Chapter 4
An Open-Source Tool-Chain for Performance Analysis

Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay

Abstract Modern supercomputers with multi-core nodes enhanced by accelerators as well as hybrid programming models introduce more complexity in modern applications. Efficiently exploiting all of the available resources requires a complex performance analysis of applications in order to detect time-consuming or idle sections. This paper presents an open-source tool-chain for analyzing the performance of parallel applications. It is composed of a trace generation framework called EZTRACE, a generic interface for writing traces in multipe formats called GTG, and a trace visualizer called VITE. These tools cover the main steps of performance analysis – from the instrumentation of applications to the trace analysis – and are designed to maximize the compatibility with other performance analysis tools. Thus, these tools support multiple file formats and are not bound to a particular programming model. The evaluation of these tools show that they provide similar performance compared to other analysis tools.

K. Coulomb (✉)
SysFera, Lyon, France
e-mail: kevin.coulomb@sysfera.com

A. Degomme
INRIA Rhône-Alpes, Grenoble, France
e-mail: augustin.degomme@inrialpes.fr

M. Faverge
Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
e-mail: mfaverge@eecs.utk.edu

F. Trahay
Institut Télécom, Télécom SudParis, Evry Cedex, France
e-mail: francois.trahay@it-sudparis.eu

H. Brunst et al. (eds.), *Tools for High Performance Computing 2011*,
DOI 10.1007/978-3-642-31476-6_4,

4.1 Introduction

Numerical simulation has become one of the pillars of science in many domains: numerous research topics now rely on computational simulations for modeling physical phenomenons. The need for simulation in various computer power hungry research areas, such as climate modeling, computational fluid dynamics, and astrophysics has led to designing massively parallel computers that now reach petaflops. Given the cost of such supercomputers, high performance applications are designed to exploit the available computing power to its maximum. During the development of an application, the optimization phase is crucial for improving the efficiency. However, this phase requires extensive understanding of the behavior and the performance of the application. The complexity of supercomputer hardware, due to the use of NUMA architectures or hierarchical caches, as well as the use of various programming models like MPI, OpenMP, MPI+threads, MPI+GPUs and PGAS models, makes it more and more difficult to understand the performance of an application. Due to the complexity of the hardware and software stack, the use of convenient analysis tools is a great help for understanding the performance of an application. Such tools permit the user to follow the behavior of a program and to spot its problematic phases.

This paper describes a complete set of tools designed for performance analysis, from the instrumentation of parallel applications using EZTRACE to the analysis of their execution with VITE. This open-source tool-chain provides a convenient and performant means to understand the behavior of an application.

The remainder of this paper is organized as follows: in Sect. 4.2, we present various research related to performance analysis. The design of EZTRACE – our instrumentation framework – is described in Sect. 4.3. Section 4.4 presents the GTG tracing library. Section 4.5 provides an overview of our trace visualization tool named VITE. The results of experiments conducted on EZTRACE are discussed in Sect. 4.6. Finally, in Sect. 4.7, we draw a conclusion and introduce future work.

4.2 Related Work

Since the advent of parallel programming and the need for optimized applications, numerous work has been conducted on performance analysis. Tools were designed for tracing the execution of parallel applications in order to understand their behavior. Some of these tools are specific to a particular programming model – MPE [4] targets MPI applications, POSIX THREAD TOOL [6] aims at applications that use pthreads, OMPTRACE [3] instruments OpenMP applications, ... – Others, such as VAMPIRTRACE [11], TAU [13] or SCALASCA [8], provide multiple modules and thus support multiple programing models. Instrumenting custom API or applications can be achieved with these tools by manually or automatically instrumenting the code.

The format of the trace generated by a tracing tool is usually specific, leading to incompatibility between performance analysis tools. Generic trace formats were designed to meet the needs of several tools. The PAJ format [7] permits the user to depict the execution of a program in a generic and hierarchic way. The OPEN TRACE FORMAT [9] (OTF) provides a generic and scalable means of tracing parallel applications more adapted to MPI applications using various communicators.

Exploring a trace file thus requires a tool designed for a particular trace format. For instance, OTF traces can be viewed with VAMPIR [10], TRIVA [12] displays PAJ traces, and the files generated with MPE can be visualized with JUMPSHOT [4]. TAU and SCALASCA embed their own trace file viewer. The lack of multiformat trace viewers forces users to switch from one system to another, depending on the tracing tool in use. A new complete tool-chain – from the application tracing to the trace analyzer – able to manipulate several trace formats, would allow users to use the most relevant format for each application to analyze while keeping the same tools.

4.3 Instrumenting Applications with EZTRACE

EZTRACE [14] has been designed to provide a simple way to trace parallel applications. This framework relies on *plugins* in order to offer a generic way to analyze programs; depending on the application to analyze or on the point to focus on, several modules can be loaded. EZTRACE provides predefined *plugins* that give the ability to the user to analyze applications that use MPI libraries, OpenMP or Pthreads as well as hybrid applications that mix several of these programming models. However, user-defined *plugins* can also be loaded in order to analyze application functions or custom libraries.

EZTRACE uses a two-phases mechanism for analyzing performance. During the first phase that occurs while the application is executed, functions are intercepted and events are recorded. After the execution of the application, the *post-mortem* analysis phase is in charge of interpreting the recorded events. This two phase mechanism permits the library to separate the recording of a function call from its interpretation. Moreover, a post-mortem analysis also reduces the overhead of profiling a program; during the execution of the application, the analysis tool should avoid performing time-consuming tasks such as computing statistics or interpreting function calls. It thus allows the user to interpret a function call event, and so a complete execution trace, in different ways depending on the point he/she wants to focus on, just by using different interpretation modules provided by the EZTRACE.

4.3.1 Tracing the Execution of an Application

During the execution of the application, EZTRACE intercepts calls to the functions specified by *plugins* and records events for each of them. Depending on the type of

```
int submit_jobs(int nb_jobs)
BEGIN
 ADD_VAR("Number of jobs", nb_jobs)
 CALL_FUNC
 EVENT("New jobs")
END

void do_work()
BEGIN
 RECORD_STATE("Working")
END
```

Fig. 4.1 Example of function instrumentation using the script language

functions, EZTRACE uses two different mechanisms for interception. The functions defined in shared libraries can be overriden using LD_PRELOAD. When the EZTRACE library is loaded, it retrieves the addresses of the functions to instrument. When the application calls one of these functions, the version implemented in EZTRACE is called. This function records events and calls the actual function. The LD_PRELOAD mechanism cannot be used for functions defined in the application since there is no symbol resolution. In that case, EZTRACE uses the DYNINST [2] tool for instrumenting the program on the fly. Using DYNINST, EZTRACE modifies the program to record events at the beginning and/or at the end of each function to instrument.

For recording events, EZTRACE relies on the FXT library [5]. FXT provides efficient support for recording traces of actions both in kernel and user space. However, EZTRACE only uses the user space feature. In order to keep the trace size as compact as possible, FXT records events in a binary format that contains only the minimum amount of information: a timestamp, an *event code* and optional parameters. During the initialization, FXT pre-allocates a buffer. When recording an event, FXT uses atomic operations for ensuring the trace consistency when multiple threads are used. At the end of the application, FXT flushes the trace to the disk.

4.3.2 Instrumenting an Application

Since EZTRACE uses a two-phases mechanism, plugins are organized in two parts: the description of the functions to instrument, and the interpretation of each function call. During the execution of the application, the first part of the plugin is in charge of recording calls to a set of functions as described in Sect. 4.3.1. The second part of the plugin is in charge of adding semantic to the trace. EZTRACE provides plugins for major parallel programming libraries (MPI, OpenMP, PThread, etc) but also allows user-defined plugins designed for custom libraries or applications. For example, the PLASMA linear algebra library [1] is shipped with an EZTRACE plugin.

In order to ease the creation of a plugin, we designed a compiler that generates EZTRACE modules from a simple script file. As depicted in Fig. 4.1, such a script

consists of a list of functions to instrument and the interpretation of each function. In this example, when the function `submit_jobs` is called, EZTRACE increases the value of a counter, calls the original function, and creates an event. A call to `do_work` is represented as a change of the state in the output trace. While some performance analysis tools allow users to specify a set of custom functions to instrument, choosing the representation of the corresponding function calls is usually impossible. Our script language gives the possibility to the users to easily create new EZTRACE modules. Since the compiler generates C files, advanced users can tune the created module to fit their needs.

4.4 Creating Trace Files with GTG

During the *post mortem* analysis phase, EZTRACE browses the recorded events and interprets them. It can then generate statistics – such as the length of messages, the duration of critical sections, etc. – or create a trace file for visualizing the application behavior. For generating trace files, the post-mortem analysis of EZTRACE relies on the Generic Trace Generator (GTG) library.[1] GTG provides an high-level interface to generate traces in different format such as Paje or OTF. This permits EZTRACE to use a single interface for creating traces in multiple formats. Thus, it can generate PAJ traces or OTF files without any modification, just giving at runtime the desired format. This high-level interface can also be used for any application supervision by any user who cares less about tracing performance such as to follow memory consumption and want to combine the tracing and the post-mortem conversion steps.

4.4.1 Overview of GTG

Although trace formats are different, most of them rely on the same structures and provide similar functionalities as it is depicted in Fig. 4.2. A set of hierarchical *containers* (1) represents processing entities such as processes, threads, or GPUs. These containers have *states* (2) that depict events that start at time T_1 and end at time T_2 – the execution of a function, the processing of a computing kernel, a pending communication, etc. – Some *events* (3) (sometimes defined as *markers*) are immediate (i. e. $T_1 = T_2$), and can represent the release of a mutex, the submission of a job, etc. Most trace formats also provide a way to track a *counter* (4) such as the total allocated memory, the number of pending jobs or the number of floating point operations per second. In order to symbolize the interaction between containers, trace formats often provide a *link* (5) feature: a couple of *events* that may happen

[1] Available under the CeCILL-C license at http://gtg.gforge.inria.fr/

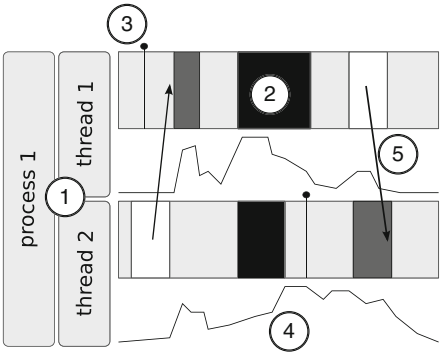

Fig. 4.2 Features commonly provided by trace formats

on different *containers*. This permits the viewer tool to represent for example: communications between processes, or signals between threads.

GTG provides a simple interface for manipulating these features. This interface then calls one of the available modules depending on the output trace format.

4.4.2 Interaction Between GTG *and* EZTRACE

Once EZTRACE is running along with the application, FXT traces are generated. The second part of EZTRACE is based on GTG, and transforms the raw traces to real meaningful traces. First, a meaning is added (for example 42 represents an `MPI_Send` request according to the MPI plugin). The semantic can represent links, events, states, etc. The hierarchical structure of the API functions is inspired by the generic PAJ trace format, thus any trace format should be matched. For instance, OTF traces can be generated. The containers can have states (*'This thread is in this function'*), notify events, or count relevant data (number of messages, memory used, number of jobs, etc). This step is based on the plugins (plugins give different meaning to the symbols). Using the EZTRACE convert tool based on GTG, one can add meaning, define containers, and describe what is happening in a function.

4.5 Analyzing Trace Files with VITE

The trace files generated by tools such as GTG or VAMPIRTRACE can be parsed for extracting statistics – such as the average message size – however, understanding the behavior and the performance of an application requires a more convenient tool. In this Section, we present VITE[2] – which stands for *Visual Trace Explorer* – an open-source multi-format trace visualizer.

[2]Freely available at http://vite.gforge.inria.fr/

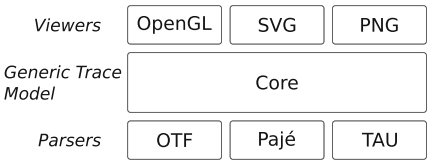

Fig. 4.3 Modules architecture in ViTE

4.5.1 A Generic Trace Visualizer

Originally, the Paj [7] trace visualizer was designed to analyze parallel applications using a simple yet generic trace format. The decline of Paj led students to design a new Paj trace viewer. ViTE was designed as a generic trace visualizer, and additional trace formats such as OTF and TAU were added later. To manage multiple formats, ViTE relies on a module architecture as depicted in Fig. 4.3.

A set of modules are in charge of parsing traces and filling the generic data structure. ViTE implements parsers for several trace formats: OTF, Paj, extended Paj (a multiple files Paj format), and TAU formats. Filling the generic data structure is a critical part of ViTE : traces may have millions of events and their processing – storing events, browsing through the event list, finding associated data, etc. – has to be efficient. The last modules are in charge of rendering traces. Such a module uses the data structure to display the trace as requested by the user. A graphical interface based on QT and OpenGL allows an user-friendly browsing of the trace. Additional rendering modules generate SVG or PNG files depicting traces to easily export the results.

Although trace formats are different in their design, most of them provide similar functionalities. ViTE implements a generic data structure and manipulates abstract objects representing the different features defined by trace formats. This abstraction permits the developers to easily implement additional parsers for new trace formats, while rendering traces in a homogeneous way.

4.5.2 Displaying Millions of Events

As depicted in Fig. 4.4, ViTE is able to display millions of items. To manage such performances, an efficient data structure and an efficient rendering is needed. The data structure is based on binary trees, as depicted in Fig. 4.5. Each tree is built over a sorted list of known size. Hence the construction of the data structure is not optimal (building lists then trees) but the conversion is not so slow (knowing the size of the list, the tree is build in linear time) and it offers an efficient data structure. Moreover, this transformation allows an internal modification, where for instance states are not recorded but state changes are. It avoids storing both the start and the duration of the state: we only store a state change that occurs at time t. This avoids storing an unneeded floating value for each state. Thanks to these trees, any element can

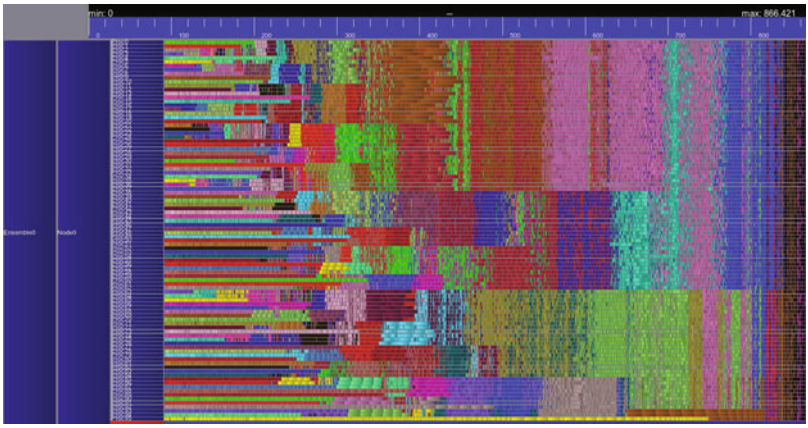

Fig. 4.4 Example of trace display with ViTE. The trace contains ten millions events

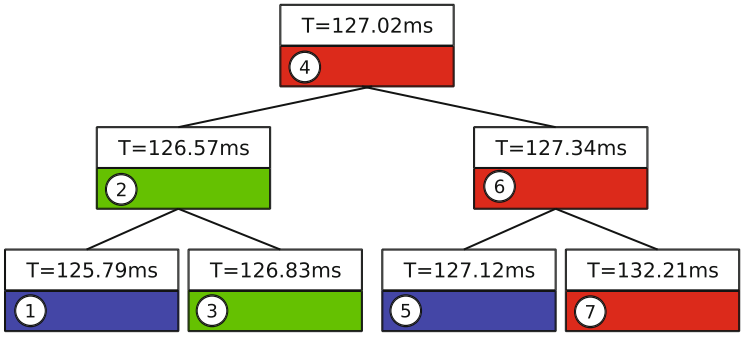

Fig. 4.5 Example of sorted binary tree representation of events

be accessed in logarithmic time and unneeded branches of the tree can be avoided while browsing.

The binary tree structure is also useful for rendering the trace. In order to avoid creating millions of graphical elements, portions of the trace have to be summarized. ViTE uses a resolution parameter for eliminating the events that are too small to be rendered: if a node and his father are too close, then the resolution will not be enough for displaying them, and it is useless to keep on browsing all the nodes between the two (the subtree of the node on the same side as the father).

For example, when rendering the binary tree depicted in Fig. 4.5 with a resolution of 1 ms, ViTE browses event *# 4*. It then handles *# 2*. Since the interval between events *# 2* and *# 4* is lower than the resolution ($T_{\#4} - T_{\#2} < 1$ ms), event *# 3* is not taken into account. Event *# 1* is then handled normally. Then, ViTE processes event *# 6*. Event *# 5* is skipped since it is beneath the resolution ($T_{\#6} - T_{\#4} < 1$ ms) and event *# 7* is handled normally. As a result, the number of elements to display,

as well as the number of nodes to browse, is limited increasing the rendering performance. If the user zooms in, the resolution decreases and the same algorithm is used.

The rendering is also critical; OpenGL has been chosen after benchmarking several solutions based on Qt, GTK, SDL, GNUStep and JAVA. Despite the fact that Qt and GTK could provide a better and easier interaction with the trace, the OpenGL engine, with our own mouse placement detection, appeared to be the most scalable solution. Moreover, on some machines, OpenGL can benefit from hardware optimization with the GPU.

4.6 Evaluation

When analyzing the performance of parallel applications that generate millions of events, the performance of the analysis tool is important. The overhead of the instrumentation should be as low as possible, and the visualization tool should allow a smooth browsing of the resulting trace. In this Section, we assess the performance of EZTRACE. We evaluate the raw performance of the instrumentation mechanisms used in EZTRACE on a synthetic benchmark as well as on application kernels.

The results of this evaluation were obtained on the CLUSTER0 platform. It is composed of 32 nodes, each being equipped with two 2.2 GHz dual-core OPTERON (2214HE) CPUs featuring 4 GB of memory. The nodes are running Linux 2.6.32 and are interconnected through MYRINET MYRI-10G NICs. We compare EZTRACE with VAMPIRTRACE in its 5.9 version.

4.6.1 Overhead of Trace Collection

In order to evaluate the raw overhead of program instrumentation, we use an MPI ping pong program. We measure the latency obtained for 16-bytes messages. We instrument this program using the automatic (i.e. using LD_PRELOAD) and manual (i.e. using DYNINST) mechanisms described in Sect. 4.3.1, then we compare the overhead of using EZTRACE or VAMPIRTRACE to the performance obtained without instrumentation. For VAMPIRTRACE, the automatic instrumentation is obtained by using its MPI module. The manual instrumentation is obtained by inserting call to VT_USER_START and VT_USER_END in the application.

Table 4.1 shows the results we obtained. Using VAMPIRTRACE automatic instrumentation degrades the latency by 1.1 µs while the manual instrumentation causes an overhead of 700 ns. The difference is due to the fact that VAMPIRTRACE generates events at the entry and the exit of functions in both instrumentations, but it also generates a *SendMessage* or *ReceiveMessage* event when the MPI module is selected.

Table 4.1 Results of the 16-bytes latency test

Method	Open MPI (μs)	VampirTrace (μs)	EZTrace (μs)
Automatic	4.99	6.12	5.68
Manual	4.99	5.71	5.67

Table 4.2 NAS Parallel Benchmark performance for Class A and B

				VampirTrace		EZTrace		
Kernel	Class	# Proc.	OpenMPI (s)	Time (s)	Overhead (%)	Time (s)	Overhead (%)	# Events/s
BT	A	4	70.57	70.58	0.01	70.39	−0.26	825
CG	A	4	2.64	2.68	1.52	2.68	1.52	12,546
EP	A	4	9.61	9.69	0.83	9.72	1.14	5
FT	A	4	6.63	6.67	0.55	6.62	−0.20	22
IS	A	4	0.63	0.64	2.13	0.62	−1.06	482
LU	A	4	42.08	42.15	0.17	41.39	−1.64	12,282
MG	A	4	5.04	5.06	0.46	5.07	0.66	2,978
SP	A	4	166.25	165.94	−0.18	166.32	0.04	696
BT	B	36	26.08	25.83	−0.97	26.37	1.10	59,350
CG	B	32	16.29	16.46	1.02	16.60	1.88	192,667
EP	B	32	4.81	4.79	−0.42	4.76	−1.04	81
FT	B	32	11.76	11.61	−1.30	11.55	−1.81	255
IS	B	32	0.97	0.96	−1.03	0.96	−1.03	2,580
LU	B	32	33.75	34.11	1.07	33.67	−0.24	–
MG	B	32	2.14	2.16	0.78	2.13	−0.62	215,515
SP	B	36	51.18	51.98	1.57	52.07	1.75	59 922

Instrumenting the application with EZTRACE causes an overhead of 700 ns for both mechanisms. This is because EZTRACE records events at the entry and the exit of functions for both manual and automatic modes. The *SendMessage* and *ReceiveMessage* events are generated during the *post mortem* phase.

4.6.2 NAS Parallel Benchmarks

In order to evaluate the overhead of EZTRACE on more realistic computing kernels, we also measure its performance for NAS application kernels. The experiment was carried out with 4 computing processes for Class A and 32 processes (or 36 for BT and SP that require a square number of processes) for Class B. We instrument MPI functions of these kernels with EZTRACE and VAMPIRTRACE automatic modules.

Table 4.2 summarizes the results we obtained. Since EZTRACE *post mortem* phase crashes for the LU kernel for Class B, the number of events in the resulting OTF trace is not reported. The results show that instrumenting these kernels with EZTRACE or VAMPIRTRACE does not significantly affect the performance:

variation of the execution time is less than 2 %. This experiments also show that intensive event recording kernels – such as MG or CG for Class B – do not suffer from the overhead of the instrumentation.

4.7 Conclusion and Future Work

Programming a parallel application that efficiently exploits a supercomputer becomes more and more tedious due to the increasing complexity of hardware – multicore processors, NUMA architectures, GPGPUs, etc. – and the use of hybrid programming models that mix MPI, OpenMP or CUDA. Tuning such an application requires the programmer to precisely understand its behavior.

We proposed in this paper an open-source tool-chain for analyzing the performance of modern parallel applications. This software suite is composed of EZTRACE – a generic framework for instrumenting applications –, GTG – a tool for generating traces in multiple formats –, and VITE – a trace visualizer that supports several trace formats –. These tools were designed to provide an open-source alternative to other performance analysis tools, while allowing interoperability with other tools such as Vampir or TAU. The evaluation shows that this genericity does not imply extra overheads since EZTRACE provides similar performance when compared to VAMPIRTRACE.

In the future, we plan to study more precisely the performance of the whole software suite and to improve it. Additional modules are to be developped in EZTRACE in order to allow the analysis of programs running CUDA or OpenCL. We also plan to improve EZTRACE performance analysis capabilities so that it can detect programming or runtime issues such as network congestion or insufficient overlap of communication and computation. Future work concerning GTG includes the support for other trace formats – such as TAU – and enhancing the API. We also plan to merge VITE and TRIVA [12] projects. TRIVA is based on PAJ software and provides new ways of displaying information such as treemaps, or network graphs that will benefit to ViTE. On the other side, TRIVA will benefit from the multi-format parser and from the OpenGL display.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the plasma and magma projects. In: Journal of Physics: Conference Series, vol. 180, p. 012037. IOP Publishing, Bristol (2009)
2. Buck, B., Hollingsworth, J.: An API for runtime code patching. Int. J. High Perform. Comput. Appl. **14**(4), 317–329 (2000)
3. Caubet, J., Gimenez, J., Labarta, J., DeRose, L., Vetter, J.: A dynamic tracing mechanism for performance analysis of OpenMP applications. In: Eigenmann, R., Voss, M.J. (eds.) OpenMP Shared Memory Parallel Programming, pp. 53–67. Springer, Berlin/London (2001)

4. Chan, A., Gropp, W., Lusk, E.: An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. Sci. Program. **16**(2–3), 155–165 (2008)
5. Danjean, V., Namyst, R., Wacrenier, P.: An efficient multi-level trace toolkit for multi-threaded applications. In: Euro-Par 2005 Parallel Processing, pp. 166–175. Springer, Berlin/New York (2005)
6. Decugis, S., Reix, T.: NPTL stabilization project. In: Linux Symposium, vol. 2, p. 111. Ottawa, Canada (2005)
7. de Kergommeaux, J., de Oliveira Stein, B.: Pajé: an extensible environment for visualizing multi-threaded programs executions. In: Euro-Par 2000 Parallel Processing, pp. 133–140. Springer, Berlin (2000)
8. Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010)
9. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.: Introducing the open trace format (OTF). In: Computational Science–ICCS 2006, pp. 526–533. Springer, Berlin/New York/ Heidelberg (2006)
10. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M., Nagel, W.: The Vampir performance analysis tool-set. In: Tools for High Performance Computing, pp. 139–155. Springer, Berlin (2008)
11. Muller, M., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.: Developing scalable applications with Vampir, VampirServer and VampirTrace. In: Proceedings of the Minisymposium on Scalability and Usability of HPC Programming Tools at PARCO, Jülich, Germany. 4 to 7 september (2007)
12. Schnorr, L.M., Huard, G., Navaux, P.O.: Triva: interactive 3d visualization for performance analysis of parallel applications. Future Gener. Comput. Syst. **26**(3), 348–358 (2010)
13. Shende, S., Malony, A.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287 (2006)
14. Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., Dongarra, J.: EZTrace: a generic framework for performance analysis. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Newport Beach. IEEE, Piscataway (2011)