

# Extending Scalasca's Analysis Features

Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube,  
and Zoltán Szebenyi

**Abstract** Scalasca is a performance analysis tool, which parses the trace of an application run for certain patterns that indicate performance inefficiencies. In this paper, we present recently developed new features in Scalasca. In particular, we describe two newly implemented analysis methods: the root cause analysis which tries to identify the cause of a delay and the critical path analysis, which analyses the path of execution that determines the application runtime. Furthermore, we present time-series profiling, a method that allows to explore time-dependent behavior of an application. Finally, we extended the means of Scalasca and its output format CUBE to define and display topologies.

## 1 Introduction

Today, high performance computers provide the computing power which is required by the complexity of many scientific computations. However, providing larger and more powerful computer systems is useless if the applications do not make efficient use of the available resources. Especially since the clock rate will no longer continue to grow, the performance increase is due to increasing the parallelism of the computer systems.

However, the complexity of parallel programs is much higher than sequential programs. Furthermore, the hardware is more complex, too. Instead of a single processor, the programmer must deal with e.g. networks, data transfer rates and hierarchical memory. Thus, the optimization becomes much more difficult.

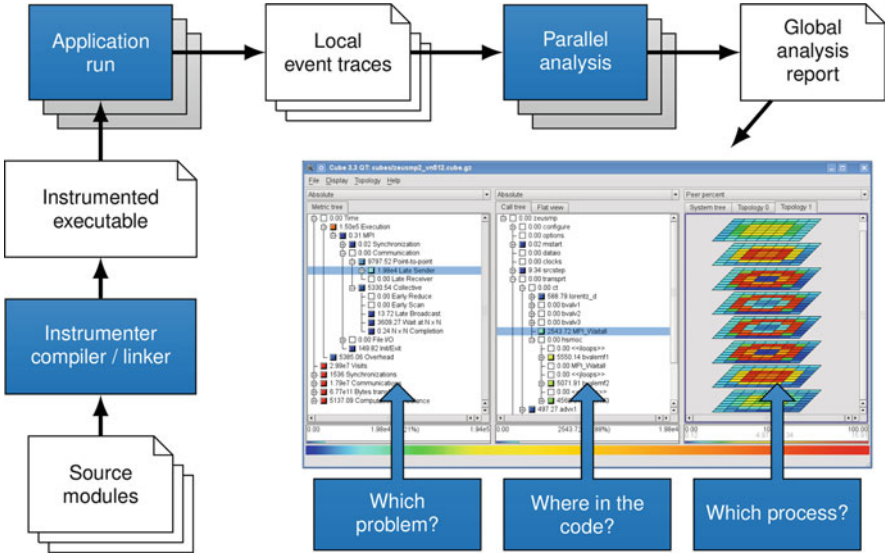
---

D. Lorenz (✉) · B. Mohr · A. Strube

Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany  
e-mail: [d.lorenz@fz-juelich.de](mailto:d.lorenz@fz-juelich.de)

D. Böhme · Z. Szebenyi

German Research School of Simulation Science GmbH, Aachen, Germany



**Fig. 1** The performance analysis workflow with Scalasca

Before a programmer can optimize the performance of his program, he must understand the performance behavior of his program and identify causes of performance reduction. In order to analyze the performance of a program, various performance analysis tools have been developed, like Scalasca [6], Vampir [8], TAU [13], Periscope [7] and Paraver [11]. Although each of the tools has its specific features and methods for analysis and display of results, there are some common techniques.

Some tools [1, 6, 13] can create a so called *profile*. It consists of aggregated statistics of performance metrics like runtime and/or number of visits for every function or call path and/or for every thread. These statistics give an overview over the execution.

Another approach is to record all events, e.g. function entry/exit, and its associated performance data like timestamps in a so called *trace*. This allows a fine grained analysis of the application. The user can visualize the trace directly, e.g., with Vampir [8] or Paraver [11]. However, manually searching the whole trace is very tedious. Another possibility, which is used by Scalasca [6], is to automatically examine an event trace for certain patterns of inefficient behavior. This search guarantees to cover the whole trace.

Figure 1 shows the performance analysis workflow with Scalasca. To instrument his application, a user must prefix the original compile and link command with the scalasca instrumenter. During application run, the events are recorded into a trace file. The Scalasca parallel trace analyzer analyses the trace and writes an analysis output which can be interactively explored in the CUBE graphical user interface (GUI). The CUBE GUI has three panes. The left panes allows to select a metric,

e.g. time waited, due to a late sender. The middle pane shows a call tree which provides information where in the code, a particular issue appeared. The right pane shows the distribution of values over all processes for the selected call path.

Scalasca uses a highly scalable event-trace replay mechanism for its analysis. Hereby, the trace for every process is stored in a separate file. The trace analyzer is launched as a separate program after the target application finished and runs on the same number of processes as the application ran. Every analysis process traverses the trace of one application process. Whenever the application process communicated with another process, Scalasca exchanges information, too. Furthermore, a backward replay is also possible.

In this paper, we will present extensions and new features to Scalasca. First, we added two new analysis methods to our automatic trace analysis. The first new method is called *root-cause analysis*, which identifies the root causes of wait states in MPI synchronization points. It is described in Sect. 2. Afterwards, Sect. 3 describes the second new analysis method, called *critical-path analysis*. It is able to detect inefficiencies that otherwise may be hidden through data aggregation.

Usually, profiling tool designs assume that the iterations of an iterative application behave basically the same. However, this is not generally true. Section 4 presents a new feature to analyze time dependent behavior.

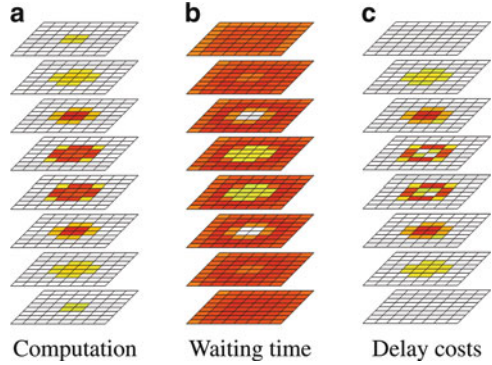
Section 5 describes the enhancements of the possibilities to define and display topologies.

## 2 Root Cause Analysis

So far, Scalasca's trace analysis could identify wait states at MPI synchronization points. Wait states, which are intervals through which a process is idle while waiting for a delayed process, are a primary symptom of load imbalance in parallel programs. The new root-cause analysis [2] also identifies the root causes of these wait states and calculates the costs of delays in terms of the waiting time that they induce.

A *delay* is the original source of a wait state, that is, an interval that causes a process to arrive belatedly at a synchronization point, causing one or more other processes to wait. Besides simple computational overload, delays may include a variety of behaviors such as serial operations or centralized coordination activities that are performed only by a designated process. The *costs* of a delay are the total amount of wait states it causes. Wait states can also themselves delay subsequent communication operations and produce further indirect wait states. This *propagation effect* does not only add to the total costs of the original delay, but also creates a potentially large temporal and spatial distance in between a wait state and its original root cause. The root-cause analysis closes this gap by mapping the costs of a delay onto the call paths and processes where the delay occurs, offering a high degree of guidance in identifying promising targets for load or communication balancing. Together with the analysis of wait-state propagation effects, the delay

**Fig. 2** Distribution of computation time, waiting time, and total delay costs in Zeus-MP/2 across the  $8 \times 8 \times 8$  three-dimensional computational domain. *Red colors indicate high values.* (a) Computation. (b) Waiting time. (c) Delay costs



costs enable a precise understanding of the root causes and the formation of wait states in parallel programs.

We applied the delay analysis to a variety of real-world MPI programs. One example is the astrophysics code Zeus-MP/2, where we studied the formation of wait states in a simulation of a 3D blast wave over 100 time steps on 512 processes. Around 12.5 % of the program’s total CPU allocation time is waiting time. Scalasca’s report browser can visualize the Cartesian process topology of a program, which we use in Fig. 2 to illustrate the relation between waiting and delaying processes in terms of their position within the computational domain. Obviously, there is a computational load imbalance between the central and outer ranks of the domain. Accordingly, the underloaded processes exhibit a significant amount of waiting time (Fig. 2b). Our analysis shows that about 70 % of the waiting time was indirectly caused by wait-state propagation. Examining the delay costs reveals that almost all the delay originates from the border processes of the central, overloaded region (Fig. 2c). The distribution of the workload explains this observation: Within the central and outer regions, the workload is relatively well balanced. Therefore, communication within the same region is not significantly delayed. In contrast, the difference in computation time between the central and outer region causes wait states at synchronization points along the border, which subsequently propagate towards the outer domain border. By pinpointing one subroutine and three computational loops with particularly high delay costs, the delay analysis also helped isolating the imbalanced source-code regions that lead to the wait states.

### 3 Critical Path Analysis

Our search for compact yet powerful means to uncover inefficiencies in parallel programs has led us to revisit the critical path as a key performance structure. Although the power and expressiveness of the critical path has been demonstrated in previous work, critical-path techniques only play a minor role in current

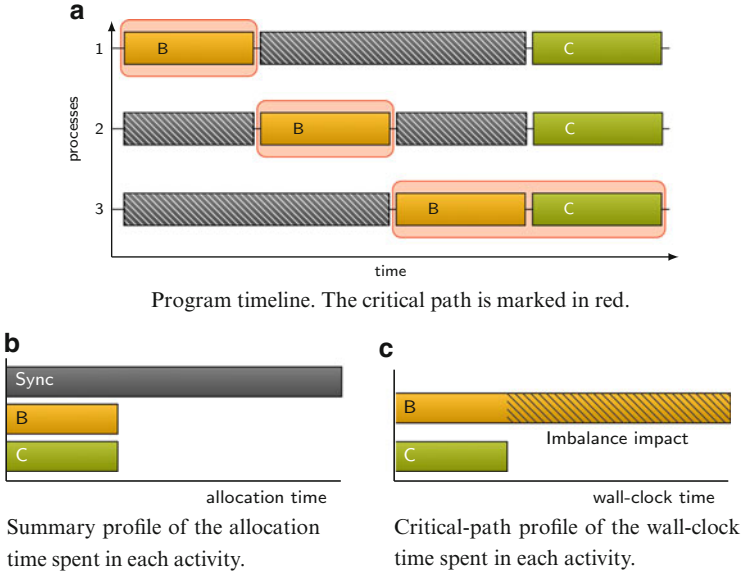
performance analysis tools. This arises partly from the difficulty in isolating the critical path, but also from the inability to extract intuitively accessible insight from the available information. Our work [3] addresses both issues. We leverage Scalasca's parallel trace replay technique to isolate the critical path in a highly scalable way. Also, instead of exposing the lengthy critical-path structure to the user in its entirety, we use the critical path to derive a set of compact performance indicators, which provide intuitive guidance about load-balance characteristics and quickly draw attention to potentially inefficient code regions.

The critical path is the longest path through a program activity graph that does not include wait states. Thus, it determines the length of program execution. Prolonging activities on the critical path increases program runtime, whereas shortening them (usually) reduces it. In contrast, optimizing an activity not on the critical path only increases waiting time, but does not affect the overall runtime.

Our critical-path analysis produces a critical-path profile, which represents the time an activity spends on the critical path. In addition, we combine the critical-path profile with per-process time profiles to create a *critical-path imbalance* performance indicator. This critical-path imbalance corresponds to the time that is lost due to inefficient parallelization in comparison with a perfectly balanced program. As such, it provides similar guidance as prior profile-based load imbalance metrics (e.g., the difference of maximum and average aggregate workload per process), but the critical-path imbalance indicator can draw a more accurate picture. The critical path retains dynamic effects in the program execution, such as shifting of imbalance between processes over time, which per-process profiles simply cannot capture. Because of this, purely profile-based imbalance metrics regularly underestimate the actual performance impact of a given load imbalance. As an extreme example, consider a program like the example in Fig. 3, where a function is serialized across all processes but runs for the same amount of time on each. Purely per-process profile based metrics would not show any load imbalance at all, whereas the critical-path imbalance indicator correctly characterizes the functions serialized execution as a performance bottleneck.

Both delay analysis and critical-path analysis are implemented as extensions to the automatic wait-state detection of the Scalasca performance analysis toolset, leveraging its scalable, post-mortem event-trace analysis. The analyzer traverses the traces in parallel, iterating over each process-local trace, and exchanges data required for the performance analysis at each recorded synchronization point using a communication operation similar to the one originally used by the program.

Other than the pure wait-state analysis, the delay and critical-path analysis require an additional, backward replay over the trace. A backward replay processes a trace backwards in time, from its end to its beginning, and reverses the role of senders and receivers. Overall, the analysis now consists of two stages: (1) a parallel forward replay that performs the wait state analysis and annotates communication events with information on synchronization points and waiting time incurred; and (2) a parallel backward replay that identifies the delays causing each of the wait states detected during forward replay, calculates their costs, and extracts the critical path. Starting at the endmost wait states, the backward replay allows delay costs

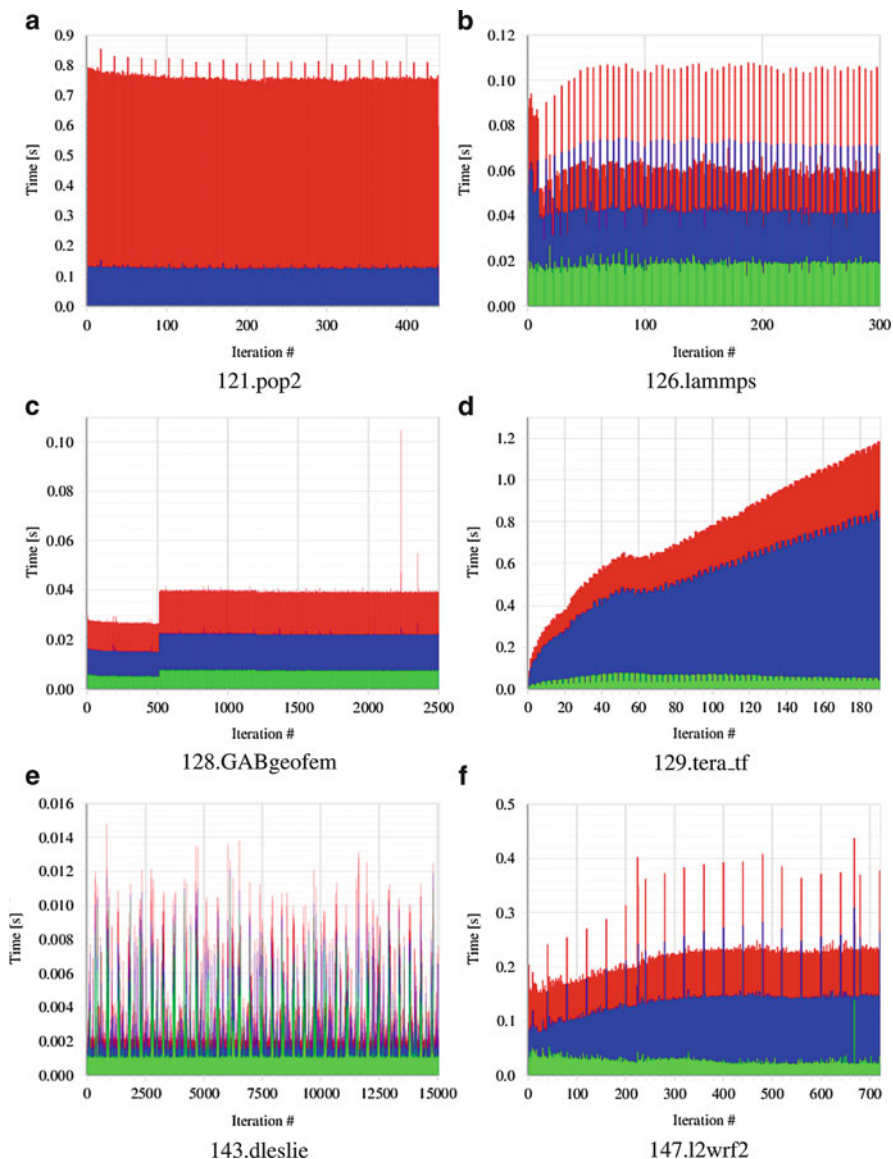


**Fig. 3** Analysis of dynamic performance effects: The serialized execution of function B, as seen in the timeline (a), goes unnoticed in a summary profile (b), but is correctly identified as a performance bottleneck by the critical-path imbalance indicator (c)

to travel from the point where they materialize in the form of wait states back to the place where they are caused by delays. The backward replay also facilitates the critical-path analysis, since the route of the critical path through the program cannot be determined without knowing the end of the execution. For MPI programs, the critical path runs between `MPI.Init` and `MPI.Finalize`. Our critical-path search begins by determining the MPI rank that entered `MPI.Finalize` last, which marks the endpoint of the critical path, and then exploits the lack of wait states on the critical path: whenever a wait state is found on the currently active path, the search proceeds on the MPI rank that caused the wait state. This way, we follow the entire critical path backwards through the trace.

## 4 Time-Series Profiling

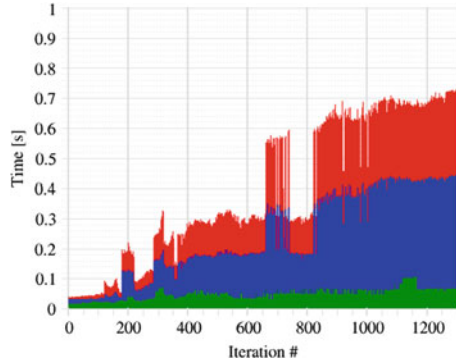
Call path profiling accumulate multiple visits of the same call path. If all visits of the same call path behave basically the same, all visits are well represented by the resulting statistics. For iterative applications, the user usually assumes very similar behavior of each iteration. However, this assumption is not always true. Szebebyi et al. [15] have shown on examples from the SPEC MPI benchmarks (see Fig. 4) and on the coulomb solver PEPC [16] (see Fig. 5) that some applications change their behavior for different iterations.



**Fig. 4** Runtime of iterations of the SPEC MPI benchmark codes 121.pop, 126.lammps, 128.GABgeofem, 129.tera\_tf, 143.dleslie, and 147.l2wrf2

To display time-dependent behavior of iterative applications, the Scalasca measurement system can now record separate profiles for every iteration of the main loop. For this purpose, a user can manually instrument the body of the main loop using the EPIK API. TAU [13] and Score-P [10] provide similar concepts named dynamic timer, or dynamic region, respectively.

**Fig. 5** Point-to point communication time of the PEPC code for each program iteration



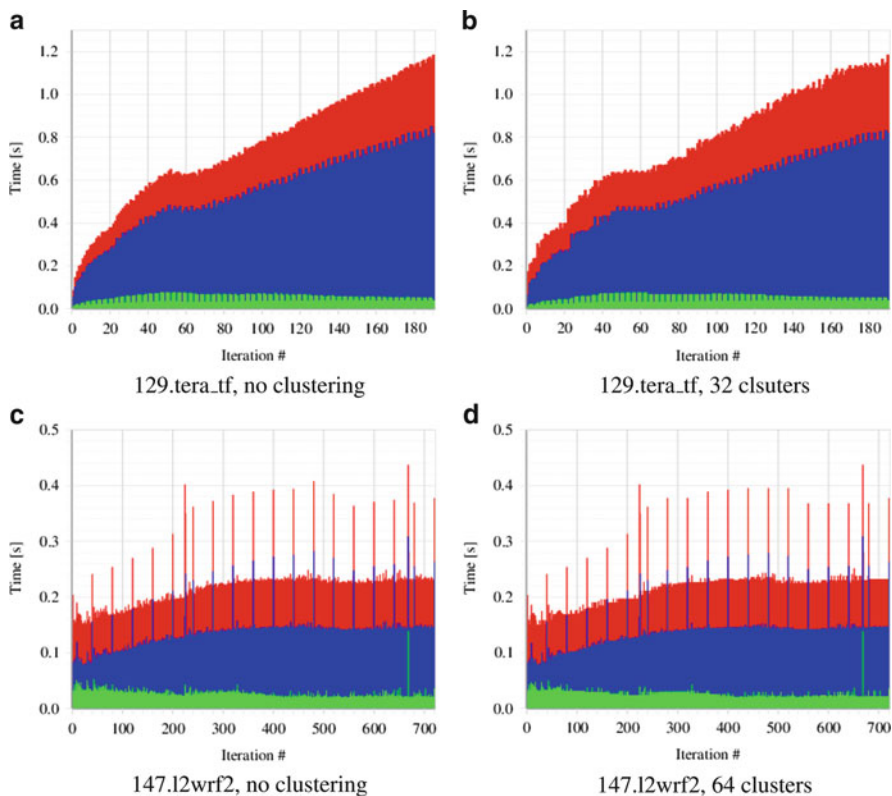
However, for applications with many iterations such a time-series profile may become very large. To reduce the memory requirements of the profile, we added a mechanism which clusters similar iterations to a single profile sub-tree instance [14]. The maximum number of clusters is configurable. However, we only cluster iterations with a similar structure. Iterations which contain different call paths are never clustered together. The time dynamics can be reconstructed from a mapping table, which stores the cluster associated with each iteration.

In case of the SPEC MPI benchmarks, the clustered profiles match the profiles without clustering very well, even if only 64 clusters are used. Figure 6 shows the comparison between the runtimes of iterations with clustering and without clustering. In the case of clustering the more complex PEPC coulomb solver, count based metrics show already a good match when using 64 clusters. However, time based metrics require more clusters for a good match [14].

## 5 Topologies

Scalasca obtains performance data for every process/thread of the application. In many cases, the hardware, the network, the communication system (e.g. MPI), or the application define neighborhood/dependency relationships and/or structure on these processes/threads, called topologies. However, network structure, hardware hierarchy, and application neighborhood relationships may significantly influence the performance of an application. Thus, we extended Scalasca's capabilities to record and display topologies. The previous capabilities of Scalasca to represent topologies were limited to 3-dimensional Cartesian topologies, i.e. each element has an unique set of coordinates in a 3-dimensional realm. Now, the topologies defined in Scalasca can have any number of dimensions. Furthermore, an application can define more than one topology, e.g., to compare the results for hardware topology and an algorithms domain topology.

Scalasca can work with the following types of Cartesian topologies:

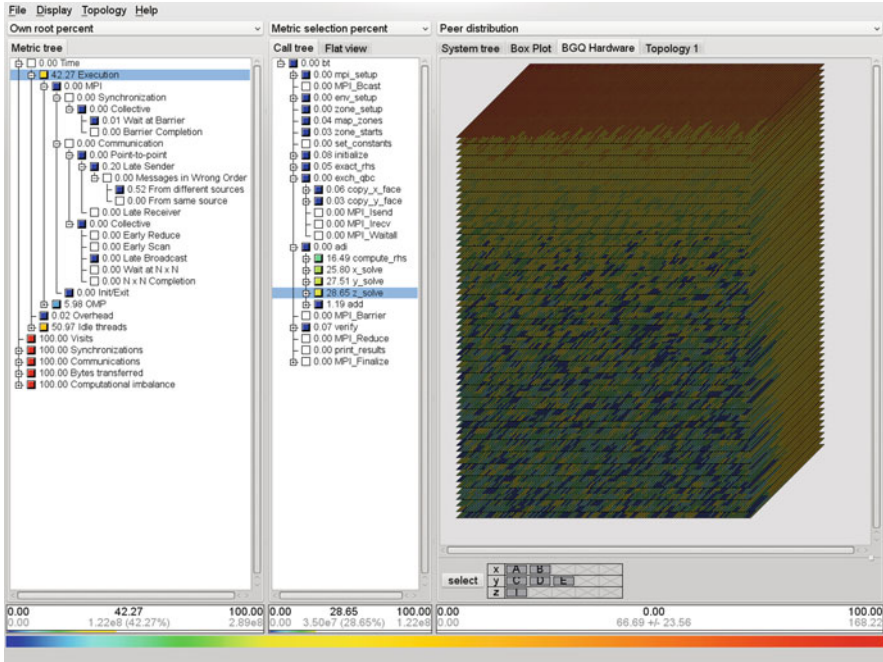


**Fig. 6** comparison between the runtimes of each iteration without clustering and with clustering for the SPEC MPI benchmarks 129.tera\_tf, and 147.l2wrf2. (a) 129.tera\_tf, no clustering. (b) 129.tera\_tf, 32 clusters. (c) 147.l2wrf2, no clustering. (d) 147.l2wrf2, 64 clusters

## 5.1 Hardware Topologies

Supercomputers, such as the IBM Blue Gene series, can report where on the machine processes run. This is useful in tightly-coupled programs, where distance information can provide insights on communication delays. This information is collected automatically by Scalasca during measurement.

Going further with the example of the Blue Gene series, the Blue Gene/Q model of supercomputer has a five-dimensional torus network, which is fully supported by Scalasca. Besides the support of topologies with more than 3 dimensions on measurement and analysis, Scalasca now has a “folding” mechanism which allows the user to select three-dimensional slices of the topology for visualization. Scalasca now also supports names for all the topologies and their dimensions, thus, the topologies can be easily identified. The names of the dimensions also are helpful to identify the individual processes/threads in complex hardware topologies such as the Blue Gene/Q’s (see Fig. 7).



**Fig. 7** CUBE showing the Blue Gene/Q's 5D Hardware Topology. The screenshot shows 524,288 threads on the Jülich BlueGene/Q machine

## 5.2 Processes X Threads

Scalasca now automatically creates a two-dimensional virtual topology that shows all OpenMP threads belonging to each process.

## 5.3 Runtime Mapping

On MPI programs, for instance, Scalasca can show how the processes are distributed in ranks and the relationship between them. This information is also collected automatically by Scalasca during measurement. Scalasca now supports any number of dimensions. In the future, topologies whose communicator was named using the `MPI_Comm_set_name` function will have this name shown in the topology's tab.

## 5.4 Algorithm Domain

The MPI specification does not enforce a strict mapping between neighbor ranks and the hardware, neither between ranks and the problem decomposition, which is

application specific. In some specific cases, the user might benefit from creating a virtual topology that is independent from the hardware and from the MPI mapping, being specific to the application. These topologies can be created manually by the user, using our API.

## 6 Future Work

As computational science evolves and new programming paradigms emerge, new challenges for performance analysis appear. Thus, we will continue to improve Scalasca. This section describes some of the ongoing developments and planned features for Scalasca.

First, we will replace Scalasca's native instrumentation and measurement system by Score-P [10]. Score-P is a common instrumentation and measurement system, initially used by Periscope [7], Scalasca [6], TAU [13], and Vampir [8]. This implies that these tools will also share common data formats for tracing and profiling. Scalasca will remain as a pure trace analyzer for traces written in the Score-P OTF2 [4] trace format. Score-P profiles and Scalasca trace analysis reports will be written in the CUBE4 format, which is the more scalable successor of the current CUBE3.

The common Score-P measurement stack improves the interoperability of the tools. Furthermore, it reduces the need for each tool to maintain its own instrumentation and measurement system, and thus, allows tool developers to focus on the specific analysis strengths of their tools.

With respect to future trace analysis enhancements, we plan to extend the current OpenMP analysis of Scalasca with the analysis of OpenMP tasks. Score-P can already record task events [9, 12]. However, we must extend Scalasca's profile construction algorithm and we want to add some task specific patterns to its analysis.

At the Barcelona Supercomputing Centre, a new tasking system was developed, named OmpSs [5]. We want to support measurements of applications using OmpSs. Therefore, we will implement instrumentation and measurement support for OmpSs in Score-P. Furthermore, we want to create the task analysis general enough that it covers also OmpSs tasks. A third new analysis feature that we are working on is the analysis of one-sided communication and PGAS languages.

## References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* **22**, 685–701 (2010). DOI <http://dx.doi.org/10.1002/cpe.v22:6>

2. Böhme, D., Geimer, M., Wolf, F., Arnold, L.: Identifying the root causes of wait states in large-scale parallel applications. In: Proc. of the 39th International Conference on Parallel Processing (ICPP, San Diego, CA, USA), pp. 90–100. IEEE Computer Society (2010). DOI 10.1109/ICPP.2010.18
3. Böhme, D., de Supinski, B.R., Geimer, M., Schulz, M., Wolf, F.: Scalable critical-path based performance analysis. In: Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China (2012)
4. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W., Wolf, F.: Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In: Proc. of the Intl. Conference on Parallel Computing (ParCo) 2011, Ghent, Belgium, *Advances in Parallel Computing*, vol. 22, pp. 481–490. IOS press (2012). DOI 10.3233/978-1-61499-041-3-481
5. Ferrer, R., Planas, J., Bellens, P., Duran, A., Gonzalez, M., Martorell, X., Badia, R., Ayguade, E., Labarta, J.: Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In: Languages and Compilers for Parallel Computing, *LNCs*, vol. 6548, pp. 215–229 (2011). URL [http://dx.doi.org/10.1007/978-3-642-19595-2\\_15](http://dx.doi.org/10.1007/978-3-642-19595-2_15)
6. Geimer, M., Wolf, F., Wylie, B., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* **22**(6), 702–719 (2010). DOI 10.1002/cpe.1556
7. Gerndt, M., Ott, M.: Automatic performance analysis with Periscope. *Concurrency and Computation: Pract. Exper.* **22**(6), 736–748 (2010). DOI 10.1002/cpe.1551
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M., Nagel, W.: The Vampir performance analysis tool set. In: Tools for High Performance Computing, pp. 139–155. Springer (2008)
9. Lorenz, D., Philippen, P., Schmidl, D., Wolf, F.: Profiling of OpenMP tasks with Score-P. In: Proc. of Third International Workshop of Parallel Software Tools and Tool Infrastructures (PSTI 2012), Pittsburgh, PA, USA (2012)
10. an Mey, D., Biersdorff, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S.S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A unified performance measurement system for petascale applications. In: Competence in High Performance Computing 2010 (CihPC), pp. 85–97. Gauß-Allianz, Springer (2012). DOI 10.1007/978-3-642-24025-6\_8
11. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVÉR: A tool to visualize and analyze parallel code. In: WoTUG-18: Transputer and occam Developments, pp. 17–31 (1995)
12. Schmidl, D., Philippen, P., Lorenz, D., Rössel, C., Geimer, M., an Mey, D., Mohr, B., Wolf, F.: Performance analysis techniques for task-based OpenMP applications. In: 8th Int. Workshop of OpenMP (IWOMP), *LNCs*, vol. 7312, pp. 196–209. Springer, Berlin / Heidelberg (2012)
13. Shende, S., Malony, A.D.: The TAU parallel performance system. *International Journal of High Performance Computing Applications* **20**(2), 287–331 (2006)
14. Szebenyi, Z., Wolf, F., Wylie, B.J.N.: Space-efficient time-series call-path profiling of parallel applications. In: Proc. of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA. ACM (2009). DOI 10.1145/1654059.1654097
15. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: SCALASCA parallel performance analyses of SPEC MPI2007 applications. In: Proc. of the 1st SPEC International Performance Evaluation Workshop (SPEW), Darmstadt, Germany, *Lecture Notes in Computer Science*, vol. 5119, pp. 99–123. Springer (2008). DOI 10.1007/978-3-540-69814-2\_8
16. Szebenyi, Z., Wylie, B.J.N., Wolf, F.: Scalasca parallel performance analyses of PEPC. In: Proc. of the 1st Workshop on Productivity and Performance (PROPER) in conjunction with Euro-Par 2008, Las Palmas de Gran Canaria, Spain, *Lecture Notes in Computer Science*, vol. 5415, pp. 305–314. Springer (2009). DOI 10.1007/978-3-642-00955-6\_35