École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur
de l'Université de Lyon – École Normale Supérieure de Lyon
Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 20 Janvier 2014 par

Georgios Markomanolis

Performance Evaluation and Prediction of Parallel Applications

Abstract

Analyzing and understanding the performance behavior of parallel applications on various compute infrastructures is a long-standing concern in the High Performance Computing community. When the targeted execution environments are not available, simulation is a reasonable approach to obtain objective performance indicators and explore various "what-if?" scenarios. In this work we present a framework for the off-line simulation of MPI applications.

The main originality of our work with regard to the literature is to rely on *time-independent* execution traces. This allows for an extreme scalability as heterogeneous and distributed resources can be used to acquire a trace. We propose a format where for each event that occurs during the execution of an application we log the volume of instructions for a computation phase or the bytes and the type of a communication.

To acquire time-independent traces of the execution of MPI applications, we have to instrument them to log the required data. There exist many profiling tools which can instrument an application. We propose a scoring system that corresponds to our framework specific requirements and evaluate the most well-known and open source profiling tools according to it. Furthermore we introduce an original tool called Minimal Instrumentation that was designed to fulfill the requirements of our framework. We study different instrumentation methods and we also investigate several acquisition strategies. We detail the tools that extract the *time-independent* traces from the instrumentation traces of some well-known profiling tools. Finally we evaluate the whole acquisition procedure and we present the acquisition of large scale instances.

We describe in detail the procedure to provide a realistic simulated platform file to our trace replay tool taking under consideration the topology of the real platform and the calibration procedure with regard to the application that is going to be simulated. Moreover we present the implemented trace replay tools that we used during this work. We show that our simulator can predict the performance of some MPI benchmarks with less than 11% relative error between the real execution and simulation for the cases that there is no performance issue. Finally, we identify the reasons of the performance issues and we propose solutions.

Résumé

L'analyse et la compréhension du comportement d'applications parallèles sur des plates-formes de calcul variées est un problème récurent de la communauté du calcul scientifique. Lorsque les environnements d'exécution ne sont pas disponibles, la simulation devient une approche raisonnable pour obtenir des indicateurs de performance objectifs et pour explorer plusieurs scénarios "what-if?". Dans cette thèse, nous présentons un environnement pour la simulation off-line d'applications écrites avec MPI.

La principale originalité de notre travail par rapport aux travaux précédents réside dans la définition de traces indépendantes du temps. Elles permettent d'obtenir une extensibilité maximale puisque des ressources hétérogènes et distribuées peuvent être utilisées pour obtenir une trace. Nous proposons un format dans lequel pour chaque événement qui apparaît durant l'exécution d'une application, nous récupérons les informations sur le volume d'instructions pour une phase de calcul ou le nombre d'octets et le type d'une communication.

Pour obtenir des traces indépendantes du temps lors de l'exécution d'applications MPI, nous devons les instrumenter pour récupérer les données requises. Il existe plusieurs outils d'instrumentation qui peuvent instrumenter une application. Nous proposons un système de notation qui correspond aux besoins de notre environnement et nous évaluons les outils d'instrumentation selon lui. De plus, nous introduisons un outil original appelé Minimal Instrumentation qui a été conçu pour répondre au besoins de notre environnement. Nous étudions plusieurs méthodes d'instrumentation et plusieurs stratégies d'acquisition. Nous détaillons les outils qui extraient les traces indépendantes du temps à partir des traces d'instrumentations de quelques outils de profiling connus. Enfin nous évaluons la procédure d'acquisition complète et présentons l'acquisition d'instances à grande échelle.

Nous décrivons en détail la procédure pour fournir un fichier de plateforme simulée réaliste à notre outil d'exécution de traces qui prend en compte la topologie de la plateforme cible ainsi que la procédure de calibrage par rapport à l'application qui va être simulée. De plus, nous montrons que notre simulateur peut prédire les performances de certains benchmarks MPI avec moins de 11% d'erreur relative entre l'exécution réelle et la simulation pour les cas où il n'y a pas de problème de performance. Enfin, nous identifions les causes de problèmes de performances et nous proposons des solutions pour y remédier.

Contents

Chapter 1

Introduction

1.1 Motivation

Computational Science is the third scientific way to study problems arising in various domains such as physics, biology, or chemistry. It is complementary to theory and actual experiments and consists in conducting studies *in silico.* This approach makes an heavy use of resources located in computing centers. The efficient execution of a parallel application on a certain massively parallel hardware architecture demands the user to be familiar not only with domain-specific algorithms but also parallel data structures [1, 2]. It is not rare to observe situations where an application can not fully exploit a large number of processors in its execution, thus it does not scale well [3].Such cases can be caused by a not optimized application, the computer system or both. However, through performance evaluation a scientist can observe the performance of an application and can identify bottlenecks [4, 5, 6, 7, 8, 9]. The techniques which constitute the performance evaluation field can be classified into two categories: measurement and modeling techniques [10, 11, 12]. The latter is separated by many scientists into simulation and analytical modeling. A lot of approaches have been presented to evaluate the performance of an application. Some of them correlate the performance of an application with its scientific domain [13]. More precisely, in this approach the authors take data from three domains, the application's working set, the hardware domain of the compute and network devices and the communication domain. The combination of visualizing and correlating these data, provides useful information about the behavior of the parallel application. On the other hand some researchers identify bottlenecks on a specific application which are related to the operating system such as noise, the cluster manager tool and fix them by applying some techniques to decrease the noise [14].

As the number of scientific domains producing results from *in silico* studies increases, the computing centers then have to upgrade their infrastructures in a continuous way. The resources impacted by such upgrades are of different kinds, *e.g.,* computing, network and storage. Moreover each kind of resource grows more complex with each generation. Processors have more and more cores, low latency and high bandwidth network solutions become mainstream and some disks now have access time close to that of memory.

The complexity of the decision process leading to the evolution of a computing center is then increasing. This process often relies on years of experience of system administrators and users. The former knows how complex systems work while the latter have expertise on the behavior of their applications. Nevertheless this process lacks of objective data about the performance of a given candidate infrastructure. Such information can only be obtained once the resources have been bought and the applications can be tested. Any unforeseen behavior can then lead to large but vain expenses. This procedure could be completed with less risks by using simulation techniques. Simulation may help the dimensioning of compute clusters in large computing centers. Through simulations the scientists can understand the behavior of large-scale computers. Moreover it is possible to simulate an application for a platform that is not available yet to observe the performance and decide what would be the best hardware characteristics for the needs of the computer center.

Many universities do not provide large-scale computer systems to the researchers and students. Clusters are too expensive as also their maintenance. The access to another supercomputers could cost also a significant amount of money. In such cases the option to predict the performance of an application through simulation is really important. A scientist can execute simulations without using any resource and be able to validate or not his assumptions. Some times a multi-parameter application is needed to be executed, thus it would take a lot of time to try different parameters for each execution on a supercomputer. The students can study concepts of parallel computing without using a cluster but just their simulator. However, the performance evaluation and prediction are not always easy to be done.

1.2 Issues on Performance Prediction and Evaluation of Parallel Applications

The first difficult step during the performance evaluation of a parallel application is the choose of the appropriate metrics. The metrics are essential for understanding the performance characteristics of applications on modern multicore microprocessor. They can indicate a relation between the performance bottleneck and the hardware. However, if a user is not familiar it can be a difficult task. A tool which can help the user was released by Texas Advanced Computing Center called PerfExpert [15] which basically executes an application four to five times. During each execution the tool uses different metrics and it measures them periodically. As result it proposes which metrics represent any performance issue that occur mainly during the execution of *loops* in the code. Moreover a tool called AutoScope [16] proposes code optimization techniques based on the results from PerfExpert. However, performance measurement usually is not a harmonic task. Different parts of a code can cause non similar behaviors and different metrics could represent their performance. Computational science applications are constituted by iterative methods. A common method is to study the performance of each iteration [17, 18] where in some cases we can observe non similar performance between different iterations. The further analysis of the performance of an application should be achieved through extensive study of the measurements. The main issue with a fine grained measurement is the extra overhead caused by the instrumentation tool. When a user wants to correlate performance measurements with execution time, can not be sure what percentage of these measurements are caused by the instrumentation overhead. This makes the study harder and can hide some other phenomena or create artifacts that are not always easy to explain.

The increase of the hardware complexity raised the difficulty to predict the performance of an application. Different processors are constituted by different cache memory levels and size. The performance of an application can depend on the cache size. Moreover there are many types of networks, Ethernet Gigabit [19], Infiniband [20] etc. Each of them behave different and thus some analytical models should be created in order to simulate the applications. In this work we study Ethernet Gigabit networks. The studied parallel applications are based on the Message Passing Interface (MPI) [21]. The various implementations of the MPI, MPICH [22], and OpenMPI [23], can behave a bit different but always respecting the MPI standards. For example some collective communications can behave similarly but not exactly the same. Thus the simulation of a parallel application is a difficult task where many aspects of the applications should be taken under consideration to predict correctly the performance.

1.3 Contributions

In this work we propose a new execution log format that is independent of time. This format includes volumes of computation (in number of instructions) and communications (in bytes) for each event instead of classical time-stamps. We present in detail an original approach that totally decouples the acquisition of the trace from its replay. Furthermore, several original scenarios are proposed that allow for the acquisition of large execution traces. These scenarios are only possible because of the chosen *time-independent* trace format and they can not be applied with other frameworks. We implement a new profiling tool based on our framework requirements which instruments exactly the necessary information from an application, it saves the measurement data directly into *time-independent* traces and it is more efficient than similar tools. We study the state of the art of the most well known and open sources profiling tools and we evaluate them with regard to our framework requirements. Moreover we justify which ones can be used for our work. We use various instrumentation methods and we evaluate them to show the evolution of the instrumentation overhead. As supporters of the open-science approach, we provide a step-by-step definition of an appliance to acquire execution traces on the Grid'5000 experimental infrastructure. Moreover we have our files available in public to be used from other researchers. Some tools were implemented to extract the *time-independent* traces from well known profiling tools such as TAU, Score-P and gather them into a single node. We evaluate the *time-independent* trace acquisition framework to present the overhead, the advantages and disadvantages. The trace replay tool is built on top of a fast, scalable and validated simulation kernel. The implemented simulator tool is analyzed in detail and we describe how to use it by covering all the procedure from calibrating the simulator till the execution of a simulation. Finally, we predict through simulation the performance of some non adaptive MPI applications with small relative error between the real execution and simulation. In the cases that there are some performance issues, we investigate and figure out what causes any performance issue.

1.4 Organization of the Document

The remaining of this document is organized as follows. Chapter 2 presents an introduction to some related topics. We present some benchmarks, we analyze some of them and we indicate which ones we use. Afterwards we present some techniques to evaluate the performance of an application such as the interface to access the hardware counters of the processors is introduced as also some instrumentation methods which are used to measure the performance of an application. We analyze the tracing, profiling, and statistical profiling methods and we explain the advantages and disadvantages.

In Chapter 3 we explore the *time-independent* trace acquisition framework. In the beginning we explain the motivation for this work, we introduce the *time-independent* trace format and we analyze the advantages. We provide more details about the whole framework and we mention the trace replay concept. There is an extensive analysis of the instrumentation of an application where we evaluate many well known tools with a score system which respects the framework requirements. We present our profiling library which can instrument an application with less instrumentation overhead and memory requirements. The instrumentation methods which we used are presented and compared. Moreover the acquisition of *time-independent* traces is presented with all the execution modes that can be used and also instructions of how to reproduce the experiments on Grid'5000 platform. We have implemented some tools to extract *time-independent* traces from well known profiling tools such as TAU and Score-P as also gather them from all the participating nodes into a single node. Finally we evaluate the acquisition procedure to present the overheads of our framework.

Chapter 4 presents the *time-independent* trace replay. First of all the components of the simulation are introduced with the calibration procedure where is explained to the user how to calibrate the simulator. Afterwards the simulation backends that were used for the implementation of the various versions of our simulator are presented with their differences. At last we present the simulation accuracy of some MPI applications and we show that our framework predicts their performance.

Finally in Chapter 5 we present the conclusions and future perspectives.

Chapter 2

Background on Performance Evaluation

Chapter 1, analyzing and understanding the performance of a parallel application is a complex task. Some parallel applications may scale well when increasing the number of the processors while others may suffer from scalability issues. In general we could divide applications in two categories, those that fully exploit the performance of a computer system and to the ones that solve a real problem. Performance evaluation is usually performed thanks to various instrumentation methods when an application is executed on a real platform. However, there are two other approaches that provide insight about the behavior of an application: emulation and simulation. The purpose of this chapter is to introduce some basic concepts for the better understanding of the remaining of this document.

This Chapter is organized as follows. Section 2.1 presents some micro-benchmarks and benchmarks. We explain why they are used and how useful where for our case. In Section 2.2 we detail the interface which provides access to the hardware counters of processors to measure specific metrics. Moreover we investigate available instrumentation methods for the performance measurement of an application. Finally in Section 2.3 we detail how to assess the performance on applications on resources. Three methods are analyzed with their advantages and drawbacks.

2.1 Evaluating the Performance of Resources

2.1.1 Micro-benchmarks

A micro-benchmark is a test program designed to measure the performance of a very small and specific piece of code. Micro-benchmarks tend to be synthetic kernels and can measure a specific aspect of computer system. Through them we can understand the behavior of major kernel characteristics.

Netgauge [24] is a high-precision network parameter measurement tool. It supports micro-benchmarking of many different network protocols and communication patterns. The main focus lies on accuracy, statistical analysis and easy extensibility. One of the included micro-benchmarks that we used is OS Noise which measures the noise caused by the Operating System.

OS Noise supports three different micro-benchmarking methods. Selfish Detour (selfish) is a modified version of the selfish detour benchmark proposed in [25]. The micro-benchmark runs in a tight loop and measures the time for each iteration. If an iteration takes longer than the minimum times a particular threshold, then the time-stamp (detour) is recorded. The micro-benchmark runs until it records a predefined number of detours. Fixed Work Quantum (FWQ), performs a fixed amount of work multiple times and records the time it takes for each run. Due to the fixed work approach of FWQ, the data samples can be used to compute useful statistics (mean, standard deviation and kurtosis) of the scaled noise. These statistics then can be used to characterize and categorize the amount of noise in a hardware and software environments.

During the execution of the Fixed Time Quantum (FTQ) micro-benchmark, a very small work quantum is performed until a fixed time quantum has been exceeded and for each iteration it records how many workload iterations were carried out. If there is no noise this number should be equal for every sample. When there is noise this number varies. Because the start and end time of every sample is defined, the periodicity of noise can be analyzed with this method. Finally the FTQ and FWQ micro-benchmarks are proposed from the list of benchmarks for the sequoia supercomputer [26]. In our studies, we used the FTQ micro-benchmark and the noise caused by the Operating System on different Grid'5000 clusters was computed to be at most 0.14% of the execution time. It is important to compute the percentage of the noise caused by external factors such as the operating system to know if the measurement data are biased or not. The specific amount of noise is not significant on the considered execution platforms. However, this is not a generic statement. If a scientist wants to apply some statistic methods on some performance measurement data, then the OS noise may probably be taken into account.

SKaMPI [27] is a micro-benchmark for implementations of the MPI standard. Its purpose is the detailed analysis of individual MPI operations. SkaMPI can be configured and tuned in many ways: operations, measurement precision, communication modes, packet sizes, number of processors used, etc. Moreover SkaMPI provides measurement mechanisms that combine accuracy, efficiency, and robustness. Each measurement is derived from multiple measurements of single calls to particular communication pattern with regard to the requested accuracy. SkaMPI handles systematic and statistical errors. The systematic error is caused by the measurement overhead of all calls, it is usually small and can be corrected by subtracting the duration of an empty measurement and the warm-up of the cache by a dummy call to the measurement routine before actually starting to measure. The statistical error is checked through the control of the standard error and the average execution time. We used this micro-benchmark to calibrate our simulation as presented in Section 4.3.

2.1.2 Benchmarks

Benchmarks are software packages which can make use of almost all resources in the system. Various kernels, synthetic programs and application level workloads, are called benchmarks. A subset of a benchmark can be a micro-benchmark. One main reason for benchmarking a computer system is to measure the performance and probably compare with other systems. However, we should be familiar to the kind of applications that are going to be executed on those systems to execute benchmarks that are representative of these applications. One of the most well-known benchmarks is High Performance Linpack (HPL) [28] which measures the floating point execution rate for solving a linear system of equations in double precision arithmetic on distributed-memory computers. It is used to rank supercomputers in the Top5000 [29] list which is constituted by the 500 fastest supercomputers of the world. However, this benchmark does not measure the system performance for a significant collection of important science and engineering applications. HPL rankings of computer systems are no longer strongly correlated to real application performance, especially for the broad set of HPC applications governed by differential equations, which tend to have much stronger needs for high bandwidth and low latency networks, and tend to access data using irregular patterns. Thus a new benchmark suite has been proposed, named High Performance Conjugate Gradient (HPCG) [30]. HPCG is composed of computations and data access patterns commonly found in applications.

The NAS Parallel Benchmarks (NPB) [31] have been developed at NASA Ames Research Center to study the performance of parallel supercomputers. This is a set of eight benchmark problems, each focusing on some important aspect of highly parallel supercomputing for aerophysics applications. The eight problems consist of five “kernels” and three “simulated computational fluid dynamics (CFD) applications”. The five kernels are relatively compact problems, each of which emphasizes a particular type of numerical computation. The simulated applications combine several computations in a manner that are similar to the actual order of execution in certain important CFD application codes.

Each benchmark can be executed for 7 different *classes*, denoting different problem sizes: S (the smallest), W, A, B, C, D, and E (the largest). For instance, between class A and class C, the problem size is increased by almost 4 times from one class to the next and from class C to class E the corresponding increase is almost 16 times.

The five kernels are the following:

EP: An “embarrassingly parallel” kernel. It provides an estimate of the upper achievable limits for floating point performance. This kernel is constituted only from computation and there is communication only during the verification of the solution. It generates pairs of Gaussian random deviates according to a specific scheme described in [31] and tabulates the number of pairs in successive square annuli;

MG: A simplified “multigrid” kernel. It requires highly structured long distance communication and tests both short and long distance data communication. Moreover it is memory intensive;

CG: A “conjugate gradient” method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix with a random pattern of non zeros thanks to the inverse power method. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication;

FT: This kernel performs the essence of many “spectral” codes. It is a rigorous test of long-distance communication performance. It solves numerically a certain partial differential equation (PDE) using forward and inverse FFTs;

IS: A large “integer sort”. This kernel performs a sorting operation that is important in “particle method” codes. It tests both integer computation speed and communication performance.

The three simulated CFD applications are the following:

BT: Solves a synthetic system of nonlinear PDEs using a block tridiagonal algorithm;

LU: Solve a synthetic system of nonlinear PDEs using a symmetric successive over-relaxation algorithm;

SP: Solve a synthetic system of nonlinear PDEs using a scalar pentadiagonal algorithm.

Finally there is a benchmark called Data Traffic (DT) which is communication intensive. It uses quad-trees (black hole and white hole) and binary shuffle as task graphs. It captures core functionality of distributed data acquisition and processing systems.

In out studies, we focused on the EP, DT, LU, and CG benchmarks. We used EP benchmark because it is computation intensive, DT is communication intensive, and the last two ones mix computations and communications.

2.2 Evaluating the Performance of Applications

Depending on the application there are various approaches to understand its behavior. First of all we should measure some characteristics of the studied application. The simplest performance metric is time. This way we can observe in which parts of the code the most time is spent and focus on them to further analyze the performance of the application. However, when the application is too complex, the time may not provide enough insight on performance. Then we have to instrument the application to measure other metrics and do further analysis which will guide us to the reasons that the application may not scale or suffer from other performance issues. For example we may want to know if some performance issues of the studied application are influenced by the computation, memory usage, communication between the processes or other factors.

2.2.1 Hardware Counters

The Performance Application Programming Interface (PAPI) [32] is a middleware that provides a consistent and efficient programming interface for the performance hardware counters found in major microprocessors. These counters exist as a small set of registers that count events, occurrences of specific signals related to the processor's functions. Monitoring these events facilitates the correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. PAPI provides a standard set of over 100 events for application performance tuning but not all of them are necessary available on every processor. Each event can be mapped to either single or linear combinations of native counters on each architecture. Hardware performance counters can provide insight into whole program timing, cache behaviors, branch behaviors, memory and resource contention and access patterns, pipeline stalls, floating point efficiency, instructions per cycle, subroutine resolution, process or thread attribution. With the introduction of Component PAPI, or PAPI-C, PAPI has extended its reach beyond the CPU and can now monitor system information on a range of components from CPUs to network interface cards to power monitors and more. Figure 2.1 shows the internal design of PAPI.

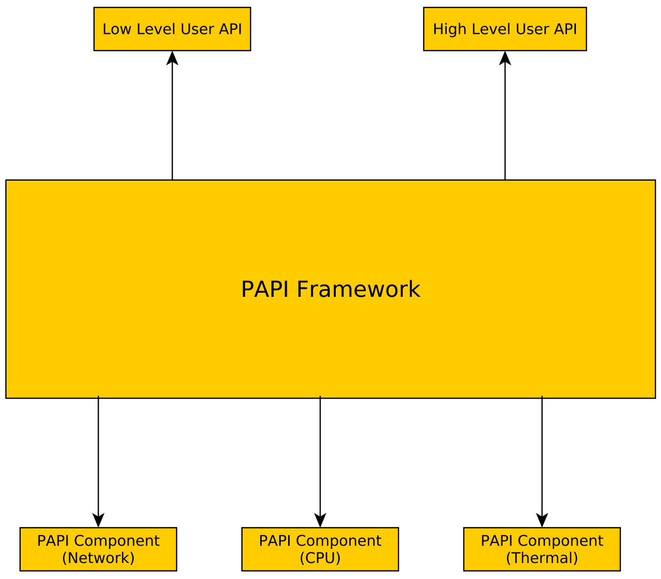


Figure 2.1: Internal deisgn of PAPI architecture.

PAPI provides two interfaces, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The Component Layer defines and exports a machine independent interface to machine dependent functions and data structures. These functions are defined in components, which

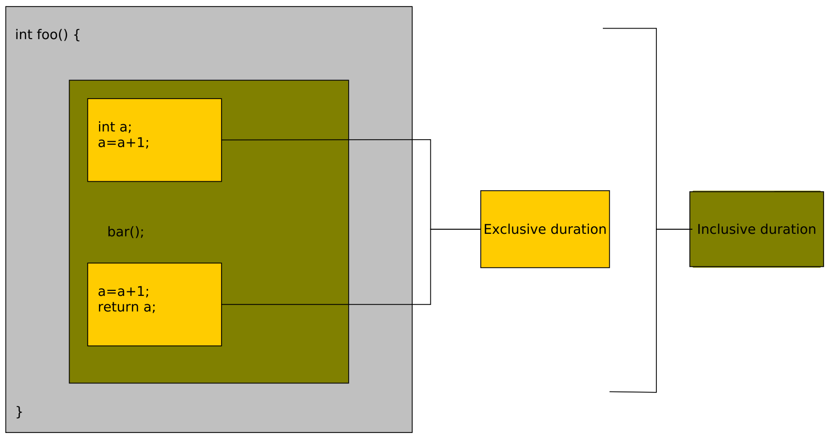


Figure 2.2: Exclusive and inclusive duration of a routine during profiling.

may use kernel extensions, operating system calls, or assembly language to access the hardware performance counters on a variety of subsystems. PAPI uses the most efficient and flexible of the three, depending on what is available. An instrumentation method can read the values of the hardware counters during the execution of an instrumented application.

2.2.2 Tracing

Observing the time-dependent performance behavior of an application requires to trace its execution. Time-stamped event traces capture what performance is being achieved when and where in the program's execution. With the advent of the distributed processing, the tracing of software events became a dominant technique for performance monitoring [33]. Moreover software event tracing is the foundation of many parallel debugging techniques [34, 35]. Tracing has also been used to study the performance of an application [36, 37, 38]. During the tracing of an application all the executed events are logged except if it is declared otherwise. Each event is associated to a timestamp and includes information related to the executed process and optional data depending on the type of the event. Event logging involves several mechanisms. This includes software instrumentation to generate events and run-time system support to store the events in a trace buffer. One issue observed on many clusters is the time synchronization across the participating nodes during an execution. After the execution of a parallel application each process creates a file that contains all the occurred events. The advantage of tracing is that we know the performance of an application at each moment for all the execution. For example if an application is constituted by iterative methods, we can extract the performance of each single iteration. However, this procedure adds overhead to the execution time and may require large hard disk space. The tracing procedure is explained in 3.1.

2.2.3 Profiling

The most common way to analyze the performance of an application is the profiling of routine execution. Basically, profiling produce statistics for each routine call by the application indicating the number of times this routine is called and the time spent executing its code, including (or not) the time spent in calls to other routines. Figure 2.2 shows the exclusive and inclusive duration of a routine. The exclusive duration is the amount of time spent while within the *foo* function excluding the time spent in function *bar* which is called by *foo*. The inclusive duration is the amount of execution time spent in *foo*, including the spent in all the busbroutines called by *foo*.

Accesses to times (or to hardware performance counters) are trivially made by inserting instrumentation probes at the beginning and ending of each routine as it can be seen in Figure 2.3 where a routine named *main*, calls a function *foo* with different argument values.

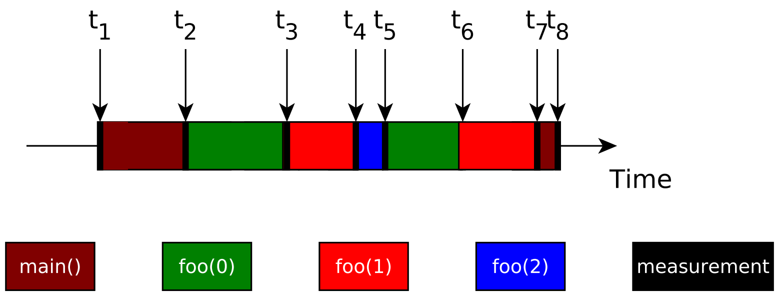


Figure 2.3: Instrumentation during profiling.

The advantage of profiles is that they can be calculated at run-time, require only a small amount of storage, and they can provide a generic glimpse of the application's performance. However, we do not know exactly where a performance issue occurs. In the case of an iterative method, if there is a performance issue with some specific iterations we can not observe them. Note that profiles can be extracted through tracing by post-processing the produced trace, but the opposite is not possible.

2.2.4 Statistical Profiling

Statistical profiling or sampling probes the target program's counter at regular intervals using operating system interrupts as we can see in Figure 2.4 where again a routine named *main* calls a routine *foo* with different arguments. Sampling profiles are less accurate and specific than full profiling, but allow the target program to run without inducing an instrumentation overhead. Usually this method is used for applications with long execution time where there is no need for detailed performance analysis.

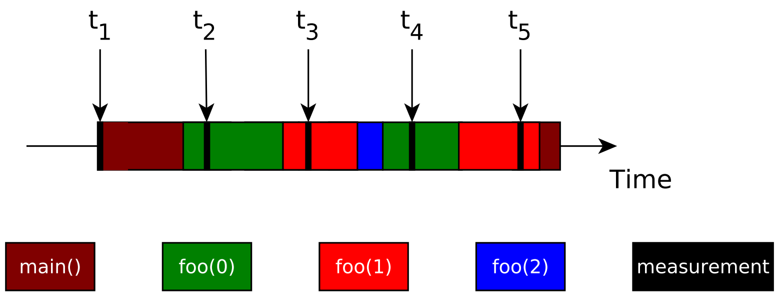


Figure 2.4: Adding probes at regular intervals for statistical profiling.

2.3 Assessing the Performance of Applications on Resources

A user can not be aware of the performance of an application on specific resources if there is no interaction between them. This interaction can be done either by executing the application on the real platform, executing the application on a virtual environment that corresponds to the resources through emulation or simulating the application to predict its performance. These three approaches constitute the main ways to assess the performance of the applications on the resources and are detailed hereafter.

2.3.1 Experiments

Large-scale computer experiments are becoming increasingly important in science. An experiment can be executed on the available resources. Thus this is the most accurate approach for observing the performance of an application on the specific resources. Moreover it is possible through the experiment to observe some unexpected phenomena that a user was not aware of. Experiments also are important for validating theories. Experimentation in a real environment is expensive, because it requires a significant number of resources available for a large amount of time. It also costs time, as it depends on the application to be actually deployed and executed under different loads, and for heavy loads long delays in execution time can be expected. Moreover the experiments are not repeatable, because a number of variables that are not under control of the tester may affect experimental results, and elimination of these influences requires more experiments, making it even more time and resource expensive.

For all our direct experiments we used the Grid'5000 [39, 40] testbed. In 2003, several teams working around parallel and distributed systems designed a platform to support experiment-driven research in parallel and distributed systems. This platform, called Grid'5000 and opened to users since 2005, was solid enough to attract a large number of users. According to [41], this platform has led to a large number of research results: 575 users per year, more than 700 research papers, 600 different experiments, 24 ANR projects and 10 European projects and 50 PhD. Grid'5000 is located mainly in France (see Figure 2.5), with one operational site in Luxembourg and a second site, not ready yet, in Porto Alegre, Brazil. Grid'5000 provides a testbed supporting experiments on various types of distributed systems (high-performance computing, grids, peer-to-peer systems, cloud computing, and others), on all layers of the software stack. The core testbed currently comprises 10 sites. Grid'5000 is composed of 24 clusters, 1,169 nodes, and 8,080 CPU cores, with various generations of technology (Intel (56.2%), AMD (43.8%), CPUs from one to 12 cores, Myrinet, Infiniband and 2 GPU clusters). A dedicated 10 Gbps backbone network is provided by RENATER (the French National Research and Education Network).

From the user point of view, Grid'5000 is a set of sites with the same software environment. It provides to the users heterogeneity which can be controlled during the experiments. The main steps identified to run an experiment are (1) reserving suitable resources for the experiment and (2) deploying the experiment on the resources. The resource scheduler on each site is fed with the resource properties so that a user can ask for resources describing the required properties (*e.g.,* 16 nodes connected to the same switch with at least 8 cores and 16 GB of memory). Once enough resources are found, they can be reserved either for exclusive access at a given time or for exclusive access when they become available. In the latter case, a script is given at reservation time, as in classical batch scheduling systems.
Several tools are provided to facilitate experiments. Most of them were originally developed specifically for Grid'5000. Grid'5000 users select and reserve resources with the OAR batch scheduler [42, 43]. Users can install their own system image on the nodes (without any virtual-

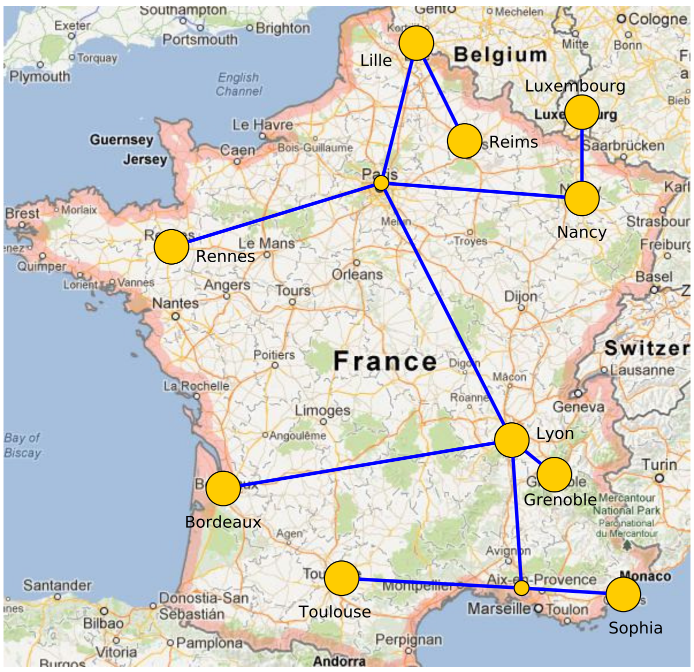


Figure 2.5: Presentation of the sites and the network of the Grid'5000 testbed.

ization layer) using Kadeploy [44]. Moreover the users can deploy a preconfigured environment on a node, called production environment.

2.3.2 Emulation

Emulation [45, 46] is an experimental approach allowing to execute real applications in a virtual environment. This allows us to reach the experimental settings wanted for an experiment. Most of the emulation solutions rely on a heavy infrastructure and mandate the use of a cluster and an emulation layer to reproduce the wanted environment. Emulators let unmodified applications run in a specific environment where relevant system calls are intercepted and mediated. So basically an emulator is used to replace a system and execute an application on the virtual system.

There are many frameworks which propose their own emulator. The Emulab-Planetlab portal [47] allows to use the interface of the Emulab [48] emulator to access the Planetlab [49] experimental platform, to emulate applications based on this platform. This helps the scientists to emulate applications considering an existing platform. EMUSIM [50] integrates emulation and simulation environments in the domain of cloud computing applications and provides a framework for increased understanding of this type of systems. Furthermore, Netbed [48] is a platform based on Emulab that combines emulation with simulation to build homogeneous networks using heterogeneous resources. Another emulator is Distem [51] which builds distributed virtual experimental environments. This tool is easily deployed on Grid'5000 platform and it can emulate a platform composed of heterogeneous nodes, connected to a virtual network described using a realistic topology model. An open problem is still the mapping of a virtual platform on a set of physical nodes. In the case that many virtual machines are on the same physical node there are limitations on the inter-node bandwidth and communication time that should be taken into account during the designing of the experiment.

2.3.3 Simulation

Simulation is a popular approach to obtain objective performance indicators on platforms that are not at one's disposal. Simulation has been used extensively in several areas of computer science for decades, *e.g.,* for microprocessor design and network protocol design. Moreover simulation can be used to evaluate a concept in early development stage. Simulations can also be used to study the performance behavior of an application by varying the hardware characteristics of a hypothetical platform. However, an application should be properly modeled, otherwise the simulation will not be accurate. A simulation is reproducible in comparison to real experiments or emulation. Furthermore, access to large-scale platforms is typically costly (possible access charges to the user, electrical power consumption). The use of simulation can not only save time but also money and resources.

The Message Passing Interface (MPI) is the dominant programming model for parallel scientific applications. Given the role of the MPI library as the communication substrate for application communication, the library must ensure to provide scalability both in performance and in resource usage. MPI development is active with an important role in the community and the new version MPI-3.0 [52] was released on September 2012 with intention to improve the scalability, performance and some other issues. 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" [53, 54, 55, 56, 57, 58, 59], and *off-line simulation* or "post-mortem simulation" [60, 61, 62, 63, 64]. In *on-line simulation* the actual code, with no or only marginal modification, is executed on a *host platform* that attempts to mimic the behavior of a *target platform*, *i.e.,* a platform with different hardware characteristics. 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 [65, 66], the simulation time can be prohibitively high when applications are not regular or have data-dependent behavior. Part of the instruction stream is then intercepted and passed to a simulator. LAPSE is a well-known on-line simulator developed in the early 90's [53] (see therein for references to precursor projects). In LAPSE, the parallel application executes normally but when a communication operation is performed a corresponding communication delay is simulated for the target platform using a simple network model (affine point-to-point communication delay based on link latency and bandwidth). MPI-SIM [54] builds on the same general principles, with the addition of I/O subsystem simulation. A difference with LAPSE is that MPI processes run as threads, a feature which is enabled by a source code preprocessor. Another project similar in intent and approach is the simulator described in [57]. The BigSim project [56], unlike MPI-SIM, allows the simulation of computational delays on the target platform. This makes it possible to simulate "what if?" scenarios not only for the network but also for the compute nodes of the target platform. These computational delays are based either on user-supplied projections for the execution time of each block of code, on scaling measured execution times by a factor that accounts for the performance differential between the host and the target platforms, or based on sophisticated execution time prediction techniques such as those developed in [55]. The weakness of this approach is that since the computational application code is not executed, the computed application data is erroneous. Data-dependent behavior is then lost. This is acceptable for many regular parallel applications, but is more questionable for irregular applications (*e.g.,* branch-and-bound or sparse matrices computations). Going further, the work in [58] uses a cycle-accurate hardware simulator of the target platform to simulate computation delays, which leads to a high ratio of simulation time to simulated time.

One difficulty faced by all above MPI-specific on-line simulators is that the simulation, because done through a direct execution of the MPI application, is inherently distributed. Parallel discrete event simulation raises difficult correctness issues pertaining to process synchronization. For the simulation of parallel applications, techniques have been developed to speed up the simulation while preserving correctness (*e.g.,* the asynchronous conservative simulation algorithms in [67], the optimistic simulation protocol in [56]). A solution could be to run the simulation on a single node but it requires large amounts of CPU and RAM resources. For most aforementioned on-line approaches, the resources required to run a simulation of an MPI application are commensurate to those needed to run the application on a real-world platform. In some cases, those needs can even be higher [58, 59]. One way to reduce the CPU needs of the simulation is to avoid executing computational portions of the application and simulate only expected delays on the target platform [56]. Reducing the need for RAM resources is more difficult and if the target platform is a large cluster, then the host platform must then be a large cluster. SMPI [66], which builds on the same simulation kernel as this work, implements all the above techniques to allow for efficient single-node simulation of MPI applications.

An alternate approach is *off-line* simulation in which a log, or trace, of MPI communication events (time-stamp, source, destination, data size) is first obtained by running the application on a real-world platform. A simulator then replays the execution of the application as if it were running on a *target platform*. This approach has been used extensively, as shown by the number of trace-based simulators described in the literature since 2009 [60, 61, 62, 63, 64]. The typical approach is to decompose the trace in time intervals delimited by the MPI communication operations. The application 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. Computation delays are typically computed by scaling the durations of the CPU bursts in the trace [61, 62, 64]. Network delays are computed based on a simulation model of the network. 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.

One limitation of these off-line simulators is that trace acquisition is not scalable. To simulate the execution of an application on a platform of a given scale, the trace must be acquired on a homogeneous platform of that same scale, so that time-stamps are meaningful. In some cases, extrapolating a smaller trace to larger numbers of compute nodes is feasible [60], but not generally applicable. Furthermore, the use of time-stamps requires that each trace be accompanied with a description of the platform on which it was obtained, so as to allow meaningful scaling of computation delays. In this work we address the trace acquisition scalability limitation by using time-independent traces.

Another limitation is that these simulators typically use simplistic network models, because they are straightforward to implement and scalable. Possible simplifications include: not using a network model but simply replay original communication delays [64]; ignoring network contention because it is known to be difficult and costly to simulate [60, 63, 68]; using monolithic performance models of collective communications rather than simulating them as sets of point-to-point communications [61, 64, 69]. Other simulators opt for accurate packet-level simulation, which is not scalable and leads to high simulation times [62].

One well-known challenge for off-line simulation is the large size of the traces, which limits the scalability of trace acquisition and can prevent running the simulation on a single node. Mechanisms have been proposed to improve scalability, including compact trace representations [61] and replay of a judiciously selected subset of the traces [63].

In this work, we focus on off-line simulation, by which a trace of a previous execution of the application is “replayed” on a simulated platform.

SimGrid

SimGrid [70] is a scientific instrument to study the behavior of large-scale distributed systems such as Grids, Clouds, HPC or P2P systems. It can be used to evaluate heuristics, prototype applications or even assess legacy MPI applications. SimGrid was conceived as a scientific instrument, thus the validity of its analytical models was thoughtfully studied [71], ensuring their realism.

Some of the key features of SimGrid are:

– A scalable and extensible simulation engine that implements several validated simulation models, and that makes it possible to simulate arbitrary network topologies, dynamic compute and network resource availabilities, as well as resource failures;
– High-level user interfaces for distributed computing researchers to quickly prototype simulations either in C or in Java;
– APIs for distributed computing developers to develop distributed applications that can seamlessly run in “simulation mode” or in “real-world mode”.

SimGrid offers four user interfaces as it can be seen in Figure 2.6.

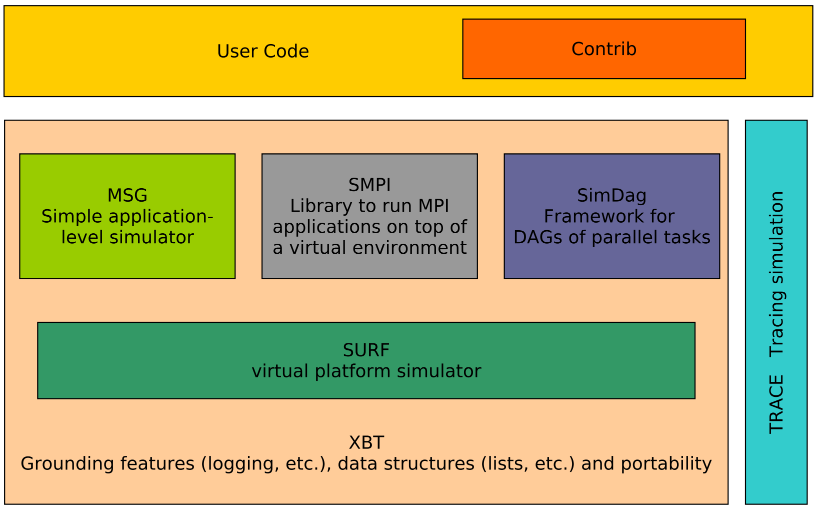

User Code
Contrib
MSG
Simple application-level simulator
SMPI
Library to run MPI applications on top of a virtual environment
SimDag
Framework for DAGs of parallel tasks
SURF
virtual platform simulator
XBT
Grounding features (logging, etc.), data structures (lists, etc.) and portability
TRACE Tracing simulation

Figure 2.6: SimGrid components.

MSG module provides an API for the easy prototyping of distributed applications by letting users focus solely on algorithmic issues. Simulations are constructed in terms of concurrent processes, which can be created, suspended, resumed and terminated dynamically, and can synchronize by exchanging data. Moreover it proved perfectly usable in other contexts, such as desktop grids [72]. An important difference between MSG and SMPI, is that all simulated processes are in the same address space. This simplifies development of the simulation by allowing convenient communications via global data structures. In other words, while MSG can accurately simulate the interactions taking place in a distributed application, including communication and synchronization delays, the simulated application can be implemented with the convenience of a single address space.

SMPI [66] is a simulator for MPI applications. Its first version provided only on-line simulation, i.e., the application is executed but part of the execution takes place within a simulation component. In Chapter 4 we present our off-line simulator based on SMPI backend. SMPI simulations account for network contention in a fast and scalable manner. SMPI also implements an original and validated piece-wise linear model for data transfer times between cluster nodes. Finally SMPI simulations of large-scale applications on large-scale platforms can be executed on a single node thanks to techniques to reduce the simulation's compute time and memory footprint.

SimDag is designed for the investigation of scheduling heuristics for applications as task graphs. SimDag allows to prototype and simulate scheduling heuristics for applications structured as task graphs of (possibly parallel) tasks. With this API one can create tasks, add dependencies between tasks, retrieve information about the platform, schedule tasks for execution on particular resources, and compute the DAG execution time.

SURF considers the platform as a set of resources, with each of the simulated concurrent tasks utilizes some subset of these resources. SURF computes the fraction of power that each resource delivers to each task that uses it by solving a constrained maximization problem: allocate as much power as possible to all tasks in a way that maximizes the minimum power allocation over all tasks, also called Max-Min fairness. SURF provides a fast implementation of Max-Min fairness. Via Max-Min fairness, SURF enables fast and realistic simulation of resource sharing (e.g., TCP flows over multi-link LAN and WAN paths, processes on a CPU). Furthermore, it enables trace-based simulation of dynamic resource availability. Finally, simulated platform configurations are described in XML using a syntax that provides an unified abstract basis for all other components of the SIMGRID project (MSG and SMPI).

XBT is a "toolbox" module used throughout the software, which is written in ANSI C for performance. It implements classical data containers, logging and exception mechanisms, and support for configuration and portability.

Now that the basic concepts of our work are analyzed, we can continue in the next chapter with the presentation of the *time-independent* trace acquisition framework.

Chapter 3

Time-Independent Trace Acquisition Framework

All the off-line MPI simulators that were reviewed in 2.3.3, rely on timed traces, *i.e.,* each occuring event is associated to a time-stamp. The duration of each computation or communication operation is then defined by the interval between two time-stamps. When it simulates an execution on a target platform that is different of the one used to get the trace, a simulator then has to apply a correction factor to these intervals. This implies to know precisely what are the respective characteristics of the host and target platforms. In other words, each execution trace must come with an accurate description of how it has been acquired. Finally, determining the right scaling factor can be tedious depending on the degree of similarity of both platforms.

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

Another advantage of this approach is the lack of clock synchronization across all the participating nodes. This can be done because we do not need to extract the durations of the communications that occur during the execution of the parallel application. This is important as there can be many issues about clock synchronization [73] that are sometimes very difficult to identify.

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

Furthermore it is mandatory to know the sequence of the events, otherwise it is not possible to create the list of the *actions* which constitutes the *time-independent* traces. The only instrumented mode that provides analytic measurement data, is the tracing one. Thus we need a profiling tool that provides tracing functionalities.

The left hand side of Figure 3.1 shows a simple computation executed on a ring of four processes. Each process computes one million instructions and sends one million bytes to its neighbor. The right hand side of this figure displays the corresponding *time-independent* trace.

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

```
0 init
0 compute 1e6
0 send 1 1e6
0 recv 3
0 finalize

1 init
1 recv 0
1 compute 1e6
1 send 2 1e6
1 finalize

2 init
2 recv 1
2 compute 1e6
2 send 3 1e6
2 finalize

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

Figure 3.1: MPI code sample of some computation on a ring of processes (left) and its equivalent time-independent trace (right).

Note that, depending on the number of processes and the number of actions, it may be preferable to split the time-independent trace to obtain one file per process. Indeed Figure 3.1 shows a simple example where all the actions can be included in the same file. However, typical MPI applications are likely to be executed by dozen of processes that perform millions of actions each. Then it is often easier to split (or actually not merge as it will be explained later) the trace to have one file per process, like in the Figure 3.2.

```
0 init
0 compute 1e6
0 send 1 1e6
0 recv 3
...
0 finalize
```

```
1 init
1 recv 0
1 compute 1e6
1 send 2 1e6
...
1 finalize
```

...

```
127 init
127 recv 2
127 compute 1e6
127 send 0 1e6
...
127 finalize
```

Figure 3.2: Example of a split time-independent trace corresponding to a larger application execution with 128 processes.

In this section we mention the main steps of the acquisition process. As it is shown in the Figure 3.3, in the beginning it is mandatory to instrument an application in order to record all the events that will occur during its execution. Afterwards, we execute the instrumented application on a parallel platform. Then from the acquired traces, we extract the *time-independent* traces, which are gathered into a single node.

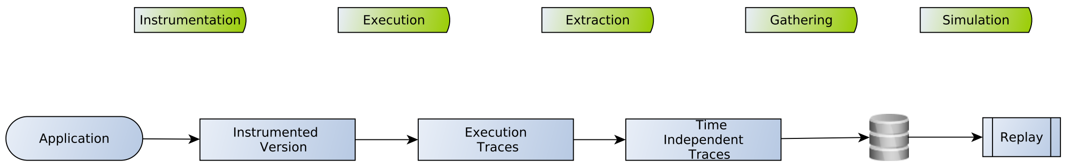


Figure 3.3: Overview of the Time Independent Trace Replay Framework.

In order to replay the *time-independent* traces, the following requirements should apply. For the computation actions, we know the amount of instructions, and the processing speed of the CPU, so we can determine the time to complete the computation. However, some algorithms should be implemented for the communication actions in order to calculate the time needed to transfer the message from the sender to the receiver in the case of a point-to-point communication and similar for other type of communications. As there are no time-stamps in the execution traces, we should take under consideration the latency and the bandwidth of the network. Furthermore it is really important to simulate also the contention that may occur during the exchange of the data between the nodes. Thus, if a simulation model can handle all the mentioned factors, then it can use the *time-independent* traces in order to predict the performance of the corresponding application.

The remaining of this Chapter is organized as follows. Section 3.1 describes the instrumentation of an application and evaluates some tracing tools. instrumentation methods. In Section 3.2, we evaluate the quality of different instrumentation methods in terms of overhead and skew. Section 3.3 details and evaluates the subsequent parts of the proposed trace acquisition framework. This presentation is illustrated by the deployment of our framework on the Grid'5000 platform. Finally, Section 3.4 concludes this chapter. Note that the replay of the *time-independent* traces will be covered in the next chapter.

3.1 Instrumenting an Application

Instrumenting an application to obtain an execution trace consists in adding probes to its code. For instance, such probes can be accesses to hardware counters or logs of the beginning and end of a function. Thanks to them, all the actions, *i.e.,* computations or communications, can be traced in a chronological order. For every function enter/exit events are saved. The information which is logged for each event are the time-stamp, event type and event-related information such as the sender of a message, the receiver, etc. A crucial difference with other instrumentation modes such as profiling and statistical profiling, is that we know the chronologically ordered sequence of the events that occur during the execution.

The left-hand side of Figure 3.4 shows an example of what occurs during tracing. We have two processes, called A and B. The first one calls the function *foo*, sends a message to process B, and then exits. Similarly process B calls the function *bar*, receives a message from process A and exits. However, it is mandatory to monitor the time through a synchronization mechanism as we need to know when a communication starts and ends.

On the right-hand side of Figure 3.4, we can observe the sequence of events per process where the name of the functions and information on events are locally encoded. In the case of merged traces, we can observe the global sequence of events for both processes thanks to the recorded time-stamps and the synchronization. The unification takes place to globally declare functions and other important information.

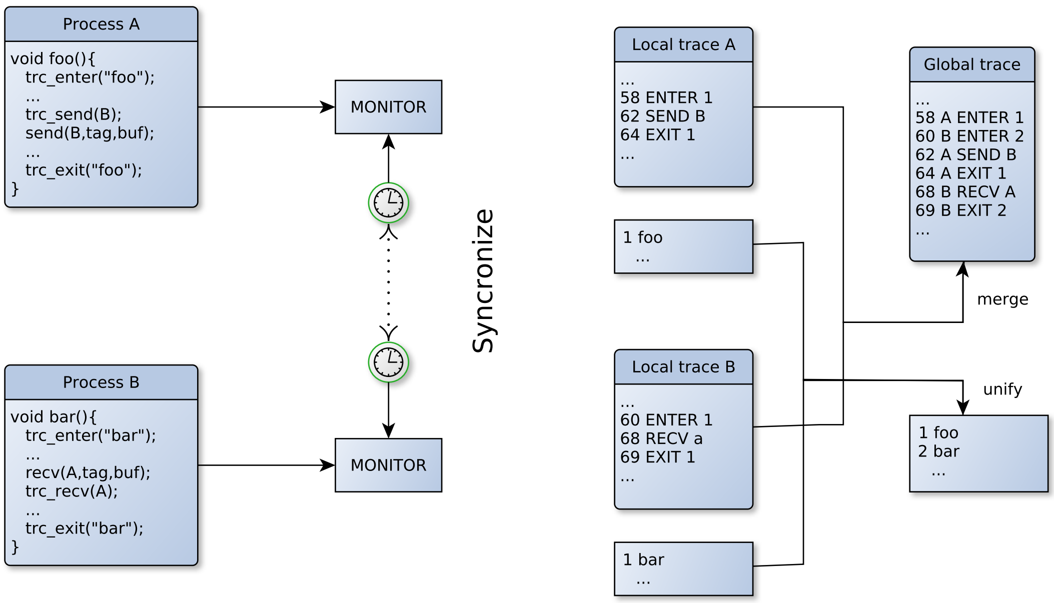


Figure 3.4: Example of tracing for two processes A and B that both call a function and exchange a message.

Our off-line simulation framework expresses the following requirements for *time-independent* traces:

- For compute parts, the trace should contain the id of the process that executes the instructions and the corresponding amount;
- For point-to-point communications, it is mandatory to know both the sender and the receiver of the message, the type (`MPI_Send`, `MPI_Recv`, etc.) and the message size in bytes;
- For collective communications, it is important to know at least the size of the message and the type of the MPI communication (`MPI_Broadcast`, etc.). However, there are some MPI calls such as `MPI_Allreduce`, where the participating processes execute also some computation, summing values or determining a maximum for example. In this case, it is mandatory to log the number of executed instructions in order to simulate the compute part;
- For synchronization calls such as `MPI_Barrier`, it is mandatory to know the MPI Communicator.

Currently our framework does not support all the MPI calls but the most common ones. Table 3.1 lists all the MPI functions for which there is a corresponding action implemented in our framework. For the collective operations, we consider that all the processes are involved as the `MPI_Comm_split` function is not implemented. When it is not possible to extract from the instrumented execution the information about the root process that initiates a collective operation, we assume that this process is the one of rank 0. Finally the `init` and `finalize` actions have to appear in the trace file associated to each process to respectively create and destroy some internal data structures that will be needed during the replay of the traces.

3.1.1 Evaluation Criteria and Scoring Method

Many profiling tools can provide useful information about the behavior of parallel applications and record the events that occur during their execution. PAPI[32] provides access to the

Table 3.1: Time-independent counterparts of the actions performed by each process involved in an MPI application.

MPI actions	Trace entry
CPU burst	`<id> compute <volume>`
`MPI_Send`	`<id> send <dst_id> <volume>`
`MPI_Isend`	`<id> Isend <dst_id> <volume>`
`MPI_Recv`	`<id> recv <src_id>`
`MPI_Irecv`	`<id> Irecv <src_id> <volume>`
`MPI_Broadcast`	`<id> bcast <volume>`
`MPI_Reduce`	`<id> reduce <vcomm> <vcomp>`
`MPI_Reduce_scatter`	`<id> reduceScatter <recv_counts> <vcomp>`
`MPI_Allreduce`	`<id> allReduce <vcomm> <vcomp>`
`MPI_Alltoall`	`<id> allToAll <send_volume> <receive_volume>`
`MPI_Alltoallv`	`<id> allToAllv <send_buffer> <send_counts> <send_displs> <receive_buffer> <recv_counts> <recv_displs>`
`MPI_Gather`	`<id> gather <send_buffer> <recv_buffer>`
`MPI_Allgatherv`	`<id> allGatherV <send_count> <recv_counts> <recv_disps>`
`MPI_Barrier`	`<id> barrier`
`MPI_Wait`	`<id> wait`
`MPI_Waitall`	`<id> waitAll`
`MPI_Init`	`<id> init`
`MPI_Finalize`	`<id> finalize`

hardware counters of a processor. In our prototype framework we measured the compute parts of an application in numbers of floating point operations (or flops). However, using another counter, such as the number of instructions, has no impact on the instrumentation overhead as long as the chosen measure does not combine the values of several counters. To select the tool that corresponds the most to the requirements expressed by our *time-independent* traces replay framework, we conducted an evaluation based on the following criteria and scoring method:

Profiling Features

Many tools exist to profile parallel applications, *i.e.,* get an idea of the behavior of the application over its execution time. A classification of these tools can be made following two axes. The first distinction depends on the type of output produced by the considered tool. It can be either a *profile* or a *trace*. A profile gives a higher view of the behavior of an application as information is grouped according to a type of operation or to the call tree. A trace gives lower level information as every event is logged without any aggregation. The second way to classify tools depends on the kind of events they can be profiled or traced. Such events are mainly related to communication or computation. A third important parameter for this evaluation is the type of value associated to each logged event, *e.g.,* volumes or timings. Finally, the instrumentation of an application is mandatory to obtain a profile or a trace. Such a instrumentation can be applied to the entire application or to a specific block of code, *e.g.,* , a particular function or a MPI call. In this evaluation we give a higher score to tools offering an automatic instrumentation feature. Such a feature lighten the burden of getting a profile from a user point of view.

With regard to the specific requirements of our time-independent trace replay framework, we adopt the following scoring scheme for the *Profiling feature* criterion:

- Tracing $\rightarrow$ 2 points
- Information on communication $\rightarrow$ 1 point
- Information on computation $\rightarrow$ 1 point
- Information on communication volumes (in bytes) $\rightarrow$ 1 point
- Information on computation volumes (in flops) $\rightarrow$ 1 point
- Automatic instrumentation of the complete application $\rightarrow$ 1 point
- Automatic instrumentation of a block of code $\rightarrow$ 1 point

Each tool can then obtain a score ranging from 0 to 8, the higher being the better.

Quality of Output

Profiling tools usually output the results in files. These files could be of various formats such as plain text, XML, or binary formats. As mentioned earlier our objective is to convert such output files into set of actions to be replayed by SimGrid. The readability of the output files is then a very important requirement. However, tools that produce binary files often come with an application or an API to extract the required information. Finally we aim at replaying executions that involve large numbers of processes. To ensure a good scalability, it is important that the trace of each process is stored in a separate file. Indeed a tool enforcing a unique trace file for the complete execution will have a poor scalability due to merging overhead. Moreover as we intend to read these traces and extract the *time-independent* ones, if there is at least one trace file per node, then we can use the same number of nodes with a parallel application to achieve our purpose. This means that even for large trace sizes we would be able to use many processors and hard disks. So the conversion will be much faster than using just one node. Then we propose the following scoring scheme (ranging from 0 to 3) for the *Output quality* criterion:

- Capacity to extract the required information $\rightarrow$ 1 point
- Direct extraction $\rightarrow$ 1 point
- One pass conversion $\rightarrow$ 1 point

Space and Time Overheads

As already mentioned, files that contain all the relevant information are created during the tracing or profiling of an application. Profiling provides aggregated data about an application. This means for example that it records the number of times a function is called and its average duration. Conversely tracing records each call independently with the timestamp and some other information which depend on what the user wants to measure. The source of the time overhead for an instrumented application are the measurement of a PAPI counter on the processor, the instrumentation of the MPI communications, and writing traces on disks. In terms of space overhead, profiling obviously leads to smaller traces, as logged events are aggregated.

Then, for *Space and Time overhead*, we adopt the following scoring scheme:

- Overhead $< 50\%$ of the initial execution time $\rightarrow$ 2 points
- $50\% < \mathit{Overhead} < 100\%$ of the initial execution time $\rightarrow$ 1 point
  The executed time of the instrumented application includes the time needed to write the produced traces on disk.
- Linear increase of the overhead $\rightarrow$ 1 point
- Constant overhead $\rightarrow$ 2 points
- Linear decrease of the overhead $\rightarrow$ 3 points
- Linear increase of trace size with regard to problem size $\rightarrow$ 1 point

– Linear increase of trace size with regard to number of processes → 1 point
– Any special technique integrated in the tool to reduce the time overhead → 1 point

Each tool can then obtain a score ranging from 0 to 8, the higher being the better.

Software Quality

Sometimes installing a profiling tool is not just a simple procedure such as executing the commands `./configure`, `make`, and `make install` in a Linux environment. In some cases many flags should be added to install a tool, or dependencies on other tools should be taken under consideration. Moreover the tools that are released under open source license are preferred compared to the ones with any non open source license. Every tool should support a variety of hardware *e.g.,* processors and software developed under various programming languages *e.g.,* C/C++/Fortran or implementations of the Message Passing Interface *e.g.,* Mpich, OpenMPI. A well written manual is also always helpful for both installing and using a tool. It also has to be up to date and inform the user about new features. Finally it is desirable that the tool is maintained in order to support all the new versions of MPI and PAPI. As result for the *software quality* criterion we adopt the following scoring scheme:

– Ease of installation → 1 point
– Software dependencies → 1 point
– License → 1 point
– Hardware compatibility → 1 point
– Support of C/C++/Fortran programming language, 1 point for each → 3 points
– Compatibility with the major MPI implementations, *i.e.,* Mpich or OpenMPI → 1 point
– Documentation → 1 point
– The project is active → 1 point
– There is a support team for this tool → 1 point

Each tool can then obtain a score ranging from 0 to 11, the higher being the better.

3.1.2 Evaluation of Tracing and Profiling Tools

Based on the aforementioned evaluation criteria, we studied in [74] the following tools: PAPI [32] which gives access to the performance counters, PerfBench [75] which can instrument only compute parts, PerfSuite [76] that provides statistical profiling for the compute parts, MpiP [77] which can apply statistical profiling for the MPI communication, MPE [78, 79] that instruments MPI communication operations, IPM [80] which supports profiling for MPI communication and compute parts. The other tools provide both profiling for communication and compute parts: Extrae [81], Scalasca [82], TAU [83], VampirTrace [84, 85], Score-P[86], and our contribution, called Minimal Instrumentation. In this section, we only include the evaluations of a selected subset of tools that are representative.

PAPI

The Performance Application Programming Interface (PAPI) is a tool developed at the University of Tennessee, Knoxville [32]. We based our evaluation on the version 3.7.0 while the last one is version 5.1.0. The differences with the last version are the support of multiple components, or counting domains, the possibility of developing new components and the support of more platforms. These recent additions do not impact our evaluation. PAPI relies on the *hardware performance counters* that are available on most processors. It provides the connection between software and hardware performance counters.

Properties of a subset of the events are presented in Listing 3.1. First the name of each hardware counter is given along with its internal code. The `Avail` column declares if the event is available and the `Deriv` column shows if this PAPI counter uses two or more counters to produce a value or there is a hardware counter that provides this information directly. Finally, the last line presents how many counters are available for the used processor.

Listing 3.1: Subset of the counters accessed by PAPI on a AMD processor.

```
The following correspond to fields in the PAPI_event_info_t structure.

    Name          Code      Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No   Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
...
PAPI_TOT_IIS 0x80000031  No    No   Instructions issued
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_INT_INS 0x80000033  No    No   Integer flops
PAPI_FP_INS  0x80000034  Yes   No   Floating point flops
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
--------------------------------------------------------------------
Of 103 possible events, 36 are available, of which 8 are derived.
```

So by using this metric, it is possible to create traces with the exact amount of executed flops.

Profiling features With regard to the *profiling features* evaluation criterion, PAPI obtains a score of 4 points. Indeed, PAPI offers tracing facilities (2 points), that allows us to obtain information about the computation (1 point) and especially about volume for each computation event (1 point). However, PAPI does not provide any mechanism to trace communication events and there is no way to trace an application automatically using only PAPI. Nevertheless it is possible to instrument an application manually.

Listing 3.2 presents an example of how to measure the number of flops computed by a block of code. In order to trace a block of code, some commands should be inserted at the beginning and the end of it. If we want to measure the amount of executed flops for the function `ssor(itmax)` in the LU benchmark, the commands `PAPIF_start` and `PAPIF_stop` should be inserted as shown in Listing 3.2.

Lines 0-1 declare the `events` array in which are saved the codes of the events to be measured. The variable `numevents` declares the number of events that PAPI should monitor and the variable `check` is used for checking the status of the commands. Furthermore the `EventSet` is the set with all events that are going to be used. The values from the counters are saved in the array `Values`. Line 2 declares two strings for printing the number of flops and the rank of the process. Line 3 declares the number of events that are going to be monitored. Line 4 sets `EventSet` to `PAPI_NULL` on the next line, the event `PAPI_FP_OPS` is saved to the events array. The variable `check` on the line 6 is always equal to the code of the current PAPI's version. Between the lines 7 and 12 the following commands are executed: the initialization of PAPI, the creation of the null set, the event is added to the null set, the measurement starts, the ssor function is called, and the measurement stops respectively. Finally from lines 13 to 16 the value of the counter is saved to the string `ch`, the id which is the rank of the process is saved to the string `ch2` and

Listing 3.2: Example of measuring the flops for the ssor iteration.

```
      integer events (1), numevents, check, EventSet
      integer Values (1)
      character(15) ch,ch2
      numevents=1
      EventSet = PAPI_NULL
      events(1) = PAPI_FP_OPS
      check = PAPI_VER_CURRENT
      call PAPIf_library_init(check)
      call PAPIF_create_eventset(EventSet, check)
      call PAPIF_add_event(EventSet, events(1),check)
      call PAPIF_start(EventSet,check)
      call ssor(itmax)
      call PAPIF_stop(EventSet, Values,check)
      write(ch,'(i10)') Values(1)
      write(ch2,'(i6)') id
      write(*,100) trim(ch2),'␣compute␣',trim(ch)
  100 format (A,A9,A)
```

with the last two lines the number of the flops and the rank are printed with the format "rank compute value" which is the format of time-independent traces.

Moreover each process saves its own trace to a separate file so if the application is executed on N processes, then there are N trace files.

Quality of output About the *quality of output* evaluation criterion, PAPI obtains a score of 3 points. PAPI allows for the extraction of information on computations (1 point). By using appropriate commands such as those in Listing 3.2 it is possible to print the number of flops according to the time-independent trace format (1 point). As result there is no need for any conversion (1 point).

Example of traces for the LU benchmark

Table 3.2 presents the first four lines of the traces of the LU benchmark, class C, executed on 4 nodes. The benchmark was traced with PAPI, hence there is no info about the communication. After each MPI call we start measuring the executed flops and stop just before the next MPI call, where we save the corresponding action into our traces.

Space and time overhead Thanks to `papi_cost`, it is possible to compute the minimal, maximal, mean and the standard deviation of the execution time of basic PAPI operations, e.g., start/pairs and reads. The results are presented in Table 3.3. More details are given in [74].

With regard to the space and time overheads, PAPI obtains the score of 6 points. In Table 3.4, we see that the time overhead is less than 50% for the LU benchmark class C (2 points). While slightly increasing, the time overhead remains in the 5-7% range, then we consider it as constant (2 points).

In Table 3.5 we can observe that when the number of the nodes is doubled then the size of the traces is almost two times bigger. We can thus claim that the increase is linear (1 point). Moreover with 32 nodes, the difference of the trace sizes from class A to C is of 25 MB per class, so the trace size increases linearly with the problem size, except for class D. Compared to the

Table 3.2: The first lines of the time-independent traces of LU benchmark, class C, 4 nodes.

```
0 compute 10632621  | 1 compute 10633358
0 compute 244390418 | 1 compute 244372181
0 compute 11612863  | 1 compute 9328093
0 compute 6492893   | 1 compute 2284893
--------------------+--------------------
2 compute 10633400  | 3 compute 10632459
2 compute 244372720 | 3 compute 244394098
2 compute 9328049   | 3 compute 9327954
2 compute 2284855   | 3 compute 6492794
```

	Min cycles	Max cycles	Mean cycles	std. deviation
Papi_start/stop	9	26381	12	26.73
Papi_read	111	28568	114	81.9
Addition	235	9537	254	10

Table 3.3: Overhead in cycles of basic PAPI operations as measured with `papi_cost`

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	696	1.58
8	345.2	362.5	5.01
16	184	195.4	6.2
32	100.1	106.6	6.5
64	53.08	56.8	7

Table 3.4: Time overhead of PAPI for the LU benchmark, Class C.

other classes, Class D works on larger matrices (14 times bigger than class C) and needs more iterations (300 vs. 250). The observed can then easily be justified and we give 1 point on this criterion.

LU, class C	
Nodes	Size (in MB)
4	11
8	23
16	46
32	94
64	193

LU, 32 nodes	
Class	Size (in MB)
A	36
B	59
C	94
D	299

Table 3.5: Space overhead of PAPI for the LU benchmark.

Software quality Considering the *software quality*, PAPI obtains the score of 11 points. The installation of the PAPI is easy without any problem but there are some restrictions by the software dependencies (1 point). PAPI depends on linux performance counters driver (`perfctr`) [87] or performance monitoring interface (`perfmon2`) [88]. For the kernels older than 2.6.30 it is needed to patch the kernel, otherwise there is at least support for `perfmon2`. In the case that a patch is needed, the configuration of the kernel has to be changed. The monitoring counters should be enabled through the `make oldconfig` command and afterwards to build and install the modified kernel (1 point). Moreover PAPI is released under the New BSD license template which is a GPL-compatible software license (1 point). PAPI is the standard tool to access the hardware counters of the CPU and it is compatible with the major platforms as shown in Table 3.6 (1 point). Furthermore PAPI supports the C/C++ and Fortran programming languages (3 points). About the documentation, the installation process is well documented on the official web page (1 point), the usage of PAPI is well explained through the `PAPI references`, the `Presentations` and the `Doxygen docs` that are available under the `Documentation` menu of the official web page (1 point). Finally the project is active, patches are released often (1 point) and there is a support team answering any question (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	Linux 2.2, 2.4,2.6	Mikael Pettersson's PerfCtr kernel patch for Linux (included)
AMD Opteron, Intel Pentium M, Core2 Series	Linux 2.6	Stephane Eranian's Perfmon2 kernel patch from sourceforge
Intel Pentium IV, D	Linux 2.2,2.4,2.6	Mikael Pettersson's PerfCtr 2.6.x kernel patch for Linux (included)
Intel Itanium II, Montecito, Montvale	Linux 2.4, 2.6	none

Table 3.6: Principal Hardware/OS combinations supported by PAPI.

Table 3.7 summarizes the results of the evaluation of PAPI.

Profiling features	Quality of output	Overheads	Quality of Software	**Total**
4	3	6	11	**24**

Table 3.7: Summary of the evaluation of PAPI.

Scalasca

Scalasca [82, 89] is an open-source toolset that can profile and trace an application. It is developed by the Forschungszentrum Jülich, Jülich Supercomputing Centre and the German Research School for Simulation Sciences, Laboratory for Parallel Programming. We based our evaluation on version 1.3.3 while the latest is version 1.4.3. When the execution of an instrumented application ends, a parallel automatic event trace analysis tool called `SCOUT` is executed.

This tool identifies and reports any logical clock violations. Since the version 1.3.3, a new environment variable is introduced that disables the execution of this tool as it does not influence the creation of the traces. The Program Database Toolkit (PDToolkit) ([90], [91]) is a framework for analyzing source code written in several programming languages and for making rich program knowledge accessible to developers of static and dynamic analysis tools. This framework is partially supported by Scalasca.

Profiling features Regarding the *profiling features*, Scalasca obtains a score of 6 points. This tool can trace an application (2 points), but it is not possible to trace the size of the message for a `MPI_Recv` call. Thus it can not extract from the trace files all the required data related to communication (0 point). All the required data about computation (1 point), the volumes in bytes (1 point) and the volumes in flops (1 point) are respectively recorded into the trace files. Finally, Scalasca can automatically trace an application (1 point).

Quality of output Scalasca provides the PEARL API [92] to create a C++/MPI tool to extract the required data from the traces (1 point). Thanks to the format of the trace files, it is possible to convert them in one pass (1 point).

Space and time overhead Considering the *space and time overhead*, Scalasca obtains a score of 8 points. According to Table 3.8, the time overhead stays under 50% (2 points). Moreover, we see that it linearly decreases as the number of participating processes increases (3 points). As we execute the benchmark with one process per node, there is always one disk per process onto which flush the traces. The size of the trace produced by a single process decreases as the total number of processes grows, hence the reduction of the overhead.

Nodes	LU benchmark, class C		
	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	893	30.33
8	345.2	449	30.07
16	184	216	17.39
32	100.1	110	9.89
64	53.08	58.6	10.39

Table 3.8: Time overhead of Scalasca for the LU benchmark, Class C.

In terms of space overhead, the evolution of the trace size is correlated to that of the size of the instance (1 point). Class D is much bigger than the other instances (larger matrices and more iterations). This bigger gap is also observed in the size of the traces. From class A to class C, size grows by a factor of 2.3 and 2.6, while from class C to class D, the trace grows by a factor of 4.7. When the number of processes is increased for a given class, we observe a linear increase of the trace size (1 point). However, a large part of the trace size comes from the tracing of a function that does not depend on the number of processes and is not relevant for our studies. Scalasca allows the user to declare that some functions do not have to be traced, which allows for a reduction the space overhead (1 point).

Software quality Although some flags have to be set to install the tool, the installation was easy (1 point). It depends on `libbfd` and `libiberty` libraries but both libraries are part of

LU, class C	
Nodes	Size (in MB)
4	2,100
8	2,200
16	2,500
32	2,900
64	3,900

LU, 32 nodes	
Class	Size (in MB)
A	476
B	1,100
C	2,900
D	36,000

Table 3.9: Space overhead of Scalasca for the LU benchmark.

standard distributions (1 point). Furthermore Scalasca is released under the New BSD license which is a GPL-compatible free software license and has been vetted as open source by the Open Source Initiative (1 point). According to Table 3.10 Scalasca is compatibile with a lot of hardware (1 point). Scalasca can trace applications which are implemented in C/C++ and Fortran (3 points) and is compatible with MPICH and OpenMPI (1 point). The documentation is well written and covers a lot of topics about the installation and usage of the tool (1 point). Finally the project is active (1 point) and there is a team providing support to the users (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit, libbfd,libiberty,Qt4
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 3.10: Principal Hardware/OS combinations supported by Scalasca.

Table 3.11 shows the overall results of the evaluation of Scalasca.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
6	2	8	11	**27**

Table 3.11: Summary of the evaluation of Scalasca.

TAU

TAU [83, 93] is an advanced profiling and tracing toolkit for the performance analysis of parallel programs. It is developed by the Department of Computer and Information Science, University of Oregon, the Forschungszentrum Jülich, Jülich Supercomputing Centre, ZAM and the Advanced Computing Laboratory, Los Alamos National Laboratory. The evaluation is based on TAU version 2.21 while the last one is version 2.22.2 which suppors new technologies and processors and a new implementation of the MPI wrapper. According to our experiments these changes do not influence the current results.

Profiling features Regarding the *profiling features*, TAU obtains a score of 8 points. TAU can trace an application (2 points), and logs data related to the computation (1 point) and the communication (1 point) into trace files. Moreover the volumes of the communication and computation are in bytes (1 point) and flops (1 point) respectively. TAU supports the PDToolkit framework so it provides the user with a method to trace all the application (1 point) or only a block of code (1 point).

Quality of output The binary trace files produced by TAU contain all the required data for the simulation. It is possible to extract them with the help of the TAU trace format reader library [94] (1 point). We used the API to implement a tool that converts TAU traces into *time-independent* traces in one pass (1 point).

Space and time overhead The time overhead of TAU is given in Table 3.12. It is under 50% of the execution time of the application (1 point) but does not decrease as the number of processes grows. In terms of space overhead, the size of the traces increases linearly along with both the number of processes (1 point) and the problem size (1 point), as shown by Table 3.13. Finally, TAU provides a way to declare that some functions are not to be traced in order to decrease the time and space overheads (1 point).

Nodes	LU benchmark, class C		
	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %)
4	685.18	983	43.46
8	345.2	493.5	42.96
16	184	239.3	30.05
32	100.1	128.8	28.67
64	53.08	71.1	33.94

Table 3.12: Time overhead of TAU for the LU benchmark, Class C.

LU, class C	
Nodes	Size (in MB)
4	5,100
8	5,400
16	6,000
32	7,300
64	9,900

LU, 32 nodes	
Class	Size (in MB)
A	1,300
B	2,700
C	7,300
D	87,200

Table 3.13: Space overhead of TAU for the LU benchmark.

Software quality The installation of TAU is easy. It can be configured thanks to compilation flags (1 point). It has a software dependency on the PDToolkit which can be respected (1 point). TAU is free under BSD style license which is GPL-compatible (1 point). According to

Table 3.14 it supports various hardware configurations (1 point). Moreover all the programming languages C/C++ and Fortran (3 points) and the major MPI implementations such as MPICH and OpenMPI (1 point) are supported. The official web site gives access to manuals about usage and installation of this tool (1 point). Finally the project is active (1 point) and there is a team for answering any question (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit (optional)
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 3.14: Principal Hardware/OS combinations supported by TAU.

The overall results of the evaluation are presented in Table 3.15.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
8	2	5	11	**26**

Table 3.15: Summary of the evaluation of TAU.

Score-P

Score-P [86, 95] has been developed by the German BMBF project SILC and the US DOE project PRIMA. It is a highly scalable tool suite for profiling, tracing, and the online analysis of HPC applications. The evaluated version is the 1.0-beta while the latest one is 1.1.1 that corrects some bugs not related with our usage. This tool is the result of the collaboration between the researchers involved in the development of many well known tools such as TAU, Scalasca and VampirTrace.

Profiling features With regard to *profiling features* criterion, Score-P achieves the score of 7 points. It can trace an application (2 points) and record all the computation (1 point) and communication events (1 point) with their volumes in flops by using PAPI (1 point) and bytes (1 point) respectively. It can automatically instrument an application (1 point).

Quality of output The Score-P tool achieves the score of 2 points for the criterion of *quality of output*. The format of the generated traces is the Open Trace Format Version 2 (OTF2) [96] which is a highly scalable, memory efficient event trace data format and will become the new standard trace format for TAU, Vampir and Scalasca. It is the common successor format for the Open Trace Format (OTF) and the Epilog trace format [97]. When the execution ends, the traces comprise the computation and the communication events with all the necessary data (1 point). From the OTF2 files it is possible to acquire the time-independent traces by using the OTF2 API and only one pass (1 point).

Space and time overhead Table 3.16 shows that the time overhead is less than 50% (2 points). Moreover the overhead decreases linearly as the number of the processes increases (3 points).

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %) (in %)
4	685.16	915.919	33.67
8	345.2	443.59	28.5
16	184	220.78	19.99
32	100.1	109.95	9.84
64	53.08	58.16	9.57

Table 3.16: Time overhead of Score-P for the LU benchmark, Class C.

According to Table 3.17 the increase of the number of the processes causes a linear increase of the size of the traces (1 point). Similarly, when the size of the problem increases, then the size of the traces increases also linearly (1 point). With the combination of PDT and the Score-P user API, code regions can be excluded from the instrumentation (1 point).

LU, class C	
Nodes	Size (in mb)
4	2800
8	2900
16	3200
32	3700
64	4900

LU, 32 nodes	
Class	Size (in mb)
A	583
B	1400
C	3700
D	48000

Table 3.17: Space overhead of Score-P for the LU benchmark.

Software quality Score-P achieves the score of 11 points regarding the *software quality* criterion. There is no issue about the installation (1 point) and no dependency on other libraries (1 point). It is available in Open Source under a BSD license (1 point). The tool is compatible with various hardware and is supported on many platforms (1 point) as it can be seen in Table 3.18.

It is compatible with the C/C++ and Fortran programming languages (3 points) and with the major MPI implementations (1 point). There is a manual which describes many different aspects of what this tool can do, how to install it and how to use it (1 point). This is a new project which aims at providing the best characteristics of some well known tools (1 point) and there is a support team for solving any problem (1 point). We aggregate the results of the evaluation in Table 3.19.

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI, PDToolkit
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 3.18: Principal Hardware/OS combinations supported by Score-P.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
7	2	8	11	**28**

Table 3.19: Summary of the evaluation of Score-P.

Minimal Instrumentation (MinI)

There are two undesirable side effects caused by complex profiling tools. First, they may induce a large time overhead. Second, the measure of hardware counters may be skewed, as the flops caused by the tool itself are also measured. Unnecessary instrumentation, in our case instrumentation that generates trace data beyond the information strictly needed for our replay framework, would then unnecessarily increase overhead and skew. In order to control the instrumentation at a lower level, we implemented our own instrumentation method, called MinI. 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 such as those in Figure 3.1. This approach is guaranteed to perform the minimal amount of instrumentation needed for our purpose.

According to both MPICH and OpenMPI implementations, the Fortran PMPI layer calls C PMPI either directly or through C MPI layer. Thus we just have to provide profiling wrappers in C to trace an application. Furthermore, to decrease the I/O overhead, we implement a simple buffering mechanism. We can declare how many events are saved in the memory before flushing them to the hard disk.

Listing 3.3 presents the instrumentation of the `MPI_Send` call, with this minimal instrumentation. The general concept is that when there is an `MPI_Send` call, we create a `compute` action with the amount of flops computed since the last MPI call. Then, `PMPI_Send` is called, and the corresponding `send` action is created, saved in the buffer and it starts measuring again the flops. We use the `PAPI_accum_counters` command which measures the corresponding PAPI metric since the beginning of the execution and increases monotonically.

Listing 3.3: Handling MPI_Send function with MinI.

```
int  MPI_Send( buf, count, datatype, dest, tag, comm )
{
 ...
  if(PAPI_accum_counters(values, 1)!=PAPI_OK) {
        printf("PAPI does not support this metric\n");
  }
  counter2=values[0];

  MPI_Type_get_name(datatype,t_data,&np);
  this_type=encode_datatype(&t_data);
  PMPI_Comm_rank( MPI_COMM_WORLD, &llrank );


  sprintf(msg,"%d compute %lld\n",llrank,counters2-counter1);
  strcat(longmsg,msg);

  returnVal=PMPI_Send( buf, count, datatype, dest, tag, comm );

  if(this_type==0) sprintf(msg, "%d send %d %d\n",
                              llrank,dest,count);
  else sprintf(msg, "%d send %d %d %d\n",
                            llrank,dest,count,this_type);
  strcat(longmsg,msg);

  if(PAPI_accum_counters(values, 1)!=PAPI_OK) {
        printf("PAPI does not support this metric\n");
  }
  counter1=values[0];
  ...
}
```

Profiling features With regard to the *profiling features* evaluation criterion, the minimal instrumentation tool, obtains a score of 7 points. This tool can trace an application (2 points) and record the communication (1 point) and computation information (1 point) with the corresponding data in bytes (1 point) and flops (1 point). Moreover it can instrument automatically an application (1 point), but all the application has to be traced.

Quality of output Regarding the *quality of output*, it achieves a score of 3 points. This tool can record all the required information (1 point), it saves directly the measured data into time-independent traces (1 point), thus there is no need to read the traces at all (1 point).

Space and time overhead About the *space and time overheads* evaluation criterion, our minimal instrumentation obtains a score of 6 points. The time overhead in Table 3.20 is around 4-5% in all the cases, then smaller than 50% (2 points) and constant (2 points). The trace sizes are given in Table 3.21. When the number of processes is doubled, the size of the traces is

Nodes	Execution time w/o tracing (in sec.)	Execution time w tracing (in sec.)	Time Overhead (in %) (in %)
4	685.18	695.8	1.55
8	345.2	360	4.29
16	184	191.63	4.15
32	100.1	105.2	5.09
64	53.08	55.8	5.12

Table 3.20: Time overhead of MinI for the LU benchmark, Class C.

increased linearly (1 point). With regard to the problem size, by changing only the class of the problem, the size of the traces is increased linearly (1 point).

LU, class C	
Nodes	Size (in MB)
4	20
8	49
16	120
32	257
64	549

LU, 32 nodes	
Class	Size (in MB)
A	99
B	161
C	257
D	804

Table 3.21: Space overhead of MinI for the LU benchmark.

Software quality MinI achieves a score of 10 points regarding the *software quality* criterion. There is no need to install the tool, just to compile it (1 point). There is no dependency except on PAPI (1 point). MinI is released under the LGPL license (1 point). Table 3.22 shows that MinI has no hardware compatibility issues (1 point). It supports C/C++ and Fortran programming languages (3 points). The MinI tool is developed according to the MPI standards, thus it supports both MPICH and OpenMPI (1 point). There are instructions on how to compile and use it (1 point). Moreover this project is active (1 point) there is a support team for answering any question (1 point).

Hardware	Operating System	Requirements
AMD Athlon/Opteron, Intel thru Pentium III, Pentium M, Core2 Series	x86-Linux	PAPI
AMD Opteron, Intel Pentium M, Core2 Series	x86_64-Linux	
Intel Pentium IV, D		
Intel Itanium II, Montecito, Montvale	IA_64-Linux	

Table 3.22: Principal Hardware/OS combinations supported by MinI.

Finally we aggregate the results of the evaluation in the Table 3.23.

Profiling features	Quality of output	Overheads	Software Quality	**Total**
7	3	6	10	**26**

Table 3.23: Summary of the evaluation of MinI.

3.1.3 Final Scores and Discussion

The final score for all the tools is presented in Table 3.24.

Tool	Profiling features	Quality of output	Space and Time Overheads	Quality of Software	**Total**
PerfBench	2	0	0	5	**7**
PerfSuite	2	0	0	10	**12**
MpiP	2	0	0	11	**13**
IPM	3	0	0	11	**14**
MPE	4	1	2	10	**17**
PAPI	4	3	6	11	**24**
Extrae	7	2	5	11	**25**
VampirTrace	7	2	5	11	**25**
MinI	7	3	6	10	**26**
TAU	8	2	5	11	**26**
Scalasca	6	2	8	11	**27**
Score-P	7	2	8	11	**28**

Table 3.24: Summary of the evaluation of the studied instrumentation tool.

First, we recall the evaluation criteria we used are driven by the requirements of our framework. Then, it is not an exhaustive evaluation of all the features of each tool. About the score, PAPI achieves a high score. However, it only provides information about computation with manual instrumentation. Then, Extrae needs a lot of time to convert the trace files into Paraver files and depending on the application, traces can be large. As the conversion to time-independent traces should be done by a serial application, the demanded time could be significant with regard to the execution of an application. Reading the provided OTF files of VampirTrace instrumentation, requires to merge them, so a global file system is demanded, or at least to gather all the traces into a single node. This is a procedure that we want to avoid because the global file system is not supported by default by all our clusters and the gathering of the traces would consume a lot of time. Although Scalasca is an efficient tool, it does not comply with all the necessary requirements expressed by our framework. TAU, is one of the tools that provide all the features and it is efficient enough according to our requirements. However, there are some cases which are going to be described later, where it demands a lot of memory in comparison with other tools. One of the newest tools, named Score-P, seems to be promising. It provides all the required information and with the OTF2 API it is possible to extract *time-independent* traces without the need for a global file system. The MinI tool succeeded to most of our tests. It has been developed to fulfill our framework requirements. It provides the time-independent traces directly after the application's execution, just by compiling MPI applications against our tool with less overhead than any other tool. The TAU, Score-P, and MinI can be used for our framework but with some restrictions. For example if we want to create an action corresponding to communication such as `MPI_Alltoallv`, then we need to know some specific information about the MPI call that only MinI provides directly.

While MinI does not achieve the highest score, it is the most efficient with regard to our framework requirements. It leads to smaller time and space overhead than other tools. Furthermore it creates the *time-independent* traces directly during the execution of the application, thus it needs less space to save trace files on hard disks. The instrumentation skew is also smaller than other tools, so the traces represent accurately the performance of an application. Another advantage is its lack of dependencies and it is easy to use by linking MinI to the desired application. For our prototype we used the TAU tool, as Score-P was not mature at that time. However, the need for an even more efficient profiling tool, led us to develop MinI.

3.2 Evaluation of Instrumentation Methods

In our prototype framework it was necessary to validate our methodology on various benchmarks. The benchmarks are constituted by computation and/or communications parts and declarations of variables and initialization/finalization phases. To validate our framework, specially as the simulation did not take memory footprint under consideration, we had to acquire the *time-independent* traces for specific parts of applications. The usual approach is called selective instrumentation. There is another reason to apply selective instrumentation. Some applications can be constituted by large number of calls to a function which does only compute and total duration whose is not significant in comparison to application's execution time. Such functions can create both time and space overhead during the instrumentation of an application. We figured out another method to decrease such time and space overhead. The main idea came out from the structure of the *time-independent* traces as there is no need to know when a non MPI function is called. From the computation point of view we only need to know the total amount of instructions between two successive MPI calls. We combine TAU with PDToolkit to instrument only MPI calls, leading to a more efficient instrumentation called "TAU-reduced". However, when we tried to test the scaling capabilities of our framework by using hundreds of nodes, TAU demands a lot of memory. This occurs because TAU provides more complex profiling features than the ones that are required by our framework. To address this issue we developed the minimal instrumentation MinI which instruments only the MPI calls of an application and saves the logged events into *time-independent* traces directly. This tool is the most efficient with regard to our framework requirements and requires less memory than other tools. In the next section we detail the "Selective" and "Tau-Reduced" instrumentation methods+

3.2.1 Selective Instrumentation

An interesting feature provided by TAU, is selective instrumentation. It can be done in different ways. One consists in listing in a separate file which functions have (or have not) to be traced. All the functions in the call path of the listed functions will also be traced. However, this technique may not be enough to isolate a given call, e.g., if a function is called twice but only one call has to be traced. A solution is then to insert two macros provided by the TAU API, namely `TAU_ENABLE_INSTRUMENTATION` and `TAU_DISABLE_INSTRUMENTATION` in the source code.

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

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

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

3.2.2 TAU-reduced

While selective instrumentation logs only the events of specific parts of an application, it still includes time overhead to instrument all the enter/exit events of a function that is called many times. For instance, this overhead may come from the instrumentation of functions that do not include MPI calls and are located within loops. While this information may be useful to users aiming at identifying performance bottlenecks in their application, it is useless to our specific usage of the produced traces.

However, the selective instrumentation feature provided by TAU gives us a way to obtain all this required information, and only it. We modified our instrumentation method as follows. First we create a file, named `exclude.pdt`, in which we indicate to TAU the list of source files that have to be excluded from the detailed instrumentation. Actually, we exclude all the source files using the special character `'*'`.

```
BEGIN_FILE_EXCLUDE_LIST
*
END_FILE_EXCLUDE_LIST
```

Then, we add the following option to the command line to take this exclusion file into account.

```
-optTauSelectFile=/path/exclude.pdt
```

Thanks to this straightforward modification, the instrumentation is "reduced" with regard to our specific needs. Indeed the performance hardware counter that measures the number of executed instructions will be triggered only when entering and exiting MPI functions. All the information related to MPI calls, *i.e.,* id of the process that calls this function and the function name and calling parameters, are traced exactly as in the previous implementation.

We expect this simple modification to have an impact on both the execution time overhead and the skew of measured numbers of instructions between non-instrumented and instrumented versions of an application. This instrumentation method is called "TAU-reduced".

3.2.3 Evaluation of the instrumentation methods

We execute four different versions of the LU factorization 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. 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 the MinI tool. 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. For each MPI process, we compute the instrumentation skew as the percentage difference in total number of instructions between an instrumented and the original application. For the instrumented application this number is computed as the average of the numbers of instructions for all the CPU bursts.

Figure 3.5 shows the skew for various 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. As expected, the TAU-full instrumentation method leads the highest skew as it is the most intrusive methods. The induced skew ranges 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 and the instrumentation instructions represents a non-negligible fraction of the executed instructions.

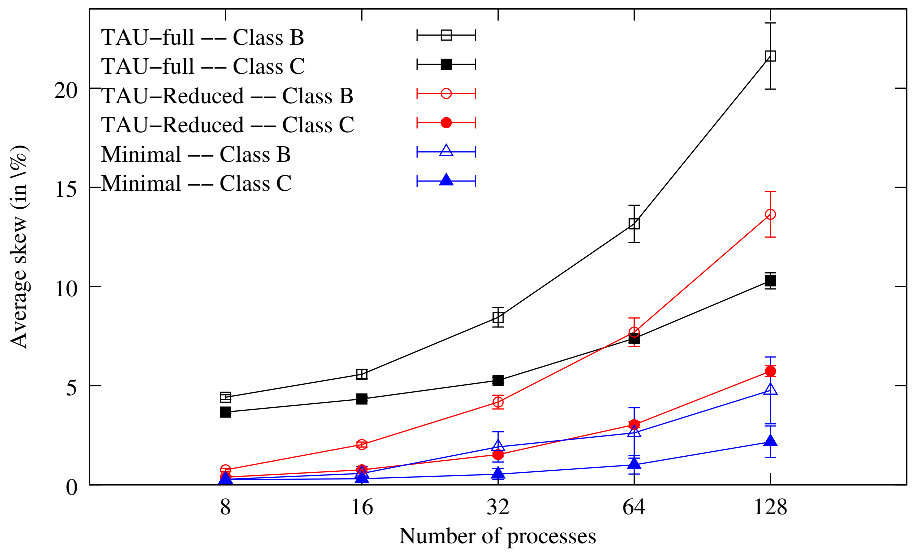


Figure 3.5: Instrumentation skew for the three instrumented LU benchmarks.

Figure 3.6 shows the overhead in terms of execution time induced by the TAU-reduced and Minimal instrumentation methods for various instances of the LU benchmark. The TAU-Full instrumentation method is not displayed here, as its high skew discards it as a valid choice for our off-line simulation framework. As for the skew, we see that the Minimal instrumentation we implemented reduces the overhead (on average by a factor of 1.6) with regard to the TAU-reduced method. While this figure indicates that the overhead can become large (up to 23.5% for the B-128 instance), in general it remains low. Indeed, the overhead induced by the Minimal instrumentation method ranges from 1.55 to 2.01 seconds for class B instances and from 0.8 to 2.6 seconds for class C instances. But 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.

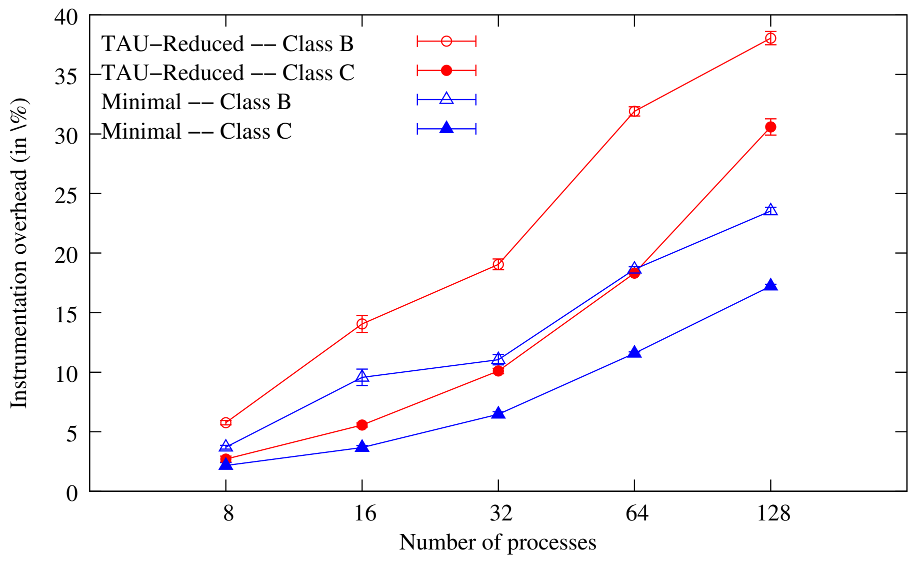


Figure 3.6: Instrumentation overhead for the reduced and minimal instrumented LU benchmarks.

3.3 Acquisition of Time-Independent Traces

When an application is instrumented then a procedure should be followed for the acquisition of the *time-independent* traces. As mentioned in Figure 3.3, the instrumented application is executed, afterwards the action lists is extracted from the instrumentation traces if necessary, and finally the *time-independent* traces are gathered into a single node.

3.3.1 Execution

There are only two requirements for obtaining a valid time-independent trace in a view to simulate 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. This is in sharp contrast with time-dependent traces, 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 processor. This is the way in which off-line simulators obtain time-stamped traces. This mode requires as many processors 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 processor. This allows for the acquisition of traces for larger instances of the application than can be executed in regular mode on the same cluster. The folding factor is only limited by the available amount of memory on the processors.

Composite mode: execution on heterogeneous or multiple non-identical clusters. In this mode, the user can aggregate disparate processors 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 processors of a same family to prevent inconsistencies in the execution.

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 processor in a non-homogeneous platform.

3.3.2 Using the Grid'5000 Platform to Acquire Time-Independent Traces

Building an Appliance

Several base system images are made available by the Grid'5000 technical staff. Most of the images are based on the Debian Linux distribution and can be deployed on all Grid'5000 sites. There exist images based on three different versions of the Debian distribution, but one of them is aging (`Lenny`), the `Squeeze` is the one used for our experiments and the `Wheezy` image is the latest Debian release at the time of writing.

– `squeeze-x64-base` is a minimal environment with no support of NFS and LDAP services. Only the drivers necessary to the support of the Grid'5000 infrastructure and high-speed network interconnect are installed.
– `squeeze-x64-nfs` is based on `squeeze-x64-base` and enables the LDAP and NFS services.
– `squeeze-x64-big` is based on `squeeze-x64-nfs` and includes various pre-installed packages such as `cmake`, `gfortran`, `OpenMPI`, `taktuk` [98], as well as several packages for development, system tools, and editors. All these tools provide a stable platform for configuring the operating system and execute various experiments.

In [99] are presented all the procedures to build an image that could be used to acquire *time-independent* traces. Initially we describe how to deploy an image on Grid'5000. In the case where the kernel is earlier than version 2.6.31, we provide instructions to patch the kernel to enable access to the hardware counters of the processor. Then we detail all the steps to install the tracing tools, instrument and compile an application. Moreover the tools for trace post-processing should be installed on the deployed image. It is needed to save the image for future deployment and set the appropriate environment variables for TAU. Finally, we give all the commands to execute an instrumented application and acquire the *time-independent* traces for all the available execution modes.

3.3.3 Trace Extract Tools

One of the tasks that may be needed after the execution of an instrumented application, is the conversion of the instrumented traces into *time-independent* ones. This is necessary only if the MinI tool is not used, otherwise the *time-independent* traces are created automatically. We are going to describe the tools that convert instrumentation traces into *time-independent* ones by presenting how they handle some of the MPI calls.

TAU

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

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

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

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

`events.<node>.edf.`

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

Figure 3.7 presents some entries of an event file generated by TAU that corresponds to some MPI functions. An entry corresponding to the access to a hardware counter that measures the number of instructions, some user events related to the size of the messages for some MPI communications as it is mentioned by their names.

```
4 MPI 0 "MPI_Init()  " EntryExit
19 MPI 0 "MPI_Send()  " EntryExit
21 MPI 0 "MPI_Irecv()  " EntryExit
22 MPI 0 "MPI_Wait()  " EntryExit
26 MPI 0 "MPI_Allreduce()  " EntryExit
28 MPI 0 "MPI_Barrier()  " EntryExit
30 MPI 0 "MPI_Finalize()  " EntryExit

1 TAUEVENT 1 "PAPI_TOT_INS" TriggerValue

DefUserEvent event id 8 user event name "Message size sent to all nodes",
monotonically increasing = 0
DefUserEvent event id 11 user event name "Message size for broadcast",
monotonically increasing = 0
DefUserEvent event id 27 user event name "Message size for all-reduce",
monotonically increasing = 0
DefUserEvent event id 9 user event name "Message size for all-to-all",
monotonically increasing = 0
```

Figure 3.7: TAU - Entries example from an event file.

As the trace files generated by TAU are binary files, there is a need for an interface to extract information. Such an API is provided by the TAU Trace Format Reader (TFR) library [94]. This tool provides the necessary functions to handle a trace file, including a function to read events.

It also defines a set of eleven callback methods, that correspond to the different types of events that appear in a TAU trace file. For instance there are callbacks for entering or exiting a function and triggering a counter. The implementation of these callback methods is let to the charge of the developer.

We thus developed a C/MPI parallel application, that implements the different callback methods of the TFR library. This program basically opens, in parallel, all the TAU trace files and read them line by line. For each event, the corresponding callback function is called. To illustrate how this tool extracts the necessary data to produce a time-independent trace, we detail the case of some calls to MPI functions. Figure 3.8 presents the parameters of the different callbacks related to the `MPI_Send` function call on process 1 in a readable format. Each line starts by the process id, the thread id, the time (in ms) at which the event occurred and the name of the event. The remaining fields are event dependent.

```
1  1 0 91832 EnterState    19
2  1 0 91832 EventTrigger   1  93686103
3  1 0 91840 EventTrigger   8  8
4  1 0 91843 SendMessage  0 0  8  1 0
5  1 0 91993 LeaveState    19
6  1 0 91832 EventTrigger   1  93686172
```

```
1 send 0 8
```

Figure 3.8: TAU - List of callbacks related to a call to the `MPI_Send` function and the corresponding *time-independent* trace.

As mentioned earlier, the event that corresponds to a `MPI_Send` is tagged as `EntryExit` in the event file with the event id 19. The first occurring callback will then be on the `EnterState` event (line 1). The matching `LeaveState` event (line 5) defines the scope of events related to the function call. There are more four events, two of them (lines 2 and 6) correspond to the hardware counter measuring the number of instructions, as identified in the event file. These two events are used to respectively end the CPU burst preceding the MPI call and starts the next one. The number of instructions computed within an MPI call, mainly due to buffer allocation costs, are ignored as they are accounted for by the network model. The last two events are related to the sent message. The `EventTrigger` on line 3 only provides the size of the message (8 bytes), which is not enough to build an entry in the *time-independent* trace. The `SendMessage` event (line 4) gives more information, namely the process and thread ids of the receiver, the size of the message, and the MPI tag and communicator for this communication.

Note that for asynchronous and collective communications, the extraction process is more complex. For instance, the mandatory information to write the entry corresponding to a `MPI_Irecv`, *e.g.,* the receiver id, are given by a `RecvMessage` event which generally occurs within the `MPI_wait` function. This implies to implement some lookup techniques to retrieve all the necessary parameters. Figure 3.9 presents the parameters of the different callbacks related to the `MPI_Irecv` and `MPI_Wait` functions calls on process 1 in a readable format. TFR provides a method to seek the current position during the reading of the traces to a specific event. Thus during the call to the `MPI_Irecv` function, the position of the next event is saved. When the tool reads the call to the `MPI_Wait` function it creates the corresponding *time-independent* actions and it moves back to the position of the saved event to continue reading the trace.

Similarly, the callbacks for the `MPI_Bcast` function are presented in Figure 3.10 with its corresponding *time-independent* trace. The difference is that the line 3 calls the `EventTrigger` with id 11 which declares the size of the message for broadcast procedure (4 bytes).

The `MPI_Allreduce` function is presented in Figure 3.11. On line 3 there is a call to

```
1 0 91800 EnterState     6
1 0 91800 EventTrigger   1  93675408
1 0 91817 LeaveState     6
1 0 91832 EventTrigger   1  93678423
..
1 0 92009 EnterState     9
1 0 92009 EventTrigger   1  93743774
1 0 92021 EventTrigger  10  8
1 0 92025 RecvMessage  0 0 8 0 1
1 0 92047 LeaveState     6
1 0 92047 EventTrigger   1  93767165
```

```
1 Irecv 0 8
1 wait
```

Figure 3.9: TAU - List of callbacks related to calls to the `MPI_Irecv` and `MPI_Wait` functions and the corresponding *time-independent* trace.

```
1 0 147298 EnterState    10
1 0 147298 EventTrigger   1  22336408
1 0 147440 EventTrigger  11  4
1 0 147441 LeaveState    10
1 0 147441 EventTrigger   1  22336512
```

```
1 bcast 4
```

Figure 3.10: TAU - List of callbacks related to a call to the `MPI_Bcast` function and the corresponding *time-independent* trace.

`EventTrigger` with id 27 which declares the size of the message for all-reduce procedure (40 bytes). Moreover in the *time-independent* trace we declare the amount of executed instructions to complete the reduction operation (600,124 instructions). The action `comm_size` declares the total number of processes that participate to this collective communication (4 processes in this case).

```
1 0 2.133e+06 EnterState    26
1 0 2.133e+06 EventTrigger   1 1231991913
1 0 2.143e+06 EventTrigger  27 40
1 0 2.143e+06 LeaveState    26
1 0 2.143e+06 EventTrigger   1 1232592037
```

```
1 comm_size 4
1 allReduce 40 600124
```

Figure 3.11: TAU - List of callbacks related to a call to the `MPI_Allreduce` function and the corresponding *time-independent* trace.

However, this callback mechanism is not always sufficient. For instance, as shown by Figure 3.12, the information extracted from the texttttMPI_Alltoallv function only gives the total number of bytes exchanged during the execution of this function and not the size of each exchanged message.

We do not know the values of the displacements of the array that are given as input during the call to the `MPI_Alltoallv` function. Also the value 4196661 corresponds to the sum of the elements that the process exchanges with other participating processes.

```
1 0 359590 EnterState   8
1 0 359590 EventTrigger 1 373796917
1 0 382392 EventTrigger 9 4196661
1 0 382400 LeaveState   8
1 0 382400 EventTrigger 1 373815406
```

Figure 3.12: TAU - List of callbacks related to a call to the `MPI_Alltoallv` function.

Score-P

Similarly, after the execution of an instrumented program with Score-P, many files are produced. Initially, there is the anchor file named `<archive_name>.otf2` which contains meta data related to the archive organization, declaring where are stored the other files. The global definition file `<archive_name>.def` stores all global and unified definition records. There are also local definition and trace files, one for each MPI process. The file `<node>.def` stores mapping tables used to map identifiers that are not global during the measurement. The `<node>` is the rank of the MPI process whose execution is logged in the files. The trace file `<node>.evt` is constituted by all the local event records that occur during the execution of the application. There is only one event file per MPI process. Each event file contains information about all the traced functions. For any function, an event file stores its `locationID` which is the rank of the MPI process that called this function and the time-stamp in ticks. The rest are event dependent, it can be the `regionID` which is the name of the function, or the `metricID` which is the id of the used PAPI counter. It is important to mention that the global definition file contains the time resolution used for the measurement of time, thus the time-stamps of the events should be divided by the time resolution in order to extract the time in seconds.

Score-P creates binary files, so it is mandatory to use the OTF2 reading interface [100] to extract the required information. We developed a C/MPI parallel application with a similar number of callbacks as for TAU. This program opens, in parallel, all the local definition and trace files and read them line by line. However, we had to copy the anchor and the global definition files across all the nodes with the help of the `taktuk` tool because they only exist on the root node. This happens because Score-P is considering the existence of a global file system. However, thanks to the structure of the OTF2 file format we can read the traces in parallel.

We present hereafter some examples of MPI functions to illustrate how our `trace_extract_scorep` extraction tool works. The following example in Figure 3.13 shows the callbacks related to the `MPI_Send` function in a readable format.

```
1  METRIC    1  33410119812629  0 1 1819794045
2  ENTER     1  33410119812629  MPI_Send 114
3  MPI_SEND  1  33410119817713  0 0 1 8
4  METRIC    1  33410119848237  0 1 1819796231
5  LEAVE     1  33410119848237  MPI_Send 114
```

```
1 send 0 8
```

Figure 3.13: Score-P - List of callbacks related to a call to the `MPI_Send` function and the corresponding *time-independent* trace.

The first column declares the name of the event, then are the process id, the time in ticks and the event dependent data. For the `METRIC` events (lines 1 and 4), are declared the id of the measured hardware counter, and its value. For the `ENTER/EXIT` events (lines 2 and 5), there are the name of the function with its id. Finally for the event `MPI_Send` after the time are declared the thread id of the receiver, the MPI communicator, the MPI tag and the size of the message (in bytes).

Figure 3.14 presents the parameters of the different callbacks related to the `MPI_Irecv` and `MPI_Wait` functions calls on process 1 in a readable format. The data layour is similar. The OTF2 reading interface provides a method to seek to a specific event during the reading of the local event trace file. Thus the procedure is similar to the one described for TAU.

```
1   METRIC             1  33410119788305  0 1 1819790170
2   ENTER              1  33410119788305  MPI_Irecv 91
3   MPI_IRECV_REQUEST  1  33410119799825  106
4   METRIC             1  33410119805525  0 1 1819792897
5   LEAVE              1  33410119805525  MPI_Irecv 91
6   ..
7   METRIC             1  33410119857221  0 1 1819797299
8   ENTER              1  33410119857221  MPI_Wait 129
9   MPI_IRECV          1  33410119876577  1 0 1 8 106
10  METRIC             1  33410119879337  0 1 1819800706
11  LEAVE              1  33410119879337  MPI_Wait 129
```

```
1 Irecv 0 8
1 wait
```

Figure 3.14: Score-P - List of callbacks related to a call to the `MPI_Irecv` and `MPI_Wait` functions and the corresponding *time-independent* trace.

On the line 3 the `MPI_IRECV_REQUEST` event, declares that the `MPI_Irecv` function was called. Then, the event `MPI_IRECV` (line 9) marks the end of this MPI function while the `MPI_Wait` function has already been called.

The `MPI_Bcast` function is presented in Figure 3.15. The events `MPI_COLLECTIVE_BEGIN` and `MPI_COLLECTIVE_END` declare the start and the end of an MPI collective communication `MPI_Bcast` function. This event provides the MPI communicator, the root node, the sent and the received sizes of the messages (in bytes).

```
1   METRIC                1  31463791461307  0 1 64206241
2   ENTER                 1  31463791461307  MPI_Bcast 11
3   MPI_COLLECTIVE_BEGIN  1  31463791461307
4   MPI_COLLECTIVE_END    1  31463791511091  BCAST 0 0 0 4
5   METRIC                1  31463791511091  0 1 64210570
6   LEAVE                 1  31463791511091  MPI_Bcast 11
```

```
1 bcast 4
```

Figure 3.15: Score-P - List of callbacks related to a call to the `MPI_Bcast` function and the corresponding *time-independent* trace.

The `MPI_Allreduce` function is presented in Figure 3.16. On line 4, after the name of the MPI function, are declared the MPI communicator and the sum of sent and received bytes. In this example four processes were involved, thus the size of each message for the `allreduce` action is 40 bytes. Moreover from the `METRIC` events we calculate how many instructions are computed during the reduction operation.

As for TAU, it is not possible to extract all the necessary data regarding a call to the `MPI_Alltoallv` function and other similar collective operations because the traces provide only generic values about the size of the messages. For example Figure 3.17 presents callbacks related to the `MPI_Alltoallv` function. The `alltoallv` event provides the MPI communicator and the sum of the sent and received size of the messages (in bytes).

```
METRIC                1  31469683710435  0 1 1455206247
ENTER                 1  31469683710435  MPI_Allreduce 3
MPI_COLLECTIVE_BEGIN  1  31469683710435
MPI_COLLECTIVE_END    1  31469701519843  ALLREDUCE 0 160 160
METRIC                1  31469701519843  0 1 1455731447
LEAVE                 1  31469701519843  MPI_Allreduce 3
```

```
1 allReduce 40 525200
```

Figure 3.16: Score-P - List of callbacks related to a call to the MPI_Allreduce function and the corresponding *time-independent* trace.

```
METRIC                0  25779008790450  0 1 18240739155
ENTER                 0  25778956847070  MPI_Alltoallv 5
MPI_COLLECTIVE_BEGIN  0  25778956847070
MPI_COLLECTIVE_END    0  25779008790450  ALLTOALLV 0 8388608 8398040
METRIC                0  25779008790450  0 1 18240984476
LEAVE                 0  25779008790450  MPI_Alltoallv 5
```

Figure 3.17: Score-P - List of callbacks related to a call to the MPI_Alltoallv function.

In this case we do not know exactly the size of the messages that a process sends or receives and the displacements of the array that are given as input in the MPI_Alltoallv function. We should mention that the last two elements of the MPI_COLLECTIVE_END declare the size of the messages (in bytes) that were sent and received respectively.

MinI

An application instrumented with MinI directly produces *time-independent* traces during its execution, thus there is no need for a separate tool. Nevertheless, we present how MinI handles calls to some MPI functions. This is a more generic approach and for sake of simplicity we do not show the whole code for each MPI call, *e.g.,* all the variables are considered as declared.

One of the main advantages in comparison to the previous tools is that we can extract the data type of each message for all the MPI communications. In order to use this information into the *time-independent* traces, we use the function encode_data_type which basically maps each MPI data type to an integer (Figure 3.18).

MinI creates the *time-independent* traces as a wrapper on calls to PMPI. Figure 3.19 shows the corresponding code for the MPI_Send function.

On line 3 the value of the PAPI counter is saved into the values array and we declare that only one counter is used. On line 4 we the save the data type of the sent data in the variable t_data. We obtain the rank of the MPI process through a call to PMPI_Comm_rank (line 6) and produce the compute action that preceeds this send (line 8). From lines 11 to 16, we produce the send action that comprises the rank of the sender, the send keywork, the rank of the receiver, the message syze, and optionally the message type. The type is optional to reduce the size of the produce traces when the same type is used by the whole application, e.g., MPI_DOUBLE_PRECISION. On line 16, PMI_Send is called to actually perform the communication. The last two lines initiate the count of the number of instructions that will be stopped in the next MPI call.

In the calles to MPI_Irecv and MPI_wait, we rely on a temporary buffer to save a local time-independent trace between these two calls. We obtain the id of the sender from the MPI_Status data structure during the MPI_Wait call.

```
int encode_data_type(const char *dat) {
  int res=0;
  char datat[30];
  if(!strcmp("MPI_DOUBLE_PRECISION",dat) || !strcmp("MPI_DOUBLE",dat))
      res=0;
  else if(!strcmp("MPI_INTEGER",dat) || !strcmp("MPI_INT",dat))
      res=1;
  else if(!strcmp("MPI_CHARACTER",dat) || !strcmp("MPI_CHAR",dat))
      res=2;
  else if(!strcmp("MPI_SHORT",dat))
      res=3;
  else if(!strcmp("MPI_LONG",dat))
      res=4;
  else if(!strcmp("MPI_REAL",dat) || !strcmp("MPI_FLOAT",dat))
      res=5;
  else if(!strcmp("MPI_BYTE",dat))
      res=6;

  return res;
}
```

Figure 3.18: MinI – Encoding MPI data types to an integer.

```
int  MPI_Send( buf, count, datatype, dest, tag, comm){
  PAPI_accum_counters(values, 1);
  MPI_Type_get_name(datatype,t_data,&np);
  this_type=encode_datatype(&t_data);
  PMPI_Comm_rank( MPI_COMM_WORLD, &llrank );

  sprintf(msg,"%d compute %lld\n",llrank, values[0]-previous_value);

  if(!this_data)
    sprintf(msg, "%d send %d %d\n", llrank,dest,count);
  else
    sprintf(msg, "%d send %d %d %d\n", llrank,dest,count,this_type);

  returnVal = PMPI_Send( buf, count, datatype, dest, tag, comm );

  PAPI_accum_counters(values, 1);
  previous_value=values[0];
}
```

```
1 send 0 8
```

Figure 3.19: MinI – Example of code for the `MPI_Send` function and the corresponding *time-independent* trace.

The `MPI_Bcast` function is given in Figure 3.23. Here we just show another way to reduce trace size thanks to default values. If a collective is rooted in process of rank 0 and the data type is `MPI_DOUBLE_PRECISION`, we do not include this information in the trace. Otherwise a `bcast` action will have two more entries.

For the `MPI_Allreduce` function, presented in Figure 3.24, each process computes the number of instructions it has executed during this call and report it in the produced action. Again, saving the data type is optional.

Similar the `MPI_Bcast` function, is presented in Figure 3.20. The most details are similar to other MPI functions and are not presented, however we wanted to show that during a collective communication the root node is known. We considered the default value of the root node equal to zero, thus are not declared into the *time-independent* traces in the case that the values of both root node and the `this_type` are equal to zero. In this example the message size is 4 bytes, the root node is equal to one and the data type value is equal to two, which means from the function `encode_data_type` that the message's data type is `MPI_CHAR`.

```
int  MPI_Bcast(buffer, count, datatype, root, comm){
  if(!root || !this_type)
    sprintf(msg, "%d bcast %d %d %d\n", llrank,count);
  else
    sprintf(msg, "%d bcast %d\n",llrank,count,root,this_type);
}
```

```
1 bcast 4 1 2
```

Figure 3.20: MinI – Example of code for the `MPI_Bcast` function and the corresponding *time-independent* trace.

Similar the `MPI_Allreduce` function, is presented in Figure 3.21. In this case, we compute the instructions that the call to the `MPI_Allreduce` function executes and are declared in the corresponding action with the message's data type if is not the default one.

```
int   MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm){
  PAPI_accum_counters(values, 1);
  ins1=values[0];

  returnVal = PMPI_Allreduce( sendbuf, recvbuf, count,
                              datatype, op, comm );

  PAPI_accum_counters(values, 1);

  sprintf(msg, "%d comm_size %d\n",llrank,global);
  if(!this_type)
    sprintf(msg, "%d allReduce %d %lld %d\n",
                     llrank,count,values[0]-ins1);
  else
    sprintf(msg, "%d allReduce %d %lld\n",
                     llrank,count,values[0]-ins1,this_type);

  PAPI_accum_counters(values, 1);
  ins1=values[0];
}
```

```
1 allReduce 40 525200
```

Figure 3.21: MinI – Example of code for the `MPI_Allreduce` function and the corresponding *time-independent* trace.

The code for the function `MPI_Alltoallv` in MinI is more complicated. hen we only present in Figure 3.22 the output of this wrapper.

```
1 allToAllV 2106083 524214 529389 523091 520458 0 524214 1053603 1576694
2100323 524214 525845 524419 525031 0 524214 1050059 1574478 1
```

Figure 3.22: MinI – Action corresponding to a call to `MPI_Alltoallv` in a time-independent trace.

The elements of this action depend on the number of participating processes. In this example we execute this collective operation on 4 processes. The first element after the name of the action is the length of the send buffer, the next four values correspond to the number of elements respectively sent to the other processes, then the other four values correspond to the displacements according to the MPI standards. Symmetrically we find the length of the receive buffer, and the values of the number of elements that the processors can receive and their displacements. The last value which is equal to one, declares that all the previous values are integers.

Once extracted, and whatever the extraction method, time-independent traces can be injected into the simulation framework. However, they first have to be gathered on a single node onto which the simulation will take place.

3.3.4 Trace Gathering

One challenge for trace acquisition is the time needed to aggregate (potentially large) trace files generated on a large number of compute nodes. This corresponds to a standard "gather" collective communication operation. The Grid'5000 platform provides a tool called `kaget` to gather remote files into a single node. This tool is based on `taktuk` and provides a simple interface without the need for the user to be familiar with the complicated commands of `taktuk`. We tested this tool to assess its performance and unfortunately the performance was not satisfying. Then, we implemented another tool based on a different gathering algorithm called `trace_gather`. It is based on a K-nomial reduction tree that allows for an 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. This tool can be efficient by picking an appropriate K value given to the number of trace files and the number of compute nodes. Figure 3.23 shows how this tool handles the traces produced on 16 nodes.

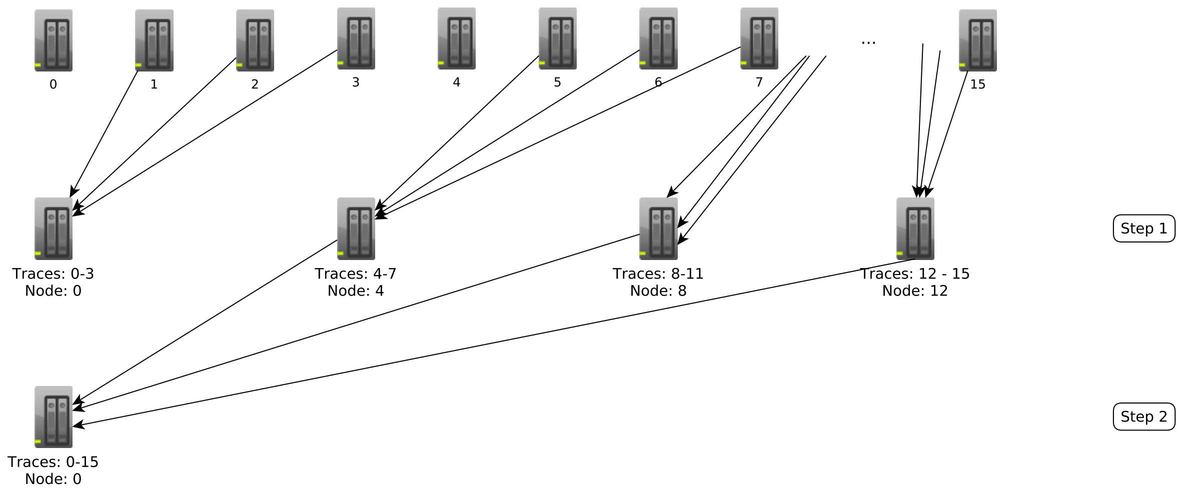


Figure 3.23: Example of the gathering algorithm for 16 nodes and arity of tree equal to four.

Table 3.25 presents a comparison of the relative performance of `kaget` and `trace_gather`. Our tool is 2.46 and 2.11 times faster than `kaget` on average for the instances B-64 and C-64 respectively. The results indicate that we should use our implementation to gather the *time-independent* traces into a single node. Moreover our tool can compress data before the first step to transfer less data between the nodes.

	Gathering time for LU (in sec)					
	B-64			C-64		
	Kaget	Trace_gather	$\frac{Kaget}{Trace\_gather}$	Kaget	Trace_gather	$\frac{Kaget}{Trace\_gather}$
min	12.33	4.93	2.5	19.27	10.06	1.92
max	13.98	5.7	2.45	23.98	10.4	2.31
avg	13.07	5.32	2.46	21.6	10.26	2.11
std	0.51	0.25	2.04	1.51	0.12	12.58

Table 3.25: Comparing the gathering tools `kaget` and `trace_gather` for the LU benchmark, 64 nodes, and classes B and C.

3.3.5 Analysis of Trace sizes

We used two instrumentation methods with TAU. Table 3.26 shows the TAU trace sizes with both methods. With TAU-reduced the trace sizes are decreased from 35% to 44% in comparison to the selective instrumentation. This leads to smaller time and space overheads.

Table 3.26: TAU trace sizes and number of actions for different instances of the LU benchmark.

#Processes	TAU trace size (in MB)			
	Selective Instrumentation		TAU-reduced	
	Class B	Class C	Class B	Class C
8	334	531	188	298
16	741	1,200	450	714
32	1,600	2,500	973	1,600
64	3,200	5,100	2,100	3,300
128	6,600	11,000	4,300	6,800

One issue with off-line simulation is the large size of the traces. The *time-independent* trace format was improved through our different versions of our framework. Although we studied various benchmarks we present the *time-independent* trace sizes only for the LU benchmark. For EP, each process executes a big block of instructions. Then the trace of a given process has only one line. For DT, the number of actions is also relatively small. The processes that execute the most actions are *comparators* that receive data from four other processes, merge them, and forward the aggregated data. Consequently the total trace size for both benchmarks is at most a few hundreds of kilobytes for class C instances. With LU, computations and communications are interleaved and each process logs a lot of actions. The size of the *time-independent* traces depends on the number of actions executed by the processes. Table 3.27 shows trace sizes for different instances of the LU benchmark as well as the number of actions in each trace for the prototype framework [101].

The size of *time-independent* traces grows linearly with the number of processes which is explained by the evolution of the number of traced events. We also see that the size of the time-independent traces grows from a constant factor of 1.6 from class B to class C which is also directly related to the number of actions. Table 3.27 presents the trace sizes for the selective instrumentation method. The ratio of size to the number of actions, that denotes the average number of characters per action, is roughly constant (from 14.72 to 15.29). Moreover the traces are extracted from the main core of the benchmark. It does not include the initialization

Table 3.27: Sizes of time-independent traces and number of actions for different instances of the LU benchmark for the prototype framework.

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

and finalization phases. Table 3.28 shows trace sizes obtained with the MinI instrumentation method for the whole benchmark. Then there are more actions for each instance in comparison to Table 3.27. Dividing the trace size by the number of actions also leads to a roughly constant ratio (between 14.45 and 15.94). Note that while the number of the actions are increased from 0.3% up to 0.5%, the trace sizes are globally decreased from 0.2% to 1.7%.

Table 3.28: Sizes of time-independent traces and number of actions for different instances of the LU benchmark produced with the MinI tool.

#Processes	Trace size (in MB)		#Actions (in millions)	
	Class B	Class C	Class B	Class C
8	29.6	48	2.04	3.24
16	72	116.8	4.89	7.78
32	159	255	10.6	16.87
64	339	550	22.83	36.33
128	711	1,200	47.3	75.25

However, our trace format is not optimized for space. Authors have proposed compact trace representations [61, 102]. In our case, we could devise a binary trace format that would remove most of the redundant characters. Alternately, we can simply use compression algorithms for our text trace files, which reduces trace file size significantly but not as much as if a custom compact trace format were used. When showing the scalability of our trace acquisition procedure in the next section we first present results without any compaction/compression, and then some results using compression. Now that the whole acquisition procedure has been presented in detail, we can evaluate it in order to identify any bottleneck.

3.3.6 Evaluation of the Acquisition Procedure

Trace Acquisition Scalability

We have already mentioned the three different instrumentation methods, *selective instrumentation*, *TAU-reduced*, and *Minimal Instrumentation* where each one evolves the previous one. We studied our prototype framework (that uses the selective instrumentation) mainly on AMD processors, thus we used the *bordereau* [103] cluster. The *bordereau* cluster comprises 93 2.6 GHz Dual-Proc, Dual-Core AMD Opteron 2218 nodes. All these nodes are connected to a single 10 Gigabit switch. One way to demonstrate the scalability of our trace acquisition procedure is to investigate the distribution of elapsed times for each step of the procedure, *i.e.,* execution, instrumentation, extraction, and gathering. Figure 3.24 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 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.

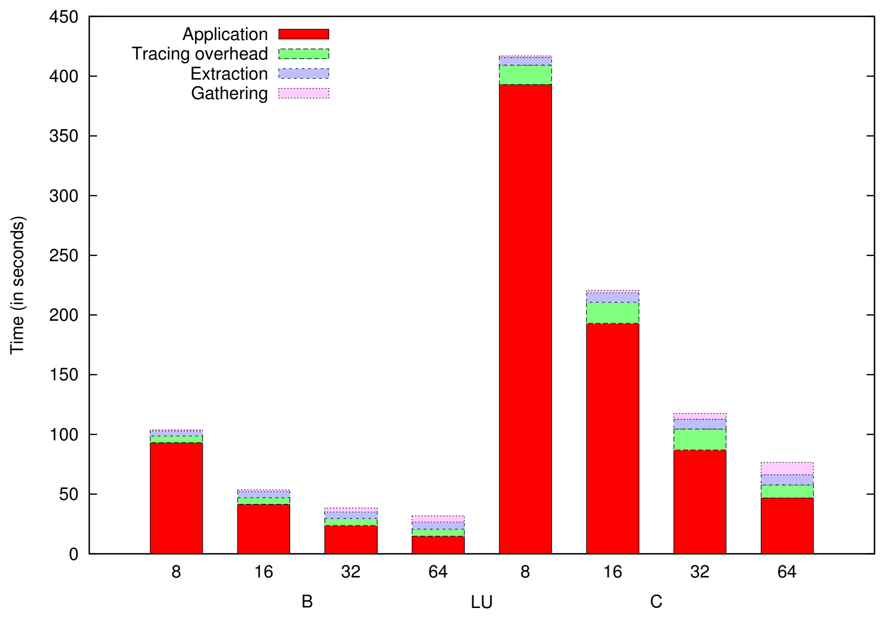


Figure 3.24: Distribution of the acquisition time for different instances of the LU benchmark in the *regular* acquisition mode, on *bordereau* cluster with selective instrumentation.

The acquisition time is strictly related to the production of the *time-independent* traces i.e., the instrumentation, extraction, and gathering represent at most 53.34% of the total acquisition time. The worst value is obtained for class B with 64 processes where the benchmark execution time is the smallest. However, large number of processes are generally used to solve large problem instances. Then we can conclude that the extra overhead required to get a *time-independent* trace can be afforded. Moreover such traces need to be acquired only once and can then be used to explore much more "what-if" scenarios that with timed traces.

As the *bordereau* cluster was aging with many technical issues and we wanted to evaluate also our framework on Intel processors, we used for the next version of our framework the *graphene* cluster. It has been already mentioned that the nodes of this cluster are spread over four cabinets, thus we expect to have some communication delays in comparison to the *bordereau* cluster. However, thanks to the TAU-reduced instrumentation method, the trace sizes are smaller than before, so the instrumentation overhead should be smaller with regard to both time and space.

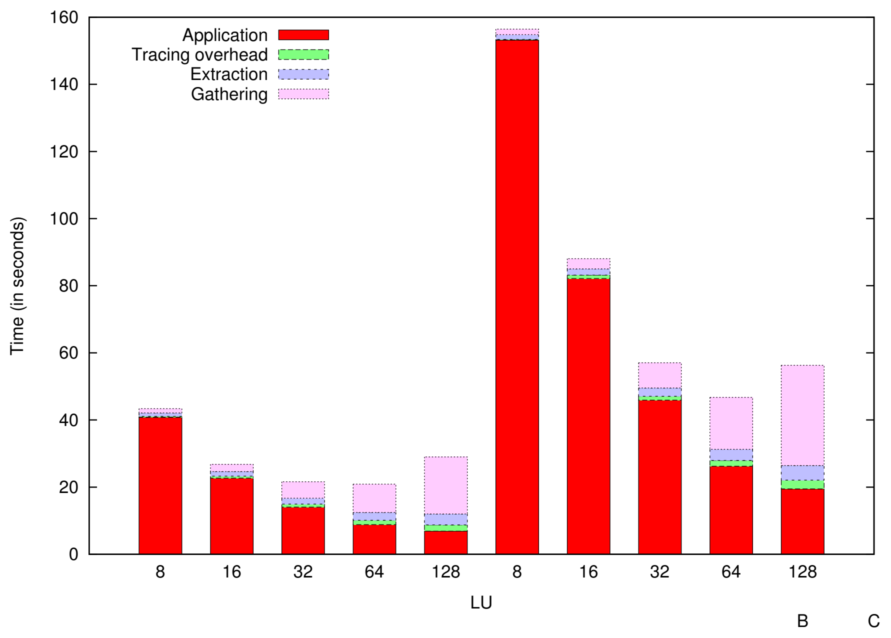


Figure 3.25: Distribution of the acquisition time for different instances of the LU benchmark in the *regular* acquisition mode on graphene cluster with TAU-reduced instrumentation.

In Figure 3.25 the instrumentation, extraction, and gathering represent at most 76.24% of the total acquisition time for class B with 64 processes. 77% of this time overhead is caused by the gathering procedure which depends on the network speed and topology. To compare the first two versions of our framework, we executed some experiments on the *graphene* cluster. Figure 3.26 shows the evolution of the instrumentation overhead. With the first implementation of our framework, the overhead goes from 9.8% to 37.9%, which is consistent with the experiments on the *bordereau* cluster. The proposed modifications make the overhead range shrink to [0.7% - 27.1%]. More interestingly, we can see a stable growth of this overhead as the number of processes increases.

Moreover the TAU-reduced method instruments only the MPI calls of an application. According to Table 3.26 the trace sizes are smaller, then the extraction of the *time-independent* traces from these TAU traces takes less time than the prototype implementation of our framework. However, the gathering of the traces takes more time because of the *graphene*'s network topology.

Then we used the MinI tool for acquiring the *time-independent* traces. It has been already mentioned that an instrumented application with MinI produces directly the *time-independent* traces, thus there is no extraction step. Considering that it uses only the absolute necessary features to profile an application, MinI is also more efficient than TAU. Moreover we used a compression option in the gathering tool to decrease the duration of this procedure. The obtained results are presented in Figure 3.27.

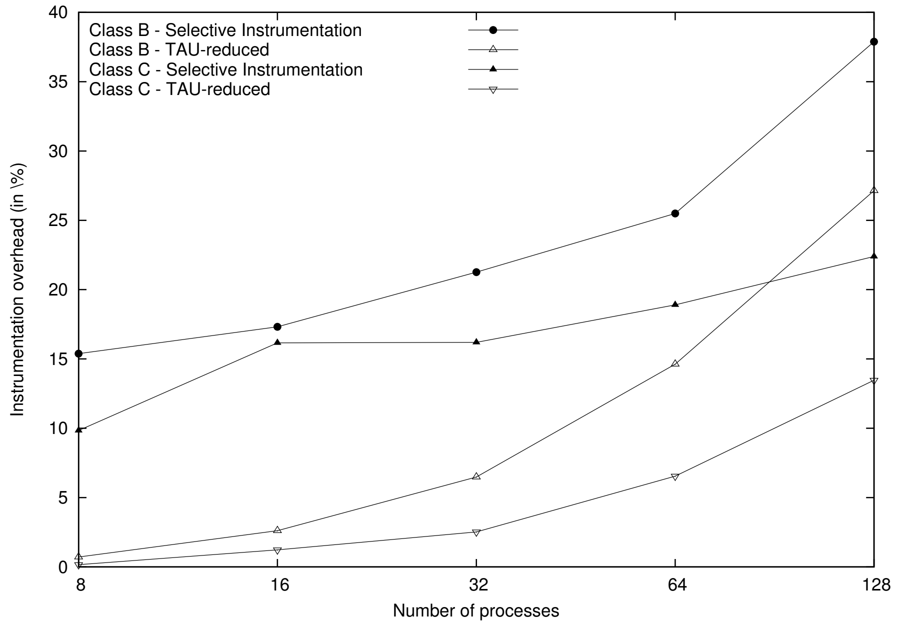


Figure 3.26: Evolution of the instrumentation overhead induced by the selective instrumentation and the TAU-reduced when the number of processes varies. Results obtained on *graphene*.

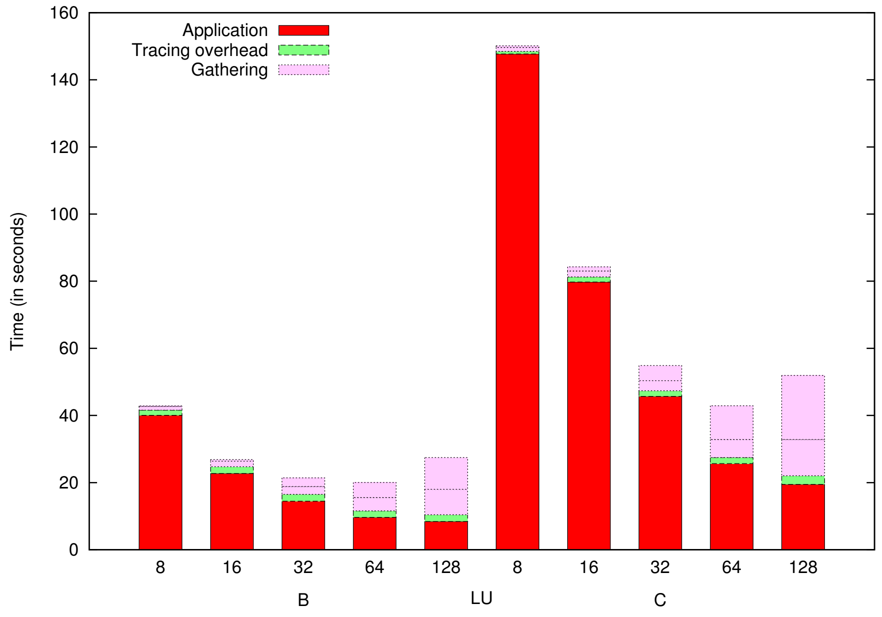


Figure 3.27: Distribution of the acquisition time for different instances of the LU benchmark in the *regular* acquisition mode on graphene cluster with minimal instrumentation.

The main observation on the results in Figure 3.27 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 part of Figure 3.27 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 then be divided by up to three. 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 process 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 3.6, are due to cases in which each process performs little computation due to distribution of a small dataset across many processes. Thus our final framework provides less tracing overhead than TAU and direct compatibility with the appropriate trace format.

The results in Figure 3.27 are for at most 128 processes. To further 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 two larger-scale experiments. In the first experiment, we use folded execution to execute the instrumented LU benchmark on a single node of a cluster called *St-Rémi*. This node comprises two twelve-core processors that share 48 GiB of memory. 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 3.29 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 (with selective instrumentation enabled), the Scalasca, and the Score-P profiling tools. We find that TAU cannot produce traces (i.e., it fails) for the larger benchmark instance due to large memory footprint. The results for our minimal instrumentation are in line with those achieved by Scalasca, with some improvement over Scalasca in all cases. But it is important to remind that Scalasca does not produce traces with all the information we need for our purpose, so the Scalasca results in the table are optimistic. The memory footprint of Score-P is three to four times larger than that of our minimal instrumentation. For large instances, such as the class D executed with 256 processes, Score-P begins to swap, leading to prohibitively long acquisition times. 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.

Table 3.29: Executing the instrumented LU benchmark using folding on a single node.

	Time in minutes / Memory footprint in GiB			
Instance	TAU Reduced	Minimal Instrumentation	Scalasca	Score-P
B - 256	2.8 / 21.4	1.9 / 1.65	2.1 / 2.8	1.8 / 5.2
C - 1024	N/A	12.9 / 7.95	16.3 / 12.9	27.3 / 29.3
D - 256	N/A	47.4 / 15.4	55.2 / 16.9	> 420 / N/A

In the second experiment, we run a very large-scale application on a heterogeneous platform, i.e., combining the folded and composite modes. More specifically, we run class E of the LU benchmark, with 16,384 processes. To run this large instance, we distributed trace acquisition across 778 compute nodes in 18 clusters at 9 geographically distant sites of the Grid'5000 experimental testbed. 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. As for the previous experiment, TAU fails to produce a trace due to its large memory footprint. We did not use Scalasca as it does not meet all the requirements for obtaining a useful trace. We stopped the execution with Score-P after more than nine hours, without obtaining the desired trace. Our minimal instrumentation method was successful in acquiring the trace, producing a 1.45 TiB time-independent trace. 53 minutes were needed to execute the instrumented application. 16 minutes were needed to gather and aggregate individual trace files on a single node using a K-nomial tree. This time includes a gzip compression step. Decompressing the files on a single node is a more expensive process that added around 40 minutes of computation. For such a large trace, using compression/decompression was worthwhile. Note that systems exist for adaptive compression so as to achieve a good trade-off between compression time and data transfer time [104]. In the end, our approach was 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.

Evaluation of the Execution Modes

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

Table 3.30 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-(u, v)) acquisition modes, instrumented with the selective instrumentation. 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 used for the acquisition. Finally when both modes are combined, CF-(u, v) means that the execution is distributed over u sites and that v processes run on each CPU. In the composite mode, y is the number of sites used for the acquisition. Finally when both modes are combined, CF-(u, v) means that the execution is distributed over u sites and that v processes run on each CPU. Again, we use only one core per node.

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

Table 3.30: Evolution of the execution time of instrumented class B instance of EP, DT and LU executed by 64 processes (43 for DT) with regard to the acquisition mode. Results obtained on the *bordereau* and *gdx* clusters using one core per node.

	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.)	20.73	52.96	88.66	179.07	347.27	689.18	37.54	79.19	134.05	277.25	505.64
	Ratio to regular mode	1	2.55	4.28	8.64	16.75	33.25	1.81	3.82	6.47	13.37	24.39
EP	Execution Time (in sec.)	1.99	4.06	8.33	16.29	36.42	69.05	2.67	5.38	10.77	21.52	43.09
	Ratio to regular mode	1	2.04	4.18	8.18	18.3	34.7	1.34	2.7	5.41	10.81	21.65

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

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

An interesting property of *time-independent* traces is exemplified 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 and 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. Preventing such a behavior would require an accurate description of the acquisition platform along with the trace. With time-independent traces, the simulated time is totally independent of the acquisition scenario. Only slight variations (under 1%) are observed caused by hardware counter accuracy issue, and in fact a dedicated platform is not even required.

Similarly, Table 3.31 shows the execution time of the LU benchmark (class B) instrumented with the MinI tool. The execution times are obtained on the *graphene* cluster 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 performance, around 8Gflop/s per core, as measured with the HPLinpack benchmark, and we use one core per node in this experiment.

Table 3.31: 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.*

	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

We obtained results and trends similar to those in Table 3.30 with regard to the slowdown factors. However, we observed an impact of the network of the second cluster, *paradent* on the performance of the different acquisition modes. Indeed, while the performance of the folded mode is improved (from 11.21% to 24.72%) thanks to the minimal instrumentation, the composite mode takes more time (around 15%). Moreover, when both modes are combined, the improvement is lower than with the folded mode alone (from 3.24% to 21.4% only). Nervertheless, the benefits offered by the minimal instrumentation in terms of overhead and skew do not compromise the use of the folded and composite acquisition modes.

3.4 Conclusion

In this chapter we presented in detail the different steps of our time-independent trace acquisition framework. We explained what occurs during the instrumentation of an application and introduced the *time-independent* trace format. We analyzed all the possible actions of a *time-independent* trace to show the corresponding requirements that a tracing tool should respect to produce such traces. According to these requirements we evaluated many open source profiling tools among the most popular. As none of the studied fulfilled our requirements without inducing important overheads, instrumentation skew, and/or large memory footprint, we also proposed another instrumentation method to decrease the instrumentation overhead and improve the accuracy of the measured values both for time and hardware counters. This original instrumentation method is called MinI. This tool was developed with the necessary options only and provides a direct compatibility with the *time-independent* traces as this is its default trace output format, leading to minimal instrumentation overhead and skew. Moreover four execution modes were proposed and we showed how we can take advantage of most of them in comparison to other frameworks. We also explained how we can use our framework on the Grid'5000 platform. Furthermore we presented in detail two tools, one for extracting *time-independent* traces produced by TAU and Score-P and another one for gathering these traces into one single node. We did further analysis of trace sizes for both TAU tool and *time-independent* trace framework. An evaluation of the full acquisition procedure was conducted to show the improvement of our framework since the prototype version and present in detail the corresponding overheads. Finally we succeed in applying our framework on 778 compute nodes by executing an instrumented application with 16,384 processes and acquire the corresponding *time-independent* traces to prove its scalability. Now that *time-independent* traces can be acquired for an application, we can use them to predict the performance of applications. The replay of time-independent traces will be detailed in the next Chapter.

Chapter 4

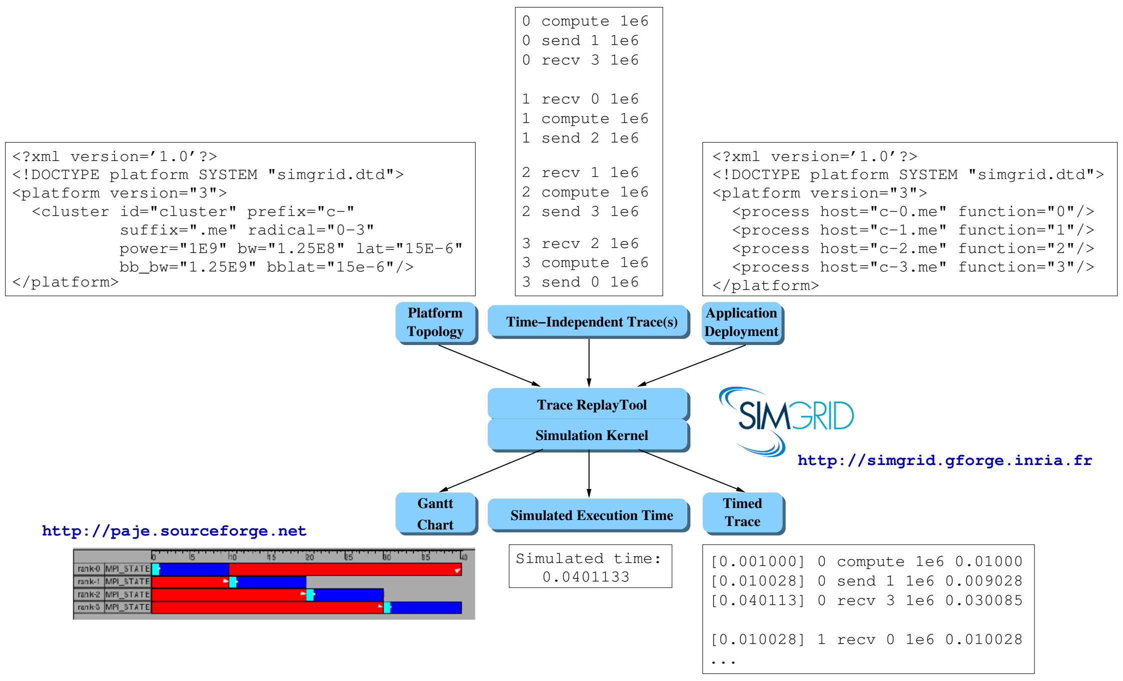
Time-Independent Trace Replay

In the previous Chapter we described the *time-independent* trace acquisition framework. We detailed all the procedures required to obtain a *time-independent* trace from an application. In the current Chapter we present how to replay *time-independent* traces by using a simulator based on the SimGrid simulation toolkit.

Since release 3.3.3, SimGrid allows users to describe an applicative workload as a time-independent trace. As shown in the upper part of Figure 4.1, three input files are needed to replay such traces with SimGrid. Apart from the *time-independent trace(s)*, descriptions of the *simulated platform* and of the *deployment* of the application, *i.e.,* how simulated processes are mapped onto simulated processors, are also needed. 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 "what if?" scenarios can be explored without modifying of the simulator and instead simply changing the input files.

The platform file on the upper left part of Figure 4.1 declares a cluster of 4 nodes with the *cluster* tag. This tag is used to quickly declare a bunch of homogeneous machines. The *cluster* is a meta-tag from SimGrid point of view. It represents an Autonomous System (AS) in which is defined some optimized routing mechanisms. There are have four nodes in this cluster, named c-[0-3].me, whose processing power is one billion instructions per second. The cluster's network bandwidth is 1Gb/s (or 1.25E8B/s), the latency for both the cluster links and the backbone are 15 microseconds, and the backbone bandwidth is 10Gb/s (or 1.25E9B/s). The *time-independent* trace in the upper middle part correspond to the example in Figure 3.1, where each process on a ring computes one million instructions and sends one million bytes to its neighbor. The deployment file on the upper right corner of Figure 4.1 maps the simulated processes on the platform. For example the simulated process of rank 0 is mapped to the node c-0.me. Our replay framework can produce various types of output as indicated in the bottom part of Figure 4.1. First we can obtain 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 a time-stamped 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 [105].

The remaining of this Chapter is organized as follows. In Section 4.1 we go further than the



```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="3">
  <cluster id="cluster" prefix="c-"
           suffix=".me" radical="0-3"
           power="1E9" bw="1.25E8" lat="15E-6"
           bb_bw="1.25E9" bblat="15e-6"/>
</platform>
```

```
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
```

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="3">
  <process host="c-0.me" function="0"/>
  <process host="c-1.me" function="1"/>
  <process host="c-2.me" function="2"/>
  <process host="c-3.me" function="3"/>
</platform>
```

```
Simulated time:
  0.0401133
```

```
[0.001000] 0 compute 1e6 0.01000
[0.010028] 0 send 1 1e6 0.009028
[0.040113] 0 recv 3 1e6 0.030085

[0.010028] 1 recv 0 1e6 0.010028
...
```

Figure 4.1: Overview of the *time-independent* replay framework.

simple platform description given in Figure 4.1 by detailing how to describe and instantiate realistic platforms. Section 4.2 details how is implemented the trace replay tool. Finally Section 4.3 presents some simulation results.

4.1 Realistic Platform Descriptions

Reflecting Hierarchical Topologies

Before illustrating on an example how to describe compute clusters in order to replay *time-independent*, we recall the motivations that led to the proposition of a scalable multi-purpose representation of network topologies in the SimGrid toolkit [106]. The key concept is to provide users with the ability to adapt the representation to their needs. Most current distributed systems mainly rely on three types of possible interconnected and mixed networks:

– System and local area networks (SAN and LAN) are organized in a very hierarchical way or with large switches;
– National Research and Education Networks (NRENs) interconnect systems through their gateways on a higher level of networks, called backbone which are scattered at national scale;
– Privately and idenpendently managed networks.

Such networks are interconnected in a hierarchical way ans large scale distributed systems can thus be seen as a hierarchical aggregation of networks. Each network is independent of the others and can have a different structure, hence the name of Autonomous Systems (AS). Each AS can be connected with lower or higher level ASes. For each AS there is at least one gateway to compute the route between two ASes even if the one is on higher level. There are various graph representations for the network topology and routing such as *Flat*, *Dijkstra* [107], *Floyd* [107] etc., but one main problem is that they do not exploit the hierarchy and the regularity of the platform [108]. The SimGrid toolkit handles most of these routing schemes and can determine routes between hosts belonging to different ASes by looking for the first common ancestor in the hierarchy.

In what follows, we detail the different steps that allowed us to propose a highly realistic description of the *graphene* cluster that was used in many of our experiments. As mentioned in the previous Chapter, *graphene* 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. The simplest description of this cluster is given by Figure 4.2

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
  <AS id="AS_graphene" routing="Full" >
     <cluster id="AS_sgraphene1"
              prefix="graphene-" suffix=".nancy.grid5000.fr" radical="0-143"
              power="1E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
  </AS>
</platform>
```

Figure 4.2: Simplest platform file describing the *graphene* cluster.

The different attributes of the `<cluster>` tag provide the minimal information to describe the cluster's characteristics and performance. The name of the nodes that compose the cluster and formed from the `prefix`, `suffix`, and `radical` attributes. The radical is a classical regular expression that allows users to describe non-contiguous ranges of nodes. In this example, the names are then `graphene-[0-143].nancy.grid5000.fr`. The `power` attribute indicates the processing speed of a node in number of instructions processed per second. The four remaining attributes respectively define the latency and bandwidth of the network links the connect each node to a backbone (prefixed by `bb_`). The main problem with this simplistic description is that it does not reflect the hierarchical organization in cabinets of the cluster.

Figure 4.3 improves this description by distinguishing the four cabinets as four ASes named `AS_graphene[1-4]`, each having a different range of nodes. These four ASes belong to a single AS, called `AS_graphene`, that represents the whole cluster. Each cabinet is connected to a central backbone, called `switch-graphene`, with its own bandwidth and latency. As the main AS follows the `Full` routind method, we have to declare how cabinets are interconneted, thank to the `<ASroute>` tags. For sake of simplicity we show only the connection between two pair of cabinets. The `<ASroute>` tag declares a route between two ASes. In our example, this route is between `AS_graphene1` and `AS_graphene2`. The access point (or *gateway*) of the source (resp. destination) AS is given in the `gw_src` (resp. `gw_dst`) attribute. There is a single link between these two gateways, namely `switch-graphene`.

As mentioned in Figure 4.1, another input file is required by our trace reaply tool. It is the deployment file that maps each simulated process on a node of the platform. A typical example of deployement for the *graphene* cluster is presented in Figure 4.4. It shows that the the process of rank 0 is mapped onto the `graphene-0.nancy.grid5000.fr` node. Similar information is given for all the other nodes in the cluster.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
  <AS id="AS_graphene" routing="Full" >
     <cluster  id="AS_sgraphene1"
             prefix="graphene-" suffix=".nancy.grid5000.fr" radical= "0-38"
             power="1E9" bw="1.25E8" lat="15E-6"  bb_bw="1.25E9" bb_lat="15E-6"/>
     <cluster  id="AS_sgraphene2"
             prefix="graphene-" suffix=".nancy.grid5000.fr" radical= "39-73"
             power="1E9" bw="1.25E8" lat="15E-6"  bb_bw="1.25E9" bb_lat="15E-6"/>
     <cluster  id="AS_sgraphene3"
             prefix="graphene-" suffix=".nancy.grid5000.fr"  radical= "74-103"
             power="1E9" bw="1.25E8" lat="15E-6"  bb_bw="1.25E9" bb_lat="15E-6"/>
     <cluster  id="AS_sgraphene4"
             prefix="graphene-" suffix=".nancy.grid5000.fr"  radical= "104-143"
             power="1E9" bw="1.25E8" lat="15E-6"  bb_bw="1.25E9" bb_lat="15E-6"/>

      <link id="switch-graphene" bandwidth="1.25E9" latency="5E-4"/>

      <ASroute src="AS_sgraphene1" dst="AS_sgraphene2"
            gw_src="graphene-AS_sgraphene1_router.nancy.grid5000.fr"
            gw_dst="graphene-AS_sgraphene2_router.nancy.grid5000.fr">
            <link_ctn id="switch-graphene"/>
      </ASroute>

        <ASroute src="AS_sgraphene1" dst="AS_sgraphene3"
               gw_src="graphene-AS_sgraphene1_router.nancy.grid5000.fr"
               gw_dst="graphene-AS_sgraphene3_router.nancy.grid5000.fr">
               <link_ctn id="switch-graphene"/>
        </ASroute>
    ...
  </AS>
</platform>
```

Figure 4.3: Example - Platform file for graphene cluster with network topology.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
<process host="graphene-0.nancy.grid5000.fr" function="0"/>
<process host="graphene-1.nancy.grid5000.fr" function="1"/>
...
<process host="graphene-142.nancy.grid5000.fr" function="142"/>
<process host="graphene-143.nancy.grid5000.fr" function="143"/>
</platform>
```

Figure 4.4: Deployment file for *graphene* cluster.

Calibration and Instantiation

A necessary step for obtaining accurate performance predictions through simulation is the calibration of the simulation tool. In our context, calibration is used to determine the rate at which a CPU processes instructions, the latency and bandwidth of communication links, and some other factors. These values are then used to instantiate the platform description file shown in Figure 4.3. Such a calibration strongly depends on both application and execution environment. Indeed different types of computation may lead to different processing rates on a given CPU. This is mainly due to how efficiently the computation can use the different levels of cache. Moreover the performance of a given computation may differ with regard to the processor brand.

Calibration of the CPU processing power To instantiate the `power` attribute of the `<cluster>` tag that represents the processing power of each host in the cluster, we proceed as follows. A *small instrumented instance* of the target application, *e.g.,* a class A instance for NAS parallel benchmarks, is run on a *small version* of the cluster to describe, *e.g.,* on up to four nodes. This allows us to determine the number of instructions of each event as long as the time spent to computed them. Then we can determine a processing rate of each single action, compute a weighted average on each process, and get an average processing rate for all the process set. To smooth the runtime variations, we repeat this procedure five times and compute an average over the five runs. We use this final value to instantiate the SimGrid platform file. It depends on the application, compilation options and target cluster characteristics. This calibration step allows us to acquire *time-independent* traces on any execution environment, as explained in Chapter 3, and simulate their compute part on a correctly instantiated description of the target cluster. For instance, the description given in Figure 4.5 has been updated with a processing power that correspond to a calibration of LU benchmark on the *graphene* cluster. The obtained rate is of 3.68E9 instructions per second and is homogeneous across the nodes.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">

<AS id="AS_graphene" routing="Full" >
  <cluster id="AS_sgraphene1"
           prefix="graphene-" suffix=".nancy.grid5000.fr" radical="0-38"
           power= "3.68E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
  <cluster id="AS_sgraphene2"
           prefix="graphene-" suffix=".nancy.grid5000.fr" radical="39-73"
           power= "3.68E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
  <cluster id="AS_sgraphene3"
           prefix="graphene-" suffix=".nancy.grid5000.fr" radical="74-103"
           power= "3.68E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
  <cluster id="AS_sgraphene4"
           prefix="graphene-" suffix=".nancy.grid5000.fr" radical="104-143"
           power= "3.68E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
   ...
  </AS>
</platform>
```

Figure 4.5: Updating the processing power after the calibration of the LU benchmark.

When analyzing the simulation results obtained for the LU benchmark in cite [101], we observed that the calibration procedure described above prevented us to obtain accurate predictions for some specific conditions on specific models of processors. More precisely we noted that we were generally overestimating the processing rate, leading to shorter simulated times. For instance, on some simulated clusters, using a calibration based on the execution of a class A instance with 4 processes prevented us to correctly predict the execution time of a class C instance executed with 8 processes. We suspected the large difference of data size managed by each process between the two instances to be the source of this inaccuracy. Then we investigated the use of additional hardware counters, such as the number of L2 cache misses or stalled cycles, to find some correlation with the processing rate. Even though we did not find anything relevant, having to rely on this counter would have been detrimental to our framework. Indeed, we explained in Section 3.3 that one of the acquisition mode allowed by *time-independent* traces consists in executing several MPI processes on a single core. In this case, the processes compete for the cache and the values of the hardware counters cannot be used anymore. Then we have to propose another method that does not compromise the benefit of the *time-independent* traces.

The rationale of the calibration method was to limit the number of resources and the time needed to do the calibration as much as possible. Experiments shown that while using only as few resources as four cores does not raise any issue, limiting the calibration to a small problem size hide some important phenomena. As already mentioned, in the particular case of the LU factorization, as soon as the share of the matrix owned by each process exceeds the capacity of the L2 cache, the performance drops, with a direct impact on the instruction rate. To circumvent this issue, we propose to complete our calibration procedure, when it is needed, with runs on larger problem sizes but on the same number of resources. In practical terms, we run the class B and C instances in addition to the class A instance, still on four nodes. Then we determine three different instruction rates, one per studied class. Depending on whether the current instance handles data that fit in the L2 cache or not, we use the rate coming from the calibration based on class A or the one that corresponds to the instance class.

Note that the use of this cache-aware calibration is mandatory only for the cases where the L2 cache of a processor is small (1MB per core only). On the *graphene* cluster that has a L2 cache of 2MB per core, all the instances do fit in cache. Calibrating the simulator with a run of the class A instance with four processes is then enough.

Communication Calibration 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 for networks ranging from local-area to wide-area networks. In [66], a simulator called SMPI, was added to the SimGrid toolkit for the on-line simulation of MPI applications on a single node. In order to specialize the generic network model for cluster interconnects, a model has been added to the SimGrid simulation kernel that takes into account the specifics of MPI implementations that use TCP on cluster interconnects. In [109] we propose a hybrid network model where more parameters are used to take under consideration the congestion which can occur during the execution of a parallel application. For the declared bandwidth in the `<cluster>` tags, the nominate value of the link is used. The measurements presented in Figure 4.6 were obtained according to the following protocol. To avoid any measurement bias, the message size is exponentially sampled from 1 byte to 100MiB. We ran two "ping" and one "ping-pong" experiments. The ping experiments aim at measuring the time spent in the `MPI_Send` (respectively `MPI_Recv`) function by ensuring that the receiver (respectively sender) is always ready to communicate. The ping-pong experiment allows us to measure the transmission delay. We ran our analysis on the whole set of raw measurements rather than on averaged values for each message size to prevent behavior smoothing and variability information loss. The rationale is to study the asynchronous part of MPI (from the application point of view) without any a priori assumptions of where switching may occur. We distinguish three modes of operation in Figure 4.7: asynchronous, detached, and synchronous. The variable k is the message size (in bytes), S_a, and S_d are the size thresholds of the asynchronous communication and detached communication respectively. In Figure 4.7, P_s and P_r represent the processes of the send and receiver respectively and the variable T_i declares the durations of the corresponding task. More information about this hybrid network model can be found in [109].

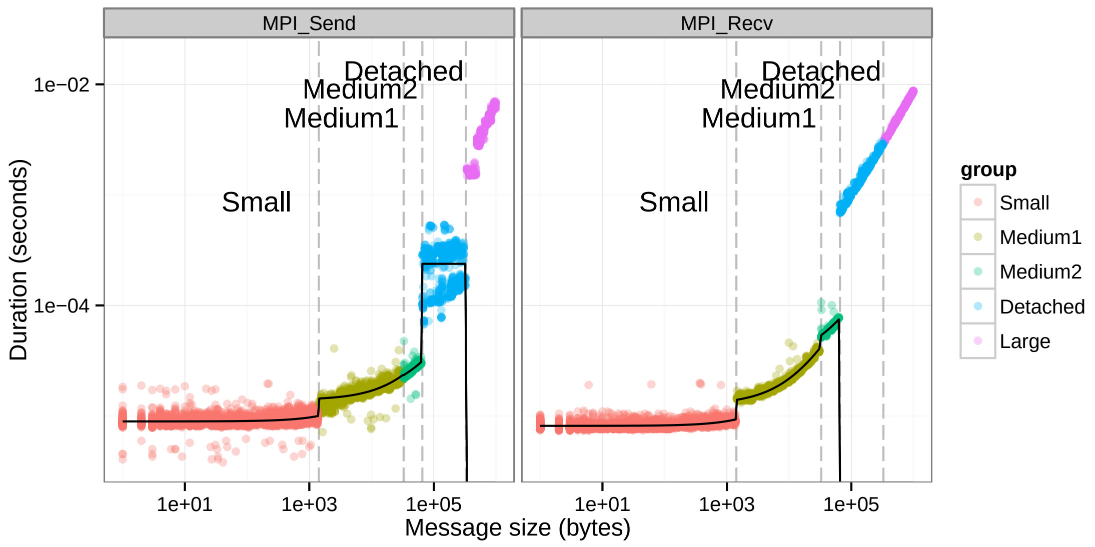
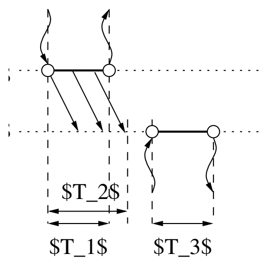


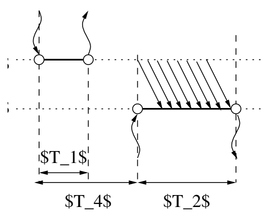
Figure 4.6: `MPI_Send` and `MPI_Recv` duration as a function of message size.

As illustrated by Figure 4.6, the duration of each mode can be accurately modeled through linear regression. It allows us to clearly identify different modes interpreted as follows:

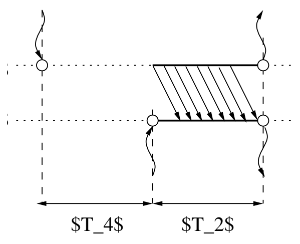
- **Small** (when $k \leq 1,420$): this mode corresponds to messages that fit in a TCP packet and are sent asynchronously by the kernel. As it induces memory copies, the duration significantly depends on the message size;
- **Medium** (when $1,420 < k \leq 32,768$ or $32,768 < k \leq 65,536 = S_a$): these messages are still sent asynchronously but incur a slight overhead compared to small messages, hence a discontinuity at $k = 1420$. The distinction at $k = 32,768$ does not really correspond



(a) SMPI Asynchronous mode ($k \leq S_a$)



(b) SMPI Detached mode ($S_a < k \leq S_d$)



(c) SMPI Synchronous mode ($k > S_d$)

Figure 4.7: The "hybrid" network model of SMPI in a nutshell.

to any particular threshold on the sender side but is visible on the receiver side where a small gap is noticed. Accounting for it is harmless and allows for a better linear fitting accounting for MPI/TCP peculiarities;

– **Detached** (when $65,536 < k \leq 327,680 = S_d$): this mode corresponds to messages that do not block the sender but require the receiver to post the reception before the communication actually takes place;

– **Large** (when $k > 327,680$): for such messages, both sender and receiver synchronize using a rendez-vous protocol before sending data. Except for the waiting time, the durations on the sender side and on the receiver side are very close.

All the parameters and values for the instantiation of this hybrid for the described cluster can be declared thanks to a configuration header in the XML file, as shown in Figure 4.8.

```
1   <?xml version='1.0'?>
2   <!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
3   <platform version="3">
4     <config id="General">
5       <prop id="workstation/model" value="compound"/>
6       <prop id="network/model" value="SMPI"/>
7
8       <prop id="smpi/async_small_thres" value="65536"/>
9       <prop id="smpi/send_is_detached_thres" value="327680"/>
10
11      <prop id= "smpi/os"
12            value="0:8.93009e-06:7.654382e-10; 1420:1.396843e-05:2.974094e-10;
13                   32768:1.540828e-05:2.441040e-10; 65536:0.000238:0; 327680:0:0"/>
14
15      <prop id= "smpi/or"
16            value="0:8.140255e-06:8.395881e-10; 1420:1.269952e-05:9.092182e-10;
17                   32768:3.095706e-05:6.956453e-10; 65536:0:0; 327680:0:0"/>
18
19      <prop id="smpi/bw_factor"
20            value="0:0.400977; 1420:0.913556; 32768:1.078319; 65536:0.956084; 327680:0.929868"/>
21
22      <prop id= "smpi/lat_factor"
23            value="0:1.35489; 1420:3.437250; 32768:5.721647;65536:11.988532; 327680:9.650420"/>
24    </config>
25
26    <AS id="AS_graphene" routing="Full" >
27      <cluster id="AS_sgraphene1"
28               prefix="graphene-" suffix=".nancy.grid5000.fr" radical="0-38"
29               power="3.68E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
30      <cluster id="AS_sgraphene2"
31               prefix="graphene-" suffix=".nancy.grid5000.fr" radical="39-73"
32               power="3.68E9" bw="1.25E8" lat="15E-6" bb_bw="1.25E9" bb_lat="15E-6"/>
33       ...
34   </AS>
35  </platform>
```

Figure 4.8: Configuring the hybrid network model.

First, we have to declare that the default network model is not the one to used by creating a compound (combination of CPU and network models) using the SMPI hybrid network model (lines 5 and 6). Then we declare the message size at which the protocol change from eager to rendez-vous (`async_small_thres` at 65,536 bytes) and from rendez-vous to detached (`send_is_detached_thres` at 327,680 bytes) (lines 7 and 8). From line 10 to line 17, we declare the values for the sender and receiver overheads (`smpi/os` and `smpi/or`. The values of these parameters are given by triplets $x : y : z$. For messages whose size is greater than x, the simulation kernel applies a delay of $z + x \times y$. This delay, that depends on the message size, basically corresponds to copying the message in memory. Finally, correction factors for the latency and bandwidth are declared from lines 19 to 23. This allows the model to mimic the behavior shown by Figure 4.6. They are declared as couples $x : y$. For messages whose size is greater than x, the factor y is applied to the nominal bandwidth (resp. latency).

All the presented values were obtained on the *graphene* cluster. The "ping" measurements are used to instantiate the values of o_s, O_S, o_r, and O_r for small to detached messages which are the sender overhead, the overhead per byte at the sender side, the receiver overhead and the overhead per byte at the receiver side respectively. By subtracting $2(o_r + k.O_r)$ from the round trip time measured by the "ping-pong" experiment, and thanks to a piece-wise linear regression, we can deduce the values of latency and bandwidth correction factors.

The last phenomenon to reflect in our description to become completely realistic is the impact of network contention on a point-to-point communication between two processors in a same cabinet and between two processors in two different cabinets. To quantify this impact, we artificially create contention and measure the bandwidth as perceived by the sender and the receiver. We place ourselves in the large message mode where the highest bandwidth usage is observed and transfer 4 MiB messages.

In a first experiment we increase the number of concurrent pings from 1 to 19, i.e., half the size of the cabinet. As the network switch is well dimensioned, this experiment fails to create contention: We observe no bandwidth degradation on either the sender side or the receiver side. Our second experiment uses concurrent `MPI_Sendrecv` transfers instead of pings. We increase the number of concurrent transfers from 1 to 19 and measure the bandwidth on the sender (B_s) and receiver (B_r) side. A multi-port model, as assumed by delay models, would estimate that $B_s + B_r = 2 \times B$ since communications do not interfere with each other. However, both fail to model what actually happens, as we observe that $B_s + B_r \approx 1.5 \times B$ on this cluster.

We model this bandwidth sharing effect by enriching the simulated cluster description, as seen in Figure 4.9. Each processor is provided with three links: an uplink and a downlink, so that send and receive operations share the available bandwidth separately in each direction; and a specific link, whose bandwidth is $1.5 \times B$, shared by all the flows to and from this processor. The consequence is that there is no more backbone (and thus no more `bb_lat` and `bb_bw` attributes in the `<cluster>` tag. Experiments shown it was more realistic to connect the nodes directly to the router of the cluster and model communications as follows. The `sharing_policy` of the links that connect the nodes to the router is declared to `FULLDUPLEX`. This means that two links (named `UP` and `DOWN`) are created during the parsing of the XML file to model pertubation coming from ACK returning packets [110]. Then the `limiter_link` attribute creates a pseudo-shared link to limit the bandwidth that can achieved with the fullduplex links. This link has no latency and a bandwidth of 1.875E8 B/s (that is $1.5 \times 1.25E8B/s$) and it will create contention when messages are sent from both directions with a new peak bandwidth value. When the messages are sent only to one direction the initial bandwidth value is respected. Finally, the `loopback_lat` and `loopback_bw` atributes respectively declare the values of the latency and the bandwidth of the link used when a process sends a message to another process mapped on the same node.

Preliminary experiments on other clusters show that this contention parameter seems constant for a given platform, with a value somewhere between 1 and 2. Determining this parameter requires benchmarking each cluster as described in this section. The set of experiments is available on this web page [111].

This modification is not enough to model contention at the level of the whole *graphene*

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">

  <config id="General">
    <prop id="workstation/model" value="compound"/>
    <prop id="network/model" value="SMPI"/>

    <prop id="smpi/async_small_thres" value="65536"/>
    <prop id="smpi/send_is_detached_thres" value="327680"/>

    <prop id="smpi/os"
          value="0:8.93009e-06:7.654382e-10; 1420:1.396843e-05:2.974094e-10;
                 32768:1.540828e-05:2.441040e-10; 65536:0.000238:0; 327680:0:0"/>

    <prop id= "smpi/or"
          value="0:8.140255e-06:8.395881e-10; 1420:1.269952e-05:9.092182e-10;
                 32768:3.095706e-05:6.956453e-10; 65536:0:0; 327680:0:0"/>

    <prop id= "smpi/bw_factor"
          value="0:0.400977; 1420:0.913556; 32768:1.078319; 65536:0.956084; 327680:0.929868"/>

    <prop id= "smpi/lat_factor"
          value="0:1.35489; 1420:3.437250; 32768:5.721647;65536:11.988532; 327680:9.650420"/>
  </config>

  <AS id="AS_graphene" routing="Full" >
    <cluster id="AS_sgraphene1" prefix="graphene-" suffix=".nancy.grid5000.fr"
             radical="0-38" power="3.68E9" bw="1.25E8" lat="15E-6"
             sharing_policy="FULLDUPLEX" limiter_link="1.875E8"
             loopback_lat="1.5E-9" loopback_bw="6000000000"></cluster>

    <cluster id="AS_sgraphene2" prefix="graphene-" suffix=".nancy.grid5000.fr"
             radical="39-73" power="3.68E9" bw="1.25E8" lat="15E-6"
             sharing_policy="FULLDUPLEX" limiter_link="1.875E8"
             loopback_lat="1.5E-9" loopback_bw="6000000000"></cluster>

    <cluster id="AS_sgraphene3" prefix="graphene-" suffix=".nancy.grid5000.fr"
             radical="74-103" power="3.68E9" bw="1.25E8" lat="15E-6"
             sharing_policy="FULLDUPLEX" limiter_link="1.875E8"
               loopback_lat="1.5E-9" loopback_bw="6000000000"></cluster>

    <cluster id="AS_sgraphene4" prefix="graphene-" suffix=".nancy.grid5000.fr"
             radical="104-143" power="3.68E9" bw="1.25E8" lat="15E-6"
             sharing_policy="FULLDUPLEX" limiter_link="1.875E8"
               loopback_lat="1.5E-9" loopback_bw="6000000000"></cluster>


  <link id="switch-backbone1" bandwidth="1162500000" latency="1.5E-6"
       sharing_policy="FULLDUPLEX"/>
  <link id="switch-backbone2" bandwidth="1162500000" latency="1.5E-6"
       sharing_policy="FULLDUPLEX"/>

  <link id="explicit-limiter1" bandwidth="1511250000" latency="0"
       sharing_policy="SHARED"/>
  <link id="explicit-limiter2" bandwidth="1511250000" latency="0"
       sharing_policy="SHARED"/>


    <ASroute src="AS_sgraphene1" dst="AS_sgraphene2"
         gw_src="graphene-AS_sgraphene1_router.nancy.grid5000.fr"
         gw_dst="graphene-AS_sgraphene2_router.nancy.grid5000.fr"
         symmetrical="NO"
             <link_ctn id="switch-backbone1" direction="UP"/>
             <link_ctn id="explicit-limiter1"/> <link_ctnid="explicit-limiter2"/>
             <link_ctn id="switch-backbone2" direction="DOWN"/>

     </ASroute>
     <ASroute src="AS_sgraphene2" dst="AS_sgraphene1"
         gw_src="graphene-AS_sgraphene2_router.nancy.grid5000.fr"
         gw_dst="graphene-AS_sgraphene1_router.nancy.grid5000.fr"
         symmetrical="NO"
             <link_ctn id="switch-backbone2" direction="UP"/>
             <link_ctn id="explicit-limiter2"/> <link_ctnid="explicit-limiter1"/>
             <link_ctn id="switch-backbone1" direction="DOWN"/>
    </ASroute>

 </AS>
</platform>
```

Figure 4.9: Realistic description of the graphene cluster.

cluster. As said eralier, this cluster is made of four cabinets interconnected through 10Gb links. Experiments show that these links become limiting when shared between several concurrent pair-wise communications between cabinets. This effect is captured by describing the interconnection of two cabinets as three distinct links (uplink, downlink, and limiting link). In Figure 4.9, this correspond to lines 49 to 76. The links named `explicit-limiter[1-2]` induce the same behavior as the `limiter_link` attribute of the `<cluster>` tag. The bandwidth of the limiting links is set to 13 Gb/s as measured. But experiments shown that application are never able to use more than 93% of the nominal bandwidth across cabinets. Then we applied this correction directly in the XML file (lines 49, 51, 54, and 56). Finally note that the routes between two cabinets are not symmetrical anymore (see lines 63 and 72) to take the uplink, downlink, and limiting links into account.

If we refer to Figure 4.1, we now have all the necessary inputs to replay a *time-independent* trace: a realistic and carefully instantiated description of the target cluster (Figure 4.9), a deployment file (Figure 4.4, and *time-independent* trace acquired according to the framework described in Chapter 3. In the next Section, we introduce the simulation backends used in the different implementations of our *time-independent* replay tool.

4.2 Simulated Trace Replay

Our simulation framework is tightly connected to the SimGrid project [70] that provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments. SimGrid relies on a scalable and extensible simulation engine and offers several user APIs. The MSG API allows users to describe an application as a set of concurrent processes and provides a simple mailbox-based communication protocol. While initially aimed at studying scheduling algorithms, the MSG API proved to be perfectly usable in other contexts and became the most widely used API of SimGrid. Then, to replay time-independent traces, we first wrote a simulator, on top of the simulation kernel using the MSG API.

This simulator had to include a function that corresponds to the expected behavior of a given action. This has to be done for each action that may appear in a trace. We present hereafter the implementation of some typical and representative actions.

Figure 4.10 shows the necessary code for the `compute` action. The format of this action is `<id> compute <volume of instructions>`, and is passed to this function as an array of strings (`action`), one for each field of the entry. Once the amount of instructions to compute has been extracted (line 2), it is possible to create the corresponding SimGrid task (line 3). The task is executed (line 6) and destroyed (line 7) as soon as the computation of `amount` instructions has been simulated. The duration of the action is logged on line 8.

```
1  static void action_compute(const char *const *action) {
2    const char *amount = action[2];
3    msg_task_t task = MSG_task_create("task", parse_double(amount), 0, NULL);
4    double clock = MSG_get_clock();
5
6    MSG_task_execute(task);
7    MSG_task_destroy(task);
8    log_action(action, MSG_get_clock() - clock);
9  }
```

Figure 4.10: MSG code for the *compute* action.

Figure 4.11 shows the implementation of the `send` action whose format is `<id> send <dst_id> <volume>`. Line 3 retrieves the message `size` while line 6 creates the name of the *mailbox* in which the message will be sent. This name combines the rank of the sender and that of the receiver. Line 8 creates a MSG communication task and sends it in the mailbox. Finally line 11 calls a function that check is some asynchronous communications are completed and can be cleaned.

```
1   static void action_send(const char *const *action) {
2     char to[250];
3     double size = parse_double(action[3]);
4     double clock = MSG_get_clock();
5
6     sprintf(to, "%s_%s", MSG_process_get_name(MSG_process_self()), action[2]);
7
8     MSG_task_send(MSG_task_create(to, 0, size, NULL), to);
9
10    log_action(action, MSG_get_clock() - clock);
11    asynchronous_cleanup();
12  }
```

Figure 4.11: MSG code for the *send* action.

Figure 4.12 shows the MSG commands for the `Isend` action whose format is very similar to that of its synchronous equivalent `<id> Isend <dst_id> <volume>`. For sake of simplicity, we only comment the additional code needed to handle asynchronous communications. Each simulated process has access to a shared data structure, called `globals`, that comprises different counters and two dynamic arrays, `isends` and `irecvs` that respectively contain the pending asynchronous sends and receives. The `MSG_task_isend` function returns a handler (lines 9-10) that is pushed at the ends of the `isends` dynamic array (line 11).

```
1   static void action_Isend(const char *const *action) {
2     char to[250];
3     souble size = parse_double(action[3]);
4     double clock = MSG_get_clock();
5     process_globals_t globals =
6         (process_globals_t) MSG_process_get_data(MSG_process_self());
7
8     sprintf(to, "%s_%s", MSG_process_get_name(MSG_process_self()), action[2]);
9     msg_comm_t comm =
10        MSG_task_isend(MSG_task_create(to, 0, size, NULL), to);
11    xbt_dynar_push(globals->isends, &comm);
12
13    log_action(action, MSG_get_clock() - clock);
14    asynchronous_cleanup();
15  }
```

Figure 4.12: MSG code for the *Isend* action.

The `Irecv` action (`<id> Irecv <src_id> <volume>`) depicted in Figure 4.13 is a bit more complex. The basic principle of this action is to retrieve the MSG communication task sent by the sender. To do this, we build the name of the mailbox for this communication from the ranks of the sender and receiver (line 7). Then we use another dynamic array of the global data structure to store the MSG task. First we add an empty task at the end of the dynamic array (line 10), and then we use give a pointer on it as parameter of the `MSG_task_irecv` function (lines 11-14). Finally, line 15 stores the handle on the asynchronous communication in the `irecvs` dynamic array.

```
1   static void action_Irecv(const char *const *action) {
2     char mailbox[250];
3     double clock = MSG_get_clock();
4     process_globals_t globals =
5         (process_globals_t) MSG_process_get_data(MSG_process_self());
6
7     sprintf(mailbox, "%s_%s", action[2],
8             MSG_process_get_name(MSG_process_self()));
9     msg_task_t t = NULL;
10    xbt_dynar_push(globals->tasks, &t);
11    msg_comm_t c =
12        MSG_task_irecv(xbt_dynar_get_ptr
13                       (globals->tasks, xbt_dynar_length(globals->tasks) - 1),
14                       mailbox);
15    xbt_dynar_push(globals->irecvs, &c);
16
17    log_action(action, MSG_get_clock() - clock);
18    asynchronous_cleanup();
19  }
```

Figure 4.13: MSG code for the *Irecv* action.

In time-independent traces, as in MPI applications, `Irecv` actions are usually followed by `wait` actions that act as synchronization barriers for asynchronous communications. Figure 4.14 presents the code for the `wait` action (`<id> wait`). After having checked that this `wait` was precedeed by an `Irecv` (lines 8-10), this function gets the last element from the `globals->irecv` dynamic array (line 12) and waits for its completion (line 13). Lines 14-16 clean up things, one the asynchronous communication has completed and released the `MSG_comm_wait` function.

```
1   static void action_wait(const char *const *action) {
2     msg_task_t task = NULL;
3     msg_comm_t comm;
4     double clock = MSG_get_clock();
5     process_globals_t globals =
6         (process_globals_t) MSG_process_get_data(MSG_process_self());
7
8     xbt_assert(xbt_dynar_length(globals->irecvs),
9                "action wait not preceded by any irecv: %s",
10               xbt_str_join_array(action, " "));
11
12    comm = xbt_dynar_pop_as(globals->irecvs, msg_comm_t);
13    MSG_comm_wait(comm, -1);
14    task = xbt_dynar_pop_as(globals->tasks, msg_task_t);
15    MSG_comm_destroy(comm);
16    MSG_task_destroy(task);
17
18    log_action(action, MSG_get_clock() - clock);
19  }
```

Figure 4.14: MSG code for the *wait* action.

We complete this presentation of the implementation of some actions using the MSG API with a collective communication operation, namely the `bcast` action (`<id> bcast <volume>`).

Such functions are more complex, as shown in Figure 4.15. Moreover, the MSG API forces us to make some unrealistic assumptions. For instance, a flat tree rooted in process of rank 0 is used to implement the broadcast operation, while actual MPI implementations adapt the diffusion algorithm to the message size and number of participating processes, and can be initiated by any process. This implementation first checks that the size of the MPI communicator has been declared (thanks to a `comm_size` action in the trace) on lines 13-14. Then each process builds a specific mailbox name to receive data from the root process (lines 18 and 26 or 36). The behavior is different for the process of rank 0 that sends asynchronously a message to every other processes and then waits for their reception (lines 20-34). The other processes just synchronously receive the sent message (line 37).

```
1  static void action_bcast(const char *const *action){
2    int i;
3    char *bcast_identifier;
4    char mailbox[80];
5    double size = parse_double(action[2]);
6    msg_task_t task = NULL;
7    const char *process_name;
8    double clock = MSG_get_clock();
9
10   process_globals_t counters =
11       (process_globals_t) MSG_process_get_data(MSG_process_self());
12
13   xbt_assert(communicator_size, "Size of Communicator is not defined, "
14              "can't use collective operations");
15
16   process_name = MSG_process_get_name(MSG_process_self());
17
18   bcast_identifier = bprintf("bcast_%d", counters->bcast_counter++);
19
20   if (!strcmp(process_name, "0")) {
21     XBT_DEBUG("%s: %s is the Root", bcast_identifier, process_name);
22
23     msg_comm_t *comms = xbt_new0(msg_comm_t, communicator_size - 1);
24
25     for (i = 1; i < communicator_size; i++) {
26       sprintf(mailbox, "%s_0_%d", bcast_identifier, i);
27       comms[i - 1] =
28           MSG_task_isend(MSG_task_create(mailbox, 0, size, NULL),
29                          mailbox);
30     }
31     MSG_comm_waitall(comms, communicator_size - 1, -1);
32     for (i = 1; i < communicator_size; i++)
33       MSG_comm_destroy(comms[i - 1]);
34     xbt_free(comms);
35   } else {
36     sprintf(mailbox, "%s_0_%s", bcast_identifier, process_name);
37     MSG_task_receive(&task, mailbox);
38     MSG_task_destroy(task);
39   }
40
41   log_action(action, MSG_get_clock() - clock);
42   xbt_free(bcast_identifier);
43 }
```

Figure 4.15: MSG code for the *bcast* action.

Once all the actions are implemented, the trace replay tool has to *register* all of them thanks to the `MSG_action_register` function. These calls made in the `main` function of the simulator link the action keywords (as defined in Table 3.1) to the functions presented above. For instance the `main` function of the simulator includes the following calls:

```
  xbt_replay_action_register("compute", action_compute);
  xbt_replay_action_register("send", action_send);
  xbt_replay_action_register("Isend", action_Isend);
  xbt_replay_action_register("Irecv", action_Isend);
  xbt_replay_action_register("wait", action_wait);
  xbt_replay_action_register("bcast", action_bcast);
```

The last step consists in calling function `MSG_action_trace_run` that takes either a trace file name or `NULL` as input. When no file name is given, it means that there is one trace file per process, whose name is given in the platform file, as shown below.

```
<process host="graphene-1.nancy.grid5000.fr" function="1"/>
  <argument value="SG_process1.trace"/>
</process>
```

This first implementation of the trace replay tool based on the MSG API forced us to mimic the behavior of MPI primitives, *e.g.,* collectives operations or protocols depending on the message size, with crude simplifications. Moreover it decouples this effort on the off-line simulation of MPI applications to that on on-line simulation also taking place within the SimGrid project with SMPI. Consequently we completely rewritten our tool between the publications of [101] and [112] to change the simulation backend and rely on SMPI instead of MSG.

As explained in Chapter 2, SMPI is another user API of SimGrid dedicated to the on-line simulation of MPI applications [66]. It is, as the present off-line approach, built on top of the simulation kernel of the SimGrid toolkit and thus also benefits of its fast, scalable, and validated network models. A salient property of SMPI is its capacity to simulate MPI applications on a single node. SMPI implements about 80% of the MPI 2.0 standard, including most of the network communication related functions. To leverage the accurate network models underlying SMPI, remove unwanted simplifactions, and factor the efforts on the simulation of MPI applications, we have reimplemented the trace replay mechanism directly within SMPI. This rewriting leads to many beneficial changes. First, it simplifies the way *actions*, *i.e.,* events stored in the trace are passed to the simulation kernel.

To illustrate this, we show the implementations of some actions that correspond to specific MPI calls. Figure 4.16 presents the new version of the compute action. Lines 2 and 4 log the duration of this action. This action is actually a simple wrapper on the `smpi_execute_flops` function that belongs to the internal API of SMPI. It simulates the execution of a certain amount of flops (the default unit in SimGrid), or any other work unit provided that the processing power of processors is declared in a coherent unit.

```
1 static void action_compute(const char *const *action){
2   double clock = smpi_process_simulated_elapsed();
3   smpi_execute_flops(parse_double(action[2]));
4   log_timed_action (action, clock);
5 }
```

Figure 4.16: SMPI version of the *compute* action.

The `send` action shown in Figure 4.17 has a slightly different format compared to the MSG version: `<id> send <dst_id> <volume> [<datatype_id>]`. Indeed, instrumentation tools such as TAU, Score-P, and Scalasca express the message size in bytes which is the default data type used by our replay tool. However, the MinI can also log the data type of the message buffer. Then we added a correspondance table that maps each MPI data type to an integer. If this integer is given in the last field of the action then the correct data type is used (lines 6-9). This is done whenever the data type can be extracted. Again, the new version is just a wrapper on the appropriate SMPI internal function (line 11).

```
1  static void action_send(const char *const *action){
2    int to = atoi(action[2]);
3    double size=parse_double(action[3]);
4    double clock = smpi_process_simulated_elapsed();
5
6    if (action[4])
7       MPI_CURRENT_TYPE = decode_datatype(action[4]);
8    else
9       MPI_CURRENT_TYPE = MPI_DEFAULT_TYPE;
10
11   smpi_mpi_send(NULL, size, MPI_CURRENT_TYPE, to , 0, MPI_COMM_WORLD);
12   log_timed_action (action, clock);
13 }
```

Figure 4.17: SMPI version of the *send* action.

The difference in the `Isend` action (`<id> Isend <dst_id> <volume> [<datatype_id>]`) shown in Figure 4.18 is that we store the `MPI_Request` built by the SMPI function in a global array of arrays (line 14). Each position corresponds to a process and contains all the pending requests for this process.

```
1  static void action_Isend(const char *const *action){
2    int to = atoi(action[2]);
3    double size=parse_double(action[3]);
4    double clock = smpi_process_simulated_elapsed();
5    MPI_Request request;
6
7    if (action[4])
8       MPI_CURRENT_TYPE = decode_datatype(action[4]);
9    else
10      MPI_CURRENT_TYPE = MPI_DEFAULT_TYPE;
11
12   request = smpi_mpi_isend(NULL, size, MPI_CURRENT_TYPE, to, 0,MPI_COMM_WORLD);
13
14   xbt_dynar_push(reqq[smpi_comm_rank(MPI_COMM_WORLD)],&request);
15
16   log_timed_action (action, clock);
17 }
```

Figure 4.18: SMPI version of the *Isend* action.

The implementation of the `Irecv` action presented in Figure 4.19 (`<id> Irecv <src_id> <volume> [<datatype_id>]` is very similar. It just calls the `smpi_mpi_irecv` function (line 12) that builds a `MPI_Request` which is stored at the right position, *i.e.,* given by the receiver's rank, in the global array (line 14). This request will be handled by the `wait` action.

```
1   static void action_Irecv(const char *const *action){
2     int from = atoi(action[2]);
3     double size=parse_double(action[3]);
4     double clock = smpi_process_simulated_elapsed();
5     MPI_Request request;
6
7     if(action[4])
8        MPI_CURRENT_TYPE = decode_datatype(action[4]);
9     else
10       MPI_CURRENT_TYPE = MPI_DEFAULT_TYPE;
11
12    request = smpi_mpi_irecv(NULL, size, MPI_CURRENT_TYPE, from, 0,  MPI_COMM_WORLD);
13
14    xbt_dynar_push(reqq[smpi_comm_rank(MPI_COMM_WORLD)],&request);
15
16    log_timed_action (action, clock);
17  }
```

Figure 4.19: SMPI version of the *Irecv* action.

As mentioned earlier, the `wait` action retrieves the last posted `irecv` request (line 9) and then waits for its completion (line 11). A check is done to ensure that there were an asynchronous receive to wait for (lines 6-7).

```
1   static void action_wait(const char *const *action){
2     double clock = smpi_process_simulated_elapsed();
3     MPI_Request request;
4     MPI_Status status;
5
6     xbt_assert(xbt_dynar_length(reqq[smpi_comm_rank(MPI_COMM_WORLD)]),
7         "action wait not preceded by any irecv: %s", xbt_str_join_array(action," "));
8
9     request = xbt_dynar_pop_as(reqq[smpi_comm_rank(MPI_COMM_WORLD)], MPI_Request);
10
11    smpi_mpi_wait(&request, &status);
12
13    log_timed_action (action, clock);
14  }
```

Figure 4.20: SMPI version of the *wait* action.

We end this presentation of the SMPI versions of actions with the `bcast` action whose format is `<id> bcast <volume> [<root>] [<datatype_id>]`. This action has two optional parameters to explicitely declare which process initiates the broadcast and the type of data to communicate. This removes one of the unrealistic assumption made in the MSG version, *i.e.,* having the collective operations always rooted on process of rank 0. The other issue related to the underlying algorithm is directly solved by SMPI. Indeed, calling the `smpi_mpi_bcast` function allows the trace replay tool to benefit of the adaptive selection of algorithms for collective operation implemented by SMPI. Depending on the message size, the number of processes, and the MPI implementation (OpenMPI, MPICH, or StarMPI), the best algorithm will be automatically selected in a seamless way.

```
static void action_bcast(const char *const *action){
  double size = parse_double(action[2]);
  double clock = smpi_process_simulated_elapsed();
  int root=0;
  MPI_CURRENT_TYPE= MPI_DEFAULT_TYPE;

  if(action[3]) {
    root= atoi(action[3]);
    if(action[4])
      MPI_CURRENT_TYPE = decode_datatype(action[4]);
  }

  smpi_mpi_bcast(NULL, size, MPI_CURRENT_TYPE, root, MPI_COMM_WORLD);

  log_timed_action (action, clock);
}
```

Figure 4.21: SMPI version of the *bcast* action.

This presentation showed that SMPI already implements a lot of functionalities in comparison to MSG. It is a better choice to use SMPI as it prevents the development of simplistic equivalents of MPI routines. It is also less error prone and difficult than relying on a programming interface that has not been designed to simulate parallel MPI applications. Using MSG means to rethink complex mechanisms such as buffering or RDMA in terms of processors putting tasks in mailboxes. On the contrary, the new implementation allows us to benefit of all the complex optimizations already offered and validated by SMPI in a seamless way.

The second main change implied by this rewriting is the user view of the replay framework. In the former version everything was exposed to the user. Everything is now embedded and replaying a Time-Independent Trace with SimGrid simply amounts to run the following program.

```
int main(int argc, char *argv[]){

 smpi_replay_init(&argc, &argv);
 smpi_action_trace_run(NULL);
 smpi_replay_finalize();

 return 0;
}
```

This code, that initializes some data structures, loads a trace, and destroys the data structures, is considered as a regular SMPI program. Then it is compiled with the `smpicc` wrapper and launched thanks to the `smpirun` command (see [66] for details), as follow

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

where `np` and `hostfile` are classical parameters of `mpirun`. Our replay tool needs a single extra parameter, *i.e.*, a file that lists the names of the trace files to associate to each process. If this file contains a single entry, all the processes will look for the actions they have to perform into the same trace. Otherwise, each process parses actions for its own trace file.

In the next section, we evaluate the accuracy of the simulation results achieved by the trace replay tool we just detailed. We present results obtained with the first prototype written with MSG as well as with the new SMPI version. Then we show how the accuracy was greatly improved thanks to a better calibration procedure and an adapted simulation backend.

4.3 Accuracy of Simulation Results

Our first prototype framework [101] was implemented with MSG API and *time-independent* traces were acquired with the selective instrumentation method described in 3.2.1. Moreover the calibration of the processing rate was not aware of the processor's cache. For the computation calibration of our prototype framework we used the already mentioned approach in Section 4.1 without the cache-aware method. The instantiation of the network parameters of the platform file was done in two steps. To set the bandwidth, we used the nominal value of the links, *e.g.,* 1 Gib for GigaEthernet links. For the latency of a communication link, we relied on the `Pingpong_Send_Recv` experiment of the SKaMPI [27] benchmark suite. Only two nodes of the target platform were then needed here to calibrate the network. We took the value obtained for a 1-byte message and divided it by six. This factor of six comes from two sources. We have to divide the ping-pong time by two to obtain the latency of a one-way message. Then we divide it by three to take the topology of a the network interconnect into account. Indeed, two nodes in a compute cluster are generally connected through two links and one switch. In case of hierarchical network, we account for this hierarchy in the determination of the latency. Finally we were studying AMD processors then we conducted our experiments on the *bordereau* cluster.

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

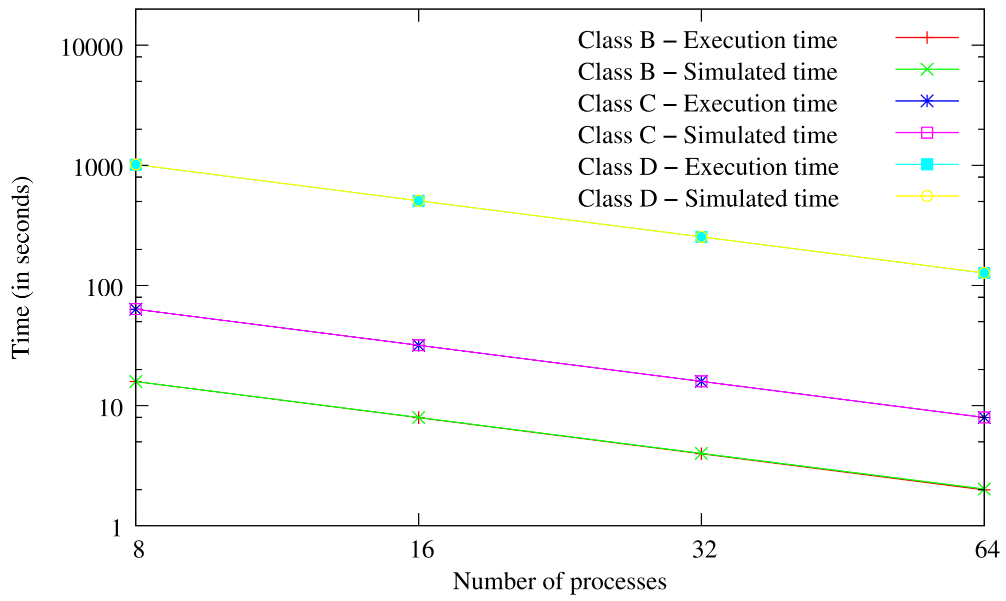


Figure 4.22: Comparison of simulated and actual execution time for the EP benchmark on the *bordereau* cluster.

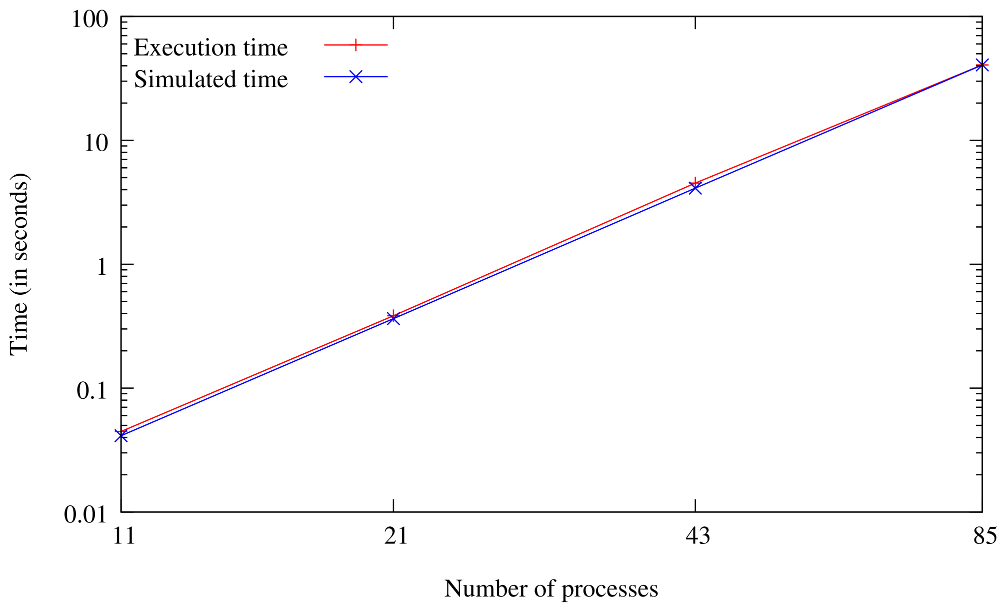


Figure 4.23: Comparison of simulated and actual execution time for the DT benchmark on the *bordereau* cluster.

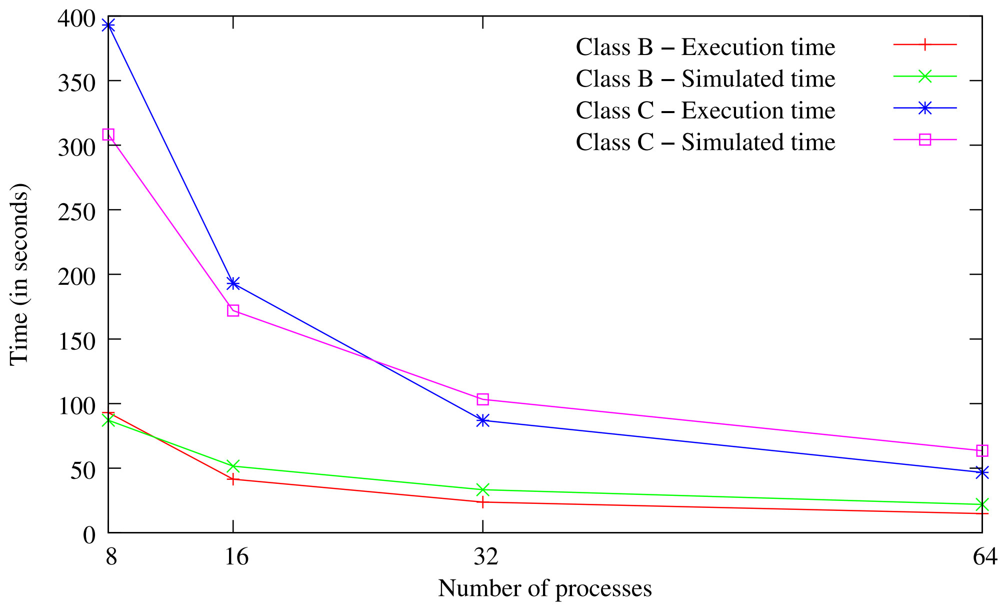


Figure 4.24: Comparison of simulated and actual execution time for the LU benchmark on the *bordereau* cluster.

Figures 4.22, 4.23, and 4.24 show the accuracy of the time-independent trace replay by comparing the actual execution time respectively of the EP, DT and LU benchmarks on the *bordereau* cluster for various classes with the simulated time obtained with SimGrid. EP (Embarrassingly Parallel) consists of only computations, DT (Data Traffic) consists of only communications, while LU (LU Factorization) mixes computation and communication. For DT, the class refers not only to the data size but also the number of communicating processes: classes A, B, and C involve 21, 43, and 85 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 in a single sink. We see that the simulated execution time is very accurate with low relative errors for EP (up to 1.71% for class B on 64 processes) and DT (up to -8.99%) that are dominated either by computation or communication. For LU, which is composed of millions of interleaved computation and communication actions, the obtained results are more ambivalent. Here the trace replay is able to predict the correct evolution trend, but the local relative error may be quite high (up to 47.1% for Class B on 64 processes) and more importantly is not constant. This prevents our tool to provide predictions within a fixed interval of confidence. We can see that for 8 and 16 processes mainly for class C, the absolute difference of the execution and simulated time represents a significant amount of time.

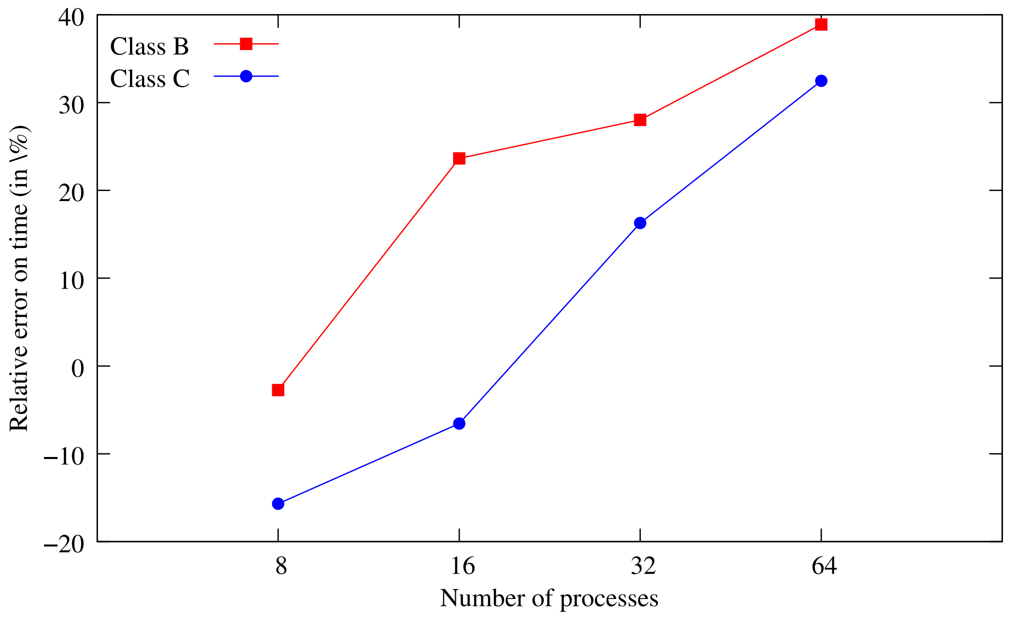


Figure 4.25: Evolution with regard to the number of processes of the relative error between execution and simulated times for the execution of the LU factorization on the *bordereau* cluster.

Figure 4.25 shows the results of Figure 4.24 from a different perspective. It shows the evolution of the relative error between the time to compute a LU factorization and its simulated counterpart when the number of involved processes vary. We see that the inaccuracy of the simulated version is not stable, but increases rather linearly with the number of processes. Moreover, our tool underestimates the execution time for small number of processes, *i.e.,* when each process owns a larger share of data, and then overestimates the execution as the number of processes grows.

This inaccuracy may have different origins related either to computations or communications. For instance it may come from the discrepancy of the measured number of instructions. Indeed this discrepancy directly impacts the calibration of the replay tool that determines the rate at which each machine can process instructions. Among other potential sources, we assume that every part of the application can be processed at the same rate while it may not be the case for some applications, due to cache affinity for instance. However, the general trend shown by Figure 4.25 seems to indicate that the main source of inaccuracy comes from a bad estimation of communication times. To confirm this impression, we made simulations in which each compute action was replaced by a delay that corresponds exactly to the time spent for this compute part during the trace acquisition. The obtained results, in which computations are thus perfectly simulated, show a similar trend. The most probable source might then be that the LU factorization, as implemented in the NAS Parallel Benchmark suite, implies a lot of small messages (less than 64 KiB). Then most of the point-to-point communications use the *eager mode* of MPI in which the sender copies the data directly into the memory of the receiver without having to wait for it to post the corresponding receive. This particular communication protocol was implemented in a too simple way, based on asynchronous communications in the first implementation of our framework. As the number of small messages increases with the number of involved processes, small inaccuracies tend to accumulate and lead to a large overall relative error.

In the second version of our framework [112] we combined the compiler optimization flag, the `TAU-reduced` instrumentation detailed in Section 3.2.2, and the cache-aware calibration method to improve the quality of the simulation of the compute part of the application. For the communication part, we benefit of the quality of the underlying SMPI layer that provides better estimations for the point-to-point communication of small messages and a tuned piece-wise linear network model. Moreover the tag called `async_small_thres` is used as described in Section 4.1.

Figure 4.26 shows the impact of the whole set of modifications on the accuracy of the simulated executions of the LU factorization. These results, obtained on the *bordereau* cluster, have to be compared to those presented by Figure 4.25.

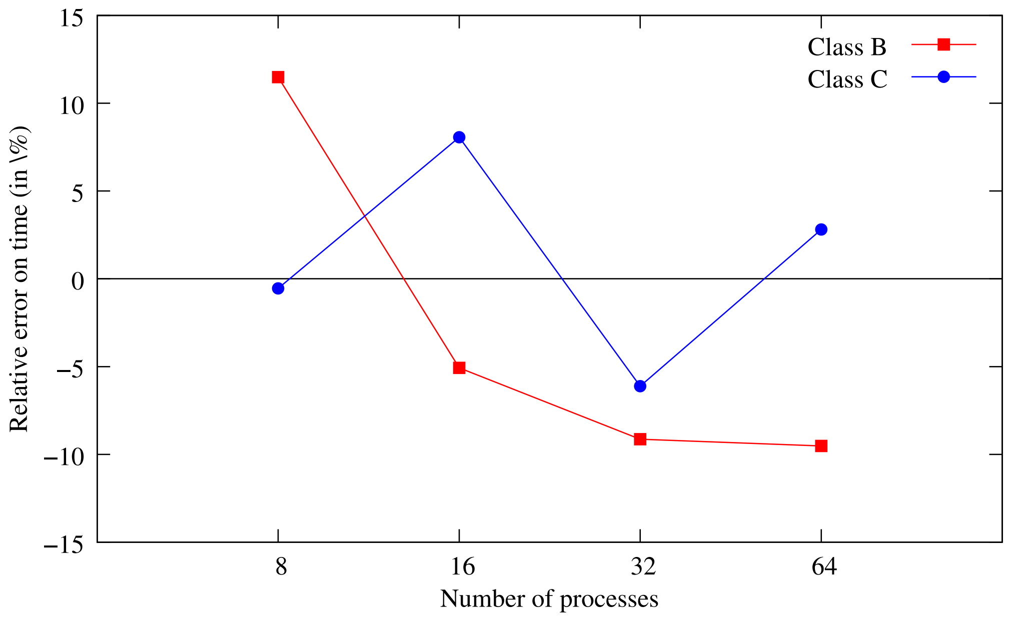


Figure 4.26: Evolution w.r.t. the number of processes of the relative error between execution and simulated times for the execution of the LU factorization with the new replay framework. Results obtained on the *bordereau* cluster.

The first outcome is a drastic reduction of the inaccuracy. With our initial implementation, the relative error linearly increased from -2.7% to 38.9% for class B and from -15.8% to 32.5% for class C. Now, the error varies between -9.5% and 11.5% for class B and between -6.1% and 8.1% for class C. Moreover, the linear increase of the error along with the number of processes, that indicated a bad simulation of the communication part, disappeared. We even note an opposite trend for class B. This can be explained by the fact that this configuration of SMPI does not capture the time to copy data in memory in the `MPI_Send` function. This leads to a slight underestimation of the simulation time that increases as more small messages are exchanged for large numbers of processes.

As the *bordereau* cluster is now aging and prone to failures and suspect behaviors. Then we completed our study with results obtained on the more recent *graphene* cluster. Figure 4.27 shows the relative error achieved under the same experimental conditions as for Figure 4.26. As on the *bordereau* cluster, the error is in a narrow interval ranging from −11.4% and -2%. Results for class B are even more stable. Again the underestimation of the simulated time should be compensated by taking memory copy into account.

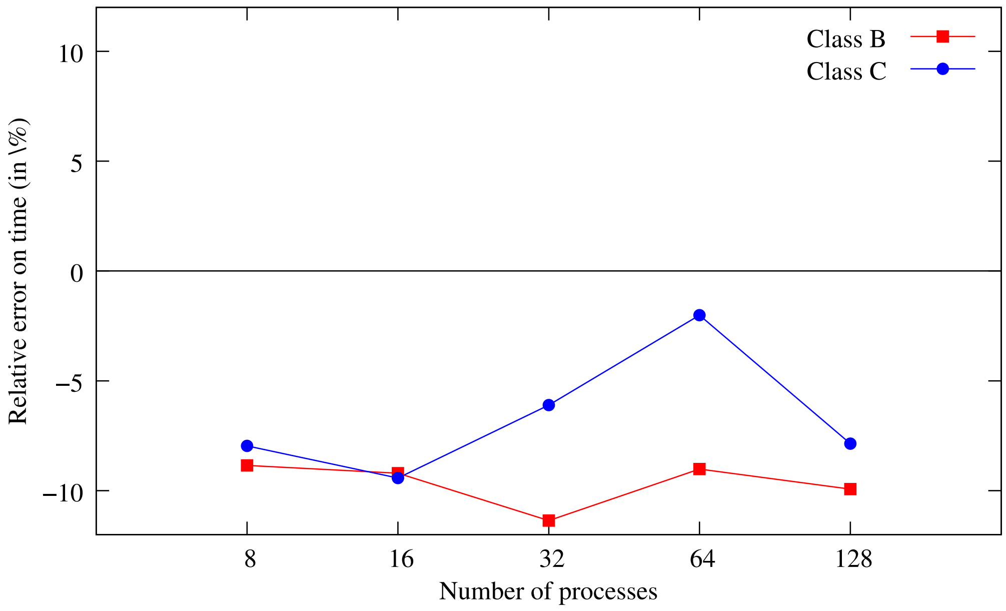


Figure 4.27: Evolution w.r.t. the number of processes of the relative error between execution and simulated times for the execution of the LU factorization with the new replay framework. Results obtained on the *graphene* cluster.

These results are not perfect yet, as some fluctuations or linear trends are still there, but there are a great improvement towards an accurate and trustworthy performance prediction tool. The reduction of around 10% of the overall inaccuracy allowed by the proposed modifications may sound common in the domain of trace-based simulation. But in our case, recall that we deal with time-independent traces that are based only on volumes and totally oblivious to both acquisition and replay platforms. For such traces a good accuracy is then harder to achieve. These results show that the more flexible approach offered by time-independent traces becomes a sound alternative for prediction only thanks to these modifications. Some issues still have to be addressed, but most of them have been identified. Finally the factorization of the code, and thus of the efforts with those made on the SMPI project will help to improve the quality of our Time-Independent Trace replay tool.

For the last version of our framework we used all the features presented in Figure 4.9 related to communications. We took under consideration the cost to copy data in memory and the network congestion that may occur during the execution. Moreover the *time-independent* traces that are now used are acquired by instrumenting the whole benchmarks and not by applying selective instrumentation. Thus the traces include initialization and finalization phases.

Figures 4.28,4.29,4.30 and 4.31 show the actual and simulated execution times for various classes of the EP, DT, LU, and CG benchmarks on the *graphene* cluster. The presented execution times are average values over ten "clean" runs, *i.e.,* the standard deviation is negligible. The *graphene* is a cluster shared by multiple users and accessed through a resource management system. Then some experiments may be impacted by other jobs as network resources are shared.

To obtain these ten undisturbed runs, we usually had to execute three to four extra runs.

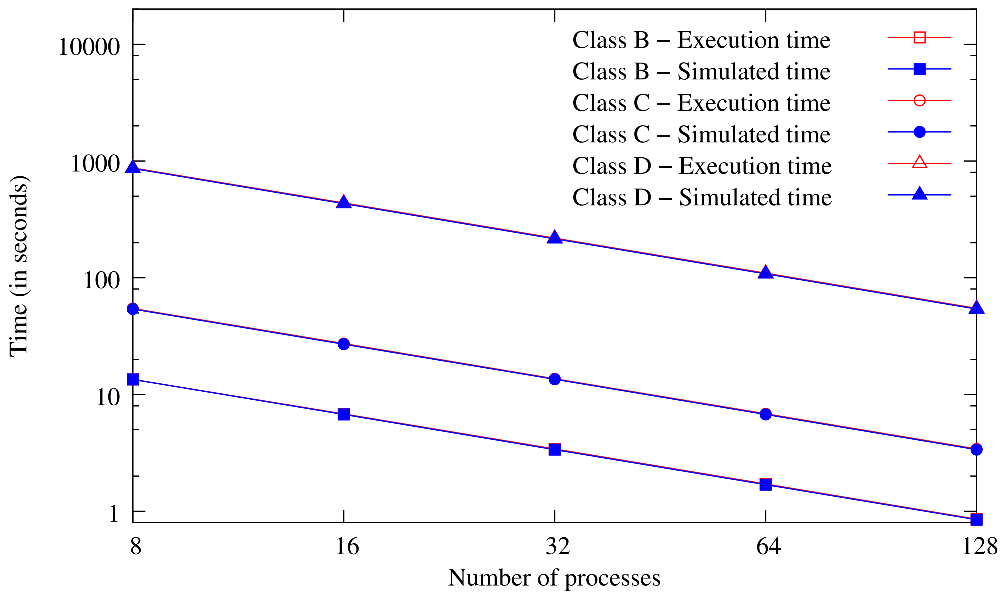


Figure 4.28: Comparison of simulated and actual execution time for the EP benchmark on the *graphene* cluster.

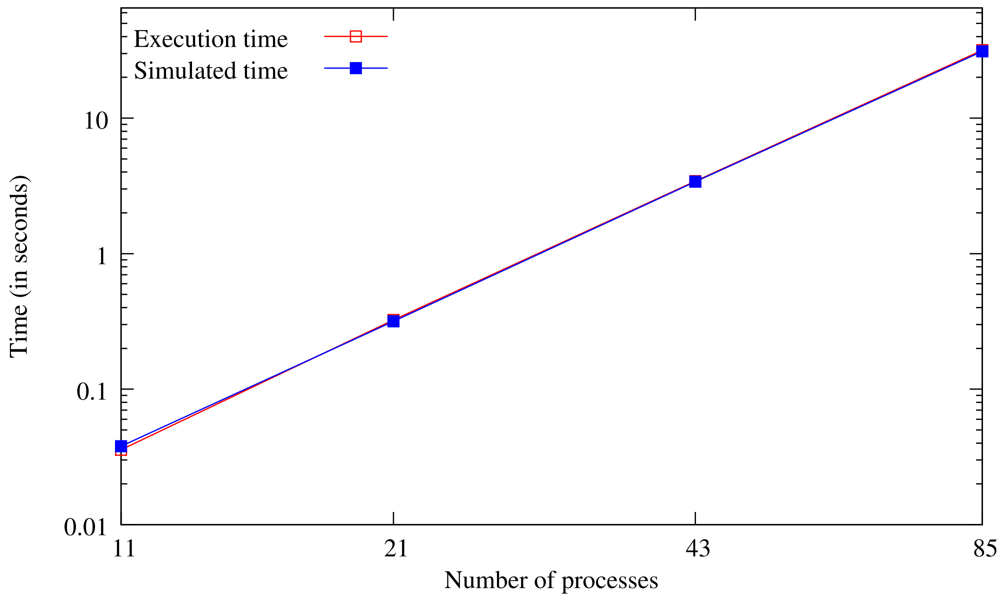


Figure 4.29: Comparison of simulated and actual execution time for the DT benchmark on the *graphene* cluster.

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 2.2 ms only), respectively. As already mentioned, these two benchmarks are dominated either by computation or communication and are then simpler to simulate.

For the LU and CG benchmarks, the simulation is less accurate with maximum absolute

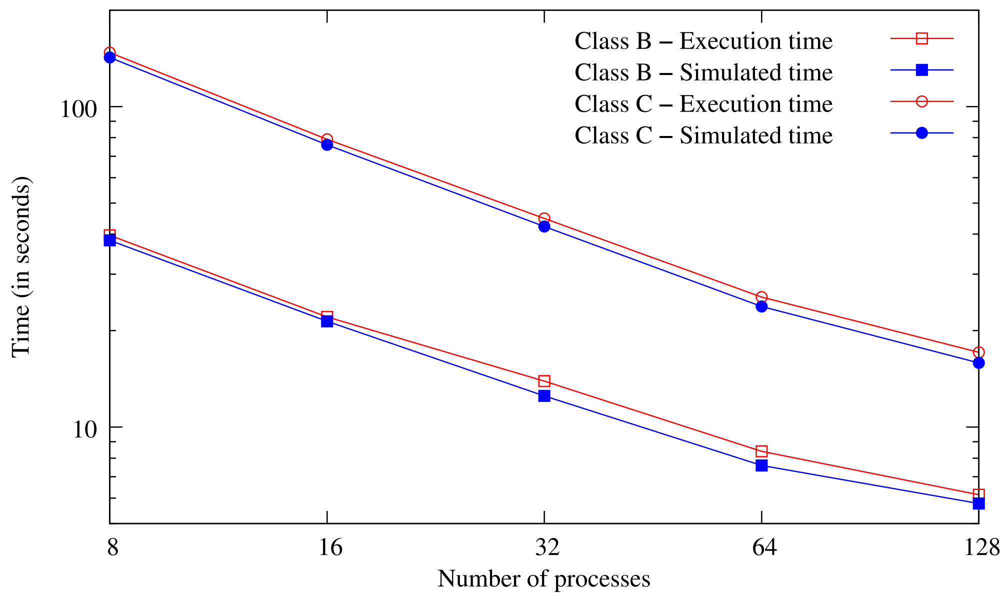


Figure 4.30: Comparison of simulated and actual execution time for the LU benchmark on the *graphene* cluster.

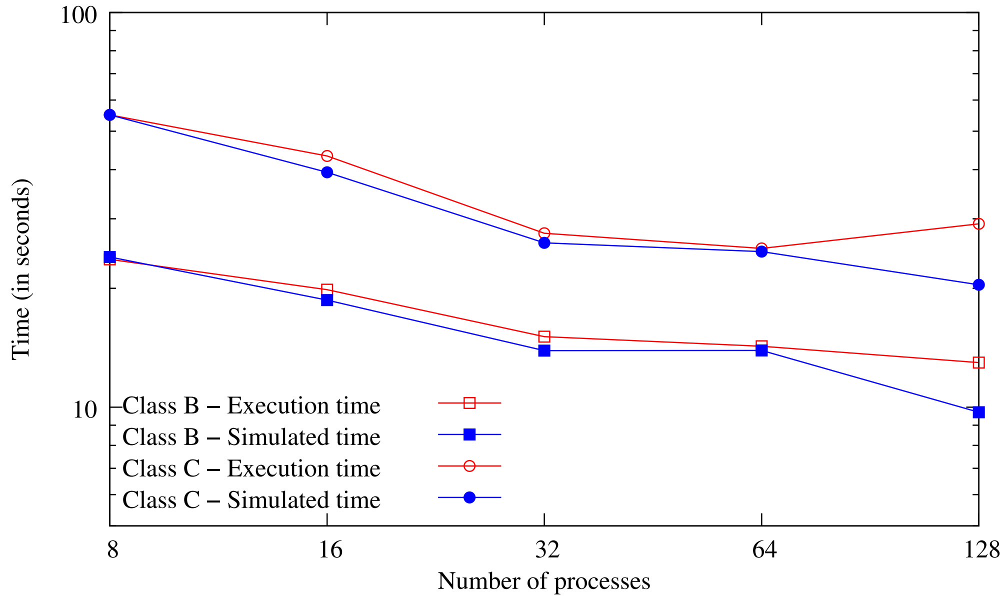


Figure 4.31: Comparison of simulated and actual execution time for the CG benchmark on the *graphene* cluster.

relative error of 10.17% and 29.95% respectively.Figure 4.32 is similar to Figures 4.26 and 4.27 but with a well calibrated network model.

We see another improvement in terms of accuracy with regard to the version and even more stable predictions. However, the simulated execution time is still underestimated. A next step, detailed in the next section is to attempt to identify (and remedy) the sources of the inaccuracies observed for the LU and CG benchmarks.

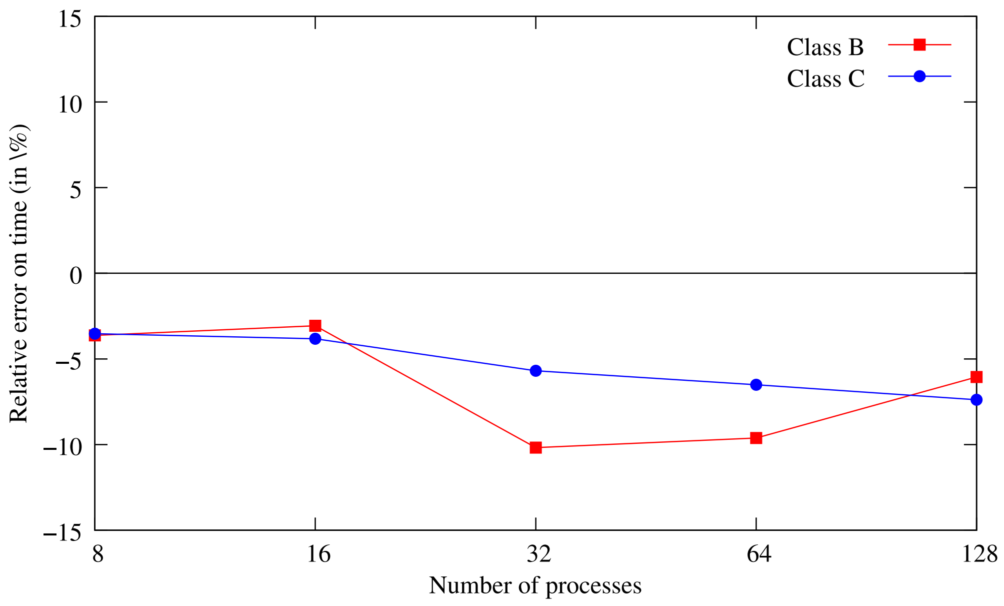


Figure 4.32: Evolution w.r.t. the number of processes of the relative error between execution and simulated times for the execution of the LU benchmark.

4.3.1 Sources of Inaccuracy

To better understand the sources of inaccuracy in the presented simulation results, we further analyze those obtained for the LU benchmark. The LU iterative benchmark alternates computations and communications at a high rate. Similar or close completion times as shown by Figure 4.30 may hide a bad, but lucky, estimation of each component of the execution time. To be sure that estimation errors on computation times are not compensated by errors on the communication times, we propose to compare (a part of) the Gantt charts of the actual and simulated executions. Such a comparison is given by Figure 4.33 that shows a period of 0.2 seconds of the execution of the LU benchmark with 32 processes. The upper part of this figure displays the actual execution while the lower corresponds to the simulation of the same phase with the simulation.

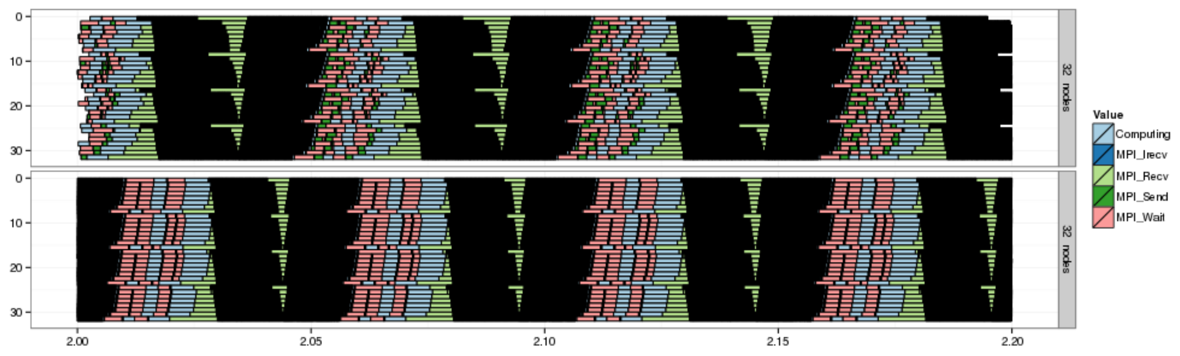


Figure 4.33: Part of the Gantt chart of the execution of the LU benchmark with 32 processes. The upper part displays the actual execution while the lower is the simulation.

Such Gantt charts are difficult to align perfectly due to the size of the initial execution traces and clock skew that requires a synchronization for the real trace. Real executions are also subject to system noise that does not occur in simulation causing some small discrepancies related to the computation time. This then prevents the observation of a perfect match. However, Figure 4.33 clearly shows that despite the small time scale, the simulation correctly renders the general communication pattern of this benchmark.

One approach to identify which aspect of the simulation, computation or communication, is not accurately predicted is to fake a *perfect* simulation of the computation times. To do this, when we acquire the *time-independent* traces in regular mode, we save the duration of each *compute* action in addition of the volume of computed instructions. Then we replace the number of instructions with the *duration* times the *processing rate*. Thus the simulated times of the compute actions correspond exactly to the measured ones. Figure 4.34 presents the results of such a "perfect" simulation.

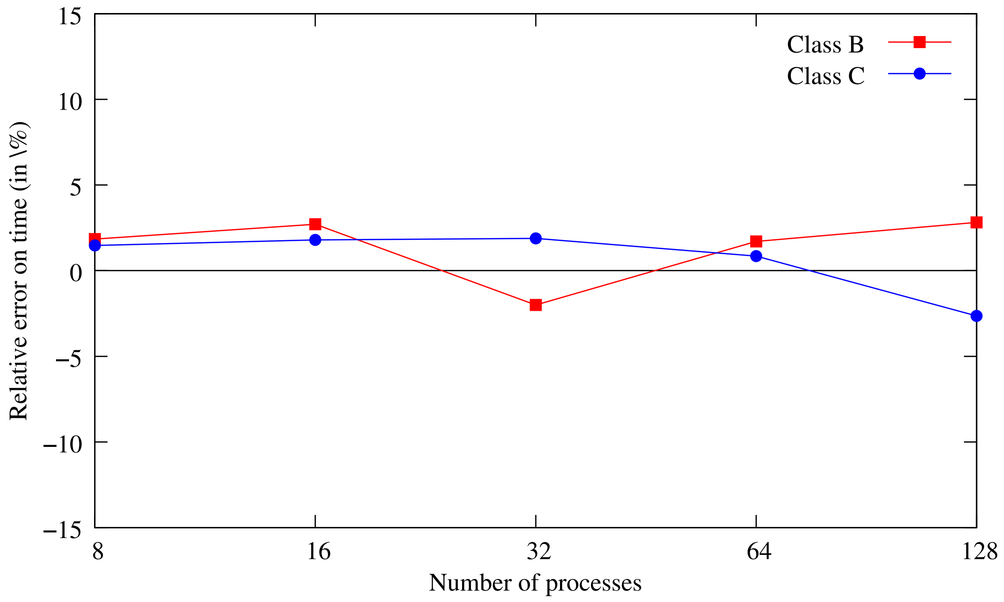


Figure 4.34: Evolution w.r.t. the number of processes of the relative error between execution and perfect simulation for the execution of the LU benchmark.

The maximum absolute relative error is 2.82%. This indicates that the network model predicts the communication with good accuracy while the calibration of the computation part could be improved. This is the cost of using a single processing rate for the whole application while it depends on many factors such as cache misses, stalled cycles on any resource, etc. One approach would be to use a probabilistic model based on the measurements of the instance used for the calibration procedure but we are still looking for an appropriate solution.

The measured times of the *compute* actions include a small instrumentation overhead, then the relative error for the "perfect" simulation can be positive wihtout any issue with the communication model. However, the predictions for class B with 32 processes and class C with 128 processes are less accurate. A further analysis show that these instances handle two specific message sizes. For class B, 32 processes there are message of 204,000 and 212,160 bytes and for class C and 128 processes there are message sizes of 129,600 and 142,560 bytes. Thus we should focus in a future work to study exactly the simulation prediction on a range of message sizes from 129,000 to 215,000 bytes.

Another issue is that calibration for hierarchical networks is a complex procedure. The instances with 64 processes use two cabinets with 8% of the communication occuring between cabinets. With 128 processes it represents 22% of the communications. However, our network calibration mainly performs the ping-pong tests within a single cabinet. Moreover 99% of the exchanged messages are small, *i.e.,* less than 65,536, and fall in the asynchronous category. Small variations in the calibration may then have a great impact of the simulation accuracy. However, the communication predictions remains pretty accurate for most of the instances.

Figure 4.35 shows the relative error between the real execution and the simulated times for the CG benchmark. We see a large error (close to 30%) with 128 processes while it remains under 9% for the other instances. Then we have to determine if the source of this gap comes from a bad estimation by the model or a problem during the actual execution. Our main suspect was the actual execution.

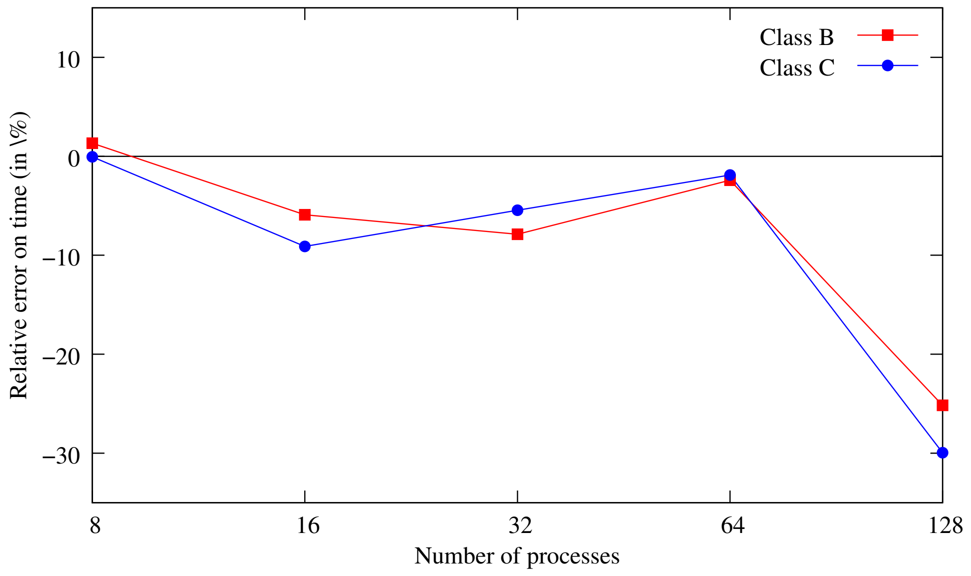


Figure 4.35: Evolution w.r.t. the number of processes of the relative error between execution and simulated times for the execution of the CG benchmark.

First experiments showed a relative error for class C with 128 processes of 56%. Then we observed the communication pattern of the CG benchmark to understand better what was happening. We extracted the communication pattern for a small part of CG benchmark which is given in Figure 4.36. This sample last from time 0.178 to time 0.187. The red links correspond message sizes of 75,000 bytes and the blue ones of messages of 8 bytes. Thanks to the knowlegde of this particular communication pattern we adapted the mapping of processes to minimize communication across cabinets. The new mapping is the presented in Table 4.1. It improved the execution time of the benchmark and also reduced the relative error from 56% to 30%.

Cabinet id	Nodes
0	1-28, 33-34, 65-66, 97-98
1	35-62
2	67-92
3	29-32,93-96,99-128

Table 4.1: Specific process mapping on nodes of the *graphene* cluster for the CG benchmark.

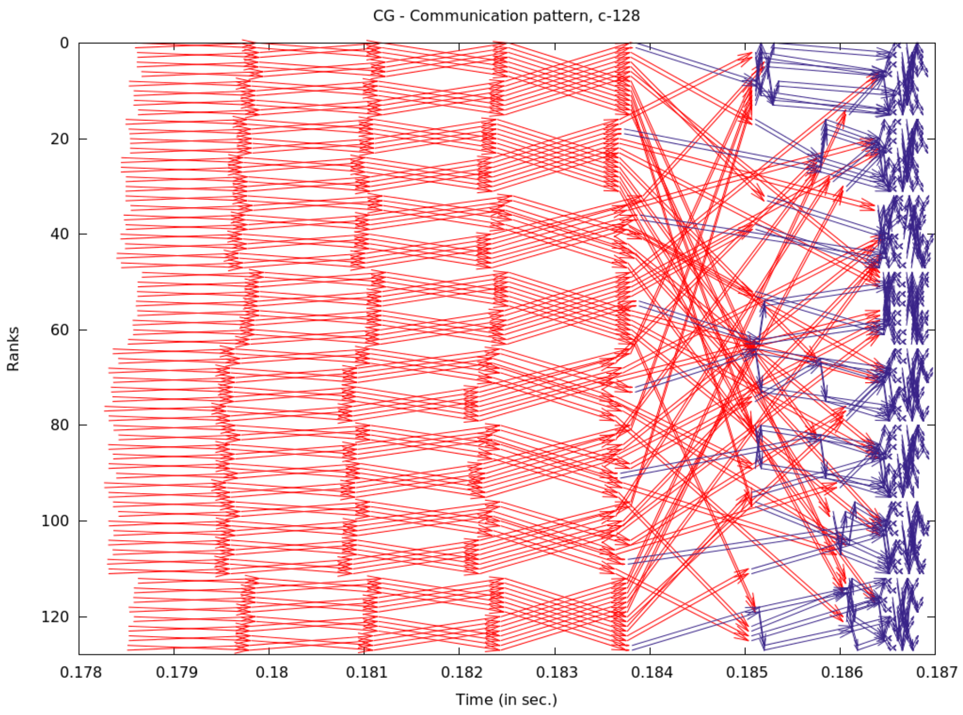


Figure 4.36: Communication pattern of a class C instance of CG for 128 processes.

To find the source of the unexpected inaccuracy for instances with 128 processes, we analyzed the Gantt chart of the execution of a class B instance (Figure4.37).

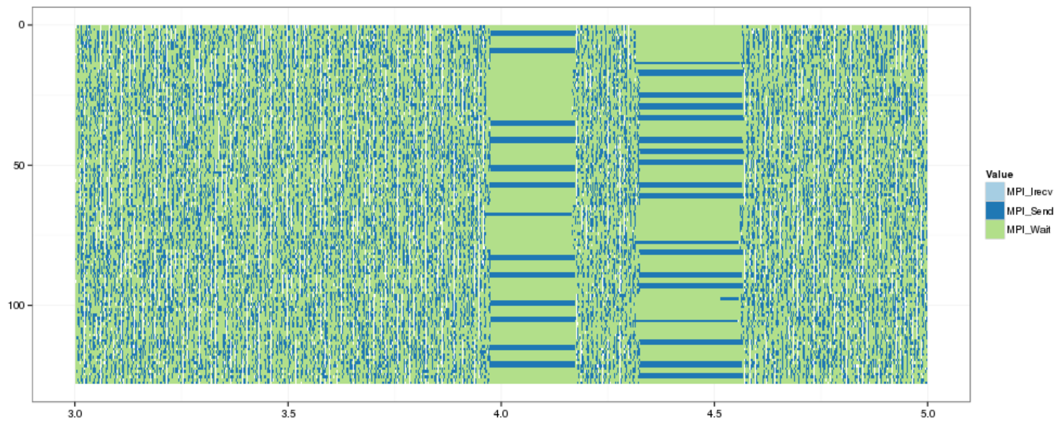


Figure 4.37: Two seconds Gantt-chart of the real execution of a class B instance of CG for 128 processes.

The execution time is 14.4 seconds while the simulated time is only of 9.9 seconds. We see two outstanding zones of `MPI_Send` and `MPI_Wait`. Such operations typically take few microseconds to less than a millisecond. Here they take 0.2 seconds. Our guess is that, due to a high congestion, the switch drops packets and slows down one (or several) process to the point where it stops sending until a timeout of .2 seconds is reached. Because of the 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 bad behavior, the real execution would take 9.55 seconds, which is extremely close to the corresponding simulated time. The same phenomenon was observed for class C. This behavior of the interconnection network can be considered as a bug that needs to be fixed in a production environment.

So we are able to predict the performance of MPI applications and indicate the reasons when there are some performance issues. Our next step is to study how much time do we need to simulate these applications.

4.4 Simulation Time

The time to replay a time-independent trace is directly related to the number of actions. As mentioned in Section 3.3.5, the traces for the EP and DT are relatively small, up to a few hundreds of kilobytes, and are composed of a small number of actions, up to a few thousands for DT. Consequently EP and DT traces for 128 processes can be replayed in less than 0.2 seconds. The large number of actions in the LU traces (see Tables 3.27 and 3.28) implies a larger simulation time. Figure 4.38 presents the evolution of the simulation time as the number of processes increases for classes B and C instances of the LU benchmark. These timings were obtained on one node of the *bordereau* cluster with our prototype simulator. We should mention that this cluster is constituted by one cabinet, thus the format of the platform file is similar to that detailed in Figure 4.2.

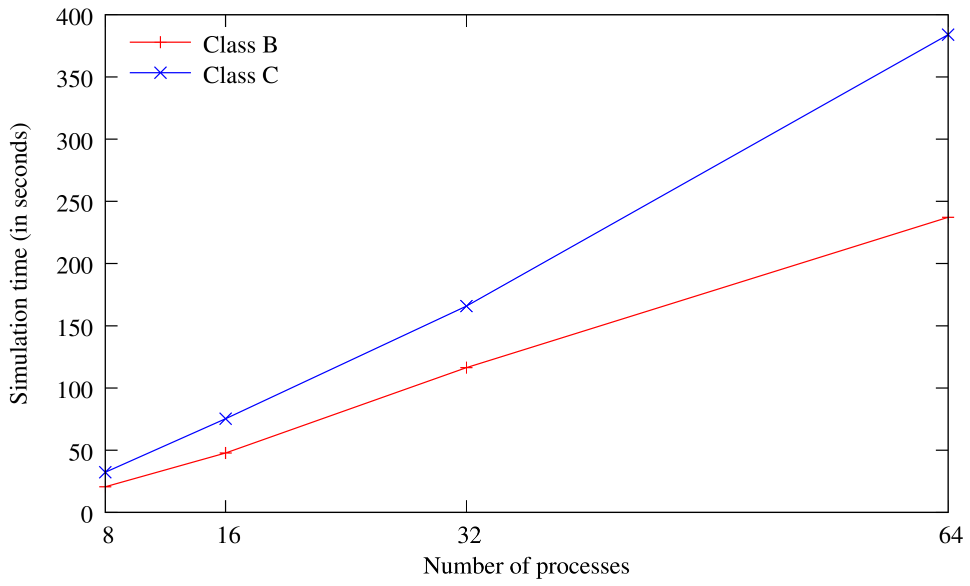


Figure 4.38: Evolution of the trace replay time with the number of processes executing a LU benchmark with the prototype simulator.

We see that the simulation time grows linearly with the number of processes. Moreover it confirms the relation between simulation time and number of actions. The average time to process with one million of actions is around 10 seconds.

The second version of our simulator was based on the SMPI layer which allowed us to save some internal context switches. These improvements lead to faster simulations. Figure 4.39 shows the simulation times. These timings were obtained on a laptop with a 2.5GHz Intel i5-3210M processor, 8 GiB of RAM and a Samsung SSD 830 hard disk. In this case we include results with 128 nodes because the *time-independent* were acquired from *graphene* cluster. This cluster is constituted by four cabinets and the format of the platform file is similar to the one given in Figure 4.3. This has to ne noted as more complicated network topologies and other related features increase the simulation.

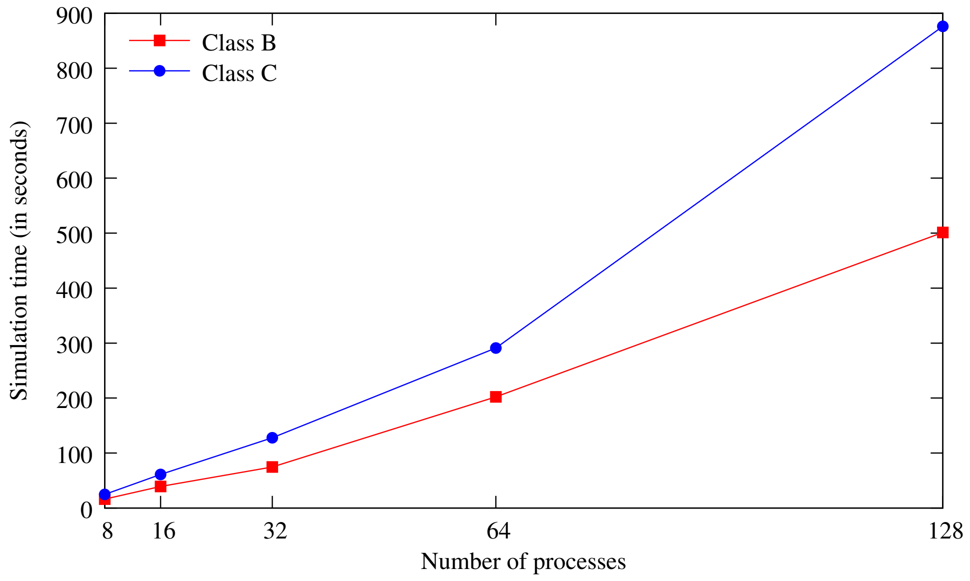


Figure 4.39: Evolution of the trace replay time with the number of processes executing a LU benchmark with the simulator based on the SMPI layer.

Similarly, the simulation time grows linearly with the number of processes for the class B and up to 64 processes for class C instances. The simulation time is more important for the C-128 instance, whose trace size is more than 1 GiB. The average time to process one million of actions is around 8 seconds for instances with up to 64 processes. For 128 processes, the processing time is a slightly higher, around 11 seconds per million of actions. These data obtained for small and medium size instances allow us to estimate the simulation time from 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 would take around 8 days on a single laptop (provided it has a large storage space). Indeed, such a huge trace contains roughly 70 billions of 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 a simulation of several compute nodes. Our *time-independent* replay framework would be a good candidate to benefit of this capacity. Traces could be distributed on several disks to allow for concurrent accesses and the time to process the traces should be greatly reduced.

Finally, when we represent the real platform as accurate as it can be, such as in Figure 4.9 then the simulation time increases due to the simulation of finer details. This is the cost that we have to pay to get even better simulation accuracy. Figure 4.40 presents the corresponding simulation times.

Again the simulation time grows linearly with the number of processes for the class B and up to 64 processes for class C instances. The average time to proceed with one million of actions is 8.6 seconds for instances with up to 64 processes. With 128 processes, the processing time is higher, around 13 seconds per million of actions. It should be noted that the *time-independent* traces for this case are 0.5% larger than the previous ones as we did not apply selective instrumentation, thus the traces correspond to the whole benchmark. The timings were also obtained on a laptop with a 2.5GHz Intel i5-3210M processor, 8 GiB of RAM and a Samsung SSD 830 hard disk.

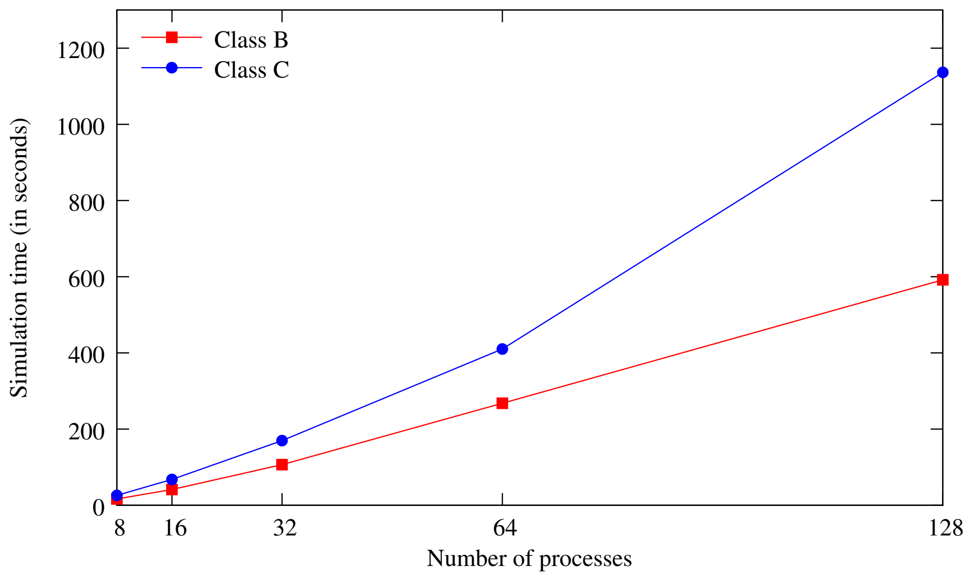


Figure 4.40: Evolution of the trace replay time with the number of processes executing a LU benchmark with our SMPI simulator and calibrating all the network parameters.

4.5 Conclusion

In this chapter we presented all the aspects of our trace replay framework and its issues. We explained how to declare to our simulator a platform file that represents the network topology of a cluster. The calibration procedure was detailed and analyzed step-by-step to be easy for a reader to understand and follow the same procedure. We also explained how to take into account the cache memory of a processor with regard to application and the problem size. The implemented simulators were analyzed in details and some examples were given to illustrate their capabilities. Moreover we mentioned some cases for which well-known profiling tools can not really provide enough data according to our requirements in comparison to the Minimal Instrumentation tool presented in the previous Chapter. We showed that our trace replay tool is able to predict the performance of MPI applications such as the EP, DT, LU, and CG benchmarks. The various implemented simulators were analyzed to show the improvement that occurred thanks to the applied modifications. We observed some performance issues and we identified the reasons that cause them. For the LU benchmark we know that we should integrate another model for the computation part to increase further the performance prediction accuracy. Moreover for the CG benchmark we figured out an improved mapping approach to increase the performance and we identified hardware issues related to high congestion and drop packets from the side of switch. The performance prediction for most instances of the studied benchmarks is accurate and we know which parts should be improved. Finally, we presented the simulation time which is the time needed to execute a simulation. We discussed also that although the simulation time was improved because of some modifications, in the case that we want to declare a more analytical platform file to improve the accuracy then we pay this cost by increasing the simulation time.

Chapter 5

Conclusions and Perspectives

Massive parallel computer systems are necessary in many scientific fields. A lot of computational science applications are executed on clusters or supercomputers to provide results on many different topics such as weather modeling, life sciences, etc. Moreover, researchers are investigating various related fields where the behavior of parallel computer systems is studied and new programming models to take into account the next generations of supercomputers are developed. The difficulty of achieving a good scale during the execution of some parallel applications while we increase the number of the processors is known and depends on many factors. Performance evaluation can provide an insight about the performance of an application and indicate possible bottlenecks. However, supercomputers are not available to all the researchers.

The work presented in this Thesis focuses on the performance evaluation and prediction of parallel applications. Simulation is a popular approach to obtain objective performance indicators platforms that are not at one's disposal. As presented in Chapter 1, there are numerous issues that increase the difficulty of our task. Our purpose was to acquire *time-independent* traces for an application and predict its performance through simulation. The prediction of the performance with good accuracy was really challenging and we had to address many issues.

In order to clarify the context of the current work, in Chapter 2 we presented some necessary concepts for our framework. Two micro-benchmarks were introduced, the OS Noise and SkaMPI. We used the first one to measure the noise caused by the Operating System on Grid'5000 platform. The second one was used to measure the latency of the network and some other data for the calibration of the communication. Moreover we mentioned the various benchmarks that constitute the NAS Parallel Benchmarks suite as our main work is based on the benchmarks EP, DT, LU and CG that belong to this suite. Three instrumentation methods, tracing, profiling and statistical profiling were introduced with the PAPI interface for handling the values of the hardware counters on the processors. We described also the advantages and disadvantages of the instrumentation methods. Similarly we analyzed the three methods for assessing the performance of applications on the resources, experiments, emulation, and simulation. Our simulator is built on top of the SimGrid simulation toolkit, thus we introduced further the SimGrid layers.

Although all the available off-line simulators which simulate MPI applications use time-stamp traces, we developed a different approach. In Chapter 3 we detailed the *time-independent* trace acquisition framework. We proposed a new execution log format that is time independent. Thus, the acquisition procedure of this trace is totally decoupled from its replay. Heterogeneous and distributed platforms with processors that belong to the same processor family, *e.g.,* i386 or amd64, can then be used to get traces without impacting the quality of the simulation which is not possible with other tool. We defined the requirements for our framework and we evaluated accordingly most of the well-known and open-source profiling tools. Our requirements were decomposed to a score system to allow the evaluation of the tools and their easy comparison. Moreover we have developed our own profiling tool which is more efficient than similar tools for our framework. This tool is memory efficient, instruments only the necessary data and it provides some internal information from the MPI profiling layer that can not be extracted automatically from other tools. For the scientists that want to use some of the well-known profiling tools such as TAU and Score-P, we have developed the corresponding tools to extract the *time-independent* traces, however all the necessary information for some MPI collective communications are not provided by all the tools. We identified issues with instrumentation time overhead and a discrepancy of the measured number of instructions. We proposed a different instrumentation method to decrease these issues and we compared with the initial one. The issue with the extra measured amount of instructions was important because the computation workload was close to the one of the instrumented application but we improved it. One more important contribution of our framework is the option to use various execution modes. In this way we can use even different hardware or a smaller platform to acquire the *time-independent* traces and predict the performance of an application for a different platform. All the experiments were carried out on Grid'5000 experimental testbed. We support open-science, thus we have documented all of our work to be able for other scientists to build an appliance to acquire *time-independent* traces on Grid'5000 experimental infrastructure. The acquisition framework was evaluated with the various instrumentation methods and different clusters to show that the overhead is acceptable considering that the *time-independent* traces can be used for many different simulations. Finally we tested our framework for large-scale experiments and our tools behave pretty efficient. We succeed in applying our framework on 778 compute nodes in 18 clusters at 9 geographically distant sites of the Grid'5000 experimental testbed.

When the *time-independent* traces are acquired, the next step is to replay them with our off-line simulator to predict the performance of the corresponding application. For this task we developed a simulator based on the SimGrid simulation toolkit. Chapter 4 presents the *time-independent* trace replay framework. First of all we had to declare the topology of the cluster under some specifications to be handled correctly from the simulator. The calibration of the simulator is an important procedure. We presented one procedure for the computation calibration based on weighted average of the computed instructions and another one for the communication calibration which is based on a more complicated procedure with a piece-wise linear model. We have described the implementation of a proposed network model that improves on this situation within SMPI, and shown that SMPI-based simulations do a better job of tracking real-world behavior than those implemented using competing simulation toolkits. Afterwards we presented the incremental procedure of developing our off-line simulator with different simulation back-ends and we showed the improvements that occurred. We also identified the reasons that the performance of some benchmarks such as LU and CG were not the expected ones. We used a calibration method which is cache-aware and figured out some switch issues. We have demonstrated that accurate modeling and performance prediction for a wide range of parallel applications requires proper consideration of many aspects of underlying communication architecture, including the breakdown of collective communication into their component point-to-point messages, the interconnect topology, and contention between competing messages that are sent simultaneously over the same link. Our priority in this work was the validation of the model at a small scale and for TCP over Ethernet networks. Such a tool would prove very useful for efforts such as the European Mont-Blanc project [113, 114], which aims to prototype exascale platforms using low-power embedded processors interconnected by Ethernet.

Perspectives

The work of this Thesis can be the base for the next version of our framework. A new computation model could be introduced to predict more accurate the performance as the power rate of a real execution usually is not stable. Our framework can support more MPI calls as both the simulator and our profiling tool can be easily extended and are available to the potential users. Moreover the simulation of a real application is one of our priorities. A next step will be to analyze its adequacy for simulating larger platforms when we can get access to such large machines for experimental purposes. The study should then extend to other kinds of interconnects (such as Infiniband) and more complicated topologies. One significant advantage is that the our *time-independent* traces used for the current work can be simulated with Infiniband network model. In general some techniques should be investigated to decrease the trace sizes and simulation time as we intend to simulate large scale instances of applications and we should avoid a huge simulation time specially compared to the real execution. In this case the parallel simulation which is under investigation could help to speed up our framework. While experimental data have been collected by hand for the present analysis, the need to collect more traces to test against various platforms is a strong incentive to automate this task. Thus, one future work will be to build a systematic validation/invalidation framework based on automatic collection of traces.

A challenge would be to investigate our framework model with regard to multicore processors and take into account the shared cache memory and all the related issues.

The proposed *time-independent* replay framework is freely distributed as part the SimGrid project under the LGPL license. It can be downloaded from `http://simgrid.gforge.inria.fr/download.html`. Moreover, the process to acquire a *time-independent* is fully described in [99].

Bibliography

[1] Krzystof Kurowski, Walter Back, Werner Dubitzky, Laszlo Gulyás, George Kampis, Mariusz Mamonski, Gabor Szemes, and Martin Swain. Complex system simulations with qoscosgrid. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 387–396, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] K. Kurowski, T. Piontek, P. Kopta, M. Mamoński, and B. Bosak. Parallel large scale simulations in the pl-grid environment. *Computational Methods in Science and Technology*, Vol. spec. iss.:47–56, 2010.

[3] Umakishore Ramachandran, H. Venkateswaran, Anand Sivasubramaniam, and Aman Singla. Issues in understanding the scalability of parallel systems. In *in: Proceedings of the First International Workshop on Parallel Processing*, pages 399–404, 1994.

[4] Xu Liu, Jianfeng Zhan, Kunlin Zhan, Weisong Shi, Lin Yuan, Dan Meng, and Lei Wang. Automatic performance debugging of spmd-style parallel programs. *J. Parallel Distrib. Comput.*, 71(7):925–937, July 2011.

[5] Philipp Gschwandtner, Thomas Fahringer, and Radu Prodan. Performance analysis and benchmarking of the intel scc. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 139–149, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.

[7] Mark Roth, Micah J. Best, Craig Mustard, and Alexandra Fedorova. Deconstructing the overhead in parallel applications. In *IISWC*, pages 59–68, 2012.

[8] Hung-Hsun Su, M. Billingsley, and A.D. George. A generalized, distributed analysis system for optimization of parallel applications. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009.

[9] Jeff R. Hammond, Sriram Krishnamoorthy, Sameer Shende, Nichols A. Romero, and Allen D. Malony. Performance characterization of global address space applications: a case study with nwchem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012.

[10] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1 edition, April 1991.

[11] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, 1991.

[12] Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner. Measurement and tuning of computer systems. In *Int. CMG Conference*, pages 793–794, 1984.

[13] Martin Schulz, Joshua A. Levine, Peer-Timo Bremer, Todd Gamblin, and Valerio Pascucci. Interpreting performance data across intuitive domains. In *ICPP*, pages 206–215, 2011.

[14] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[15] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[16] Olalekan Sopeju, Martin Burtscher, Ashay Rane, and James Browne. AutoSCOPE: Automatic Suggestions for Code Optimizations using PerfExpert. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 19–25, July 2011.

[17] H. Kotakemori, H. Hasegawa, and A. Nishida. Performance evaluation of a parallel iterative method library using openmp. In *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pages 5 pp.–436, 2005.

[18] Sameer Shende, Allen D. Malony, Alan Morris, Steven Parker, and J. Davison de St. Germain. - performance evaluation of adaptive scientific applications using {TAU}. In Anil Deane, Akin Ecer, James McDonough, Nobuyuki Satofuka, Gunther Brenner, David R. Emerson, Jacques Periaux, and Damien Tromeur-Dervout, editors, *Parallel Computational Fluid Dynamics 2005*, pages 421 – 428. Elsevier, Amsterdam, 2006.

[19] Gigabit ethernet. In *The 2001 IEEE International Symposium on Circuits and Systems (ISCAS). Tutorial Guide*, pages 9.4.1–9.4.16, 2001.

[20] G.F. Pfister. Aspects of the infiniband architecture. In *Cluster Computing, 2001. Proceedings. 2001 IEEE International Conference on*, pages 369–371, 2001.

[21] Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[22] R. Frost. Mpich performance characteristics and considerations. In *MPI Developer's Conference, 1996. Proceedings., Second*, pages 199–202, 1996.

[23] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[24] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A network performance measurement framework. In *Proceedings of High Performance Computing and Communications, HPCC'07*, volume 4782, pages 659–671. Springer, Sep. 2007.

[25] P Beckman, K Iskra, K Yoshii, S Coghlan, and A Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[26] Sequoia Benchmarks. `https://asc.llnl.gov/sequoia/benchmarks/`.

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

[28] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. Available at `http://icl.cs.utk.edu/news_pub/submissions/hpcc-challenge-intro.pdf`, 2005.

[29] The Top 500 List. `http://www.top500.org`.

[30] Jack Dongarra, Michael A. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013-4744, UTK EECS and Sandia National Labs, 2013. `http://www.netlib.org/utk/people/JackDongarra/PAPERS/HPCG-Benchmark-utk.pdf`.

[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

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

[33] Marc Abrams. Design of a measurement instrument for distributed systems. In *SIGMETRICS*, page 274, 1988.

[34] Jason Gait. A debugger for concurrent programs. *Software: Practice and Experience*, 15(6):539–554, 1985.

[35] P. A. Emrath, S. Chosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 580–588, New York, NY, USA, 1989. ACM.

[36] Allen D. Malony, James W. Arendt, Ruth A. Aydt, Daniel A. Reed, Dominique Grabas, and Brian K. Totty. An Integrated Performance Data Collection, Analysis, and Visualization System. Technical Report TTR11, Department of Computer Science, Urbana, Illinois, USA, March 1989.

[37] S. Moore, D. Cronk, S. Shende, and A. Malony. Loop-level profiling and analysis of dod applications using tau. In *HPCMP Users Group Conference, 2006*, pages 378–383, 2006.

[38] Brian J. N. Wylie, Felix Wolf, Bernd Mohr, and Markus Geimer. Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. In *Proc. of the 8th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umea, Sweden, June 2006*, volume 4699 of *Lecture Notes in Computer Science*, pages 460–469. Springer, 2007.

[39] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[40] Grid'5000. `http://www.grid5000.fr/`.

[41] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding Virtualization Capabilities to Grid'5000. Rapport de recherche RR-8026, INRIA, July 2012.

[42] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 776–783, Washington, DC, USA, 2005. IEEE Computer Society.

[43] The OAR Project. `http://oar.imag.fr/`.

[44] Emmanuel Jeanvoine, Luc Sarzyniec, and Lucas Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning for HPC Clusters. Rapport de recherche RR-8002, INRIA, June 2012.

[45] Neelam Saboo, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. Emulating petaflops machines and blue gene. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 195–, Washington, DC, USA, 2001. IEEE Computer Society.

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

[47] Kirk Webb, Mike Hibler, Robert Ricci, Austin Clements, and Jay Lepreau. Implementing the emulab-planetlab portal: Experience and lessons learned. In *In Workshop on Real, Large Distributed Systems (WORLDS*, 2004.

[48] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *In Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, 2002.

[49] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[50] Rodrigo N. Calheiros, Marco Aurélio Stelmar Netto, César A. F. De Rose, and Rajkumar Buyya. Emusim: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Softw., Pract. Exper.*, 43(5):595–612, 2013.

[51] *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, February 27 - March 1, 2013*. IEEE Computer Society, 2013.

[52] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

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

[54] Rajive Bagrodia, Ewa Deelman, and Thomas Phan. Parallel Simulation of Large-Scale Parallel Applications. *International Journal of High Performance Computing and Applications*, 15(1):3–12, 2001.

[55] Allan Snavely, Laura Carrington, Nicole Wolter, Jesús Labarta, Rosa Badia, and Avi Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC'02)*, Baltimore, MA, November 2002.

[56] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant Kale. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, April 2004.

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

[58] Edgar León, Rolf Riesen, and Arthur Maccabe. Instruction-Level Simulation of a Cluster at Scale. In *Proc. of the International Conference for High Performance Computing and Communications*, Portland, OR, November 2009.

[59] Brad Penoff, Alan Wagner, Michael Tüxen, and Irene Rüngeler. MPI-NetSim: A network simulation module for MPI. In *Proc. of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzen, China, December 2009.

[60] 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.

[61] 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 EuroPar Conference*, volume 5704 of *LNCS*, pages 135–148, Delft, August 2009.

[62] Alberto Núñez, Javier Fernández, José-Daniel Garcia, Felix Garcia, and Jesús Carretero. New Techniques for Simulating High Performance MPI Applications on Large Storage Networks. *Journal of Supercomputing*, 51(1):40–57, 2010.

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

[64] 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.

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

[66] 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.

[67] Sundeep Prakash, Ewa Deelman, and Rajive Bagrodia. Asynchronous Parallel Simulation of Parallel Programs. *IEEE Transactions on Software Engineering*, 26(5):385–400, 2000.

[68] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant Kalé. Simulation-Based Performance Prediction for Large Parallel Machines. *International Journal of Parallel Programming*, 33(2-3):183–207, 2005.

[69] Rosa Badia, Jesús Labarta, Judit Giménez, and Fransesc Escalé. Dimemas: Predicting MPI applications behavior in Grid environments. In *Proc. of the Workshop on Grid Applications and Programming Tools*, 2003.

[70] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[71] Pedro Velho and Arnaud Legrand. Accuracy study and improvement of network simulation in the simgrid framework. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 13:1–13:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[72] D. Kondo. Simboinc. `http://simboinc.gforge.inria.fr`.

[73] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale. *International Journal of Parallel, Emergent and Distributed Systems*, 25(4):241–258, Jul. 2010.

[74] Frédéric Desprez, George S. Markomanolis, and Frédéric Suter. Evaluation of Profiling Tools for the Acquisition of Time Independent Traces. Rapport technique, July 2013.

[75] PerfBench. `http://icl.cs.utk.edu/papi/custom/index.html?lid=51&slid=71`.

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

[77] Jeffrey Vetter and Michael Mccracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. In *Proccedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 123–132, Snowbird, UT, Jun 2001.

[78] Anthony Chan, William Gropp, and Ewing Lusk. *User's Guide for MPE: Extensions for MPI Programs.* Argonne National Laboratory, Mathematics and Computer Science Division, 1998.

[79] Anthony Chan, William Gropp, and Ewing Lusk. An Efficient Format for Nearly Constant-Time Access to Arbitrary Time Intervals in Large Trace Files. *Scientific Programming*, 16(2-3):155–165, 2008.

[80] Nicholas J. Wright, Shava Smallen, Catherine Mills Olschanowsky, Jim Hayes, and Allan Snavely. Measuring and understanding variation in benchmark performance. *HPCMP Users Group Conference*, 0:438–443, 2009.

[81] Extrae. `http://www.bsc.es/ssl/apps/performanceTools/`.

[82] 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.

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

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

[85] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, Wolfgang E. Nagel, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, B. Mohr, F. Peters (eds, Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with vampir.

[86] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P–A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany*, pages 1–12. Gauß-Allianz, Springer, June 2010. (to appear).

[87] Mikael Pettersson. Perfctr: the Linux Performance Monitoring Counters Driver. `http://user.it.uu.se/~mikpe/linux/perfctr/2.6/`.

[88] Perfmon2. `http://perfmon2.sourceforge.net/`.

[89] Scalasca. `http://www.scalasca.org`.

[90] Performance Research Lab. University of oregon: Tau reference guide, chapter tau instrumentaton options, `http://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch02.html`.

[91] Performance Research Lab. University of oregon: Tau user guide, chapter selectively profiling an application, `http://www.cs.uoregon.edu/research/tau/docs/newguide/ch03s03.html`.

[92] Markus Geimer, Felix Wolf, Andreas Knüpfer, Bernd Mohr, and Brian J. N. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 398–408, Berlin, Heidelberg, 2007. Springer-Verlag.

[93] Tuning and Analysis Utilities. `http://www.cs.uoregon.edu/research/tau/home.php`.

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

[95] Score-P. `http://www.score-p.org`.

[96] 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 Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium*, 2011. (to appear).

[97] F. Wolf and B. Mohr. *EPILOG binary trace-data format.* Technical report // Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich. FZJ-ZAM, 2004.

[98] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM.

[99] George Markomanolis and Frédéric Suter. Time-Independent Trace Acquisition Framework – A Grid'5000 How-to. Rapport Technique RT-0407, INRIA, May 2011. GRID5000.

[100] OTF2 Reading Interface. `http://wwwpub.zih.tu-dresden.de/~silctest/otf2/user/current/html/usage_reading_page.html`.

[101] Frédéric Desprez, George S. Markomanolis, Martin Quinson, and Frédéric Suter. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. In *Proc. of the 2nd International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 467–476, Taipei, Taiwan, September 2011.

[102] Michael Noeth, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, March 2007.

[103] Technical specification of the network interconnect in the bordereau cluster of grid'5000. `https://www.grid5000.fr/mediawiki/index.php/Bordeaux:Network`.

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

[105] Jacques Chassin De Kergommeaux and Benhur De Oliveira Stein. Pajé, an Extensible and Interactive and Scalable Environment for Visualizing Parallel Program Executions. Rapport de recherche RR-3919, INRIA, 2000.

[106] Technical specification of the network interconnect in the graphene cluster of grid'5000. `https://www.grid5000.fr/mediawiki/index.php/Nancy:Network`.

[107] Silas De Munck, Kurt Vanmechelen, and Jan Broeckhove. Improving the scalability of simgrid using dynamic routing. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 406–415, Berlin, Heidelberg, 2009. Springer-Verlag.

[108] L. Bobelin, A. Legrand, D.A.G. Marquez, P. Navarro, M. Quinson, F. Suter, and C. Thiery. Scalable multi-purpose network representation for large scale distributed system simulation. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 220–227, 2012.

[109] Paul Bedaride, Stéphane Genaud, Augustin Degomme, Arnaud Legrand, George Markomanolis, Martin Quinson, Lee Stillwell, Mark, Frédéric Suter, and Brice Videau. Improving

Simulations of MPI Applications Using A Hybrid Network Model with Topology and Contention Support. Rapport de recherche RR-8300, INRIA, May 2013.

[110] Pedro Velho, Lucas Schnorr, Henri Casanova, and Arnaud Legrand. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS) – Under revision*, 2013.

[111] How to instantiate the parameters of the SMPI model. `http://mescal.imag.fr/membres/arnaud.legrand/research/smpi/smpi_loggps.php`.

[112] Frederic Desprez, George S. Markomanolis, and Frédéric Suter. Improving the accuracy and efficiency of time-independent trace replay. In *SC Workshops*, 2012.

[113] The mont-blanc project. `http://www.montblanc-project.eu`.

[114] Nikola Rajovic, Nikola Puzovic, Lluis Vilanova, Carlos Villavieja, and Alex Ramirez. The low-power architecture approach towards exascale computing. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, ScalA '11. ACM, 2011.