Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications

Felix Wolf[1,2], Brian J. N. Wylie[1], Erika Ábrahám[1], Daniel Becker[1,2], Wolfgang Frings[1], Karl Fürlinger[3], Markus Geimer[1], Marc-André Hermanns[1], Bernd Mohr[1], Shirley Moore[3], Matthias Pfeifer[1,2], and Zoltán Szebenyi[1,2]

Abstract SCALASCA is a performance toolset that has been specifically designed to analyze parallel application behavior on large-scale systems, but is also well-suited for small- and medium-scale HPC platforms. SCALASCA offers an incremental performance-analysis process that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature of SCALASCA is its ability to identify wait states even for very large processor counts. The current version supports the MPI, OpenMP and hybrid programming constructs most widely used in highly-scalable HPC applications.

1 Introduction

Supercomputing is a key technology pillar of modern science and engineering, indispensable to solve critical problems of high complexity. World-wide efforts to build machines with performance levels in the petaflops range acknowledge that the requirements of many key applications can only be met by the most advanced custom-designed large-scale computer systems. However, as a prerequisite for their productive use, the HPC community needs powerful and robust performance-analysis tools that make the optimization of parallel applications both more effective and more efficient. Such tools not only help improve the scalability characteristics of scientific codes and thus expand their potential, but also allow domain experts to

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany
{f.wolf, b.wylie, e.abraham, d.becker, w.frings, m.geimer, m.a.hermanns, b.mohr, m.pfeifer, z.szebenyi}@fz-juelich.de

[2] Department of Computer Science and Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen University, Germany

[3] Innovative Computing Laboratory, University of Tennessee, USA
{karl, shirley}@cs.utk.edu

concentrate on the science underneath rather than to spend a major fraction of their time tuning their application for a particular machine.

As the current trend in microprocessor development continues, this need will become even stronger in the future. Facing increasing power dissipation and with little instruction-level parallelism left to exploit, computer architects are realizing further performance gains by using larger numbers of moderately fast processor cores rather than by further increasing the speed of uni-processors. As a consequence, supercomputer applications are being required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. With an exponentially rising number of cores, the often substantial gap between peak performance and the performance actually sustained by production codes [6] is expected to widen even further. Finally, increased concurrency levels place higher scalability demands not only on applications but also on parallel programming tools [10]. When applied to larger numbers of processes, familiar tools often cease to work satisfactorily (e.g., due to escalating memory requirements, failing displays, or limited I/O bandwidth).

Developed at the Jülich Supercomputing Centre in cooperation with the University of Tennessee, SCALASCA is a performance-analysis toolset that has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well-suited for small- and medium-scale HPC platforms. SCALASCA supports an incremental performance-analysis process that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature of SCALASCA is its ability to identify wait states that occur, for example, as a result of unevenly distributed workloads. Especially when trying to scale communication-intensive applications to large processor counts, such wait states can present severe challenges to achieving good performance. Compared to its predecessor KOJAK [11], SCALASCA can detect such wait states even in very large configurations of processes using a novel parallel trace-analysis scheme [3].

In this article, we give an overview of SCALASCA and show its capabilities for diagnosing performance problems in large-scale parallel applications. First, we review the SCALASCA analysis process and discuss basic usage. After presenting the SCALASCA instrumentation and measurement systems in Section 3, Section 4 explains how its trace analysis can efficiently detect wait states in communication and synchronization operations even in very large configurations of processes, before we demonstrate how execution performance analysis reports can be interactively explored in Section 5. Finally, in Section 6, we outline our development goals for the coming years.

2 Overview

The current version of SCALASCA supports measurement and analysis of the MPI, OpenMP and hybrid programming constructs most widely used in highly-scalable HPC applications written in C/C++ and Fortran on a wide range of current HPC

platforms. Usage is primarily via the `scalasca` command with appropriate action flags, and is identical for a 64-way OpenMP application on a single UltraSPARC-T2 processor or a 64k hybrid OpenMP/MPI application on a Blue Gene/P.

Figure 1 shows the basic analysis workflow supported by SCALASCA. Before any performance data can be collected, the target application must be *instrumented*, that is, it must be modified to record performance-relevant events whenever they occur. On most systems, this can be done completely automatically using compiler support; on other systems a mix of manual and automatic instrumentation mechanisms is offered. When running the instrumented code on the parallel machine, the user can choose between generating a summary report (aka profile) with aggregate performance metrics for individual function call paths, or generating event traces recording individual runtime events from which a profile or time-line visualization can later be produced. The first option is useful to obtain an overview of the performance behavior and also to optimize the instrumentation for later trace generation. Since traces tend to become very large, this step is usually recommended before choosing the second option. When tracing is enabled, each process generates a trace file containing records for all its process-local events. After program termination, SCALASCA loads the trace files into main memory and analyzes them in parallel using as many CPUs as have been used for the target application itself. During the analysis, SCALASCA searches for characteristic patterns indicating wait states and related performance properties, classifies detected instances by category and quantifies their significance. The result is a pattern-analysis report similar in structure to the summary report but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and pattern reports contain performance metrics for every function call-path and system resource which can be interactively explored in a graphical report explorer (Fig. 3). As an alternative to the automatic search, the event traces can be converted and investigated using third-party trace browsers such as Paraver [4, 7] or VAMPIR [5, 9], taking advantage of their powerful time-line visualizations and rich statistical functionality.

3 Instrumentation and Measurement

SCALASCA offers analyses based on two different types of performance data: (i) aggregated statistical summaries and (ii) event traces. By showing which process consumes how much time in which call-path, the summary report provides a useful overview of an application's performance behavior. Because it aggregates the collected metrics across the entire execution, the amount of data is largely independent of the program duration. This is why runtime summarization is the first choice for very long-running programs working on realistic input data sets and models. The summary metrics measured with SCALASCA include wall-clock time, the number of times a call-path has been visited, message counts, bytes transferred, and a rich choice of hardware counters available via the PAPI library [2].

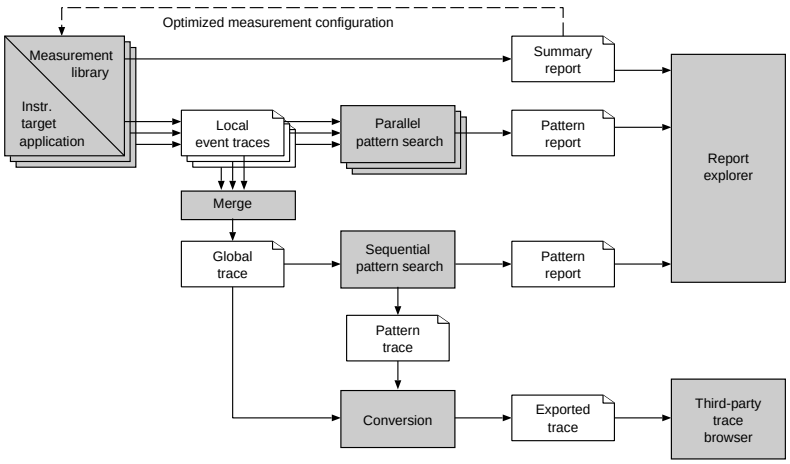

Fig. 1 SCALASCA's performance analysis workflow

In contrast, event traces allow the in-depth study of parallel program behavior. Tracing is especially effective for observing the interactions between different processes or threads that occur during communication or synchronization operations and to analyze the way concurrent activities influence each other's performance. When an application is traced, SCALASCA records individual performance-relevant events with timestamps and writes them to a trace file (one per process) to be analyzed in a subsequent step.

To effectively monitor program execution, SCALASCA intercepts runtime events critical to communication and computation activities. These events include entering and leaving functions or other code regions as well as sending and receiving point-to-point messages or participation in collective communication. Whereas the communication-related event types are crucial to study the interactions among different processes and to identify wait states, function entries and exits are needed to understand the computational requirements and the context in which the most demanding communication operations occur.

The application must be instrumented to provide notification of these events during measurement, using function calls inserted at specific important points ("events") which call into the SCALASCA measurement library. Just linking the application with the measurement library already ensures that all events related to MPI operations are properly captured. For OpenMP, a source preprocessor is used which automatically instruments directives and pragmas for parallel regions, etc., and many compilers are capable of adding instrumentation to every function or routine entry and exit. Finally, programmers can manually add their own custom instrumentation annotations in the source code for important regions (such as phases or loops, or functions when this is not done automatically by the compiler): these annotations are in the form of pragmas or macros which are ignored when instrumentation is not configured.

Instrumentation configuration and processing of source files are achieved by prefixing the SCALASCA instrumenter to selected compilation commands and the final link command, without requiring other changes to optimization levels or the build process.

```
# scalasca -instrument <compile-or-link-command>
% scalasca -instrument mpicc -c foo.c
% scalasca -instrument f90 -o bar -OpenMP bar.F
% scalasca -instrument mpif90 -o foobar -OpenMP foo.o bar.F
```

A simple means to be able to conveniently instrument an entire application, is to add a 'preparer' prefix to compile and link commands in its Makefile(s), which is undefined by default and results in a regular uninstrumented build, or when the preparer is set to the SCALASCA instrumenter then an instrumented build is produced.

```
PREP =
MPICC = $(PREP) mpicc
MPIFC = $(PREP) mpif90
foobar: bar.F foo.o
       $(MPIFC) -o $@ -OpenMP foo.o bar.F

% make PREP="scalasca -instrument"
```

The SCALASCA measurement system that gets linked with instrumented application executables can be configured to allow runtime summaries and/or event traces to be collected, along with optional hardware counter metrics. A unique experiment archive is created to contain all of the measurement and analysis artifacts, including configuration information, log files and analysis reports. When event traces are collected, they are also stored in the experiment archive to avoid accidental corruption by simultaneous or subsequent measurements.

Measurements are collected and analyzed under the control of a nexus which automatically configures the parallel trace analyzer with the same number of processes as used for measurement. This allows SCALASCA analysis to be specified as a command prefixed to the application execution command-line, whether part of a batch script or run interactively.

```
# scalasca -analyze <application-launch-command>
% scalasca -analyze mpiexec -np 65536 foo arglist
Scalasca runtime summarization experiment ./epik_foo_65536_sum
% OMP_NUM_THREADS=64 scalasca -analyze bar arglist
Scalasca runtime summarization experiment ./epik_bar_Ox64_sum %
OMP_NUM_THREADS=4 scalasca -analyze mpiexec -np 512 foobar Scalasca
runtime summarization experiment ./epik_foobar_512x4_sum
```

Although collection of runtime summarization experiments is the default, addition of the **-t** flag configures trace collection and automatic analysis (without the need for instrumentation re-configuration).

```
% OMP_NUM_THREADS=4 scalasca -analyze -t mpiexec -np 512 foobar
Scalasca trace analysis experiment ./epik_foobar_512x4_trace
```

Instrumented functions which are executed frequently, while only performing a small amount of work each time they are called, have an undesirable impact on measurement. The overhead of measurement for such functions is large compared to the execution time of the (uninstrumented) function, resulting in measurement dilation, while recording such events requires significant space and analysis takes longer with relatively little improvement in quality. This is especially important for event traces whose size is proportional to the total number of events recorded. For this reason, SCALASCA offers various mechanisms to exclude certain functions from measurement. Before writing a trace file, the instrumentation should therefore be optimized based on a visit-count summary obtained during an earlier run.

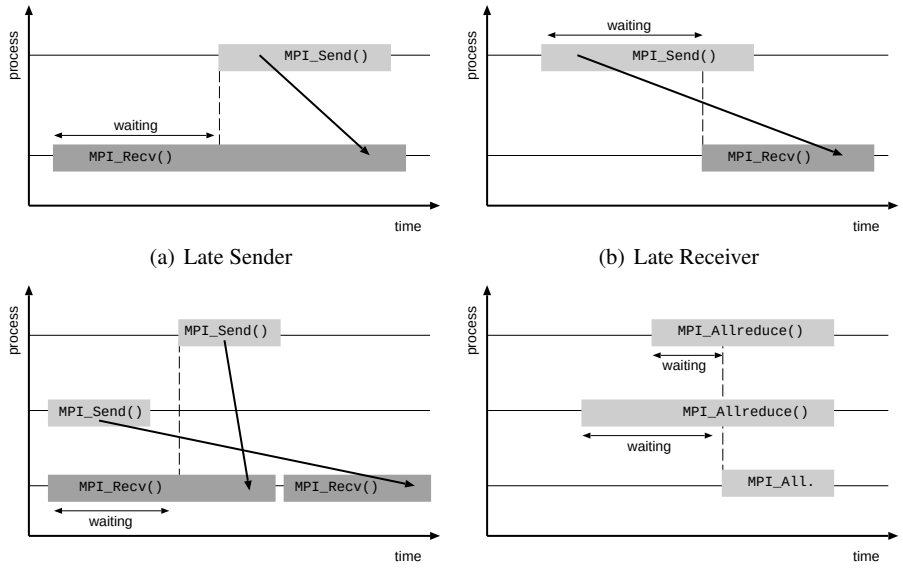

Fig. 2 Examples for patterns of inefficient behavior. Note that the combination of MPI functions used in each of these examples represents just one possible case

4 Trace Analysis

In message-passing applications, processes often require access to data provided by remote processes, making the progress of a receiving process dependent upon the progress of a sending process. If a rendezvous protocol is used, this relationship also applies in the opposite direction. Collective synchronization is similar in that its completion requires each participating process to have reached a certain point. As a consequence, a significant fraction of the time spent in communication and

synchronization routines can often be attributed to wait states that occur when processes fail to reach implicit or explicit synchronization points in a timely manner, for example, as a result of an unevenly distributed workload. Especially when trying to scale communication-intensive applications to large process counts, such wait states can present severe challenges to achieving good performance. As a first step in reducing the impact of wait states, SCALASCA provides a diagnostic method that allows their localization, classification, and quantification. Because wait states cause temporal displacements between program events occurring on different processes, their identification can be accomplished by searching event traces for characteristic patterns. A subset of the patterns supported by SCALASCA is depicted in Fig. 2.

As the first example of a typical wait state, consider the so-called *Late Sender* pattern (Fig. 2(a)). Here, a receive operation is entered by one process before the corresponding send operation has been started by the other. The time lost waiting due to this situation is at least the time difference between the two function invocations. In contrast, the *Late Receiver* pattern (Fig. 2(b)) describes the inverse situation, where a sender is blocked while waiting for the receiver when a rendezvous protocol is used (e.g., to transfer a large message). The *Late Sender / Wrong Order* pattern (Fig. 2(c)) is more complex than the previous two. Here, a receiver waits for a message, although an earlier message is ready to be received by the same destination process (i.e., message receipt in wrong order). Finally, the *Wait at N×N* pattern (Fig. 2(d)) quantifies the waiting time due to the inherent synchronization in collective n-to-n operations, such as `MPI_Allreduce`.

To accomplish the search is a scalable way, SCALASCA exploits both distributed memory and parallel processing capabilities available on the target system. Instead of sequentially analyzing a single global trace file, as done by its predecessor tool KOJAK, SCALASCA analyzes separate process-local trace files in parallel by *replaying* the original communication on as many CPUs as have been used to execute the target application itself. During the search process, SCALASCA classifies detected pattern instances by category and quantifies their significance for every program phase and system resource involved. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, SCALASCA has completed pattern searches even at the previously intractable scale of over 22,000 processes. Additionally, to allow accurate trace analyses on systems without globally synchronized clocks such as most PC clusters the trace analyzer provides the ability to synchronize inaccurate timestamps postmortem using the same scalable replay mechanism [1].

OpenMP Support and Pattern Traces

In addition to the scalable MPI trace analysis, sequential trace analysis (Fig. 1) is also provided for OpenMP and MPI one-sided RMA operations. This sequential analysis is currently the default for pure OpenMP measurements, and can be specified for an augmented analysis of MPI and hybrid measurements when desired. For large measurements, however, the additional storage space and serial analysis time required

can be prohibitive, unless very targeted instrumentation is configured or the problem size is reduced (e.g., to only a few timesteps or iterations). Other options include visual analysis using third-party trace browsers, such as Paraver and VAMPIR and the generation of pattern traces.

The first step to access these features consists of merging the local trace files generated by SCALASCA into a single global trace file. The resulting global trace file can then be searched for MPI and/or OpenMP patterns or converted and loaded into Paraver or VAMPIR. A third option was motivated by the fact that the pattern search method accumulates the severities of all of the pattern instances found to inform about the overall performance penalty. However, the temporal and spatial relationships between individual pattern instances are lost, although these relationships can be essential to understand the detailed circumstances of a performance problem. These relationships can now be retained by writing a second event trace with events delimiting individual pattern occurrences. Guided by the summary pattern report, this synthetic pattern trace can be interactively analyzed leveraging the powerful functionality of the aforementioned trace browsers.

5 Understanding Performance Behavior

After SCALASCA analysis is completed, the experiment archive may contain a summary report generated immediately at measurement completion and/or trace-analysis report(s) generated after searching event traces. These profiles have the same structure and can be viewed and manipulated using the same set of commands.

```
# scalasca -examine <experiment-archive>
% scalasca -examine epik_foobar_512x4_trace
```

Whereas a summary report includes metrics, such as time, visit counts, message statistics or hardware counters, a trace-analysis report also accounts for the times lost in different wait states. Both types of reports are stored as a three-dimensional array with the dimensions metric, call path, and system resource (e.g., process or thread). Because of the cubic structure, the corresponding file format is called CUBE. For every metric included, a CUBE report stores the aggregated value for each combination of call-path and process or thread. Motivated by the need to represent performance behavior on different levels of granularity as well as to express natural hierarchical relationships among metrics, program, or system resources, each dimension is organized in a hierarchy.

The SCALASCA analysis report explorer (Fig. 3) provides the ability to interactively browse through this three-dimensional performance data space in a convenient way. Its design emphasizes simplicity by combining a small number of orthogonal features with a limited set of user actions. Each dimension of the data space (metric, call-path, and system resource) can be shown using tree displays and allows the user to interactively explore the values of all the data points. Since the data space is large, views representing only a subspace can be selected and combined

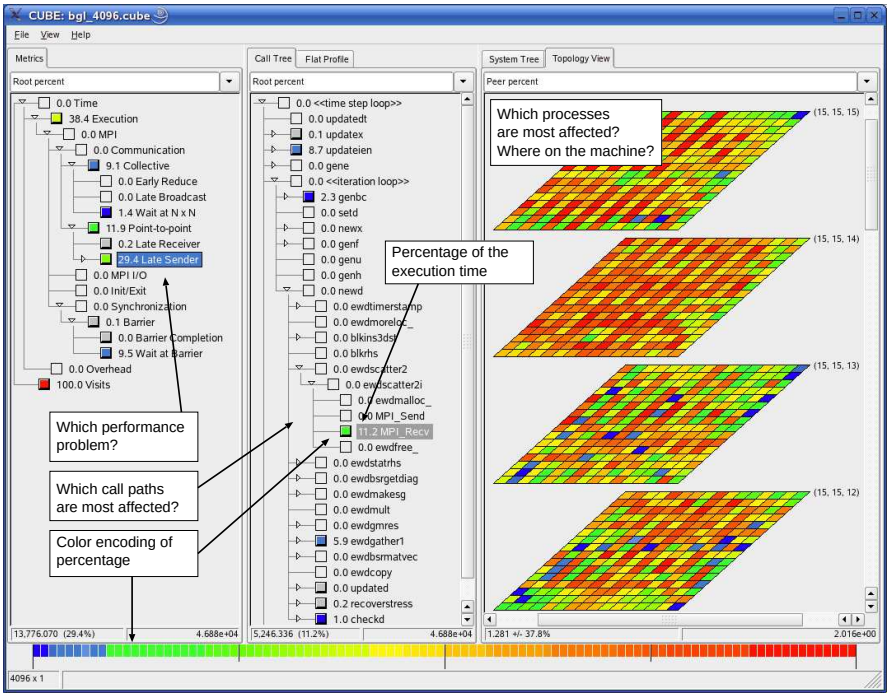

Fig. 3 The trace analysis report displayed in the report explorer indicates that 29.4% of the total time in the annotated region `<<timestep loop>>` is spent waiting due to *Late Sender* situations (left pane). The call tree (middle pane) shows that more than one-third of the waiting time is concentrated in one call-path, with its waiting time unevenly distributed across the visible section of the machine topology (right pane)

with aggregation mechanisms that control the level of detail. Two types of actions can be performed: selecting a node or expanding/collapsing a node. Whereas the first action defines a "slice" or "column" of the data space, the latter exposes/hides sub-hierarchies of the different dimensions. To help identify combinations with a high value more quickly, all values are not only shown numerically but also color-coded. To facilitate the analysis of runs on many processors, the explorer provides a scalable two- or three-dimensional Cartesian grid display to visualize physical or virtual process topologies which were recorded with measurements. The topological display is offered as an alternative to a standard tree hierarchy of machine, compute nodes, processes and threads.

With a set of command-line tools [8], CUBE reports can be combined or manipulated to allow comparisons or aggregations of different reports or to focus the analysis on specific parts of a report. Specifically, multiple reports can be averaged or merged, the difference between two reports calculated, or a new report generated after pruning specified call-trees and/or specifying a call-tree node as a new root. The latter can be particularly useful for eliminating uninteresting phases (e.g., initialization) and focusing the analysis on a selected part of the execution. These

utilities each generate new CUBE-format reports as output that can be loaded into the explorer like the original reports that were used as input.

6 Outlook

Future enhancements will aim at both further improving the functionality and scalability of the SCALASCA toolset. Whereas automatic MPI analysis has been demonstrated at very large scales, runtime summaries currently only include measurements for the OpenMP master thread, and OpenMP trace analysis is currently done serially: for hybrid applications, scalable MPI trace analysis is the default and serial OpenMP analysis is offered as an additional option. Most standard-conforming HPC applications should be measurable, however, there is no recording or analysis of MPI I/O, experimental analysis of MPI one-sided RMA operations is currently only done by the serial trace analyzer, and automatic trace analysis of OpenMP applications using dynamic, nested and guarded worksharing constructs is not yet possible.

While the current parallel trace analysis mechanism is already a very powerful instrument in terms of the number of application processes it supports, we are working on optimized data management operations and workflows that will allow us to master even larger configurations. Restrictions and inefficiencies imposed by the current CUBE-file format and data model are also being addressed to allow non-aggregatable metrics (such as rates) to be stored and accessed without the need to process and aggregate values from the entire report.

Although parallel simulations are often iterative in nature, individual iterations can differ in their performance characteristics. Another major focus of our research is therefore to study the temporal evolution of the performance behavior as a computation progresses. Our general approach is to first observe the behavior on a coarse-grained level and then to successively refine the measurement focus as new performance knowledge becomes available. Using a more flexible measurement control, we are also striving to offer more targeted trace collection mechanisms, reducing memory and disk space requirements while retaining the value of trace-based in-depth analysis.

Finally, the symptoms of a performance bottleneck may appear much later than the event causing it, on a different processor, or both. For this reason, we are currently looking for ways to establish causal connections among different pattern instances found in traces and related phenomena such as load imbalance because we believe that understanding such links can prove essential for more effective scaling strategies. First experiments with a trace-based simulator that verifies corresponding hypotheses by replaying modified traces in real time on the target system proved encouraging.

For more information on SCALASCA refer to the website `www.scalasca.org`.

Acknowledgements This work has been supported by the Helmholtz Association under Grants No. VNG-118 and No. VH-VI-228 ('VI-HPS') and by the Federal Ministry for Research and Education (BMBF) under Grant No. 01IS07005C ('ParMA').

References

1. Becker, D., Rabenseifner, R., Wolf, F.: Timestamp synchronization for event traces of large-scale message-passing applications. In: Proc. of the 14th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI), *Lecture Notes in Computer Science*, vol. 4757, pp. 315–325. Springer, Paris, France (2006)
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications **14**(3), 189–204 (2000)
3. Geimer, M., Wolf, F., Wylie, B., Mohr, B.: Scalable parallel trace-based performance analysis. In: Proc. of the 13th European Parallel Virtual Machine and Message Passing Interface Conference (EuroPVM/MPI), *Lecture Notes in Computer Science*, vol. 4192, pp. 303–312. Springer, Bonn, Germany (2006)
4. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP : A parallel program development environment. In: Proc. of the 2nd International Euro-Par Conference, pp. 665–674. Springer, Lyon, France (1996)
5. Nagel, W., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. Supercomputer **63, XII**(1), 69–80 (1996)
6. Oliker, L., Canning, A., Carter, J., Iancu, C., Lijewski, M., Kamil, S., Shalf, J., Shan, H., Strohmaier, E., Ethier, S., Goodale, T.: Scientific application performance on candidate petascale platforms. In: Proc. of the International Parallel & Distributed Processing Symposium (IPDPS). Long Beach, CA (2007)
7. Paraver: `http://www.cepba.upc.es/paraver/`
8. Song, F., Wolf, F., Bhatia, N., Dongarra, J., Moore, S.: An algebra for cross-experiment performance analysis. In: Proc. of the International Conference on Parallel Processing (ICPP), pp. 63–72. IEEE Society, Montreal, Canada (2004)
9. VAMPIR: `http://www.vampir.eu/`
10. Van De Vanter, M., Post, D., Zosel, M.: HPC needs a tool strategy. In: Proc. of the 2nd International Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS) (2005)
11. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. Journal of Systems Architecture **49**(10-11), 421–439 (2003)