI processes on the same compute node provided the RAM capacity of the node is not exceeded. Results in [7] show how using folding on a composite platform it is possible to acquire a trace for a 16,384-process execution of the LU NAS Parallel Benchmark (NPB) [2] using either 778 compute nodes aggregated from 18 geographically distant clusters or a single cluster with 20 nodes with each 24 cores and 48 GiB of RAM.

From the perspective of off-line simulation the use of time-independent traces is both a challenge and a scalability opportunity. The challenge stems from the fact that without timing information the off-line simulation cannot be based on simple scaling of communication and computation delays, but must rely on precise discrete-event simulation techniques that use accurate simulation models. The scalability opportunity is that, using state-of-the-art simulation techniques [8] and in particular techniques designed specifically for simulating MPI applications [3], it is possible to conduct such simulations on a small-scale platform, e.g., a laptop computer, rather than using a full-scale platform for "live replay" of the traces. Experimental results in [7] show that it is possible to simulate application executions with reasonable accuracy (simulated execution times within a few % of actual execution times). The scalability of the simulation is not as convincing. While the simulation techniques are shown to scale in the context of on-line simulation [8, 3], the results in [7] show that the size of the traces, i.e., the large number of individual events encoded in each trace, is a limiting factor. As a result, the simulation time is shown to grow roughly linearly with scale of the simulation.

3 Making ScalaTrace Time-Independent

3.1 Motivations

ScalaTrace uses a compact trace format but traces contain time-related information, which limits the scalability of the trace acquisition process. By contrast, Time-Independent Trace Replay removes several of the scalability constraints for trace acquisition thanks to the use of composite platforms and/or "folding". However, its scalability is limited due to the size of the traces, which are not stored compactly. Our first objective in this work is to combine both approaches so as to obtain a solution with scalable trace acquisition and trace size.

In this section we provide motivation for our work by demonstrating that the "folding" technique in Time-Independent Trace Replay, which is key to trace acquisition scalability, cannot be leveraged with time-dependent traces (such as ScalaTrace traces). Let us first focus on the recorded compute delays. We execute the LU NPB (Class C with 64 processes) with various *folding factors* (i.e., number of MPI processes per compute node) on the *graphene* cluster in the Grid'5000 testbed [5], which comprises 144 2.53GHz Quad-Core Intel Xeon x3440 nodes. For several values of the folding factor (2, 4, 8, 16, 32), we compare the distribution of the measured delays normalized to those obtained in the *non-folded* mode (i.e., one MPI process per compute node). Since the processes have to compete for CPU resources in folded executions, a natural, perhaps naive, expectation is that the measured compute delays increase roughly linearly with the folding factor.

Figure 1 shows the distribution of the ratios between the measured delays in a folded execution and those in a non-folded execution, versus the folding factor. We find that, contrary to the naive expectation, folding the execution does not have the same impact on all compute events. If we consider all events (leftmost set of boxes), we see that the Inter-Quartile Range (IQR) is large, meaning that folding impact different measured delays differently. The figure also shows data restricted to the most time-consuming events, keeping the top 50%, 25%, 10%, and 5% events in terms of measured delays. These events, and in particular the top 5% events, have delays roughly multiplied by the folding factor up to a folding factor of 16. For a folding factor of 32, the IQR is very large. We conclude that scaling down measured delays to factor out the impact of

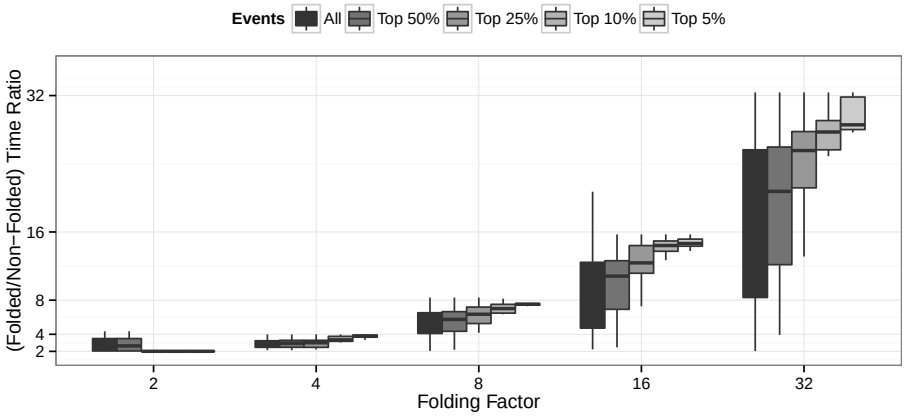


Figure 1: Distribution of the ratio between the measured delta times of a folded execution and those of a regular execution of the LU benchmark (Class C with 64 processes) across events for different folding factors.

execution folding is non-trivial, hence likely precluding the use of execution folding to increase the scalability of the trace acquisition process.

ScalaTrace records the time spent in each MPI call as communication times. These are used by ScalaReplay [18] to replay communication delays and to determine when certain asynchronous calls have to be emitted. Accurate recording of communication delays is in itself more challenging than that of computation delays. Large platforms are typically shared by multiple users who are granted access to compute nodes via a resource management system (e.g., a batch scheduler). So while compute nodes can be dedicated to a user, network links are typically shared and communication delays can be impacted by so-called *nearby jobs* [4]. Furthermore, communication delays can be impacted by operating system noise and running daemons. In the end, even on a homogeneous platform at scale, communication times can be difficult to measure consistently. Still using the same LU benchmark, but using various instances (B-8, B-16, C-8, C-16), we analyze the variation of the delays recorded by ScalaTrace. For each trace we compute the sum of the products of the number of occurrences of an event and of the recorded average delay for that event, in microseconds. In other words, we compute the cumulative computation time and communication time for each trace. The number of runs is between 5 and 7, depending on the instance.

Figure 2 shows the variability of the cumulative computation and communication times across different runs of the same application, launched using the OAR batch scheduler on Grid'5000. We made sure to request entire compute nodes from the batch scheduler and then used various number of cores within each node as described hereafter. In this way, our results are not impacted by sharing a compute node with other jobs. As expected, the computation times are stable across runs. However, there are some outliers due to executions with different numbers of cores per compute node. For B-8, B-16, and C-8, the first run was executed on 4 nodes using only one core per node, while the other runs used only one node (using all 4 cores). For C-16, the first six runs use one core per node, and the last one uses 4 cores on one node. When all the cores of a node are used, the application suffers from memory bus contention and leads to larger execution times. The number of cores used per node also impacts recorded communication delays, but this impact is counter-intuitive: using all 4 cores of a node increases the amount of intra-node communications (i.e., through shared memory), which should be faster than inter-node communications (i.e., over the network). We observe the opposite. This is because the cumulative communication times include time spent in the `MPI_Wait` function. Slowed down computation, due to memory bus contention, in fact improves process synchronization and reduces these wait times. Overall, these results highlights a drawback of time-related information in traces. Even though the execution environment is homogeneous in terms of compute resources, the process mapping impacts the measured times. It implies that a timed trace must be accompanied with a *precise* description

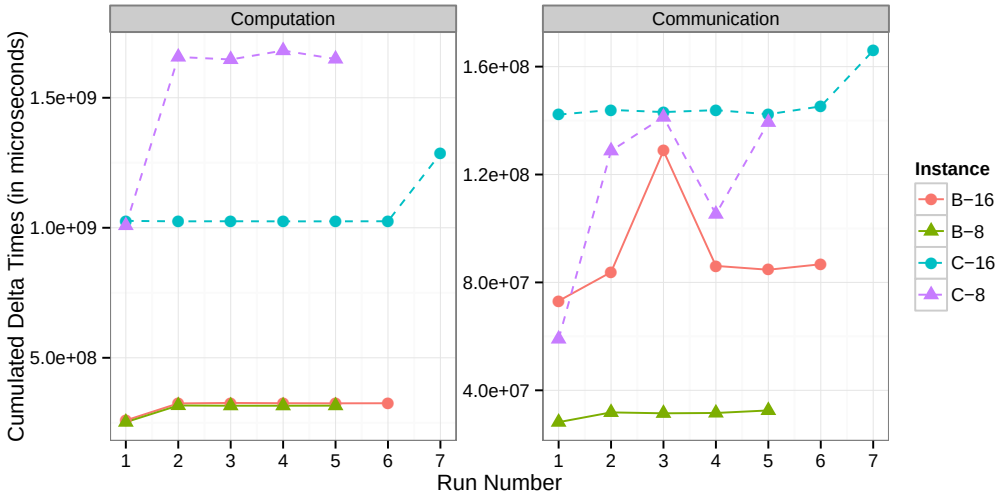


Figure 2: Variability of cumulated delta times across several traces of different instances of the LU benchmark.

of its acquisition platform, including the mapping of processes to cores, to make consistent replay possible.

Regardless of the impact of the mapping of MPI processes to cores and compute nodes, the main observation from Figure 2 is that significant variations in cumulative communication time can be observed across runs. The B-16 and C-8 instances show wide variations (e.g., +76.11% for B-16, -18.24% for C-8). These executions were performed back-to-back with the same compute nodes and for the same mapping of processes to cores. For B-16, a slightly larger cumulative computation time, which leads to smaller wait times, also contributes to the communication time variations. But this is not the case for the third run of the C-8 instance. For this run, the cause of the variation is external (we suspect network contention caused by nearby jobs).

Although time-dependent traces are used in most previously proposed tools, including ScalaTrace, they have several drawbacks. In this section we have provided quantitative evidence that, indeed, a key scalability enhancing technique proposed in [7], execution folding, is not applicable to time-dependent traces. Furthermore, our results show that obtaining consistent and realistic time measurements when tracing events is not straightforward. The mapping of processes to cores of compute nodes has to be taken into account. More importantly, communication delays can be heavily impacted by multiple factors so that two traces recorded back-to-back on the same set of compute nodes can have widely different recorded communication times. For these reasons, and as in [7] we completely forego the use of time-dependent traces in this work.

3.2 Implementation

The standard approach for instrumenting MPI applications, used by both ScalaTrace and Time-Independent Trace Replay, is to use the MPI profiling layer (PMPI). 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 instrumentation code in the implementation of all `MPI_` functions. ScalaTrace inserts pre and post stubs in these functions. In pre stubs the recording of the metric of interest, i.e., time, is stopped for the *computing* phase that precedes the MPI call and a new record is started for the impending *communication* phase. Conversely, post stubs stop the recording of the metric for the finished communication phase and start the recording for the impending computation phase. These stop and start actions are implemented by the `RecordStat()` and `ResetStat()` functions of the `Stat` class, which in turn call the `end()` and `start()` functions of the child class `StatTime`. Both these functions call `gettimeofday()` and the `end()` functions computes the *delta* elapsed time since the last call

to `start()`. Finally, events, i.e., trace entries, are associated with two instances of a `StatTime` object, one for computation phases, and another for communication phases.

Given the above clean design, modifying ScalaTrace so that it records the number of instructions executed in each computation phase instead of time is straightforward. We first added a new class, `StatInst`, which inherits from the `Stat` class and overrides the `start()` and `end()` functions. Instead of calling `gettimeofday`, these functions call the `PAPI_accum_counters()` function that accesses the current value of a specific hardware counter, as provided by the portable PAPI interface [6]. We count instructions using the `PAPI_TOT_INS` counter as in [7]. The `end()` function computes the *delta* in number of instructions since the last call to `start()`. Counter initialization and destruction are performed in the `MPI_Init` and `MPI_Finalize` functions. The number of bytes transferred in each communication phase is simply computed based on the parameters passed to MPI communication functions. Switching from time-dependent to time-independent tracing with our modified implementation of ScalaTrace is done through a single compilation flag.

3.3 Results

3.3.1 Impact on Trace Size

As mentioned in Section 1, the main scalability limit for Time-Independent Trace Replay is the verbosity of its trace format. Without any compression, the trace size directly depends on the number of recorded events. With the instrumentation method and corresponding trace format proposed in [7], called *MinI* therein, trace size grows roughly linearly with the number of processes. Reducing trace size was the principal motivation for considering ScalaTrace, whose main strength lies in the compactness of the traces it produces. This compactness preserves the complete sequence of events executed by the processes, which can be retrieved by *unrolling* a ScalaTrace trace.

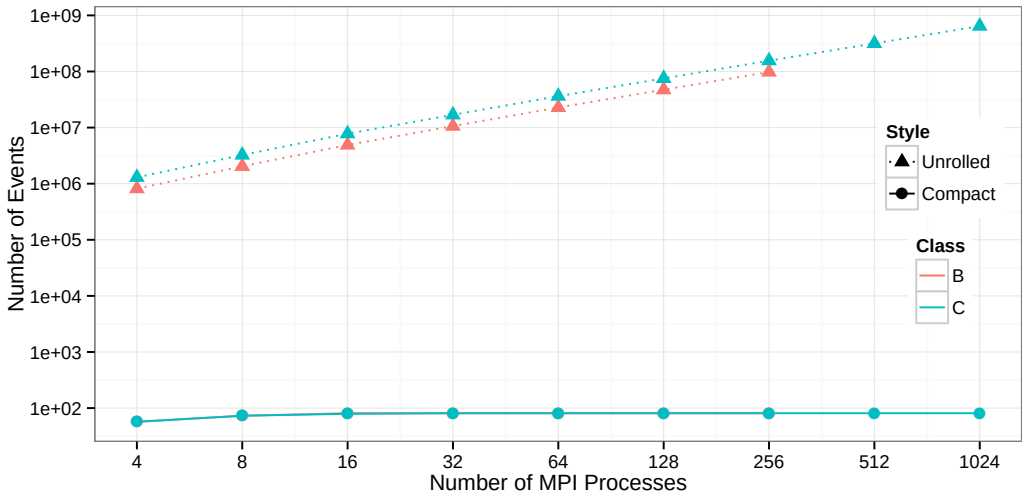


Figure 3: Evolution of the number of events stored in traces in the compact format of *ScalaTrace* and once unfolded for different instances of the LU benchmark.

Figure 3 plots the number of events stored in a trace vs. the number of processes for the LU NPB Class B and C. *Compact* refers to the ScalaTrace format, while *unrolled* corresponds to the total number of events executed by the processes computed once the trace is unrolled (which is the same number of events in the trace when stored in the verbose Time-Independent Trace format). The total number of events executed grows linearly with the number of processes. This is because 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). This number also grows by a constant factor of 1.6 when going from

class B to class C. For these instances, which perform millions of individual events, the number of *compact* events stored in ScalaTrace traces remains low (less than one hundred) and almost constant.

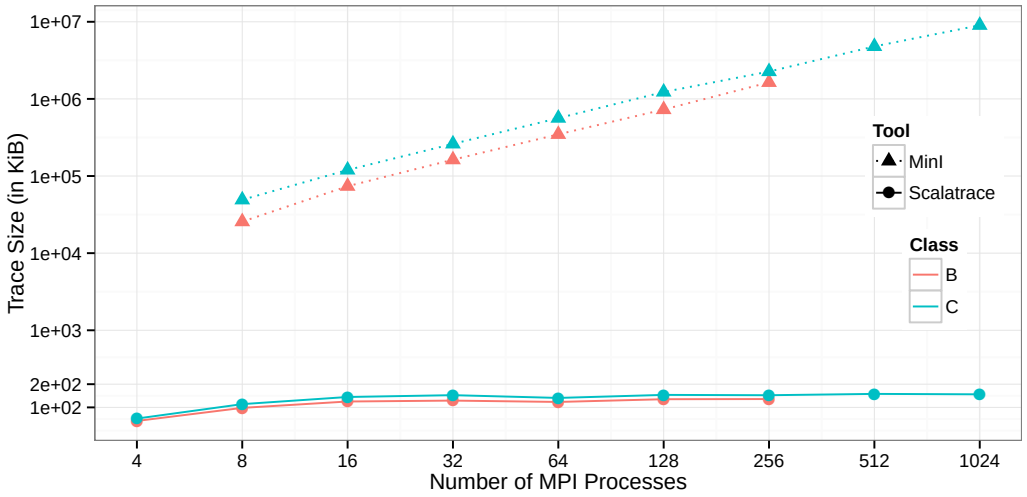


Figure 4: Evolution of the trace size in the *MinI* and *SclaTrace* formats for different instances of the LU benchmark.

The results in Figure 3 demonstrate the capacity of ScalaTrace to take a full advantage of the regularity of an application execution, in this case the LU NPB. This drastic reduction in the number of stored events directly impacts the trace size. Figure 4 plots trace size vs. number of processes the LU NPB Class B and C, for both the MinI instrumentation method in Time-Independent Trace Replay (data from Figure 3, page 10 in [7]) and ScalaTrace. With 15 ASCII characters per event on average and millions of events stored, MinI traces can be up to 8.8 GiB for class C with 1,024 processes, with trace size increasing roughly linearly with the number of processes. By contrast, ScalaTrace traces show only moderate size increase when the number of processes increases, remaining below 150 KiB. For a given problem size, we note some slight variations in the trace size that are related to the communication pattern of the application, e.g., an instance executed with only four processes has fewer events than instances with 8 processes. Importantly, increasing the problem size leads to only a slight trace size increase. This is because ScalaTrace stores large volumes (of instructions or bytes) in event histograms to ensure trace compactness.

Figures 3 and 4 combined show that replacing the verbose Time-Independent traces used in [7] by the compact traces produced by ScalaTrace solves the major scalability issue of trace size without losing the capacity to replay the exact sequence of events that composes an MPI application.

3.3.2 Impact of Folded Acquisition

The main advantage of time-independent traces that store volumes of executed instructions instead of *delta* times to represent the CPU bursts between MPI calls is that such information is oblivious to the acquisition conditions. Indeed, even though several processes share a single CPU or are scattered across heterogeneous compute nodes, the measured numbers of instructions should not vary. This is confirmed by Figure 5 that shows the distribution of the ratio between the measured number of instructions of a folded execution and those of a non-folded execution across events in the trace for different folding factors (for the LU NPB, Class C with 64 processes). The number of measured instructions does not vary significantly when the folding factor increases, except for a few outliers. We attribute these outliers to limited hardware counter resolution and to operating system noise. Overall observed variability is less than 5%. This is by contrast to the results in Figure 1, thus demonstrating that storing time-independent information enables the use of folding for

increasing the scalability of the trace acquisition process. While similar good results are reported in [7] for unrolled (time-independent) traces, the results in Figure˜ 5 are for compact ScalaTrace (time-independent) traces.

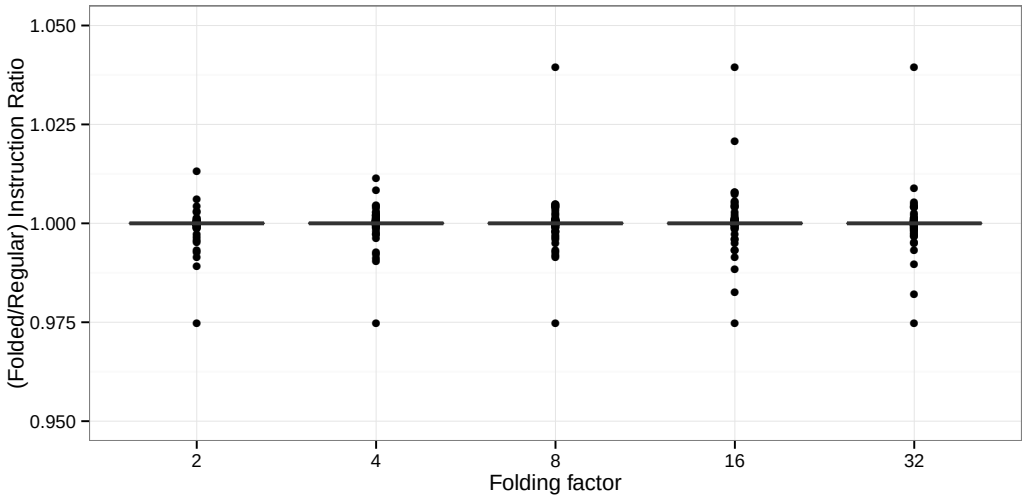


Figure 5: Ratios between the measured number of instructions for events in a folded execution and that in of a regular execution of the LU benchmark (Class C with 64 processes) vs. the folding factor.

4 Replaying ScalaTrace Traces in Simulation

4.1 Motivations

The "live trace replay" implemented by ScalaReplay must be executed on a platform at scale. This limits both the scalability of the analyses and the range of "what if?" scenarios that can be explored. Arguably, ScalaReplay is merely a means for its developers to assess whether the compactness of the traces does not lead to problematic information loss. From this standpoint, requiring a homogeneous platform at scale for trace replay is not necessarily a major drawback. ScalaReplay can actually be used to explore "what if?" scenarios by modifying the content of the traces. For instance, one could scale the delta times in the traces to *emulate* different compute or network delays such as those expected on a hypothetical platforms. However, such scaling is limited and cannot be used to explore all scenarios, such as those with transient and/or local performance degradations, different mappings of processes to compute nodes, different network topologies, or different collective communication implementations. By contrast, Time-Independent Trace Replay leverages SimGrid to enable the exploration of these scenarios, without any tempering of the traces, using off-line simulation. SimGrid leads to accurate simulation for the communication part of the application thanks to the thoroughly validated network model proposed in [3].

4.2 Implementation

The latest release of ScalaTrace includes a redesign of both the trace format and replay engine to cope with applications with inconsistent behavior across time steps. ScalaReplay can be used in two different modes: *normal* or *histo*. In the *normal* mode, each process simply traverses the trace and replays events while respecting the structure and temporal ordering of the original application. MPI calls are emitted with their original parameters and mock payloads of the right size. CPU bursts are replaced by sleeps or busy waits whose durations are extracted from the trace. The *histo* mode enables probabilistic trace replay,

meaning that senders randomly select receiver ranks from histograms. Receive operations are not posted upon parsing the trace, but instead postponed according to communication delays stored in the trace. A complex mechanism is implemented to handle pending asynchronous receive calls and to ensure that every send is matched.

In this work, we focus on applications such as the NPB, whose behavior is consistent across time steps. Moreover, the implementations of these applications rely on the MPI parameter `MPI_ANY_SOURCE`, a wildcard for the sender's rank when doing a receive operation. According to the ScalaTrace development team, this parameter causes problematic behavior when using the *histo* mode. Consequently, we use the *normal* mode in all our experiments.

To enable the simulated replay of ScalaTrace traces, we leverage the fact that ScalaReplay is an actual MPI application. As such, it can be simulated seamlessly by SMPI [9, 3]. The SMPI framework comprises: (i) an API that implements all the communication operations of the MPI-2 standard (and a subset of the MPI-3 standard) and all the collective communication algorithms of MPICH2 and OpenMPI with their selection logic; and (ii) an emulated MPI runtime. The API is implemented as one of SimGrid's user APIs, while the runtime is built on top of SimGrid's simulation kernel. SMPI supports MPI applications written in C, C++, or Fortran, which must be linked to the SMPI library. In a simulated execution with SMPI all MPI calls are intercepted, and the simulated duration of each call is determined by underlying simulation models. The compute bursts between the MPI calls are emulated, i.e., code for a CPU burst is actually executed on the simulation host, possibly scaled by a user-specified factor. Note that MPI allows the simulation of *unmodified* MPI applications (provided that they are compiled with `smpicc`, provided in the SimGrid distribution, instead of `mpicc`).

To simulate ScalaReplay with SMPI we had to address a couple of issues. First, simulating ScalaReplay as is with SMPI would be a simulation of the replay engine itself, not of the traced applications. This is because in ScalaReplay the replay logic (i.e., trace traversal, event extraction, *delta* times computations, ...) is interleaved with the MPI calls emitted to replay the traced application. To resolve this issue we have added a command line option to SMPI so that it does not perform the simulation of the CPU bursts, in our case ScalaReplay's replay logic. Second, we have modified ScalaReplay to replace the existing busy waits that emulates the *delta* times between MPI calls by the appropriate SMPI function, `smpi_execute(duration)`, where `duration` is the *delta* time extracted by ScalaTrace. When time-independent traces are replayed, we rely on the `smpi_execute_flops(numinst)` SMPID function, where `numinst` is the number of instructions extracted from the trace for the current burst. We have added compilation flags so that ScalaReplay allows simulated replay with time-dependent or time-independent traces.

4.3 Results

The main objectives in this work is to extend the capabilities of ScalaTrace by enabling a simulated, as opposed to live, replay of the compact traces it produces. More precisely, thanks to the modifications of the ScalaReplay replay engine described in the previous section and the capabilities offered by SMPI, we can simulate application execution using a single host but based on a ScalaTrace trace that has been obtained on a large-scale platform. This makes it possible to explore "what if?" scenarios, *without having to modify* the input trace, and in particular match the hardware characteristics of an existing platform.

Figure 6 shows execution time vs. number of processes for various instances of the LU NPB (class B and C). Three kinds of results for shown: (i) Simulated execution times based on *unrolled* time-independent traces (MinI); (ii) Simulated execution times based on *compact* time-independent traces (ScalaTrace); and (iii) Real execution times on the *graphene* cluster described in Section 3. The simulations are instantiated so as to match the characteristics of the real-world cluster. With Time-Independent Trace Replay (i.e., with SimGrid), this simply entails creating an XML file that describes the performance rates of the compute nodes, the latencies and bandwidths of the network links, and the physical network topology. We show results for the real-world cluster only up to 128 processes since this particular cluster has only 144 compute nodes. However, we are able to run simulations of platforms 8 times larger, i.e., of an hypothetical *graphene* cluster that would have around 1,024 compute nodes. We are able to do this because we can obtain valid time-independent traces on composite, non-homogeneous platforms on which multiple processes can run on

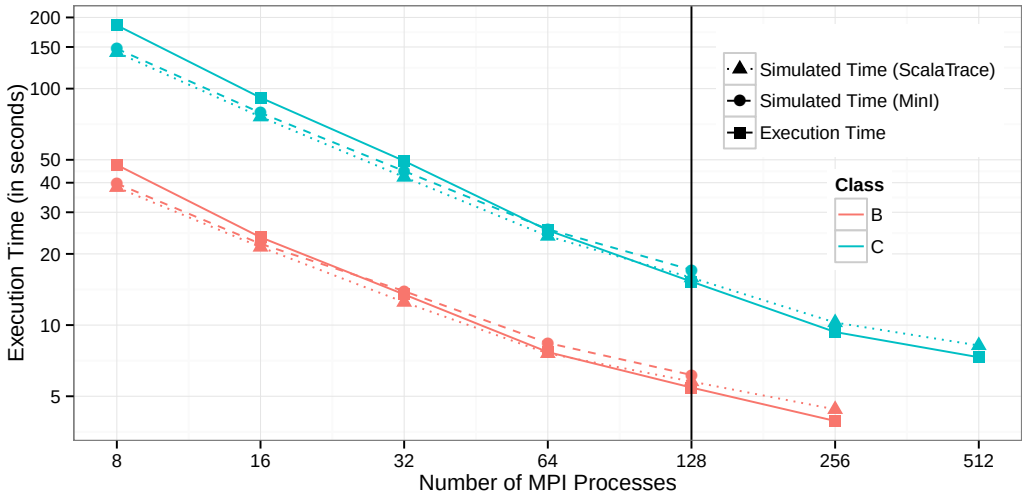


Figure 6: Simulated and actual execution times for class B and C LU benchmark vs. number of processes.

a single compute node (using folding). Being able to simulate large homogeneous platforms for which we would not have been able to obtain a time-dependent trace is an essential motivation for this work. This experiment is at a relatively modest scale (since many homogeneous clusters with more than 1,024 exist), but regardless of the maximum size of the homogeneous physical cluster at hand, or approach can be used to simulate a larger cluster scalably thanks to its use of time-independent compact event traces.

Overall, results in Figure 6 show that all three curves follow the same trends. There is little difference between the two simulated execution times, showing that simulation accuracy is not significantly diminished by using compact ScalaTrace traces, even though some information is lost in these traces compared to MinI traces. We see that simulation is not very accurate for small-scale executions (i.e., with 8 or 16 processes), with an underestimation of the execution by up to 25.6%). Accuracy improves as the number of processes increases (e.g., relative error with respect to the real-world execution is always below 11.5% for 64 processors or more). Since the main objective is to simulate the execution of applications with thousands of processes, this trend is encouraging. A possible source of simulation inaccuracy is given in [23] and is related to trace parsing in the *normal* mode. The emission of some MPI calls might be delayed, which increases execution time. This issue is solved by the *histo* replay mode, but as explained earlier this mode cannot be used for our applications. Further investigation, likely in collaboration with the ScalaTrace team, is necessary to improve the accuracy of the simulated replay even at small scales. Overall, these results show that our approach benefits from and combines the strength of ScalaTrace and Time-Independent Trace Replay so as to push the accuracy and scalability of off-line simulations of MPI applications further.

5 Related Work

Many simulators have been proposed that, like the simulator described in this work, follow the *off-line* simulation approach. For instance, since 2009 five off-line simulators of MPI applications have been described in the literature [13, 21, 24, 12]. These simulators use different simulation models 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 [21, 12]. The alternate approach 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 [19, 25]. We refer the reader to [7] for a detailed discussion of state-of-the-art off-line and on-line simulators.

Many tools have be proposed for the post-mortem performance analysis of parallel applications, typically relying to aggregated statistical information [20, 11, 14, 15]. Performance profiles generated by these tools can be used to detect performance bottlenecks, load imbalance, or undesired behaviors such as late senders/receivers or wait states. The common approach to perform this kind of analysis is to rely on pattern detection and visualization rather than on the replay of the traces. Most of these tools can also produce detailed traces in the OTF2 format [10] which is much more verbose than that of ScalaTrace.

Recent work from the ScalaTrace team is closely related to this work. ScalaJack [1] is a redesign of ScalaTrace that uses aspect-oriented programming to allow for user-customizable instrumentation. As such, it provides means to trace numbers of instructions for CPU bursts as explained in Section 3. A combination of ScalaBenchGen, a ScalaTrace tool to automatically benchmarks representative of event traces, and an off-line simulator called xSim has been proposed in [16]. xSim relies on parallel discrete event simulation. Is is highly scalable provided it is executed on a large number of resources, whereas our proposed approach executes a SimGrid simulation on a single machine. Also, unlike this work it cannot process traces directly, but instead simulates synthetic benchmark generated from the traces. Note that software for these two projects are not publicly available at the time of writing.

6 Conclusion and Future Work

In this paper we have proposed an approach for scalable off-line simulations of MPI applications. We have combined two existing and complementary tools, ScalaTrace and Time-Independent Trace Replay, so as to capitalize on their respective strengths while alleviating several of their limitations. We have explained our motivations, described a usable implementation and presented results that illustrate the feasibility and the benefits of the combination. This work was done in the spirit of Open Science and reproducible research. Both ScalaTrace and SimGrid are free software available online. Code, traces, and experiments are publicly available on http://github.com/frs69wq/ScalaTrace-TI. This work has highlighted a number of open issues, which we plan to address in future work in collaboration with the ScalaTrace development team.

Acknowledgments

This work is partially supported by the SONGS ANR project (11-ANR-INFRA-13). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Srinath Krishna Ananthakrishnan and Frank Mueller. ScalaJack: Customized Scalable Tracing with In-situ Data Analysis. In Silva Fernando M. A., Inês de Castro Dutra, and Vítor Santos Costa, editors, *Proc. of the 20th International Euro-Par 2014 Conference on Parallel Processing*, volume 8632 of *LNCS*, pages 13–25, Porto, Portugal, August 2014. Springer.

[2] David Bailey, E. Barszcz, John Barton, D. Browning, Robert Carter, Leonardo Dagum, Rod Fatoohi, Paul Frederickson, T. Lasinski, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. The nas parallel benchmarks - summary and preliminary results. In *Proc. of Supercomputing '91*, pages 158–165, Albuquerque, NM, November 1991.

[3] Paul Bédaride, Augustin Degomme, Stéphane Genaud, Arnaud Legrand, George S. Markomanolis, Martin Quinson, Mark Stillwell, Frédéric Suter, and Brice Videau. Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. In *Proceedings of the 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Denver, CO, November 2013.

[4] Abhinav Bhatele, Kathryn Mohror, Steve H. Langer, and Katherine E. Isaacs. There Goes the Neighborhood: Performance Degradation due to Nearby Jobs. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, Denver, CO, November 2013. ACM.

[5] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *IJHPCA*, 20(4):481–494, 2006.

[6] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Internation Journal of High Performance Computing and Applications*, 14(3):189–204, 2000.

[7] Henri Casanova, Frédéric Desprez, George S. Markomanolis, and Frédéric Suter. Simulation of MPI Applications with Time-Independent Traces. *Concurrency and Computation: Practice and Experience*, 2014. Available online (early view): http://onlinelibrary.wiley.com/doi/10.1002/cpe.3278/pdf.

[8] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899 – 2917, 2014.

[9] Pierre-Nicolas Clauss, Mark Stillwell, Stéphane Genaud, Frédéric Suter, Henri Casanova, and Martin Quinson. Single Node On-Line Simulation of MPI Applications with SMPI. In *Proc. of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Anchorage, AK, May 2011.

[10] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *Proc. of the ParCo Conference – Applications, Tools and Techniques on the Road to Exascale Computing*, pages 481–490, Ghent, Belgium, September 2011.

[11] Markus Geimer, Felix Wolf, Brian Wylie, and Bernd Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7):375–388, 2009.

[12] Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 78–84, Weimar, Germany, February 2009.

[13] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, pages 597–604, Chicago, IL, June 2010.

[14] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias Müller, and Wolfgang Nagel. The Vampir Performance Analysis Tool-Set. In *Proc. of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 139–155, Stuttgart, Germany, July 2008.

[15] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, WolfgangE. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of the 5th International Workshop Tools for High Performance Computing*, pages 79–91, 2012.

[16] Mahesh Lagadapati, Frank Mueller, and Christian Engelmann. Tools for Simulation and Benchmark Generation at Exascale. In *Proc. of the 7th Parallel Tools Workshop*, Dresden, Germany, September 2013.

[17] Michael Noeth, Prasun Ratn, Franck Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, 2009.

[18] Prasun Ratn, Franck Mueller, Bronis R. de Supinski, and Martin Schulz. Preserving Time in Large-scale Communication Traces. In *Proc. of the 22nd Annual International Conference on Supercomputing*, pages 46–55, 2008.

[19] Rolf Riesen. A Hybrid MPI Simulator. In *Proc. of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

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

[21] Mustafa Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proc. of the 15th International Euro-Par Conference on Parallel Processing*, number 5704 in LNCS, pages 135–148. Springer, August 2009.

[22] Xing Wu and Frank Mueller. ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Programs. *ACM TOPLAS*, 34(1):5, 2012.

[23] Xing Wu and Frank Mueller. Elastic and Scalable Tracing and Accurate Replay of Non-Deterministic Events. In Allen Malony, Mario Nemirovsky, and Samuel Midkiff, editors, *Proc. of the 27th Annual International Conference on Supercomputing*, pages 59–68, Eugene,OR, June 2013. ACM.

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

[25] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.