Scalasca v2: Back to the Future

Ilya Zhukov, Christian Feld, Markus Geimer, Michael Knobloch, Bernd Mohr, and Pavel Saviankou

Abstract Scalasca is a well-established open-source toolset that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes. The latest Scalasca v2 release series is based on the community instrumentation and measurement infrastructure Score-P, which is jointly developed by a consortium of partners from Germany and the US. This significantly improves interoperability with other performance analysis tool suites such as Vampir and TAU due to the usage of the two common data formats CUBE4 for profiles and the Open Trace Format 2 (OTF2) for event trace data. This paper will showcase recent as well as ongoing enhancements, such as support for additional platforms (K computer, Intel Xeon Phi) and programming models (POSIX threads, MPI-3, OpenMP4), and features like the critical-path analysis. It also summarizes the steps necessary for users to migrate from Scalasca v1 to Scalasca v2.

1 Motivation and Introduction

The Scalasca toolset [10, 23] is a portable, well-established software package which supports the performance optimization of parallel applications by measuring and analyzing their dynamic runtime behavior. It has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray X series, but is also well-suited for small- and medium-scale HPC platforms. A distinctive feature of Scalasca is its scalable automatic trace analysis, which identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes. Scalasca is available under a 3-clause BSD open-source license.

I. Zhukov • C. Feld • M. Geimer • M. Knobloch • B. Mohr (✉) • P. Saviankou
Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre
52425 Jülich, Germany
e-mail: i.zhukov@fz-juelich.de; c.feld@fz-juelich.de; m.geimer@fz-juelich.de; m.knobloch@fz-juelich.de; b.mohr@fz-juelich.de; p.saviankou@fz-juelich.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2014*,
DOI 10.1007/978-3-319-16012-2_1

Users of the Scalasca toolset can choose between two different analysis modes: (i) performance overview on the call-path level via profiling and (ii) the analysis of wait-state formation via event tracing. Wait states often occur in the wake of load imbalance and are serious obstacles to achieving satisfactory performance and scalability. The latest release at the time of writing (v2.1) also includes a scalable critical-path analysis [3]. Analysis results are presented to the user in an interactive explorer called Cube that allows the investigation of the performance behavior on different levels of granularity along the dimensions metric, call path, and process/thread. Selecting a metric displays the distribution of the corresponding value across the call tree. Selecting a call path again shows the distribution of the associated value over the machine or application topology. Expanding and collapsing metric or call tree nodes allows to investigate the performance at varying levels of detail.

Scalasca and Cube are both under active development. New features – in many cases triggered by user requests – are developed, tested, and integrated into the main development branch. In addition, bugs that emerge are fixed and performance as well as scalability are constantly evaluated and improved. The latest Scalasca v2 release series is based on the community-driven instrumentation and measurement infrastructure Score-P [15], which is jointly developed by a consortium of partners from Germany and the US. This significantly improves interoperability with other performance analysis tool suites such as Vampir [14] and TAU [25], enabled by the usage of two new common data formats: CUBE4 for profiles and the Open Trace Format 2 (OTF2) for event trace data.

Re-architecting Scalasca on top of Score-P marked an important milestone for the Scalasca project. While the previous Scalasca v1 release series provided a rather monolithic, mostly self-contained toolset which included all components required to perform measurements and performance analyses of parallel applications in a single package, the Score-P project followed a component-oriented philosophy from the very beginning – with the intention to allow a more decoupled development of the individual components and to share them among several tools. Therefore, the changeover to Score-P naturally lend itself to also "componentize" the Scalasca toolset.

From a user's perspective, the most obvious difference between the Scalasca v1 series and Scalasca v2 is that one now has to install several packages instead of just one. In terms of actual usage, however, we tried to keep the convenience commands and their options as stable as possible, although a few changes were necessary to adapt to new features and to provide new functionality. Nevertheless, we believe that the benefits of the long-term commitment of the Score-P partners and the increased tool interoperability overall far outweigh the additional (one-time) learning effort.

The remainder of this article is structured as follows: In Sect. 2, we briefly introduce the community instrumentation and measurement system Score-P. Section 3 then describes the performance analysis toolset Scalasca, and outlines changes and differences between the old (v1) and new (v2) generation of Scalasca in more detail. Finally, Sect. 4 gives an overview of new features introduced in the latest versions of Score-P, Scalasca, and Cube, as well as future plans for enhancements.

2 Scalable Performance Measurement Infrastructure for Parallel Codes (Score-P)

The community performance measurement and runtime infrastructure *Score-P* [15] – Scalable Performance Measurement Infrastructure for Parallel Codes – is a highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications.

Until 2009, the performance tools landscape was fragmented and revealed many redundancies – not so much in the analysis parts, but in program instrumentation, measurement data acquisition, and data formats. The analysis tools Scalasca [10], Vampir [14, 27], TAU [25], and Periscope [1] all provided their native instrumentation and measurement systems, data formats, and configuration options. Although very similar in functionality, the effort to maintain a portable and scalable measurement tool with associated data formats was redundantly expended by every tool developer group. Moreover, on the user's side, dealing with different measurement tools that expose incompatible configurations, diverse feature sets, and different options for equivalent features, created discomfort and made interoperability between analysis tools difficult.

Therefore, a consortium of tool developer groups from Germany and the US as well as other stakeholders in 2009 started the Score-P project to remedy this situation. Score-P aims to provide a single, highly scalable, and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications written in C/C++ and Fortran. Score-P comes along with scalable, space-efficient, and well-defined data formats: CUBE4 for profiling and OTF2 [7] for tracing data. Based on the experience with the existing tools, the interaction with their users, and the programming paradigms most prevalent in HPC, the Score-P architecture as shown in Fig. 1 was derived.

The workflow to obtain measurement data usually consists of instrumentation, measurement configuration, and running the instrumented application. At instrumentation time, this typically means to recompile/relink the application using the `scorep` compiler wrapper as a prefix to the actual compile/link commands. This step inserts measurement probes into the application code that collect performance-related data when triggered during measurement runs. Currently, the following means of instrumentation are supported:

- Compiler-based instrumentation (entry/exit of functions)
- MPI/SHMEM library interposition
- OpenMP source-code instrumentation using OPARI2
- POSIX thread instrumentation using library wrapping
- CUDA event interception using the CUPTI library
- Source-code instrumentation via the TAU instrumentor [9]
- User instrumentation using convenient macros

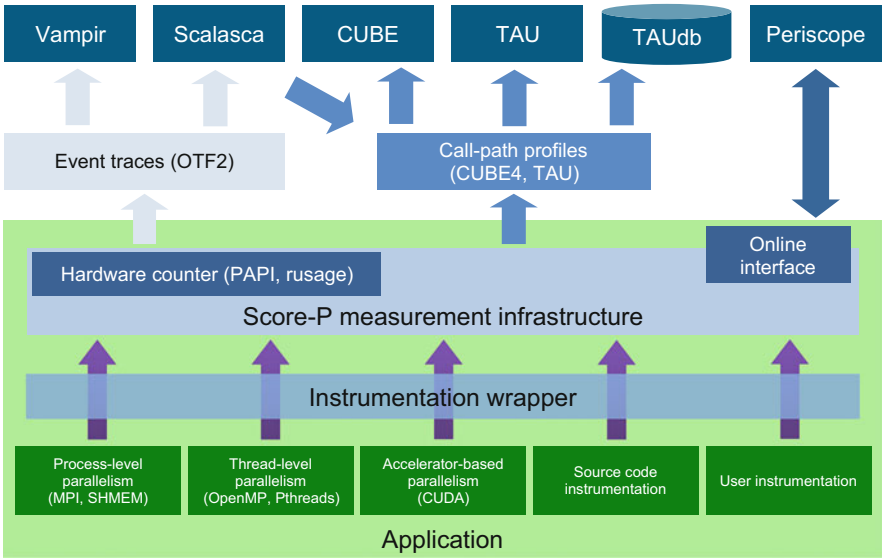

Fig. 1 Score-P architecture overview. Instrumentation wrappers capture important events during application execution and pass them to the Score-P measurement core. Here, this data is optionally combined with hardware counter information, and detailed event traces or call-path profiles are generated, which can be handled by various analysis tools. In addition, an online interface allows tools to access profiling data already during measurement

The instrumentation probes are placed at performance-relevant regions in the code, e.g., function entry and exit, thread-management functions, and communication routines. Once triggered, the probes obtain relevant performance data and pass them to the Score-P measurement core where they are augmented with timestamps and, optionally, additional metrics from hardware and OS counter sources. This enhanced event data is then processed by so-called substrates, currently either profiling and/or tracing. Substrates collect and process the measurement events thread-locally and write the results efficiently to file. The profiling substrate writes either CUBE4 or TAU profiles, while the tracing substrate outputs its data in the space-efficient OTF2 format.

Measurements can be conveniently configured using environment variables. That is, once instrumented, the application can be run in different measurement configurations without recompilation. Depending on the settings of the configuration variables, users can

- Switch between profiling and tracing,
- Apply event filters to control overhead,
- Collect additional metrics (e.g., hardware counters),
- Adapt measurement internals (e.g., size of memory buffers),

- Specify output parameters,
- And much more.[1]

After its first public release end of 2011, the Score-P infrastructure has become the default instrumentation and measurement system for Scalasca v2, Vampir, and Periscope, and is also supported by the TAU performance system as a configuration option. It is actively maintained and constantly enhanced with new features by the contributing partners, managed by a meritocratic governance model.

3 Scalasca v2

As already mentioned in the introduction, the changeover to the Score-P instrumentation and measurement system as the new foundation of the Scalasca toolset marked an important milestone for the Scalasca project. While the previous Scalasca v1 release series provided all components required to perform measurements and performance analyses of parallel applications in a single distribution package, the Score-P project follows a rather different philosophy. Here, smaller components are developed and released in a more decoupled fashion – with the intention to share them among several tools. Therefore, switching to Score-P naturally lend itself to also "componentize" the Scalasca toolset, eliminate some legacy components, and substitute a number of previously internal components by now external components from the Score-P universe:

- The custom instrumentation and measurement system of Scalasca v1 (EPIK) was replaced by Score-P.
- The libraries to read/write Scalasca's original custom trace file format (EPILOG) were substituted by OTF2.[2]
- The graphical report explorer Cube v3 as well as associated read/write libraries and command-line tools were split off as a separate component, significantly improved, and subsequently released in the Cube v4 package.
- The legacy serial trace analyzer from Scalasca's predecessor project KOJAK as well as the non-scalable trace converters from the EPILOG to the Jumpshot and Paraver trace formats were removed.

These substantial changes, however, posed a number of challenges for both the Scalasca developers and the Scalasca user community.

[1] The command `scorep-info config-vars` provides an exhaustive list of options available for a particular installation.

[2] A stripped-down EPILOG reader library is still included in Scalasca v2 to provide backwards compatibility support for existing trace files, except for traces stored in SIONlib [8] containers which are not supported.

From a developer's point of view, adjusting the code base of the remaining core components of Scalasca (i.e., its scalable trace-based analysis tools) required a significant development effort. On the one hand, code sections dealing with file I/O had to be reworked, since the APIs of the Cube v4 and OTF2 libraries differ significantly from their previous counterparts. Moreover, both Cube and OTF2 are under continuous development and now provided as external components, thus creating a need to identify – and potentially deal with – different versions of these libraries. On the other hand, OTF2 also introduced many small differences in the trace definition and event data compared to the previous EPILOG format, which required a careful inspection and adaption of Scalasca's trace analysis algorithms.

From a user's perspective, the most obvious difference between the Scalasca v1 series and Scalasca v2 is that now several packages have to be installed instead of just one. In terms of actual usage, however, we tried to keep the convenience commands (`scalasca` and its shortcuts `skin`, `scan`, and `square`) and their command-line options as stable as possible. Nevertheless, some changes were necessary, which will be described in more detail below. All in all, however, we believe that the benefits of the improved functionality, the long-term commitment of the Score-P partners, and the increased tool interoperability far outweigh the additional installation and (one-time) learning effort.

3.1 General Comparison of Scalasca v1 and Scalasca v2

As mentioned above, basically all components of Scalasca v2 had to be adjusted and/or refactored to integrate with the Score-P instrumentation and measurement system. In the following, however, we focus on the user-visible differences. That is, we compare the usage of the two generations of the Scalasca toolset to help existing users with transitioning from the old to the new version more easily. An initial comparison of their performance and scalability can be found in [28].

Table 1 summarizes the differences between Scalasca v1 and Scalasca v2 from a high-level perspective. While at first glance the table suggests that there are – except for the license – no commonalities between the two generations of Scalasca, the detailed comparisons in the following subsections will show that the differences are often, especially with respect to the Scalasca convenience commands, only marginal and in most cases motivated by improved functionality provided by Score-P.

Obviously, some usability changes can barely be avoided when replacing a core component such as the instrumentation and measurement infrastructure by a unified solution that emerged from existing measurement systems of multiple tool suites. For example, Score-P's instrumenter command uses different command-line options than the instrumenter provided by Scalasca's former measurement system EPIK. Moreover, their manual instrumentation APIs are incompatible, though the API supported by Score-P is much more feature rich. In addition, the names (and often also the valid values) of the environment variables used to configure measurements

Table 1 High-level comparison of Scalasca v1 and Scalasca v2

	Scalasca v1	Scalasca v2
Instrumentation system	EPIK	Score-P
Command line switches	Different	
Manual instrumentation API	Different	
Environment variables	Different	
Filter file syntax	Different	
Memory buffers	separate for each thread	memory pool on each process
Experiment directory name	`epik_###`	`scorep_###`
Profile format	CUBE3	CUBE4
Trace format	EPILOG	OTF2
Scalable trace I/O	Supports SIONlib	Partially supports SIONlib
License	3-clause BSD	

have changed. Therefore, it is inevitable that existing users of Scalasca transitioning to the new version have to familiarize themselves with these modifications.

Another notable change concerns the syntax of filter files, which are commonly used to control the measurement overhead by excluding source-code regions from being measured. While Scalasca v1 only supported a simple plain text file listing function names (optionally using wildcards) that should be ignored during measurement, Score-P provides the ability to filter based on function names as well as file names, also supporting wildcards. In addition, both blacklisting and whitelisting of functions/files are now supported. This, however, necessitated the introduction of an incompatible, more powerful syntax for filter files.

Finally, handling of memory buffers internal to the measurement system has changed. Scalasca v1 provided two environment variables to control its memory usage: one to define the size of a per-process buffer for definition data and another to set the size of a per-thread trace data buffer, the latter only if tracing mode is configured. By contrast, Score-P uses a memory pool on each process, whose total size can be controlled using a single variable. This approach is typically more memory efficient and in general easier to use.

Other differences such as the experiment directory prefix and the different profile and trace file formats used by the two generations of Scalasca, although apparent, usually do not affect the user directly. One notable exception are the different levels of support for writing trace data in SIONlib [8] containers. At the time of writing, the current Score-P release (v1.3) only supports SIONlib containers for OTF2 traces from pure MPI applications, while the former EPIK measurement system also included support for hybrid MPI+OpenMP codes. This currently limits the scalability of tracing in Score-P for such applications. However, extending OTF2 and Score-P to support SIONlib for all supported programming models (and their combination) is already work in progress.

3.2 Changes in the Scalasca Convenience Commands

Although there are many differences between Scalasca v1 and Scalasca v2, we tried to preserve the overall usage workflow and the set of associated convenience commands. Therefore, users already familiar with Scalasca v1 should be able to continue using these commands to accomplish the most common tasks.

Most of Scalasca's functionality can be easily accessed through the `scalasca` command, which provides action options that in turn invoke the corresponding underlying commands: `scorep` for instrumentation, `scan` for measurement execution control, and `square` for analysis report postprocessing and examination. These actions are:

- `scorep`
  is used to instrument the application during compilation/linking. The former `scalasca -instrument` (or short `skin`) command is now deprecated and only provided to assist in converting existing measurement configurations to Score-P. Hence, it tries to map the command-line options of the Scalasca v1 instrumenter onto corresponding options of the `scorep` command as far as this is possible, and prints the new command to standard output. However, to take full advantage of Score-P's functionality, it is recommended to use the `scorep` instrumenter command directly.
- `scalasca -analyze` (or short `scan`)
  is used to control the Score-P measurement environment during the execution of the target application, and to automatically initiate Scalasca's trace analysis after measurement completion in case tracing was requested.
- `scalasca -examine` (or short `square`)
  is used to postprocess the analysis report generated by a Score-P profiling measurement and/or Scalasca's automatic post-mortem trace analysis, and to start the analysis report examination browser Cube.

Each of these commands is discussed in further detail below.

3.2.1 Scalasca Application Instrumenter

In order to perform measurements with Scalasca, the application's source code has to be prepared, a process called instrumentation. That is, special calls have to be inserted into the code which trigger measurements at performance-relevant points during application runtime. The Score-P measurement system can do this instrumentation in various ways; the most important ones are listed below.

Compiler instrumentation inserts specific instrumentation probes at the enter and exit of functions. This method is supported by most modern compilers and can usually be enabled by a particular compiler flag. It is the most convenient way to instrument an application but may result in high overhead and/or disruptive

measurements. Both of these issues can be addressed by manual instrumentation and/or measurement filtering (see Sect. 3.2.2).[3]

If compiler instrumentation is not possible or it incurs too much overhead, one can use *manual instrumentation*. Here, the user manually inserts function calls into the application's source code, using the provided manual instrumentation API. Note that the Score-P API differs from the previous EPIK API. Besides changing the common prefix of the API calls from `EPIK` to `SCOREP`, the calls themselves have been renamed to be more expressive and now also take different parameter lists. This, however, allowed to make the API more flexible and powerful, for example, to support phase and dynamic region profiling. For more detailed information on the new possibilities, please consult the Score-P user manual [24].

Compiler-based and manual instrumentation can also be used in combination, thereby allowing to mark specific regions other than function entry and exit and thus augmenting the measurement output with additional information. While manual instrumentation is very flexible, it is more time-consuming and requires extra implementation efforts.

If Score-P has been configured with support for the Program Database Toolkit (PDToolkit) [9], *automatic source-code instrumentation* can be used as an alternative to compiler instrumentation. In this case, the source code of the target application is pre-processed before compilation and appropriate calls to the manual instrumentation API are inserted automatically. PDT instrumentation allows for instrumentation-time filtering, thus potentially reducing overhead.

In the rare case that neither automatic compiler instrumentation nor automatic source-code instrumentation via PDToolkit are supported, the user can apply *semi-automatic instrumentation*. Here, instrumentation can be accomplished by means of the Open Pragma And Region Instrumenter (OPARI2) [20], a source-to-source instrumenter that was originally developed to detect and instrument OpenMP [21] directives. OPARI2 scans the source code for OpenMP directives, inserting specific instrumentation probes according to the POMP2 interface [18]. The same procedure can be used to instrument user regions defined via pragmas or to instrument directive-based non-OpenMP paradigms [13]. The advantage of this method is that the pragma-based instrumentation directives will be ignored by the compiler during "normal" compilation.

Table 2 shows a comparison between the most relevant options that can be passed to the old and the new instrumentation commands. As can be seen, the command-line options supported by the `scorep` command are very similar to those used by `scalasca -instrument` or `skin`, respectively. However, the Score-P instrumenter command provides many additional options not listed in the table, making it more flexible and feature rich than its predecessor.[4]

[3] Some compilers, e.g., Intel, provide command-line options to control instrumentation-time filtering, however the functionality provided is typically limited and very compiler (and compiler version) specific.

[4] The command `scorep --help` provides an exhaustive list of instrumentation options.

Table 2 Comparison between the `skin` (`scalasca -instrument`) and `scorep` commands

Scalasca v1		Scalasca v2		
Command	Option	Command	Option	Description
skin	-v	scorep	-v --verbose=<value>	Enable verbose mode
	-comp=all		--compiler	Turn on compiler instrumentation (default)
	-comp=none		--nocompiler	Turn off compiler instrumentation
	-user		--user	Turn on user instrumentation
	-pdt		--pdt	Process source files with PDT instrumenter
	-pomp		--pomp	Process source files for POMP directives
			--dry-run	Display executed commands. No execution
			--keep-files	Do not delete temporarily created files

3.2.2 Scalasca Measurement Collection and Analysis Nexus

The Scalasca measurement and analysis nexus (`scalasca -analyze` or short `scan`) configures and manages the collection of performance experiments of an application instrumented by Score-P. By setting different measurement and analysis configurations, users can perform various experiments using a single instrumented executable without re-instrumentation. Typically, a single performance experiment is collected in an unique archive directory that contains not only measurement and analysis reports but also a measurement log file and configuration data. Score-P uses a directory name prefix of `scorep_` for this directory, while Scalasca's EPIK measurement system used `epik_`.

Users can choose between generating a summary analysis report (profile) with aggregate performance metrics for each function call path and/or generating per-thread event traces recording time-ordered runtime events which are automatically fed to Scalasca's trace analyzer to identify potential communication and performance bottlenecks. Alternatively, traces can be used for timeline visualization using, for example, the Vampir trace browser.

Summarization is particularly useful to get a first insight into the performance behavior of the inspected application as well as to optimize the measurement configuration for subsequent trace generation. If hardware counter metrics were requested, these are also included in the summary report. By contrast, tracing provides detailed insights into the dynamic application behavior.

Measurement overhead can be prohibitive for small, frequently-called routines. Therefore both Scalasca v1 and Score-P provide measurement filtering capabilities.

To enable filtering, the user has to prepare a filter file. Scalasca v1 supports a plain text file containing the names of functions to be excluded from measurement, whereas Score-P can filter based on region name *and* source file names. While the use of wildcards is allowed by both file formats, only Score-P supports white- and blacklisting – also in combination. The enhanced filtering mechanism in Score-P requires a more powerful filtering syntax which will be explained in the following paragraph.

Score-P's filter file can contain the following two sections:

- A section with rules for *region names*. This section must be enclosed by the keywords `SCOREP_REGION_NAMES_BEGIN` and `SCOREP_REGION_NAMES_END`.
- A section with rules for filtering all regions defined in specific *source files*. This section must be enclosed by the keywords `SCOREP_FILE_NAMES_BEGIN` and `SCOREP_FILE_NAMES_END`.

Within these sections, the user can specify an arbitrary number of include and exclude rules which are processed in sequential order. An exclude rule starts with the keyword `EXCLUDE` whereas include rules start with `INCLUDE`, both followed by one or several white-space separated file or region names. Bash-like wildcards can be used in file or/and region names.

By default, all files and regions are included. After all rules of the filter file have been evaluated, files and regions that were marked as excluded are filtered, i.e., excluded from measurement.

Besides the two filter sections, users may also use comments in the filter file. Comments start with the '#' character and are effective throughout the entire rest of the line. If a region name or source file name contains the comment character '#', the user must escape it with a backslash.

Let us consider the following filter file:

```
#Exclude all regions except bar and foo
SCOREP_REGION_NAMES_BEGIN
    EXCLUDE *
    INCLUDE foo bar
SCOREP_REGION_NAMES_END
#Exclude all files except *.c
SCOREP_FILE_NAMES_BEGIN
    EXCLUDE *
    INCLUDE *.c
SCOREP_FILE_NAMES_END
```

According to this filter file only `foo` and `bar` region names in *.c source files will be recorded by the measurement system.

To collect measurements for profiling or/and tracing, additional memory for internal data structures as well as for the trace data is required. Even more memory is needed if hardware counter measurements are requested. This additional memory in Scalasca v1 and Score-P constitutes the memory buffer. Although memory

buffers are used in both measurement systems, their handling changed notably. Scalasca v1 uses separate pre-allocated memory buffers for each thread whereas Score-P uses a pre-allocated memory pool consisting of a set of chunks on each process. Score-P's memory pool dynamically distributes memory chunks to threads that demand memory to store event data.[5] This approach manages memory in a more flexible but still efficient way and thus minimizes overhead due to run-time memory allocations.

If tracing is enabled, each thread will generate a trace file containing chronologically ordered local events. However, for applications running at scale the *one file per thread* approach might impose a serious I/O bottleneck. To address this issue Score-P can leverage the SIONlib I/O library. SIONlib improves file handling by transparently mapping logical thread-local files into a smaller number of physical files.

Once a measurement run terminates, the Scalasca trace analysis is automatically initiated to quantify wait states that cannot be determined with runtime summarization. The report generated by the trace analysis is similar to the summary report but includes additional communication and synchronization inefficiency metrics.

Table 3 shows a comparison of the most relevant options and environment variables[6] that can be passed to the two different versions of the `scalasca -analyze` or `scan` command that controls measurement collection and analysis. As can be seen, the command-line options remained unchanged. By contrast, the names of all environment variables have been changed in Score-P. Score-P provides more self-explanatory variable names prefixed with `SCOREP_` rather than with `EPIK_`. We believe that these names can be more easily memorized by users than their previous counterparts.

3.2.3 Scalasca Analysis Report Explorer

Once measurement and analysis are completed, analysis reports are produced in either CUBE3 (Scalasca v1) or CUBE4 (Scalasca v2) format, which can be interactively explored with the Cube GUI and processed by various Cube command-line utilities. Although the CUBE format has been changed considerably with the new generation of Scalasca, the Cube GUI and utilities provide the same functionality as in the previous version.

In the usual workflow, the `scalasca -examine` (or short `square`) convenience command will post-process intermediate analysis reports produced by the measurement and analysis to further derive additional metrics and construct a hierarchy of measured and derived metrics. The Cube browser will then present the final

[5]The size of the chunk, a so-called page, is configurable using the `SCOREP_PAGE_SIZE` environment variable. The other memory-related configuration variable is `SCOREP_TOTAL_MEMORY`, denoting the total amount of memory reserved by each process.

[6]For Score-P, the command `scorep-info config-vars` provides the list of all environment variables (and their default values) that are available for the current installation.

Table 3 Comparison of the `scalasca -analyze` (`scan`) command-line options and the most important, associated environment variables

Command switch	Environment variable [default value]		Description
	Scalasca v1	Scalasca v2	
-s	EPK_SUMMARY [1]	SCOREP_ENABLE_ PROFILING [1]	Enable runtime summarization
-t	EPK_TRACING [0]	SCOREP_ENABLE_ TRACING [0]	Enable trace collection and analysis
-e <dir_name>	EPK_TITLE	SCOREP_EXPERIMENT_ DIRECTORY	Experiment archive to generate and/or analyze
-f <filter_file>	EPK_FILTER	SCOREP_FILTERING_ FILE	File specifying measurement filter
	ESD_BUFFER_SIZE [100000], ELG_BUFFER_SIZE [10000000]	SCOREP_TOTAL_ MEMORY [16384000]	Total memory in bytes for the measurement system
	EPK_METRICS[a]	SCOREP_METRIC_ PAPI[b]	List of counter metrics
	ELG_SION_FILES [0]	SCOREP_TRACING_ MAX_PROCS_PER_ SION_FILE [1024]	Maximum number of processes that share one sion file

[a] metrics provided as a colon-separated list
[b] by default metrics provided as a comma-separated list (separator can be configured with *SCOREP_METRIC_PAPI_SEP* environment variable)

report. The command-line options supported by `scalasca -examine/square` did not change during the transition from Scalasca v1 to Scalasca v2. Therefore users can continue to work with already familiar commands and command-line options during this stage of the workflow.

4 New Features and Future Plans

Score-P, Scalasca and Cube are under active development. New features, also on user request, are developed, tested, and integrated into the main development branch. Bugs that emerge are fixed and performance is constantly evaluated and improved. In this section, we highlight recently added features and improvements, and outline our plans for future work.

4.1 Score-P

In collaboration with our Score-P development partners

- Technische Universität Dresden,
- Technische Universität München,

- RWTH Aachen University,
- German Research School for Simulation Sciences (Aachen), and
- University of Oregon

we provide a small number of feature releases each year. Each feature release is usually followed by one or more bugfix releases. We do not follow a fixed release schedule but ship a new version if one or more user-relevant features become available.

4.1.1 Score-P v1.3

The last feature release was Score-P v1.3, released August 2014. Significant new features in Score-P v1.3 include:

- Basic support for the K Computer and Fujitsu FX10 systems. Fujitsu systems are cross-compile systems using rarely encountered Fujitsu compilers. The porting effort was significant, but as Score-P is a joint infrastructure, it had to be done just once.[7]
- Support for instrumenting programs which use Symmetric Hierarchical Memory access (SHMEM) library calls for one-sided communication. SHMEM implementations from Cray, Open MPI, OpenSHMEM, and SGI are supported.
- Basic support for POSIX threads instrumentation. At the moment Score-P is capable of tracking creation and joining of POSIX threads as well as important synchronization routines.[8]
- Support for CUDA [19] versions 5.5 and 6.0.
- Improved event size estimation in `scorep-score` to adapt memory requirements for subsequent trace experiments.

[7] The Tofu network topology will be supported in a subsequent release.

[8] Currently supported POSIX threads routines are `pthread_create`, `pthread_join`, `pthread_mutex_init`, `pthread_mutex_destroy`, `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`, `pthread_cond_init`, `pthread_cond_destroy`, `pthread_cond_signal`, `pthread_cond_broadcast`, `pthread_cond_wait`, and `pthread_cond_timedwait`. The following thread management functions are currently not supported and will abort the program: `pthread_exit` and `pthread_cancel`. The usage of `pthread_detach` will cause the program to fail if the detached thread is still running after the end of the main routine. These limitations will be addressed in an upcoming version of Score-P. Note that currently every thread creation needs to be instrumented.

4.1.2 Upcoming Score-P v1.4

Score-P v1.4, likely to be released by the end of 2014, might include some of the features we and our project partners are currently working on. The prominent user-visible ones are:

- Basic OpenCL [26] support by intercepting OpenCL library routines using library wrapping.
- Intel Xeon Phi support for native and symmetric mode.
- GCC plugin as an alternative to the default compiler instrumentation, allowing for instrumentation-time filtering.

The next release will also include improved POSIX threads support, internal refactorings, and bugfixes.

4.1.3 Future Plans

There are a number of features we are working on that eventually will appear in a future release. Some of them are listed below:

- Experimental support for sampling as an alternative to instrumentation-based measurement, potentially combined with library wrapping and interposition, will give opportunity to collect measurements with lower overhead than with full instrumentation [12].
- Basic OpenACC [5] instrumentation using the OpenACC tools interface.
- A refactored OPARI2 allows to easily add source-to-source instrumentation directives, i.e., C/C++ pragmas and Fortran special comments, beyond the already supported OpenMP ones.
- Hardware and application-specific Cartesian topologies that were already supported by Scalasca v1 will be ported to Score-P and Scalasca v2.
- MPI-3 [17] support. MPI-3.0 is a major update to MPI adding over 100 new functions and extensions to collective and one-sided communication, which we expect to be used by scientific applications in the near future. We plan to add support for MPI-3 features in multiple steps.

  In the current release, MPI-3 functions are silently ignored by the measurement system, so first we will provide basic wrappers for all new MPI-3 functions. The next step is adding support for the newly introduced non-blocking collectives and neighborhood collectives. Non-blocking collectives are similar to non-blocking point-to-point routines, i.e., they have the same semantics as their blocking counterparts but can be used to overlap computation and communication. Neighborhood collectives on the other hand introduce a new semantic to collective operations, as only neighbors in a defined (Cartesian or distributed graph) topology communicate with each other. We will initially concentrate on Cartesian topologies and tackle graph topologies later. Concurrently, we are

working on support for the new RMA features to analyze applications using one-sided communication. In all cases profiling support will be implemented first. Tracing support for these new features may require new OTF2 event records and adaptations in Scalasca (for analysis) and Vampir (for visualization).

MPI-3 also offers new Fortran 2008 bindings, which will probably be used once Fortran 2008 features are exploited by more scientific applications. Currently, all MPI wrappers are implemented in C with Fortran-to-C conversion to support the Fortran bindings in MPI-2. We will need to implement new native Fortran wrappers to support the Fortran 2008 bindings in MPI-3.

Finally, we plan to support the MPI Tools interface (MPI_T). MPI_T exposes performance (and control) variables internal to the MPI library to external tools to help the advanced user in fine-tuning the MPI usage in their application (or control the MPI library accordingly). Score-P will primarily focus on performance variables, though also control variables might be exploited by tools like Periscope.

- New programming models or extensions of existing ones:
  - Support of relevant OpenMP 4.0 [21] features, e.g., target and cancellation directives, as well as support for the OpenMP Tools Application Programming Interface for Performance Analysis (OMPT) [6].
  - Support for OmpSs [4], an extension to OpenMP to support asynchronous parallelism with data dependencies.
  - Support for Qt, ACE, and TBB threading models using Intel's dynamic binary instrumentation tool Pin [16].
  - Enhanced tasking support, in particular for MTAPI [11], using Pin as well as library wrapping.
- Basic support for the Windows platform, however, not targeting distributed-memory applications. The preferred instrumentation method will be Pin.
- New substrates – substrates are the event consumers – besides profiling and tracing. We work on a generic substrate layer that allows arbitrary processing of event data. One could think of online tuning of runtime parameters or streaming event data to reduce memory demands.
- Improved support for threading. As of Score-P v1.3, the user can either analyze programs using OpenMP *or* POSIX threads. Additionally all thread creations need to be instrumented. This is particularly challenging if the thread creation takes place in a shared library. A future version of Score-P will allow the combination of OpenMP and POSIX threading. Additionally, events from uninstrumented threads will be handled gracefully.

4.2 Scalasca v2

The first version of the Scalasca v2 series based on Score-P v1.2, OTF2 v1.2, and CUBE v4.2 was released in August 2013. Besides a new build system based on

GNU autotools, a significant amount of code refactoring, and support for the new data formats, this release included only minor functional changes and fixes for a number of bugs uncovered during the refactoring process. This initial release then served as a basis for all subsequent developments.

4.2.1 Scalasca v2.1

Scalasca v2.1, the first feature release based on the new infrastructure, was released in August 2014 and included the following new features:

- Support for the K Computer and Fujitsu FX10 systems, to bring Scalasca's platform support in line with Score-P v1.3.
- Improved detection of Late Receiver pattern instances. Previously, Late Receiver instances in non-blocking point-to-point MPI communication could only be detected in operations finalizing a single request (e.g., `MPI_Wait`). The enhanced detection algorithm now also supports multi-request finalization calls, such as `MPI_Waitall`.
- Critical-path analysis [3] for MPI- and/or OpenMP-based applications. The critical-path analysis determines the call-path profile of the application's critical path, highlighting the call paths for which optimization will prove worthwhile and thus guiding optimization efforts more precisely.

4.2.2 Scalasca v2.2

The upcoming Scalasca v2.2 release, planned for early 2015, is likely to include a number of additional features which will be briefly described below.

- Intel Xeon Phi support for native and symmetric mode.
- Basic OpenMP task support. So far, Scalasca's automatic trace analyzer has been unable to handle traces including events related to OpenMP tasking, thus not even allowing an MPI-only analysis for applications using OpenMP tasks in their computational phases. We therefore extend the analysis infrastructure to support tasks, handle (potentially) multiple call stacks, and track the asynchroneous task behavior. However, this initial tasking support will not include any task-specific analyses.
- Basic support for create/wait threading models (e.g., POSIX threads). We intend to implement some basic analyses that should work in many cases, however, the current OpenMP-based replay analysis approach used by Scalasca's trace analyzer may impose some limitations.
- In multi-threaded programs, lock contention can be a serious performance bottleneck. We are therefore investigating a more advanced analysis to identify waiting time due to lock contention, supporting both fork/join threading models (e.g., OpenMP) as well as create/wait threading models (e.g., POSIX threads).

- Finally, we plan to integrate an updated, production-ready version of the delay analysis research prototype developed for Scalasca v1 [2]. The delay analysis is capable to identify the root causes of wait states that materialize at communication or synchronization operations (i.e., the delays/imbalances causing them), and to quantify their overall costs.

4.2.3 Future Plans

In future releases, we plan to further improve the analysis capabilities of Scalasca's automatic trace analyzer in various ways:

- MPI-3 support. Currently, Scalasca's trace analyzer focuses exclusively on communication and synchronizition operations as defined by MPI-2.2. However, once MPI-3 support has been added to the Score-P measurement system, the trace analysis will be enhanced to also support MPI-3 features. In this context, non-blocking and neighborhood collective operations are of particular interest. While extending the existing trace analysis algorithms to handle non-blocking collectives should be fairly straightforward, analyzing neighborhood collectives requires enhanced wait-state detection algorithms which take the neighborhood information into account. Another area of interest is the extension of the existing Scalasca RMA analysis to support the new RMA functionality of MPI-3.
- Improved threading support. The support for create/wait thread models planned for Scalasca v2.2 is very limited and needs to be improved in various ways. For example, the timestamp correction algorithm implemented in the Scalasca trace analyzer currently only supports fork/join threading models. Thus, correcting the timestamps of an application trace using, for example, MPI and POSIX threads in combination may lead to unexpected clock condition violations due to corrections applied on the thread executing the MPI calls but not on the other threads. In addition, the critical-path and delay analyses have to be extended accordingly to support all threading models handled by the wait-state search carried out by the automatic trace analyzer.
- Another topic of active research are analyses related to task-based programming models. However, this is still preliminary work that needs thorough investigation.

4.3 Cube v4

Cube v4 and its predecessor Cube v3 both provide a graphical user interface to visually examine profile and trace analysis reports, as well as a set of libraries and tools to read, write, and manipulate such reports. Although, CUBE4 and CUBE3 reports have a very similar structure in the metric tree, call tree, and system tree, a completely new format for storing the measured data was introduced. While Cube v3 used a pure XML file format, Cube v4 uses a binary format to store the

experiment data, whereas XML is only used to store metadata information. The new format provides both greater scalability and flexibility. In addition, Cube v4 introduced the following new extensions:

- A Java reader library allows TAU to import data from CUBE4 files into PerfExplorer [22] and opens the opportunity to use CUBE4 files on Android in the future.
- A flexible remapping based on specification files enables the fine-grained control of metrics and metric hierarchies that are created during remapping (e.g., for POSIX threads, CUDA, etc.).
- Support for new platforms, e.g., Windows, K Computer, and Fujitsu FX10.

In addition, we want to highlight two features we believe will benefit many users of Cube v4: derived metrics and the visualization of high-dimensional Cartesian topologies.

4.3.1 Derived Metrics in Cube

Derived metrics are a very powerful and flexible tool allowing users to define and calculate new metrics directly within Cube. While Cube's predefined metrics are stored in the analysis report, the values of derived metrics are calculated on-the-fly when necessary according to user-defined arithmetic expressions formulated using a domain-specific language called *CubePL*.[9]

A first version of *CubePL* was introduced with Cube v4.1, which had very limited functionality and allowed only simple expressions for derived metrics. The latest implementation of *CubePL* available in Cube v4.2.3 supports different kinds of derived metrics which will be explained in the following example. This version features an improved memory layout and added an extended set of predefined variables. Finally, it also provides better support for string and numeric variables, thereby allowing the user to define very sophisticated metrics.

The following simple example introduces the basic concept behind derived metrics and explains why different kinds of derived metrics are needed. Let us assume that an application with the following call tree was executed:

```
main
 - foo
 - bar
```

where *main*, *foo* and *bar* are function calls, with *foo* and *bar* being called from *main*. Further, assume that the number of floating-point operations (`FLOP`) has been measured for every call-path, as well as the respective execution time. Note that it is important to distinguish between inclusive and exclusive values of metrics. An *inclusive* metric value correponds to the execution of the function including all

[9] *CubePL* stands for **Cube P**rocessing **L**anguage.

function calls inside it (e.g., the inclusive metric value of *main* is the value measured for itself plus the values of *foo* and *bar*). An *exclusive* metric value only corresponds to the execution of the function itself excluding the values of functions called from it. In this example exclusive values are stored for every call-path.

To calculate the floating-point operations per second (`FLOPS`) as a derived metric for every call path, the naïve approach would be to define a new metric `FLOPS` and calculate its values for every call path using the formula $\frac{FLOP_c}{time_c}$, where c is either `main`, `foo`, or `bar`, and store the resulting values as a data metric within the corresponding Cube profile.

With this approach, however, a problem arises when the user would like to get an inclusive value of the metric `FLOPS` for the `main` call path. In this case, the Cube library would sum up all values of the metric `FLOPS` for every region and therefore deliver the result of the expression

$$\frac{FLOP_{main}}{time_{main}} + \frac{FLOP_{foo}}{time_{foo}} + \frac{FLOP_{bar}}{time_{bar}}$$

However, this expression is the *sum of FLOPS of every region* instead of the intended *FLOPS of* `main`. So, the correct calculation is

$$\frac{FLOP_{main} + FLOP_{foo} + FLOP_{bar}}{time_{main} + time_{foo} + time_{bar}}$$

which gives the desired floating-point operations per second for `main`.

As can be seen, the calculation of the $\frac{FLOP}{time}$ ratio should be performed as a last step – after the aggregation of the corresponding metric values `FLOP` and `time` is done. We call this a *postderived* metric. In contrast, the case where the derived metric should be calculated before the aggregation of the involved metrics is called a *prederived* metric.

The corresponding *CubePL* expression for the postderived FLOPS metric reads:

```
metric::flop()/metric::time()
```

As another slightly more complicated example of a postderived metric, consider the *CubePL* expression for the average time per visit of a function:

```
metric::time(i)/metric::visits(e)
```

Here the inclusive value of the time metric has to be used to include the time spent in child functions. However, for the visits metric, the exclusive value is necessary to only account for the calls to the function itself.

Generally, we advise users to start with the examples shipped with Cube and use those as a starting point to develop their own derived metrics.

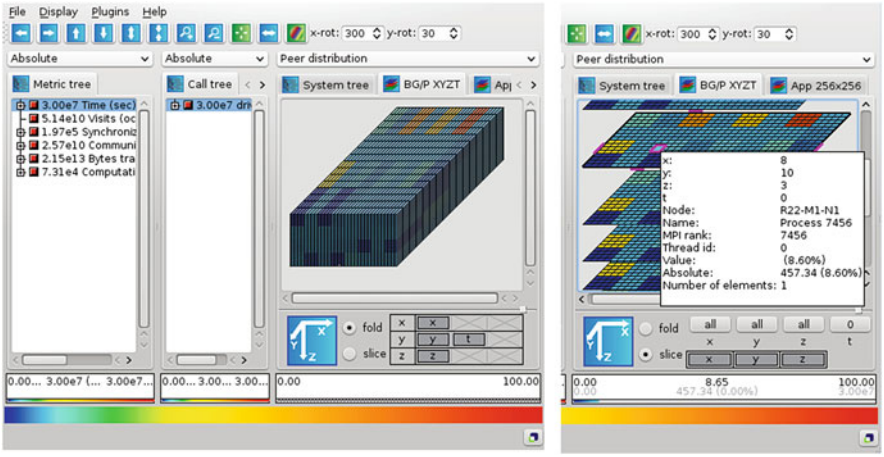

Fig. 2 Cube screenshot showing folding and slicing of multi-dimensional Cartesian topologies

4.3.2 Visualization of High-Dimensional Topologies in the Cube GUI

Many parallel applications define *virtual topologies* to express neighborhood relationships between processes, e.g., based on the chosen domain decomposition. Often, such virtual topologies are specified as multi-dimensional Cartesian grids. Another type of topologies are *physical topologies* reflecting the structure of the hardware the application run on. A typical three-dimensional physical topology is given by the (hardware) nodes in the first dimension, and the arrangement of cores/processors on nodes in further two dimensions. Further, some machines exhibit specific network layouts, like the IBM Blue Gene/Q 5-D torus network.

Until version 3, Cube was only able to handle Cartesian topologies up to three dimensions (where the folding of the Blue Gene tori into 3 dimensions was hard-coded). This situation was significantly improved in Cube v4. The Cube display now supports multi-dimensional Cartesian grids, where grids with higher dimensionality can be sliced or folded down to two or three dimensions for presentation – where the details of slicing and folding can be controlled by the user. An example of a visualization of a four-dimensional topology[10] is shown on Fig. 2.

4.3.3 Future Plans

Cube v4 is under continuous development, with two major feature additions coming up in the near future:

[10]Note that the topology toolbar is only enabled when a topology is available to be displayed.

- Plugin interface: We are in the process of designing an open plugin interface for the Cube GUI, which will allow users to develop problem-specific analyses based on CUBE4 data. We will provide various example plugins for users and are planning to establish an open repository for third-party plugins.
- Another major milestone is the transition of the Cube GUI from a monolithic architecture to a client/server architecture. Here, a client running on, e.g., the user's desktop machine can connect to a server started on the HPC system where the experiments have been collected. This will remove the necessity to transfer large Cube files between machines or to remotely execute the Cube GUI using X11 forwarding, and should also lead to increased scalability.

5 Conclusion

Like any other actively used software framework, the Scalasca parallel performance analysis toolset is constantly under development. Besides fixing bugs uncovered by our rigorous testing or reported by users, we also develop new features – often inspired by feedback from our active user community. In addition, the HPC landscape is also constantly changing due to the introduction of new computer systems and components, which in turn require new versions of parallel programming paradigms or sometimes even trigger the invention of new paradigms requiring further changes, additions, and adaptations in all performance tools. To better cope with this situation, the Scalasca v2 series is now based on the community-developed performance measurement and runtime infrastructure Score-P.

This change required significant redesign and implementation efforts of the Scalasca tool components as outlined in this paper. However, we believe that we were successful in minimizing the user-visible changes to the absolute necessary. After one year of deployment of the new Score-P based version of Scalasca, we see already a much quicker adaption of the infrastructure to new architectures and paradigms, with much more to come in the short term. Sharing development efforts for the parallel program instrumentation and measurement components with other tool developers also allowed us to provide better documentation as well as to improve the development and testing processes, while at the same time freeing some highly-needed resources for new and improved analysis features.

Acknowledgements The authors would like to use this opportunity to thank the HPC community in general and the users of our tools in particular for their feedback, feature requests, and bug reports. Without them, our tools would not be as good as they are now. We also want to thank the entire Scalasca and Score-P development teams for many fruitful discussions, insights, and sharing their experiences.

References

1. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: an online-based distributed performance analysis tool. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 1–16. Springer, Berlin/Heidelberg (2010)
2. Böhme, D., Geimer, M., Wolf, F., Arnold, L.: Identifying the root causes of wait states in large-scale parallel applications. In: Proceedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, pp. 90–100. IEEE Computer Society (2010)
3. Böhme, D., de Supinski, B.R., Geimer, M., Schulz, M., Wolf, F.: Scalable critical-path based performance analysis. In: Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, pp. 1330–1340. IEEE Computer Society (2012)
4. Bueno, J., Planas, J., Duran, A., Badia, R., Martorell, X., Ayguade, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: Proceedings of the 26th IEEE International Parallel Distributed Processing Symposium (IPDPS), Shanghai, pp. 557–568. IEEE Computer Society (2012)
5. CAPS, CRAY, NVIDIA, PGI: The OpenACC application programming interface. http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf (2013)
6. Eichenberger, A.E., Mellor-Crummey, J.M., Schulz, M., Wong, M., Copty, N., DelSignore, J., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: OpenMP tools application programming interfaces for performance analysis. In: Proceedings of the 9th International Workshop on OpenMP (IWOMP), Canberra. LNCS, vol. 8122, pp. 171–185. Springer, Berlin/Heidelberg (2013)
7. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open trace format 2 – the next generation of scalable trace formats and support libraries. In: Proceedings of the International Conference on Parallel Computing (ParCo), Ghent. Advances in Parallel Computing, vol. 22, pp. 481–490. IOS Press (2012)
8. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proceedings of ACM/IEEE SC09 Conference, Portland (2009)
9. Geimer, M., Shende, S.S., Malony, A.D., Wolf, F.: A generic and configurable source-code instrumentation component. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) Proceedings of the International Conference on Computational Science (ICCS), Baton Rouge. Lecture Notes in Computer Science, vol. 5545, pp. 696–705. Springer (2009)
10. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The *Scalasca* performance toolset architecture. Concurr. Comput.: Pract. Exp. **22**(6), 702–719 (2010)
11. Gleim, U., Levy, M.: MTAPI: parallel programming for embedded multicore systems. http://www.multicore-association.org/pdf/MTAPI_Overview_2013.pdf (2013)
12. Ilsche, T., et al.: Combining instrumentation and sampling for trace-based application performance analysis. In: Proceedings of 8th Parallel Tools Workshop, Stuttgart. Springer (To appear)
13. Jiang, J., Philippen, P., Knobloch, M., Mohr, B.: Performance measurement and analysis of transactional memory and speculative execution on IBM Blue Gene/Q. In: Proceedings of the 20th Euro-Par Conference, Porto. Lecture Notes in Computer Science, vol. 8632, pp. 26–37. Springer (2014)
14. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis toolset. In: Tools for High Performance Computing (Proceedings of the 2nd Parallel Tools Workshop, July 2008, Stuttgart), pp. 139–155. Springer (2008)
15. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P – a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proceedings of 5th Parallel Tools Workshop, Dresden, pp. 79–91. Springer (2012)

16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. **40**(6), 190–200 (2005)
17. Massage Passing Interface Forum: MPI: a message-passing interface standard version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (2012)
18. Mohr, B., Malony, A.D., Hoppe, H.C., Schlimbach, F., Haab, G., Hoeflinger, J., Shah, S.: A performance monitoring interface for OpenMP. In: Proceedings of Fourth European Workshop on OpenMP (EWOMP), Rome (2002)
19. NVIDIA: CUDA toolkit documentation. http://docs.nvidia.com/cuda/ (2014)
20. OPARI2 web page. http://www.vi-hps.org/tools/opari2.html (2014)
21. OpenMP Architecture Review Board: The OpenMP API specification for parallel programming, version 4.0. http://openmp.org/wp/openmp-specifications/ (2013)
22. PerfExplorer web page. http://www.cs.uoregon.edu/research/tau/docs/perfexplorer/ (2014)
23. Scalasca Performance Analysis Toolset web page. http://www.scalasca.org (2014)
24. Score-P User Manual. Available as part of the Score-P installation or online at http://www.score-p.org (2014)
25. Shende, S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
26. The Khronos Group: OpenCL 2.0 API specification. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf (2014)
27. Vampir web page. http://www.vampir.eu/ (2014)
28. Zhukov, I., Wylie, B.J.N.: Assessing measurement and analysis performance and scalability of Scalasca 2.0. In: Proceedings of the Euro-Par 2013: Parallel Processing Workshops, Aachen. LNCS, vol. 8374, pp. 627–636. Springer (2014)