DOI: 10.14529/jsfi140207

Scalable parallel performance measurement and analysis tools – state-of-the-art and future challenges

B. Mohr[1]

Current large-scale HPC systems consist of complex configurations with a huge number of potentially heterogeneous components. As the systems get larger, their behavior becomes more and more dynamic and unpredictable because of hard- and software re-configurations due to fault recovery and power usage optimizations. Deep software hierarchies of large, complex system software and middleware components are required to operate such systems. Therefore, porting, adapting and tuning applications to today's complex systems is a complicated and time-consuming task. Sophisticated integrated performance measurement, analysis, and optimization capabilities are required to efficiently utilize such systems. This article will summarize the state-of-the-art of scalable and portable parallel performance tools and the challenges these tools are facing on future extreme-scale and big data systems.

Keywords: parallel programming, performance tools, extreme scale computing.

Introduction

Current large-scale HPC systems consist of complex configurations with a huge number of components. Each node has multiple multi-core sockets and often one or more additional accelerator units in the form of many-core nodes (e.g., Intel Xeon Phi), GPGPUs or FPGAs, resulting in a heterogeneous system architecture. Caches, memory and storage are attached to the components on various levels and are shared between components in varying degrees. Typically, there is a (huge) imbalance between the computational power of the components and the amount of memory available to these components. Thus, deep software hierarchies of large, complex system software and middleware components are required to efficiently use such systems.

Therefore, porting, adapting and tuning applications to today's complex systems is a complicated and time-consuming task. Sophisticated integrated performance measurement, analysis, and optimization capabilities are required to efficiently utilize such systems. However, designing and implementing efficient and useful performance tools is extremely difficult because of the same reasons. While the system and middleware software layers are designed to be transparent, they are typically not transparent with respect to performance. This *performance intransparency* will result in escalation of unforeseen problems to higher layers, including the application. This is not really a new problem, but certain properties of current large-scale and future extreme-scale systems significantly increase its severity and significance:

- At extreme-scale scale, there always will be failing components in the system, which has a large impact on performance. A real-world application will probably never run on the exact same configuration twice.
- Load balancing issues limit the success even on moderately parallel systems, and the challenge of data locality will become another severe issue which has to be addressed by appropriate mechanisms and tools.
- Dynamic power management, e.g., at hardware level inside a CPU or accelerator, will result in performance variability between cores, nodes, and across different runs. The alternative to run at lower but system-wide consistent speed without dynamic power adjustments may not be an option in the future.

[1]Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany

- The challenge of predicting application performance at extreme scale will make it difficult to detect a performance problem if it is escalated undetected to the application level.
- The ever growing higher integration of components into a single chip and the use of more and more hardware accelerators makes it more difficult to monitor application performance and move performance data out of the system unless special hardware support will be integrated into future systems.

Altogether this will require an integrated, collaborative, and holistic approach to handle performance issues and correctly detect and analyze performance problems. In the following, we will summarize the state-of-the-art of scalable and portable parallel performance tools. Next, we discuss the various methods and approaches regarding tool portability, scalability, and integration. The article closes with an outlook on the challenges tools will face on future extreme-scale and big data systems.

1. Overview of state-of-the-art parallel performance tools

Research on parallel performance tools has a long history. First tools appeared at the same time as the first parallel computer systems back in the 1980s and early 90s [1]. Meanwhile, many performance instrumentation, measurement, analysis and visualization tools exist, and summarizing and listing all major methods, approaches and tools is impossible in a short journal paper. Therefore, in the following section we concentrate on the major tool sets that are (i) *portable*, i.e., they can be used on more than one of today's dominating architectures: Cray, IBM BlueGene, Fujitsu K computer, and Linux/UNIX clusters, (ii) *scalable*, i.e., it was demonstrated that they can be successfully used for an application executing on a couple of thousand nodes, (iii) *versatile*, i.e., they allow the performance analysis of all levels of today's HPC systems: message passing between nodes, multi-threading and multi-tasking inside nodes, and offloading to accelerators, and (iv) *supported*, i.e., there are well-established groups or organizations behind them which maintain and further develop them.

The structure of the following subsections is modelled after and re-uses excerpts from the VI-HPS Tools Guide [2], which also includes descriptions of further tools.

1.1. TAU

TAU [3, 4] is a comprehensive profiling and tracing toolkit that supports performance evaluation of programs written in C++, C, UPC, Fortran, Python, and Java. It is a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. TAU supports both direct measurement as well as sampling modes of instrumentation and interfaces with external packages such as PAPI [25] or tools like Score-P, Scalasca, and Vampir (all described in the following sections). TAU is available under a BSD-style license.

Typical Workflow

TAU allows the user to instrument the program in a variety of ways including rewriting the binary using *tau_rewrite* or runtime pre-loading of shared objects using *tau_exec*. Source-level instrumentation typically involves substituting a compiler in the build process with a TAU compiler wrapper. This wrapper uses a given TAU configuration to link in the TAU library. At

runtime, a user may specify different TAU environment variables to control the measurement options chosen for the performance experiment. This allows the user to generate call-path profiles, specify hardware performance counters, turn on event-based sampling, generate traces, or specify memory instrumentation options.

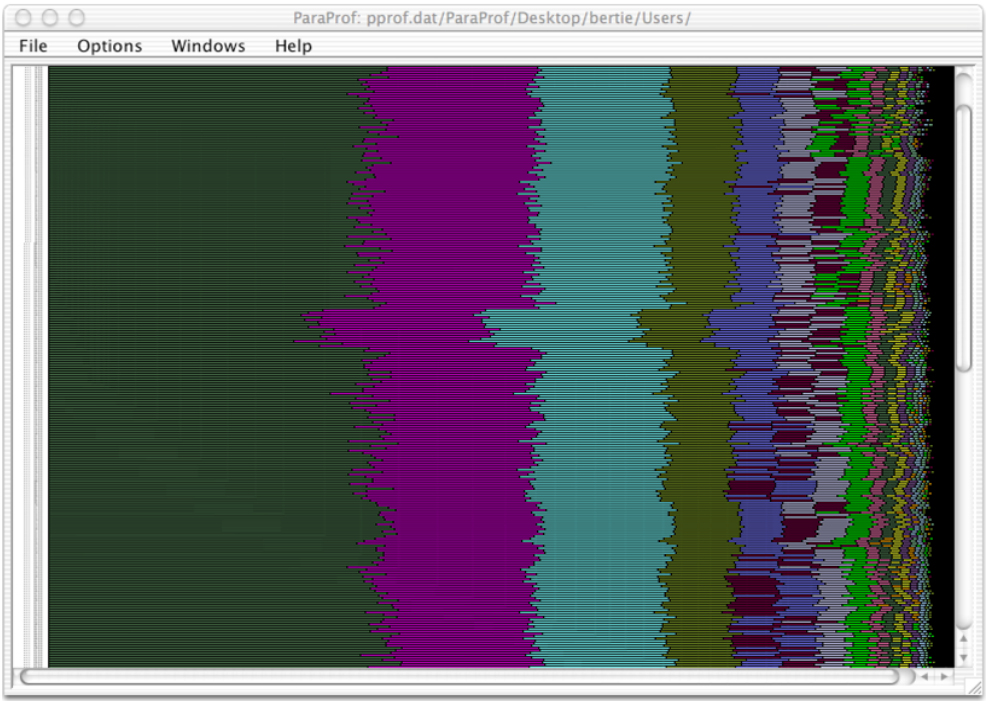

Figure 1. TAU main profile window. Each line shows a flattened histogram of a selected performance metric for a specific thread or task. The different colors represent the different program regions (e.g., functions, loops, parallel sections, etc.). The display allows to get a quick overview via comparing the differences/commonalities of the different threads/tasks. Clicking on a line label brings up more detailed information about the selected thread or task. Clicking on a colored region brings up more detailed information about the selected program region. The menu bar allows to choose the performance metric shown or to launch further, more detailed displays like a call-graph or communication matrix display. Results from larger performance experiments can be more easily analyzed with the 3D displays of TAU, see fig. 2.

Performance-analysis results may be stored in TAUdb, a database for cross-experiment analysis and advanced performance data mining operations using TAU's PerfExplorer tool [5]. It may be visualized using ParaProf, TAU's profile browser that can show the extent of performance variation and compare executions, see fig. 1.

Supported platforms

IBM Blue Gene/P/Q, NVIDIA and AMD GPUs and Intel MIC systems, Cray XE/XK/XC30, SGI Altix, Fujitsu K Computer (FX10), NEC SX-9, Solaris & Linux clusters (x86/x86_64, MIPS, ARM), Windows, Apple Mac OS X.

Supported Runtime Layers

MPI, OpenMP (using GOMP, OMPT, and Opari instrumentation), Pthread, MPC Threads, Java Threads, Windows Threads, CUDA, OpenCL, OpenACC.

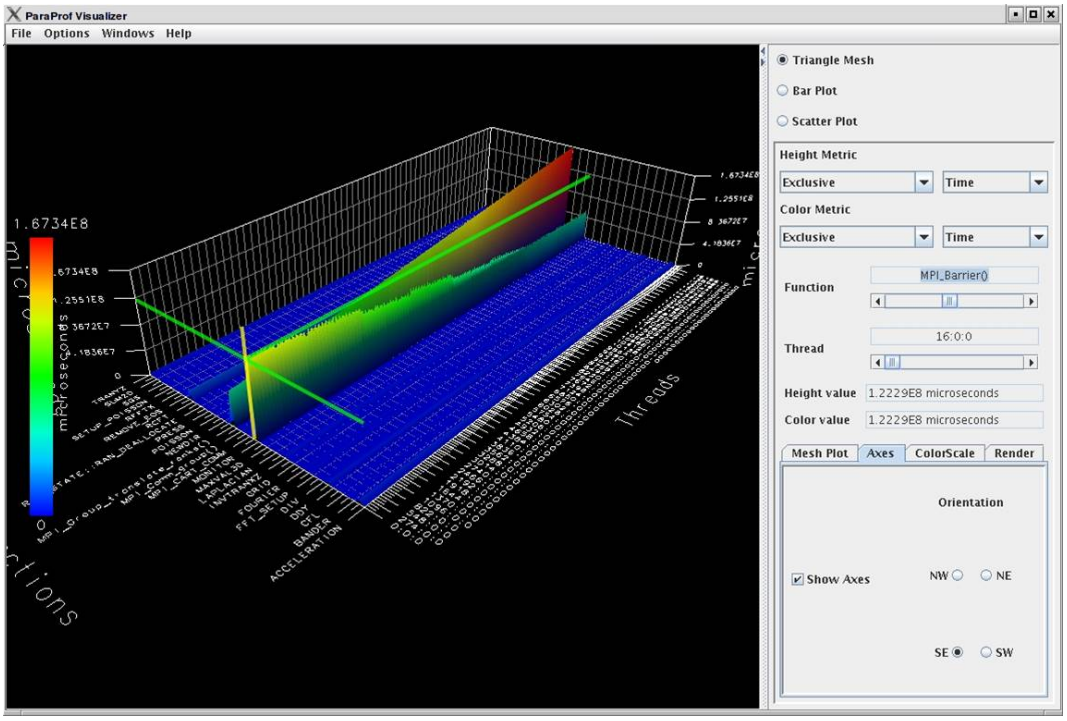

Figure 2. TAU 3D profile view. It displays the distribution of two selected performance metrics (encoded via height and color) over the axes program regions and threads/tasks.

1.2. HPCToolkit

HPCToolkit [6, 7] is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multi-core desktop systems to the largest supercomputers. HPCToolkit provides accurate measurements of a program's work, resource consumption, and inefficiency, correlates these metrics with the program's source code, works with multilingual, fully optimized binaries, and has very low measurement overhead. HPCToolkit's measurements provide support for analyzing a program's execution cost, inefficiency, and scaling characteristics both within and across nodes of a parallel system.

Typical Workflow

HPCToolkit works by sampling an execution of a multi-threaded and/or multiprocess program using hardware performance counters, unwinding thread call stacks, and attributing the metric value associated with a sample event in a thread to the calling context of the thread/process in which the event occurred. Sampling has sometimes advantages over instrumentation for measuring program performance: it requires no modification of source code and it avoids potential blind spots (such as code available in only binary form). Sampling using performance counters enables fine-grained measurement and attribution of detailed costs including metrics such as operation counts, pipeline stalls, cache misses, and inter-cache communication in multi-core and multi-socket configurations. HPCToolkit also supports computing derived metrics such as cycles per instruction, waste, and relative efficiency to provide insight into a program's shortcomings. HPCToolkit is available under a BSD-style license.

HPCToolkit assembles performance measurements into a call-path profile that associates the costs of each function call with its full calling context. A unique capability of HPCToolkit is

its nearly flawless ability to unwind a thread's call stack (which is often difficult and error-prone with highly optimized code). In addition, HPCToolkit uses binary analysis to attribute program performance metrics to full dynamic calling contexts augmented with information about call sites, source lines, loops and inlined code. Measurements can be analyzed in a variety of ways, see fig. 3: top-down in a calling context tree, which associates costs with the full calling context in which they are incurred; bottom-up in a view that apportions costs associated with a function to each of the contexts in which the function is called; and in a flat view that aggregates all costs associated with a function independent of calling context.

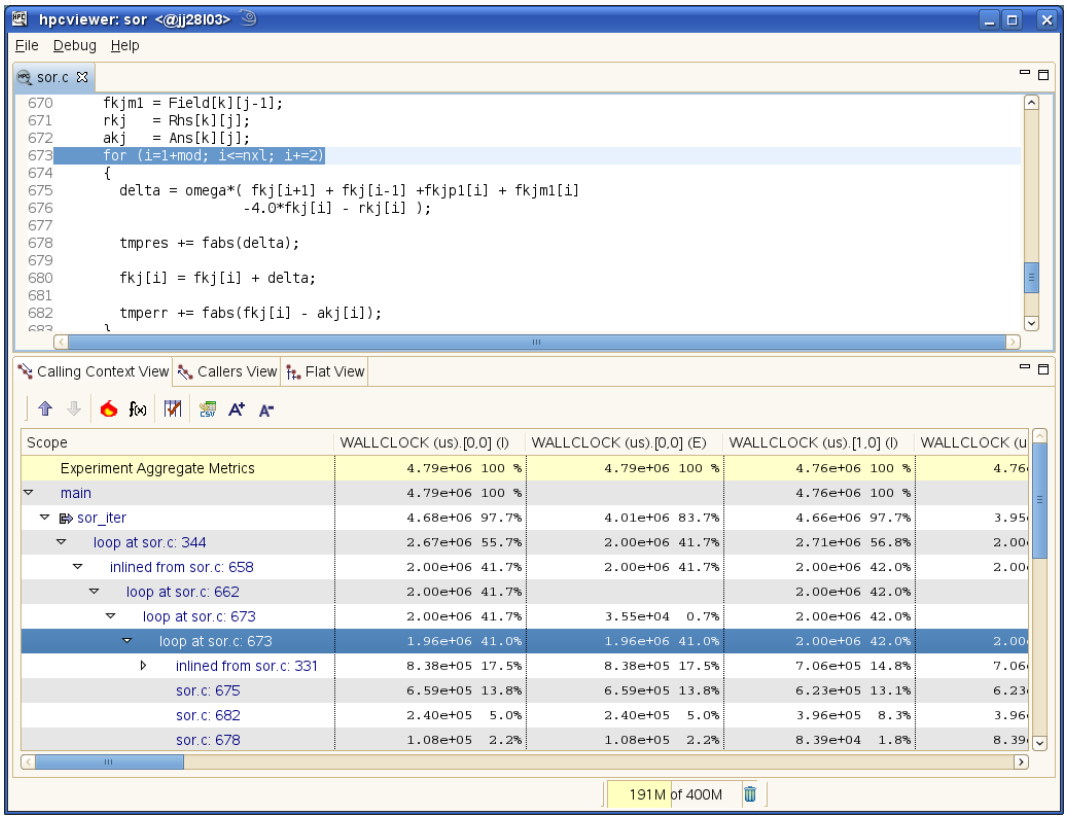

Figure 3. HPCToolkit main window (hpcviewer) showing a top-down calling context analysis. Selecting a specific program region in the call tree at the bottom displays the corresponding source code in the upper part.

By working at the machine-code level, HPCToolkit accurately measures and attributes costs in executions of multilingual programs, even if they are linked with libraries available only in binary form. HPCToolkit supports performance analysis of fully optimized code; it even measures and attributes performance metrics to shared libraries that are dynamically loaded at runtime.

HPCToolkit also helps pinpointing scaling losses in parallel codes, both within multi-core nodes and across the nodes in a parallel system. Using differential analysis of call path profiles collected on different numbers of threads or processes enables one to quantify scalability losses and pinpoint their causes to individual lines of code executed in particular calling contexts.

Supported platforms

IBM Blue Gene/P/Q, Cray XE/XK/XC30, Linux clusters (x86/x86_64).

Supported Runtime Layers

HPCToolkit allows measuring and analyzing codes which span multiple processes (within and across nodes) and/or use multi-threading (e.g., OpenMP, Pthreads, etc.). As the tool collects data on the binary level, it is programming-model agnostic; the "translation" from binary objects (e.g., a runtime system function call) to programming model user-level constructs (e.g., OpenMP critical region directive) has to be done by the user.

1.3. Extrae/Paraver

Paraver [8–10] is a performance analyzer based on event traces with a great flexibility to explore the collected data, enabling detailed analysis of metrics variability and distribution with the objective of understanding the applications' behavior. Paraver has only two types of views, but a lot of flexibility to define and correlate them. Timelines provide the evolution with time (see fig. 4) and tables (histograms, profiles) a measurement of the metrics distribution.

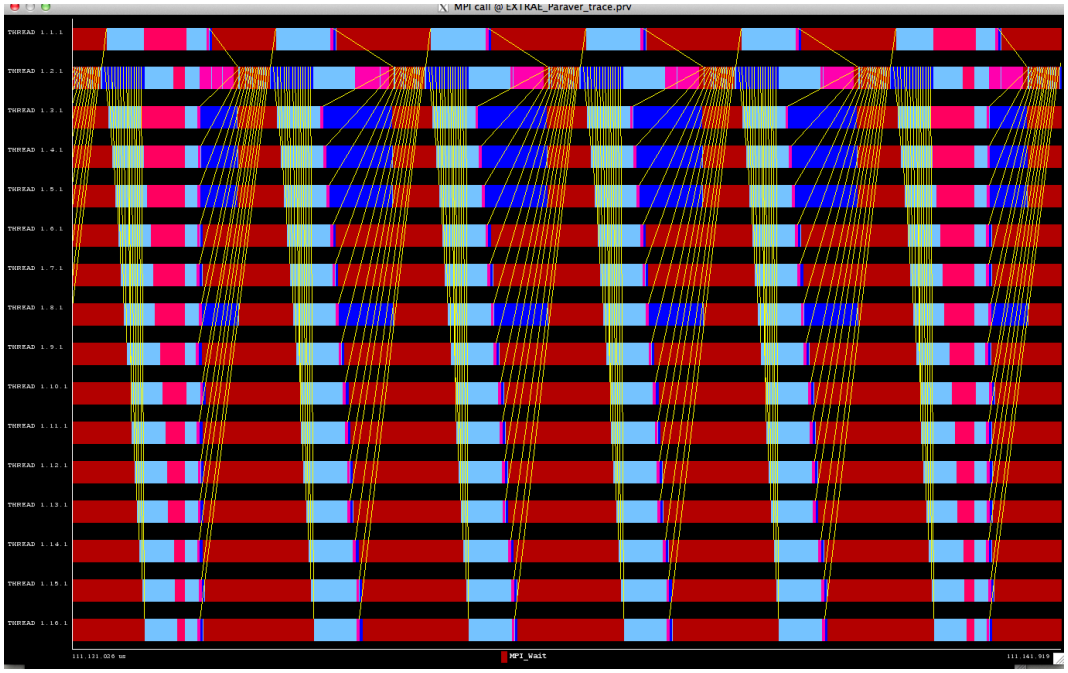

Figure 4. Paraver timeline window

To facilitate extracting insight from detailed performance data, during the last years new modules have been added implementing advanced performance analytics techniques. Clustering, tracking and folding allow the performance analyst to identify the program structure, study its evolution and look at the internal structure of the computation phases.

Typical Workflow

First, event traces are collected with the parallel program instrumentation and measurement package Extrae in a Paraver-specific format (PRV). It uses different mechanisms to insert measurement probes that vary from static interception of the runtime calls linking with the Extrae library to dynamic instrumentation using Dyninst [11]. The most frequent scenario is to use LD_PRELOAD to intercept runtime system calls of production binaries at loading time. The information collected by Extrae includes entry and exit to the programming model runtime,

hardware counters (via PAPI), call stack references, user functions, periodic samples and user events. Extrae and Paraver are available under a LPGL license.

Once the necessary trace(s) are collected, the analyst drives the full process generating and validating his/her hypothesis about the application behavior within Paraver. The initial steps would be similar for all the analyses and are provided as a basic methodology tutorial, the results of these steps would allow to decide the path to follow. The tool provides an extensive list of pre-conceived configuration files. Paraver offers a very flexible way to combine multiple views, so as to generate new representations of the data and more complex derived metrics. Once a desired view is obtained, it can be stored in a configuration file to apply it again to the same trace or to a different one.

Platform support

Linux clusters (x86/x86_64, ARM, Power), IBM Blue Gene/P/Q, Fujitsu FX10, SGI Altix, Cray XT, Intel Xeon Phi, GPUs.

Supported Runtime Layers

MPI, OpenMP, OmpSs, Pthread, CUDA, OpenCL.

1.4. Vampir

The Vampir [12–14] event trace visualizer allows to study a program's runtime behavior at a fine level of detail. This includes the display of detailed performance event recordings over time in timelines and aggregated profiles. Interactive navigation and zooming are the key features of the tool, which help to quickly identify inefficient or faulty parts of a program.

Typical Workflow

Before using Vampir, an application program needs to be instrumented and executed with the community-developed parallel program instrumentation and measurement package Score-P (see section 1.6). Running the instrumented program produces a bundle of trace files in OTF2-format [16]. Earlier versions of Vampir were relying on an internal instrumentation and measurement package called VampirTrace which produced traces in OTF-Format [15]. Both VampirTrace and Score-P are available under a BSD-style license.

After a trace file has been loaded by Vampir, the Trace View window opens with a default set of charts as depicted in fig. 5. Vampir accepts traces in OTF, OTF2, and EPILOG [17] (a format used by earlier versions of the Scalasca tool, see next section). The Vampir visualizer is a commercial package.

Platform support

IBM Blue Gene/P/Q, AIX (x86_64, POWER6), Cray XE/XK/XC30, SGI UV/ICE/Altix, Linux clusters (x86/x86_64), Windows, Apple Mac OS X.

Supported Runtime Layers

MPI, OpenSHMEM, OpenMP, Pthread, CUDA, OpenCL, OpenACC.

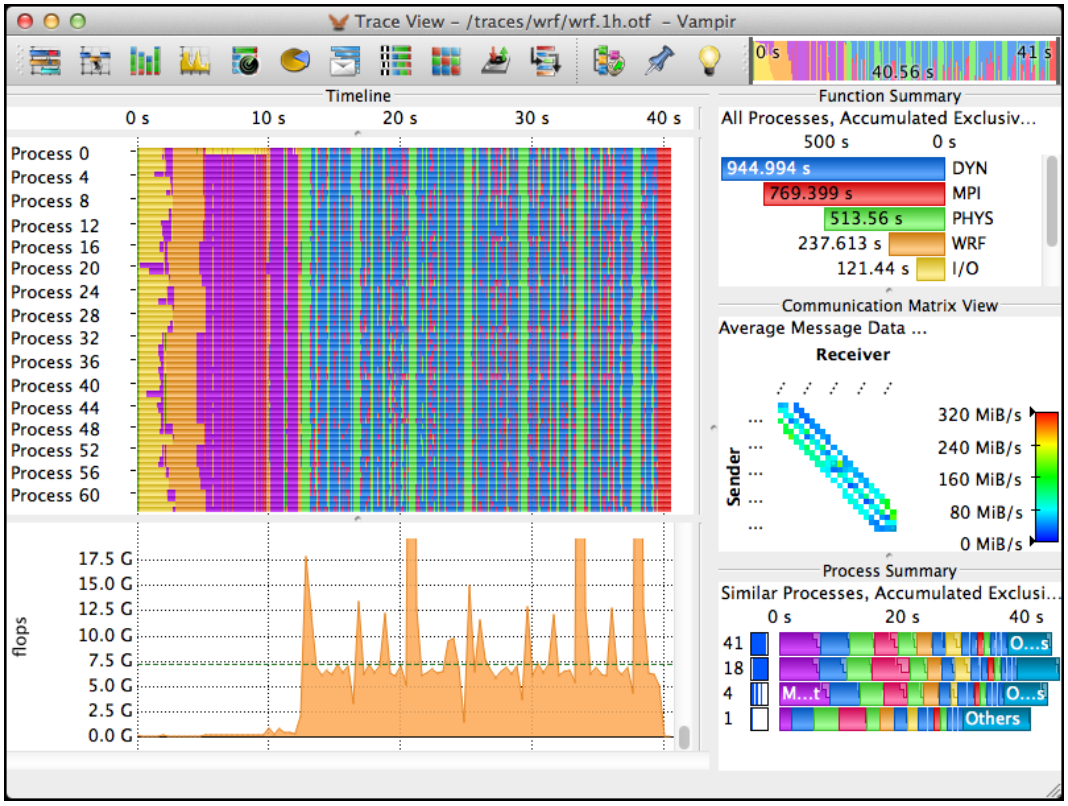

Figure 5. Vampir event trace browser. The charts shown can be divided into timeline charts and statistical charts. Timeline charts (left) show detailed event based information for arbitrary time intervals while statistical charts (right) reveal accumulated measures which were computed from the corresponding event data of the selected time interval in the timeline charts. Informational charts provide additional or explanatory information regarding timeline- and statistical charts. On top of the charts, two toolbars are available. The Charts Toolbar (top left) allows to add further charts to study I/O, inter-process communication and synchronization, and hardware performance metrics of the depicted program run. An overview of the phases of the entire program run is given in the Zoom Toolbar (top right), which can also be used to zoom and pan to the program phases of interest.

1.5. Scalasca

Scalasca [18, 19] supports the performance optimization of parallel programs by measuring and analysing their runtime behavior. The tool has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XE, but is also well suited for small- and medium-scale HPC platforms. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronization – and offers guidance in exploring their causes. Scalasca is available under a BSD 3-Clause license.

The user of Scalasca 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. The latest versions also includes a scalable critical path analysis [20] and root-cause analysis [21]. Performance-analysis results are presented to the user in an interactive explorer called Cube (fig. 6) that allows the investigation of the performance behavior on different levels of granularity along the dimensions metric, call path, and process.

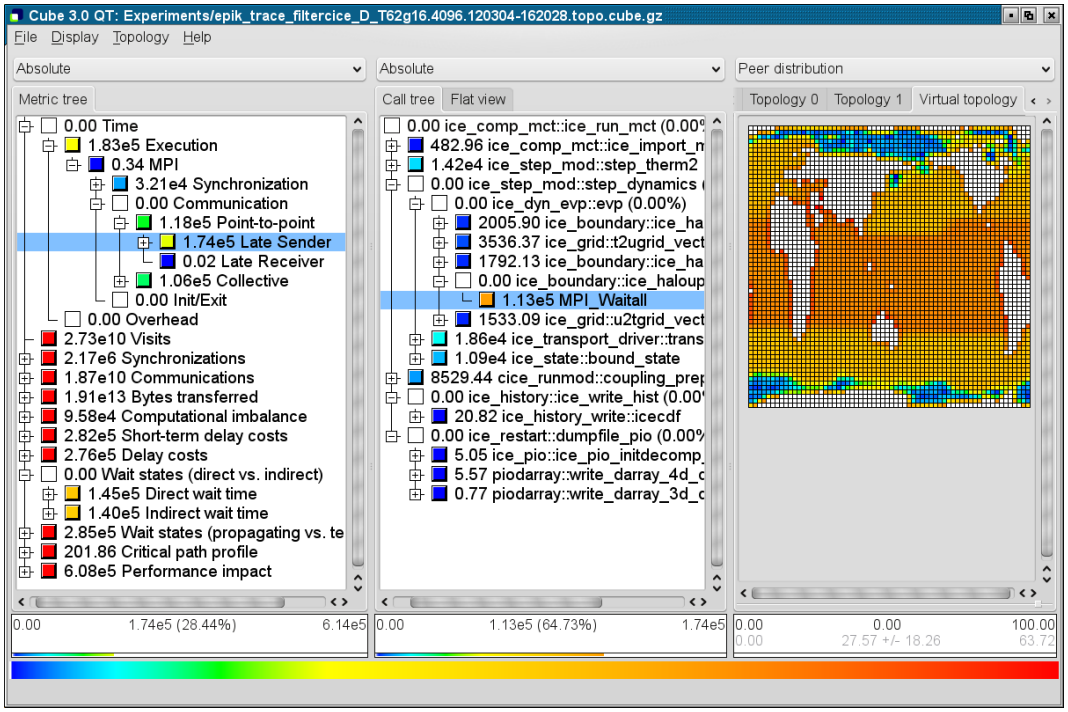

Figure 6. Scalasca's result explorer Cube. It allows interactive exploration of performance behavior along the dimensions performance metric (left), call tree (middle), and process topology (right). Selecting a metric displays the distribution of the corresponding value over the call tree. Selecting 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.

Typical Workflow

Before any Scalasca analysis can be carried out, the target application needs to be instrumented. For this task, Scalasca leverages now the community-driven instrumentation and measurement infrastructure Score-P (see section 1.6). Earlier versions of Scalasca (version 1) used an internal instrumentation and measurement package called EPIK which provided the same functionality. First, a call-path profile is collected and analyzed. After an optimized measurement configuration has been prepared based on initial profiles, a targeted event trace in EPILOG format (Scalasca version 1) or in OTF2 format (Scalasca version 2 or later) can be generated, and subsequently analyzed by Scalasca's automatic event trace analyzer after measurement is complete. This scalable analysis searches for inefficiency patterns and wait states, collects statistics about the detected instances, and identifies the critical path of the application. The analysis result can the be examined using the interactive performance report explorer Cube.

Platform support

IBM Blue Gene/P/Q, Cray XT/XE/XK/XC, SGI Altix (incl. ICE + UV), Fujitsu FX-10 & K Computer, Tianhe-1A, IBM AIX clusters, Solaris & Linux clusters (x86/x86_64, ARM)

Supported Runtime Layers

MPI, OpenSHMEM, OpenMP, OmpSs, HMPP, Pthread, CUDA.

1.6. Score-P

The Score-P [22, 23] instrumentation and measurement infrastructure is a highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis. It supports a wide range of HPC platforms and programming models. Score-P provides core measurement services for a range of specialized analysis tools, such as Vampir, Scalasca, TAU, and Periscope [24]. Further insights can be gained by employing additional tools on Score-P measurements. Score-P is available under a BSD 3-Clause license.

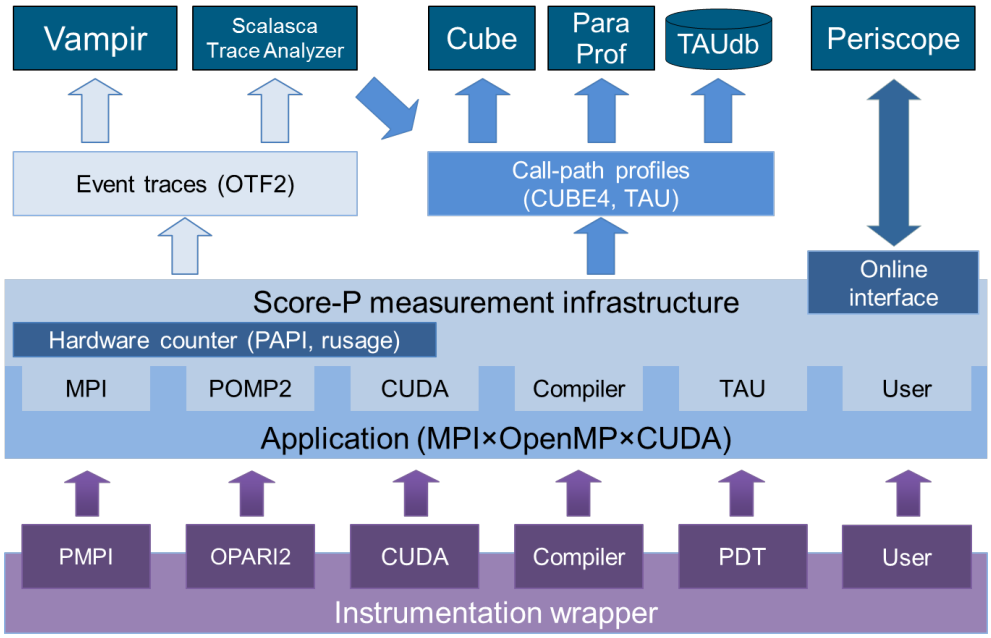

Figure 7. Score-P architecture

Typical Workflow

The overall Score-P architecture is shown in fig. 7. To create measurements, the target program must be instrumented. Score-P offers various instrumentation options. User functions can be instrumented automatically with compiler support or source-to-source instrumentation (PDT) or manually. MPI functions are monitored via library interposition (PMPI). OpenMP constructs are instrumented with the source-to-source instrumenter OPARI2.

For measurement, the instrumented program can be configured to record an event trace in OTF2 format or produce a call-path profile in CUBE4 format. Optionally, PAPI hardware counters [25] can be recorded. Filtering techniques allow precise control over the amount of data to be collected. Call-path profiles can be examined in TAU (see section 1.1) or the Cube profile browser (see section 1.5). Event traces can be examined in Vampir (see section 1.4) or used for automatic bottleneck analysis with Scalasca (see section 1.5). Alternatively, on-line analysis with Periscope is possible.

Platform support

IBM Blue Gene/P/Q, Cray XT/XE/XK, SGI Altix, K Computer, Fujitsu FX10, NEC SX-9, Solaris & Linux clusters (x86/x86_64, ARM)

Supported Runtime Layers

MPI, OpenSHMEM, OpenMP, OmpSs, HMPP, Pthread, CUDA.

2. Challenges for today's parallel performance tools

In this section, we discuss the basic challenges parallel performance tools face on today's large-scale systems: portability, scalability, and integration between tool components.

2.1. Portability

Although there was some consolidation in regard to operating systems for HPC clusters and high-end systems in the past decade – most of them are running some variant of Linux or at least Linux-compatible micro-kernels now – providing performance tools for these systems did not became much easier due to the openness and the lack of standardization of the Linux operating system. Basically no Linux cluster is exactly alike another one and each system provides a different set of compilers (e.g., GNU, Intel, PGI, IBM XL, Clang, SUN, Pathscale and others) and a different set of MPI libraries (e.g., MPICH, OpenMPI, Intel, LAM, HP, Scali, BullXMPI, SUN, IBM POE, Platform and more) in different versions. Depending on the compiler version, a different version of the OpenMP standard needs to be handled and supported by the tools; the same is true for the version of the MPI standard supported by the MPI library.

2.2. Scalability

The very large number of nodes and cores available to applications in future extreme-scale systems will be of course a severe challenge to tools, but the size of today's large-scale systems poses already problems now. In the November 2013 list of the TOP500 supercomputers, only four systems have less than 4,096 processor cores and the average is at almost 41,500 cores, which is an increase of 10,000 in just one year. Even the median system size is already over 17,400 cores. So tools must be able to handle at least tens of thousands cores and threads; ideally of course much more. Top 5 systems have multiple millions of cores. Various methods and approaches are employed by current performance tools to be able to handle the measurement and analysis of applications executing on large numbers of cores:

Scalable data collection and reduction: All tools listed in section 1 collect performance data in parallel either per process or even per thread and then use MPI to combine and merge the data either at the end of the execution or in a separate phase after the program measurement. Various I/O virtualization mechanisms (e.g., SIONlib [26]) are used to efficiently store process- and thread-local data in files. Scalasca's EPIK measurement system and Score-P no longer merge local traces into a global trace but use parallel I/O to access the separate traces concurrently and use MPI at the end of the program measurement to perform an efficient unification of measurement object identifiers and timestamp synchronization and correction. Another approach is used by Paraver: Automatic clustering, tracking and folding of trace events allows the performance analyst to identify the program structure, study its evolution and then only look at instances of the most important computation phases one at a time.

Scalable parallel data analysis: It is clear that the huge amount of performance data collected with a large-scale measurement can only be analyzed efficiently if the performance analysis components are parallelized themself. The Vampir toolset employs a distributed client/server model where the trace data is read and analyzed by a multi-node, multi-threaded server running on the target HPC system connected to a simple visualization client running on the workstation or laptop of the performance analyst. This way, the huge traces never have to be copied off the HPC system. The interactivity of the analysis process can be controlled by using more or less processes and threads for the parallel server process. The Scalasca trace analyzer also features a parallel and highly efficient performance bottleneck pattern search [18] and delay [21] and critical path [20] analysis. The algorithm is designed in a way that it uses the same number of MPI ranks and threads as the application under investigation, so the measurement and the subsequent trace analysis can be run in the same batch job. Time-consuming analysis modules from the Paraver analyzer and visualizer also have been parallelized with OmpSs.

Scalable visualizations: However, with an efficient parallel performance data collection and analysis, the bottleneck in the performance analysis process was just shifted to the result presentation phase. TAU uses 3D-displays to effectively display large amounts of data (see fig. 2) and Scalasca's report explorer Cube utilizes 2D and 3D displays to show performance data mapped on system hardware or application topologies (see fig. 6). Cube also uses hierarchical tree displays to allow browsing of performance data on various levels of detail. More research and new methods (e.g., visual data analytics) are needed here.

2.3. Integration

It is clear that only an integrated tool (environment) which can handle more than one level of parallelization – including the communication and synchronization aspects on the inter-node level, the multi-threading and multi-tasking issues at the intra-node level, and the data transfers, scheduling, and kernel invocations for attached accelerators – is a prerequisite for an efficient performance analysis process. This not only allows to investigate all performance relevant aspects of a program execution within one environment but also the influences and dependencies between the different parallelization levels. All tools listed in section 1 fall into this category. A big obstacle here is the current "zoo" of programming models available for multi-threading and off-loading; it is very hard for tools to support all of them and equally well.

However, it is just as important that performance measurement tools are integrated in some way with performance prediction and modeling tools. This way performance models can be automatically generated and/or calibrated with performance measurements and then can be much more successfully used to predict the application performance for other systems or configurations. A good example here is the Dimemas simulator which reconstructs the time behavior of a parallel application using a Paraver event trace that captures the time resource demands (CPU and network) of a parallel application as input. The target machine is modelled by a reduced set of key factors influencing the performance that model linear components like the point-to-point transfer time as well as non-linear factors like resources contention or synchronization. Using this simple model, Dimemas allows to simulate parametric studies in a very short time frame. Dimemas can generate a modified Paraver trace file as part of its output, enabling the user to conveniently examine the simulated run and understand the behavior. Another approach is

used in the DFG SPPEXA Catwalk project, where a series of profile measurements collected with Score-P, representing a scaling or parameter study, is input to a special model-generation component which automatically approximates the scaling behavior for selected metrics for all kernels of the analyzed program [30]. The generated performance model can then be used to easily locate program parts with a bad scaling behavior, e.g., program parts that will become bottlenecks on future systems with either higher core counts or less memory capacity.

Finally, even integrated and powerful tool environments cannot fulfill all requirements. Different tools have different strength and weaknesses and provide different views on the same performance behavior. Ideally, profile and trace data can easily be imported and exported by the various tools via open interfaces to make it easy to compare the results between tools and to move from one tool to another if necessary. In the first decade of the current millennium, the international tools community worked hard to achieve this goal, see fig. 8 which actually only shows the result for a selected portion of the tool landscape.

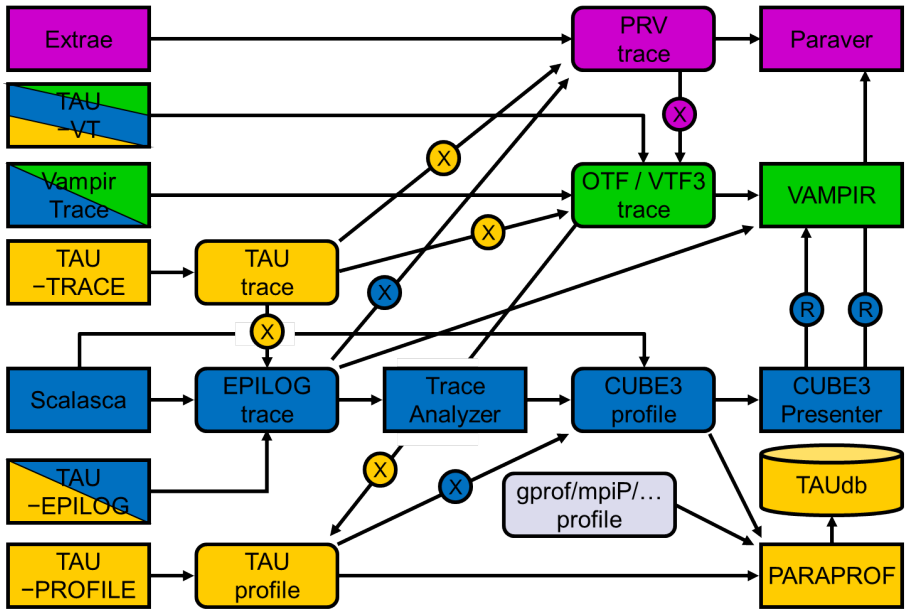

Figure 8. Tool integration efforts. The diagram shows the basic performance analysis workflows of the different tools around the year 2011. Rectangles represent tool components, boxes with round corners profile or trace data files. Lines labeled with an "X" in a circle indicate a conversion tool between the connected file formats. Lines labeled with an "R" in a circle describe Cube's ability to remotely control Vampir and Paraver [27].

.

Although kind of impressive, it quickly became clear that this complex situation is more confusing than helping for the typical tool user. This lead in the end to the community-developed performance instrumentation and measurement package Score-P which replaced the internal packages from Vampir, TAU, and Scalasca (see section 1.6), simplifying the resulting performance analysis workflow considerably. The use of common file formats like CUBE4 and OTF2 shared between a wide a range of tools allowed the definition of a well-defined and sophisticated performance analysis workflow in the EU-Russia research project HOPSA (see fig. 9). Beyond the tools described in this article, it also includes the commercial memory and threading analysis tool ThreadSpotter [28] and the system monitoring framework LAPTA [29]. For more details, especially more information about the implemented tool interactions, see [27].

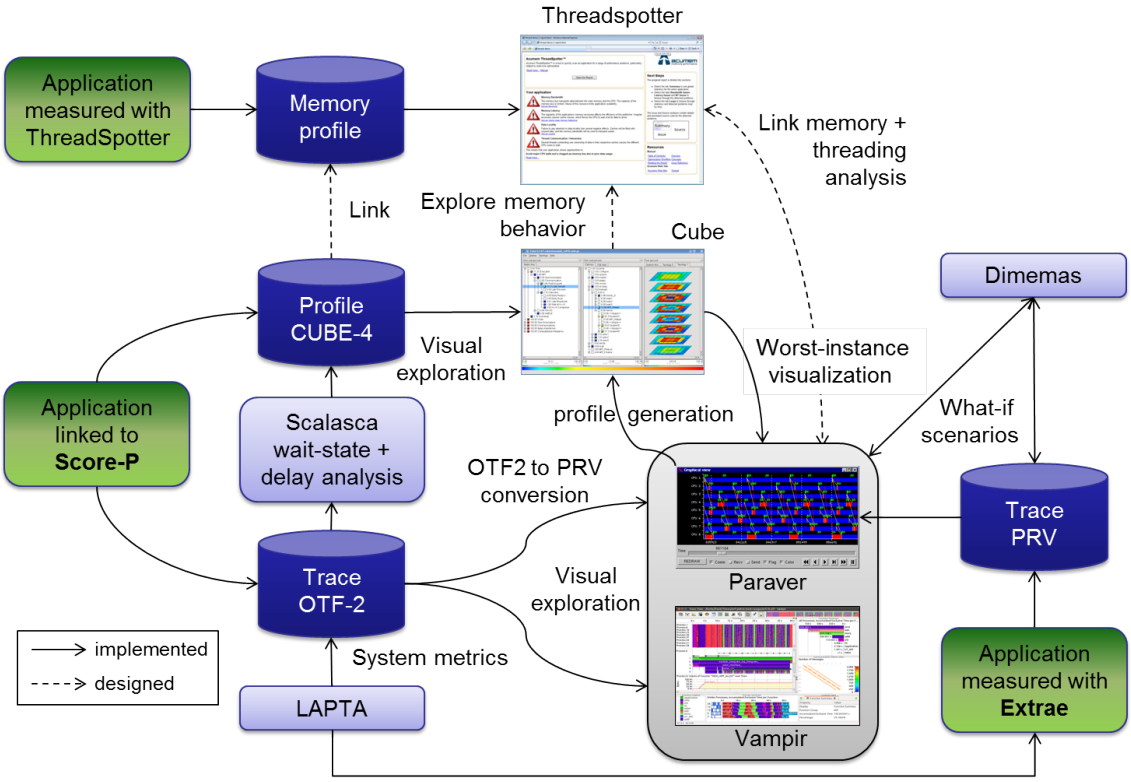

Figure 9. HOPSA workflow

3. Challenges for parallel performance tools for future extreme-scale and big data systems

A large number of approaches for performance analysis exist that have successfully applied at small and medium scale. The large amount of performance data may seem to impede the use at extreme scale. However, this is not the case as long as features like memory size and I/O capabilities scale with compute power. An instrumented application is nothing but an application with modified demands on the system executing it. This makes current approaches for performance analysis still feasible in the future as long as all involved software components are parallel and scalable. In addition to increased scalability, techniques like automatic analysis, advanced filtering, on-line monitoring, clustering, and analysis as well as data mining will be of increased importance. A combination of various techniques will have to be applied. The following considerations are key for a successful approach to performance analysis at extreme scale:

- Failover or operation with failed components in general should be performance neutral.
- An extreme scale system has to be capable to monitor the performance of components, not just the functionality.
- Hardware and software components need to provide sufficient performance details for the analysis if a performance problem unexpectedly escalates to higher levels.
- Metrics beyond FLOPs need to be developed to identify and quantify bottlenecks, to measure the sustained performance and the gap to the attainable peak performance.
- Programming models should be designed with performance analysis in mind. Part of that could be a (standardized) hidden control mechanism in the runtime system that will be

able to dynamically control – in time and space – the generation of performance data if requested.

- Performance analysis in the presence of noise requires inclusion of appropriate statistical descriptions.
- Performance analysis needs to incorporate techniques from the areas of signal processing, data mining, and visual data analytics.

Acknowledgements. *The author would like to use this opportunity to thank the HPC performance tools community in general and the Scalasca, Vampir, TAU, and Paraver teams in particular for many fruitful discussions, insights, and sharing their experiences and tools.*

References

1. R. Klar and N. Luttenberger: VLSI-based Monitoring of the Inter-Process-Communication of Multi-Microcomputer Systems with Shared Memory. Proceedings EUROMICRO '86, Microprocessing and Microprogramming, vol. 18, no. 15, 195–204, Venice, Italy, 1986.
2. Virtual Institute - High Productivity Supercomputing (VI-HPS): VI-HPS Tools Guide. Available at `http://www.vi-hps.org/tools/`.
3. S. Shende and A. D. Malony: The TAU Parallel Performance System. Intl. Journal of High Performance Computing Applications, 20(2):287–331, 2006. SAGE Publications. DOI: 10.1177/1094342006064482
4. TAU homepage. University of Oregon. `http://tau.uoregon.edu`
5. K. A. Huck and A. D. Malony: 2005. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. Proceedings ACM/IEEE conference on Supercomputing (SC '05). IEEE, Washington, DC, USA, 2005.
6. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience, 22(6):685–701, 2010.
7. HPCToolkit homepage. Rice University. `http://hpctoolkit.org`
8. J. Labarta, S. Girona, V. Pillet, T. Cortes, L. Gregoris, DiP: A parallel program development environment. Proceedings 2nd Intl. Euro-Par Conference, Lyon, France, Springer, 1996.
9. H. Servat Gelabert, G. Llort Sanchez, J. Gimenez, and J. Labarta: Detailed performance analysis using coarse grain sampling. Proceedings Euro-Par 2009 - Parallel Processing Workshops, Delft, The Netherlands, August 2009, 185–198. Springer, 2010.
10. Paraver homepage. Barcelona Supercomputing Center. `http://www.bsc.es/paraver`
11. Dyninst homepage. University of Wisconsin - Madison. `http://www.dyninst.org/`
12. M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, W. E. Nagel: Developing Scalable Applications with Vampir, VampirServer and VampirTrace. Proceedings of ParCo 2007, Jülich, Germany, 637–644, IOS Press, 2007.
13. A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, W. E. Nagel: The Vampir Performance Analysis Tool-Set. Proceedings Parallel Tools Workshop 2008, 139–155, 2008.
14. Vampir homepage. Technical University Dresden. `http://www.vampir.eu`

15. A. Knüpfer, R. Brendel, H. Brunst, H. Mix, W. E. Nagel: Introducing the Open Trace Format (OTF), Proceedings Computational Science - ICCS 2006: 6th Intl. Conference, Reading, UK, Springer, 526–533, 2006.

16. D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, F. Wolf: Open Trace Format 2 - The Next Generation of Scalable Trace Formats and Support Libraries. Proceedings of ParCo 2011, Ghent, Belgium, 481–490, IOS Press, 2012.

17. F. Wolf, B. Mohr: EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, 2004.

18. M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, B. Mohr: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, 2010.

19. Scalasca homepage. Jülich Supercomputing Centre and German Research School for Simulation Sciences. `http://www.scalasca.org`

20. D. Böhme, B. R. de Supinski, M. Geimer, M. Schulz, F. Wolf: Scalable Critical-Path Based Performance Analysis. Proceedings IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China, 1330–1340, IEEE, 2012.

21. D. Böhme, M. Geimer, F. Wolf, L. Arnold: Identifying the root causes of wait states in large-scale parallel applications. Proceedings Intl. Conference on Parallel Processing (ICPP), San Diego, CA, USA, 90–100, IEEE, 2010.

22. D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A.D. Malony, W.E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S.S. Shende, M. Wagner, B. Wesarg, F. Wolf: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Competence in High Performance Computing 2010 (CiHPC), 85–97. Springer, 2012.

23. Score-P homepage. Score-P Consortium. `http://www.score-p.org`

24. M. Gerndt and M. Ott: Automatic Performance Analysis with Periscope. Concurrency and Computation: Practice and Experience, 22(6):736–748, 2010.

25. PAPI homepage. University of Tennessee - Knoxville. `http://icl.cs.utk.edu/papi/`

26. W. Frings, F. Wolf, V. Petko:. Scalable massively parallel I/O to task-local files. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09). ACM, New York, NY, USA, Article 17, 2009.

27. B. Mohr, V. Voevodin, J. Giménez, E. Hagersten, A. Knüpfer, D. A. Nikitenko, M. Nilsson, H. Servat, A. Shah, F. Winkler, F. Wolf, and I. Zhukov: The HOPSA Workflow and Tools. Proceedings 6th Intl. Parallel Tools Workshop, Stuttgart, September 2012.

28. E. Berg, E. Hagersten: StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. Proceedings IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS-2004), Austin, Texas, USA, 2004.

29. A.V. Adinets, P.A. Bryzgalov, Vad.V. Voevodin, S.A. Zhumatiy, D.A. Nikitenko: About one approach to monitoring, analysis and visualization of jobs on cluster system (In Russian). Numerical Methods and Programming, vol. 12, 90–93, 2011.

30. A. Calotoiu, T. Hoefler, M. Poke, F. Wolf: Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. Proceedings ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA, 1–12, ACM, 2013.