Chapter 4
Recent Developments in the Scalasca Toolset

Markus Geimer, Felix Wolf, Brian J. N. Wylie,
Daniel Becker, David Böhme, Wolfgang Frings,
Marc-André Hermanns, Bernd Mohr, and Zoltán Szebenyi

Abstract The number of processor cores on modern supercomputers is increasing from generation to generation, and as a consequence HPC applications are required to harness much higher degrees of parallelism to satisfy their growing demand for computing power. However, writing code that runs efficiently on large processor configurations remains a significant challenge. The situation is exacerbated by the rising number of cores imposing scalability demands not only on applications but also on the software tools needed for their development.
To address this challenge, Jülich Supercomputing Centre creates software technologies aimed at improving the performance of applications running on leadership-class systems. At the center of our activities lies the development of Scalasca, a performance-analysis tool that has been specifically designed for large-scale systems and that allows the automatic identification of harmful wait states in applications running on hundreds of thousands of processors. In this article, we review recent developments in the open-source Scalasca toolset, highlight research activities of the Scalasca team during the past two years and give an outlook on future work.

Markus Geimer, Felix Wolf, Brian J. N. Wylie, David Böhme, Wolfgang Frings, Bernd Mohr, Zoltán Szebenyi
Jülich Supercomputing Centre,
Forschungszentrum Jülich, 52425 Jülich, Germany
e-mail: {m.geimer, b.wylie, d.boehme, w.frings, b.mohr, z.szebenyi}@fz-juelich.de

Felix Wolf, Daniel Becker, Marc-André Hermanns
German Research School for Simulation Sciences, 52062 Aachen, Germany
e-mail: {f.wolf, d.becker, m.a.hermanns}@grs-sim.de

Felix Wolf, David Böhme, Zoltán Szebenyi
RWTH Aachen University, 52056 Aachen, Germany

M.S. Müller et al. (eds.), *Tools for High Performance Computing 2009*,
DOI 10.1007/978-3-642-11261-4_4,

4.1 Introduction

Supercomputing is a key technology pillar of modern science and engineering, indispensable to solve critical problems of high complexity. The extension of the ESFRI road map to include a European supercomputer infrastructure in combination with the creation of the PRACE consortium acknowledges that the requirements of many critical applications can only be met by the most advanced custom-built large-scale computer systems. However, as a prerequisite for their productive use, the HPC community needs powerful and robust software development tools. These would not only help improve the scalability characteristics of scientific codes and thus expand their potential, but also allow domain scientists to concentrate on the underlying models 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 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 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 level actually sustained by production codes is expected to widen even further. Finally, increased concurrency levels place higher scalability demands not only on applications but also on parallel programming tools. When applied to larger numbers of cores, familiar tools often cease to work in a satisfactory manner (e.g., due to escalating memory requirements, failing renditions, or limited I/O performance).

To overcome this challenge, Jülich Supercomputing Centre creates software technologies aimed at improving the performance of applications running on leadership-class systems with hundreds of thousands of cores. At the center of our activities lies the development of Scalasca [1, 2], an open-source performance-analysis tool that has been specifically designed for large-scale systems, which allows the automatic identification of harmful wait states in applications running on very large processor configurations.

In this article, we give an overview of Scalasca and highlight research accomplishments of the Scalasca team during the past two years, focusing on the analysis of hybrid applications, the detection of wait states, and the characterization of time-dependent behavior. The latter two examples address the scalability of Scalasca regarding both the number of processes and the length of execution, respectively.

4.2 Scalasca Overview

Scalasca supports measurement and analysis of MPI applications written in C, C++ and Fortran on a wide range of current HPC platforms [3]. Hybrid codes making use of basic OpenMP features in addition to message passing are also supported. Figure 4.1 shows the basic analysis workflow supported by Scalasca. Before any performance data can be collected, the target application must be instrumented and linked to the measurement library. When running the instrumented code on the parallel machine, the user can choose between generating a summary analysis report ('profile') with aggregate performance metrics for individual function call paths and/or generating event traces recording individual runtime events from which a profile or time-line visualization can later be produced. Summarization is particularly useful to obtain an overview of the performance behavior and for local metrics such as those derived from hardware counters. Since traces tend to rapidly become very large [4], optimizing the instrumentation and measurement based on the summary report is usually recommended. When tracing is enabled, each process generates a trace file containing records for its process-local events. After program termination, Scalasca loads the trace files into main memory and analyzes them in parallel using as many processes as have been used for the target application itself. During the analysis, Scalasca searches for wait states and related performance properties, classifies detected instances by category, and quantifies their significance. The result is a wait-state report similar in structure to the summary report but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and wait-state reports contain performance metrics for every measured function call path and process/thread which can be interactively examined in the provided analysis report explorer.

4.3 Analysis of Hybrid MPI/OpenMP Codes

Although message passing is still the predominant programming paradigm used in HPC, increasingly applications leverage OpenMP to exploit more fine-grained process-local parallelism, while communicating between processes using MPI. Support for such hybrid applications in the Scalasca 1.0 release consisted of serial trace analysis of merged traces using the EXPERT analyzer from the KOJAK toolkit [5]. Extended runtime summarization and automatic parallel trace analysis support incorporated in Scalasca 1.2 provide similar analyses of hybrid OpenMP/MPI applications, within the same Scalasca instrumentation, measurement collection and analysis, and presentation usage model [6].

The OPARI source-code preprocessor inserts instrumentation for OpenMP constructs and API calls, which deliver events to the OpenMP-aware measurement library. Call-path metrics are accumulated per OpenMP thread during measurement, and collated into a complete summary report during finalization. Trace data is also analyzed in parallel with an analyzer thread for each OpenMP thread, and subse-

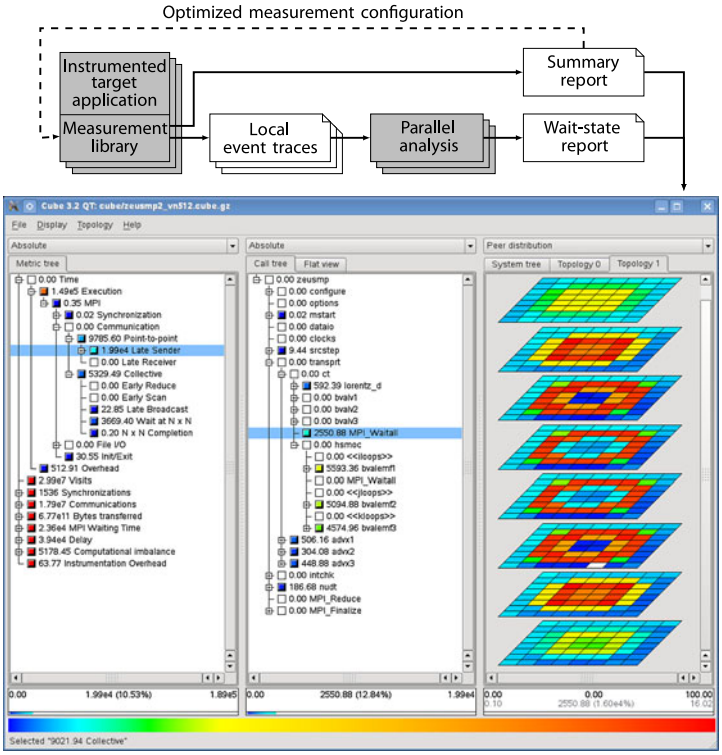


Fig. 4.1: Schematic overview of the performance data flow in Scalasca. Grey rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel. The GUI shows the distribution of performance metrics (left pane) across the call tree (middle pane) and the process topology (right pane).

quently collated into a similar pattern report. Event timestamp correction can also be applied to the trace data of OpenMP thread teams when logical consistency violations are encountered in MPI events due to unsynchronized clocks. Specific OpenMP metrics are calculated and presented alongside serial and MPI metrics in integrated analysis reports.

While the trace analysis currently remains restricted to fixed-size teams of OpenMP threads, runtime summarization identifies threads that have not been used during parallel regions. The associated time within the parallel region is distinguished as a *Limited parallelism* metric from the *Idle threads* time which includes time outside OpenMP parallel regions when only the master thread executes. This

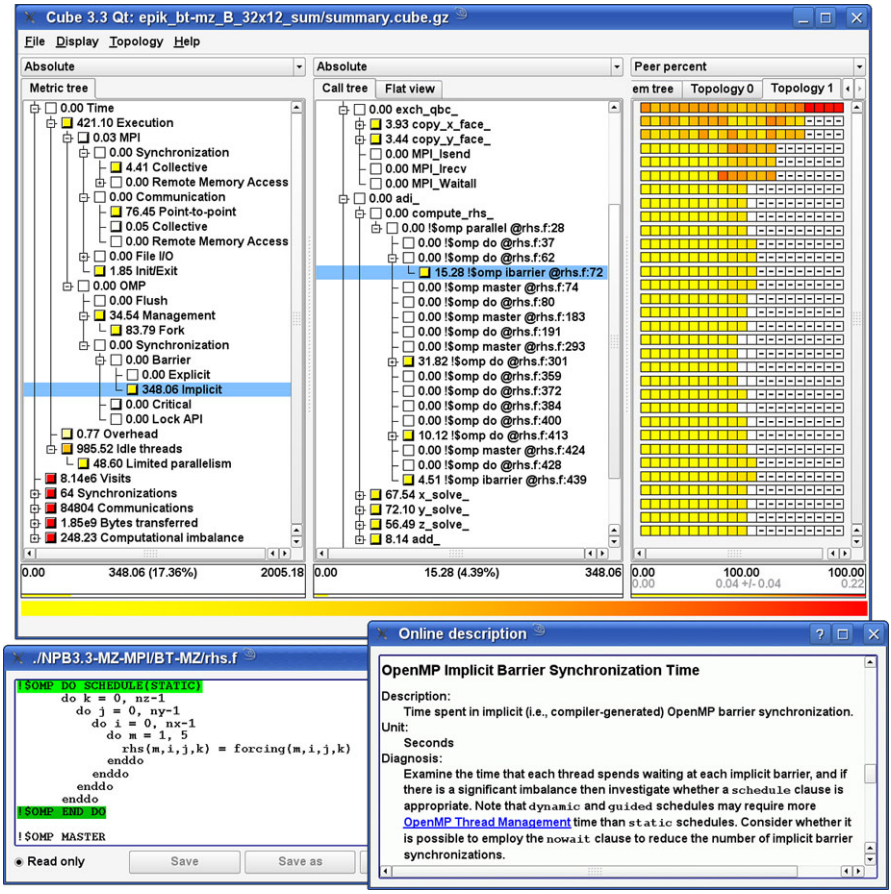


Fig. 4.2: Scalasca analysis report explorer display of a hybrid OpenMP/MPI NPB3.3 BT-MZ benchmark Class B execution on 32 Cray XT5 twin six-core compute nodes, showing OpenMP *Implicit Barrier Synchronization* time for a parallel loop in the `compute_rhs` routine (from lines 62 to 72 of source file rhs.f) broken down by thread. Higher metric values are shown darker and void thread locations in the topology pane are displayed in gray or with dashes.

matches the typical usage of dedicated HPC resources which are allocated for the duration of the parallel job, or threads which busy-wait occupying compute resources in shared environments. The number of OpenMP threads included in the measurement can be explicitly specified, defaulting to the number of threads for an unqualified parallel region when measurement commences: a warning is provided if subsequent `omp_set_num_threads` calls or `num_threads` clauses result in additional threads not being included in the measurement experiment.

Figure 4.2 shows a Scalasca analysis report from a hybrid OpenMP/MPI NAS NPB3.3 Block Triangular Multi-Zone (BT–MZ) benchmark [7] Class B execution in a Cray XT5 partition consisting of 32 compute nodes, each with two six-core Opteron processors. One MPI process was started on each of the compute nodes, and OpenMP threads run within each SMP node. In an unsuccessful attempt at load balancing by the application, more than 12 OpenMP threads were created by the first 6 MPI ranks (shown at the top of the topological presentation in the right pane), and 20 of the remaining ranks used fewer than 12 OpenMP threads. While the 49 seconds of *Limited parallelism* time for the unused cores represent only 2% of the allocated compute resources, half of the total time is wasted by *Idle threads* while each process executes serially, including MPI operations done outside of parallel regions by the master thread of each process. Although the exclusive *Execution* time in local computation is relatively well balanced on each OpenMP thread, the over-subscription of the first 6 compute nodes manifests as excessive *Implicit Barrier Synchronization* time at the end of parallel regions (as well as additional OpenMP *Thread Management* overhead), and higher *MPI Point-to-point Communication* time on the other processes is then a consequence of this. When over-subscription of cores is avoided, benchmark execution time is reduced by one third (with *MPI* time reduced by 52%, *OMP* time reduced by 20% and time for *Idle threads* reduced by 55%).

4.4 Scalable Wait-State 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. 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 communication and synchronization time can often be attributed to wait states, for example, as a result of an unevenly distributed workload. Especially when trying to scale applications to large process counts, such wait states can present severe challenges to achieving good performance.

4.4.1 Scalability

After the target application has terminated and the trace data have been flushed to disk, the trace analyzer is launched with one analysis process per (target) application process and loads the entire trace data into its distributed memory address space. Future versions of Scalasca may exploit persistent memory segments to pass the trace data to the analysis stage without involving any file I/O. While traversing the traces in parallel, the analyzer performs a replay of the application's original communication behavior [8]. During the replay, the analyzer identifies wait states

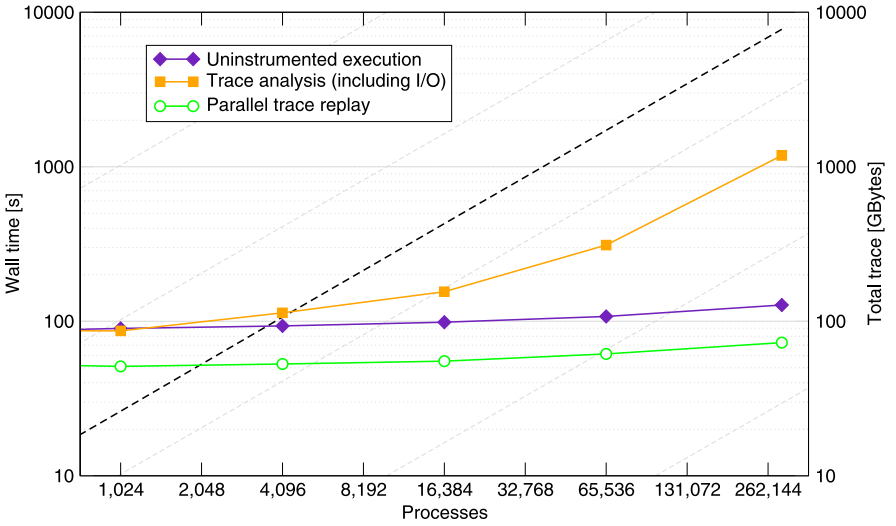


Fig. 4.3: Scalability of wait-state search for the ASCI benchmark application SWEEP3D on the JUGENE Blue Gene/P. The graph charts wall-clock execution times for the uninstrumented application and for the analyses of trace files generated by the instrumented version with varying numbers of processes. The time needed for the trace analysis replay is shown as well as that for the entire parallel analysis (including loading the traces and collating the analysis report). The black dashed line shows the linear increase in total trace size in GBytes.

in communication operations by measuring temporal differences between local and remote events after their timestamps have been exchanged using an operation of similar type. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, we can achieve good scalability at previously intractable scales. Recent scalability improvements allowed us to perform trace analyses of execution measurements with up to 294,912 processes (Figure 4.3).

4.4.2 Improvement of Trace-Data I/O

Parallel applications often store data in multiple task-local files, for example, to remember checkpoints, to circumvent memory limitations, or to record trace data in the case of the Scalasca toolset. When operating at very large processor configurations, such applications often experience scalability limitations when the simultaneous creation of thousands of files causes metadata-server contention or simply when large file counts complicate file management or operations on those files destabilize

the file system. In this context, a generic parallel I/O library called SIONlib has been developed which addresses this problem by transparently mapping a large number of task-local files onto a small number of physical files via internal metadata handling and block alignment to ensure high performance. While requiring only minimal source code changes, SIONlib significantly reduces file creation overhead and simplifies the resulting file handling, offering even the chance to achieve superior read and write performance via optimized I/O operations [9]. For the Scalasca trace collection and analysis of 294,912 processes shown in Figure 4.3, SIONlib was able to reduce the time to create the experiment archive directory and trace files from 86 minutes (for individual files) down to 10 minutes (for one file for each of the 576 BG/P I/O nodes),

4.4.3 Analysis of MPI-2 Remote Memory Access Operations

In our earlier work, we already defined wait-state patterns for MPI-2 *Remote Memory Access* (RMA) communication and synchronization, although still based on a serial trace-analysis scheme with limited scalability [10]. Taking advantage of Scalasca's scalable trace-analysis approach, we recently extended our parallel trace analyzer to detect these wait states. Using the programming paradigm of the target application, RMA-related communication and synchronization inefficiencies are now detected by exchanging data via RMA operations. In this way, we successfully performed analyses of RMA-based applications running with up to 8,192 processes. [11]

4.4.4 Delay Analysis

In general, the temporal or spatial distance between cause and symptom of a performance problem constitutes a major difficulty in deriving helpful conclusions from performance data. So just knowing the locations of wait states in the program is often insufficient to understand the reason for their occurrence. We are currently extending our replay-based wait-state analysis in such a way that it attributes the waiting times to their root causes. The root cause, which we call a *delay*, is an interval during which a process performs some additional activity not performed by its peers, for example as a result of insufficiently balancing the load. [12]

4.4.5 Evaluation of Optimization Hypotheses

Excess workload identified as root cause of wait states usually cannot simply be removed. To achieve a better balance, optimization hypotheses drawn from a delay

analysis typically propose the redistribution of the excess load to other processes instead. However, redistributing workloads in complex message-passing applications can have intricate side-effects that may compromise the expected reduction of waiting times. Given that balancing the load statically or even introducing a dynamic load-balancing scheme constitute major code changes, they should ideally be performed only if the prospective performance gain is likely to materialize. Our goal is therefore to automatically predict the effects of redistributing a given delay without altering the application itself and to determine the savings that can be realistically hoped for. Since the effects of such changes are hard to quantify analytically, we simulate these changes via a real-time replay of event traces after they have been modified to reflect the redistributed load. [13, 14]

4.4.6 Configurable Source-Code Instrumentation

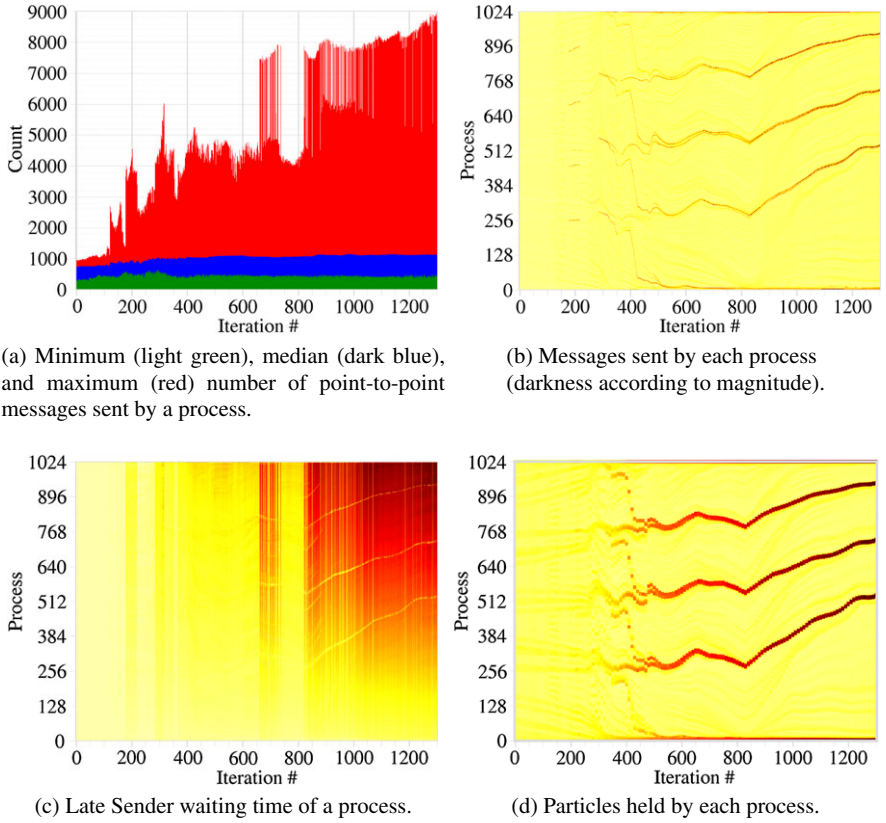
Proper instrumentation is an essential prerequisite for producing reliable performance analysis results with the Scalasca toolset. We therefore extended our instrumentation capabilities to leverage the generic and configurable source-code instrumentation component we developed in collaboration with the University of Oregon based on PDT and the TAU instrumentor. [15] This component provides flexible instrumentation specification capabilities, reducing the need to filter performance events at runtime and, thus, further reducing the measurement overhead.

4.5 Analysis of Time-Dependent Behavior

As scientific parallel applications simulate the temporal evolution of a system their progress occurs via discrete points in time. Accordingly, the core of such an application is typically a loop that advances the simulated time step by step. However, the performance behavior may vary between individual iterations, for example, due to periodically re-occurring extra activities [16] or when the state of the computation adjusts to new conditions in so-called adaptive codes [17].

4.5.1 Observing Individual Iterations

To study the time-dependent behavior, Scalasca is being equipped with iteration instrumentation capabilities (corresponding to TAU dynamic timers [18]) that allow the distinction of individual iterations both in runtime summaries and in event traces. Moreover, to simplify the understanding of the resulting time-series data, we are implementing several display tools including iteration graphs with minimum, median,



(a) Minimum (light green), median (dark blue), and maximum (red) number of point-to-point messages sent by a process.

(b) Messages sent by each process (darkness according to magnitude).

(c) Late Sender waiting time of a process.

(d) Particles held by each process.

Fig. 4.4: Gradual development of a performance problem over 1,300 timesteps of the PEPC application execution on 1,024 processors of Blue Gene.

and maximum representation (Figure 4.4a) as well as value maps to cover the full <process/thread, iteration> space for a given performance metric (Figure 4.4b).

Using prototype implementations of these new tools, we evaluated the performance behavior of the SPEC MPI2007 benchmark suite on the IBM SP p690 cluster JUMP, observing a large variety of complex temporal characteristics ranging from gradual changes and sudden transitions of the base-line behavior to both periodically and irregularly occurring peaks, including noise that varies from measurement to measurement [19]. Moreover, problems with several benchmarks that limited their scalability (sometimes to only 128 processes) were identified, such as distributing initialization data via broadcasts in 113.GemsFDTD and insufficiently large data sets for several others. Even those codes that apparently scaled well contained considerable quantities of waiting time, indicating possible opportunities for performance and scalability improvement through more effective work distributions or bindings of processes to processors.

Another real-world code with a substantially time-varying execution profile is the PEPC [20] particle simulation code, developed at Jülich Supercomputing Centre and the subject of an application liaison between the Scalasca and PEPC developer teams. The MPI code employs a parallel tree algorithm to efficiently calculate the forces the particles exert on each other and also includes a load-balancing mechanism that redistributes the computational load by shifting particles between processes. However, our analysis [21] revealed a severe and gradually increasing communication imbalance (Figure 4.4a). We found evidence that the imbalance was caused by a small group of processes with time-dependent constituency that sent large numbers of messages to all remaining processes (Figure 4.4b) in rank order, introducing *Late Sender* waiting times at processes with higher ranks (Figure 4.4c). Interestingly, the communication imbalance correlated very well with the number of particles "owned" by a process (Figure 4.4d), suggesting that the load-balancing scheme smoothes the computational load at the expense of communication disparities. Since the number of particles also influence the memory requirements of a process, we further conclude that the current behavior of concentrating particles at a small subset of processes may adversely affect scalability under different configurations. Work with the application developers to revise the load-balancing scheme and improve the communication efficiency is in progress.

4.5.2 Space-Efficient Time-Series Call-Path Profiling

While call-path profiling is an established method of linking a performance problem to the context in which it occurs, generating call-path profiles separately for thousands of iterations may exceed the available buffer space — especially when the call tree is large and more than one metric is collected. We therefore developed a runtime approach for the semantic compression of call-path profiles [22] based on incremental clustering of a series of single-iteration profiles that scales in terms of the number of iterations without sacrificing important performance details. Our approach has low runtime overhead by using only a condensed version of the profile data when calculating distances and accounts for process-dependent variations by making all clustering decisions locally.

4.6 Outlook

Besides further scalability improvements in view of upcoming systems in the range of several petaflops, we plan to extend Scalasca towards emerging programming models such as partitioned global address space languages and general-purpose GPU programming, which we expect to play a bigger role in the future. Moreover, to offer enhanced functionality and combine development efforts, we will integrate

Scalasca closer with related tools including Periscope [23], TAU [24], and Vampir [25].

Acknowledgement Financial support from the Helmholtz Association of German Research Centers through Grants VH-NG-118 and VH-VI-228 and the German Research Foundation through Grant GSC 111 is gratefully acknowledged. This research was supported by allocations of advanced computing resources provided by the John von Neumann Institute for Computing and US National Science Foundation for computations on the Jugene IBM Blue Gene/P at Jülich Supercomputing Centre and Kraken Cray XT5 at the US National Institute for Computational Sciences.

References

1. Jülich Supercomputing Centre: Scalasca. http://www.scalasca.org/.
2. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience, Proc. Workshop on Scalable Tools for High-End Computing (to appear) DOI: 10.1002/cpe.1556.
3. Wylie, B.J.N., Geimer, M., Wolf, F.: Performance measurement and analysis of large-scale parallel applications on leadership computing systems. Scientific Programming **16**(2-3) (2008) 167–181
4. Wolf, F., Freitag, F., Mohr, B., Moore, S., Wylie, B.J.N.: Large event traces in parallel performance analysis. In: Proc. 8th Workshop on Parallel Systems and Algorithms (PASA, Frankfurt/Main, Germany). Lecture Notes in Informatics, Gesellschaft für Informatik (March 2006) 264–273
5. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid MPI/OpenMP applications. In: Proc. 11th Euromicro Conf. on Parallel Distributed and Network based Processing (Genoa, Italy), IEEE Computer Society (February 2003) 13–22
6. Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.A., Mohr, B., Moore, S., Pfeifer, M., Szebenyi, Z.: Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In: Proc. 2nd HLRS Parallel Tools Workshop (Stuttgart, Germany), Springer (July 2008) 157–167 ISBN 978-3-540-68561-6.
7. Van der Wijngaart, R.F., Jin, H.: NAS Parallel Benchmarks, Multi-Zone versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, USA (July 2003)
8. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: A scalable tool architecture for diagnosing wait states in massively-parallel applications. Parallel Computing **35**(7) (2009) 375–388
9. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proc. 21st ACM/IEEE SC Conf. (SC09, Portland, OR, USA). (November 2009)
10. Kühnal, A., Hermanns, M.A., Mohr, B., Wolf, F.: Specification of inefficiency patterns for MPI-2 one-sided communication. In: Proc. 12th Euro-Par (Dresden, Germany). Volume 4128 of Lecture Notes in Computer Science, Springer (2006) 47–62
11. Hermanns, M.A., Geimer, M., Mohr, B., Wolf, F.: Scalable detection of MPI-2 remote memory access inefficiency patterns. In: Proc. 16th European PVM and MPI Conference (EuroPVM/MPI, Espoo, Finland). Volume 5759 of Lecture Notes in Computer Science, Springer (September 2009) 31–41
12. Böhme, D., Geimer, M., Hermanns, M.A., Wolf, F.: Identifying the root causes of wait states in large-scale parallel applications. Technical Report AICES-2010-1, Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen University, Germany (January 2010)
13. Hermanns, M.A., Geimer, M., Wolf, F., Wylie, B.J.N.: Verifying causality between distant performance phenomena in large-scale MPI applications. In: Proc. 17th Euromicro Int'l Conf. on

Parallel, Distributed, and Network-Based Processing (PDP, Weimar, Germany), IEEE Computer Society (February 2009) 78–84
14. Böhme, D., Hermanns, M.A., Geimer, M., Wolf, F.: Performance simulation of non-blocking communication in message-passing applications. In: Proc. 2nd Workshop on Productivity and Performance (PROPER 2009, Delft, The Netherlands). (August 2009) (to appear).
15. Geimer, M., Shende, S.S., Malony, A.D., Wolf, F.: A generic and configurable source-code instrumentation component. In: Proc. 9th Int'l Conf. on Computational Science (ICCS, Baton Rouge, LA, USA). Volume 5545 of Lecture Notes in Computer Science, Springer (May 2009) 696–705
16. Kerbyson, D.J., Barker, K.J., Davis, K.: Analysis of the weather research and forecasting (WRF) model on large-scale systems. In: Proc. 12th Conference on Parallel Computing (ParCo, Aachen/Jülich, Germany). Volume 15 of Advances in Parallel Computing, IOS Press (September 2007) 89–98
17. Shende, S., Malony, A., Morris, A., Parker, S., de St. Germain, J.: Performance evaluation of adaptive scientific applications using TAU. In: Parallel Computational Fluid Dynamics — Theory and Applications. Elsevier (2006) 421–428
18. Malony, A.D., Shende, S.S., Morris, A.: Phase-based parallel performance profiling. In: Proc. 11th Conference on Parallel Computing (ParCo, Málaga, Spain). Volume 33 of NIC Series, John von Neumann Institute for Computing (September 2005) 203–210
19. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proc. 1st SPEC Int'l Performance Evaluation Workshop (SIPEW, Darmstadt, Germany). Volume 5119 of Lecture Notes in Computer Science, Springer (June 2008) 99–123
20. Gibbon, P., Frings, W., Dominiczak, S., Mohr, B.: Performance analysis and visualization of the N-body tree code PEPC on massively parallel computers. In: Proc. 11th Conf. on Parallel Computing (ParCo, Málaga, Spain). Volume 33 of NIC Series, John von Neumann Institute for Computing (October 2005) 367–374
21. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: Scalasca parallel performance analyses of PEPC. In: Proc. 1st EuroPar Workshop on Productivity and Performance (PROPER 2008, Las Palmas de Gran Canaria, Spain). Volume 5415 of Lecture Notes in Computer Science, Springer (August 2008) 305–314
22. Szebenyi, Z., Wolf, F., Wylie, B.J.N.: Space-efficient time-series call-path profiling of parallel applications. In: Proc. 21st ACM/IEEE SC Conference (SC09, Portland, OR, USA). (November 2009)
23. Technical University of Munich: Periscope. http://www.lrr.in.tum.de/~gerndt/home/Research/PERISCOPE/Periscope.htm.
24. University of Oregon: TAU. http://www.cs.uoregon.edu/research/tau/.
25. Technische Universität Dresden: Vampir. http://www.vampir.eu/.