Detection and Visualization of Performance Variations to Guide Identification of Application Bottlenecks

Matthias Weber, Ronald Geisler, Tobias Hilbrich, Matthias Lieber,
Ronny Brendel, Ronny Tschüter, Holger Brunst, and Wolfgang E. Nagel

Center for Information Services and High Performance Computing
Technische Universität Dresden, Germany
{*matthias.weber, tobias.hilbrich, matthias.lieber, ronny.tschueter, holger.brunst, wolfgang.nagel*}*@tu-dresden.de*

***Abstract*—The identification of performance bottlenecks in parallel applications is a challenging task. Without some form of performance measurement tool, this task lacks any guidance and purely relies on trial-and-error. At the same time, data sets from parallel performance measurements are often large and overwhelming. We provide an effective solution to automatically identify and highlight several types of performance critical sections in an application run. Our approach first identifies time dominant functions of an application that are subsequently used to analyze runtime imbalances throughout the application run. We then present the resulting runtime variations in an intuitive visualization that guides the analyst to performance hot spots. We demonstrate the effectiveness of our approach in a case study with three applications, detecting performance problems and identifying their root-causes in all cases.**

I. Introduction

Today's High Performance Computing (HPC) systems feature complex architectures that require software adaption and tuning to run codes efficiently. In this regard performance analysis of parallel applications presents a challenging task. The analysis process involves measurements to gather data for performance evaluation. Existing measurement suites offer different levels of detail. Measurements with low levels of detail record coarse-grain measurement samples that are averaged across application processes. Highly detailed measurements provide many measurement samples and retain all individual measurement points. The latter is well-suited to detect performance problems that vary across processes or over time, as well as to highlight root causes for several types of performance problems. While performance measurement tools exist to capture such detailed data at scale [5], [9], the analysis of such detailed measurement data remains challenging.

Existing analysis approaches either provide automatic search methods for specific problems [6], [21] or leave the analysis task completely to the analyst and provide a visualization of the data [3], [10], [13]. While automatic search methods are limited to certain types of problems, pure visualization approaches cover the widest range of possible detectable performance problems. This makes visual analysis methods powerful and compelling. At the same time they suffer from a more user driven identification of the performance problems. We provide a new approach for visualization-based performance analysis that identifies performance hotspots of an application execution.

Ideally, applications making use of structured parallelism should exhibit a regular runtime behavior throughout the complete run and across all processes. Meaning, that for instance the duration of iterations should be similar between processes as well as from the beginning to the end of the application. If some parts of an application run slower than the other parts, this might indicate a performance problem. We refer to such a situation with the terms *runtime imbalance* or *performance variation*. With our approach we detect such imbalances and highlight areas in an application run that exhibit notably higher runtime than others. Our performance measurement data sets are so-called program traces, which are time-sorted records of timestamped application behavior. Examples for performance relevant behavior include entering or leaving a function or sending a message from one process to another. In a parallel application, each processing element can create one such trace. To detect slow performing parts we first automatically identify time-dominant functions of an application. These functions are most suitable to detect runtime imbalances. To guide the analyst, we then analyze and visualize execution time variations of these function invocations throughout the application run. Applying our approach on top of established trace visualization methods allows analysts to benefit from the full potential of the underlying analysis system, as to detect the root cause of a performance problem. In summary our approach highlights performance critical parts during an application execution, and thus, guides the analyst directly to performance problems.

Our contribution includes:

- A method to automatically identify time-dominant functions of an application run;
- A scheme to calculate runtime imbalances using time-dominant functions to reveal performance critical areas;
- A visualization of runtime imbalances in a trace visualizer, thus, ultimately guiding analysts to performance problems; and
- An application study with three examples to showcase the effectiveness of our approach.

The remainder of this paper is organized as follows. In Section II we describe related work. Sections III-VI highlight our overall approach and methodology. In these sections we detail how we identify and visualize runtime imbalances. In the case study in Section VII we verify the validity of our approach for the detection of performance problems. We analyze several trace data sets and detect different performance problems and identify their root cause.

II. Related Work

A wide range of parallel performance analysis approaches exists. We provide an overview of their basic techniques and relate them to our approach.

Parallel profilers like TAU [2] and HPC Toolkit [1] show aggregated statistics of an application run. Such profiles are well suited for an overview of the performance behavior of an application. But due to aggregation, the detection of runtime imbalances and small slow sections can be hard or even impossible.

Trace viewers like Vampir [3], Paraver [17], and the Intel Trace Analyzer [10] visualize performance traces to guide the user to potential performance problems. Usually, summary information provides the guidance that allows the analyst to find interesting spots in timeline charts that visualize the overall trace data. Since these tools present full data sets, they can detect runtime imbalances and bottlenecks. However, due to the large amount of data, analysts may easily miss performance problems during their analysis. To provide scalable visualization, Mohror et al. [13] propose a technique to compare event streams of processes. The approach takes structurally equal processes and compares their temporal behavior. If the temporal behavior of these processes is sufficiently similar, only one representative is stored to simplify the visualization. However, by basing the analysis on only a few representative processes, performance problems may easily be hidden. Additionally, root causes, such as high inter-process communication latency on a specific process, may not be included in the selected representatives.

Weber et al. [20] propose metrics to compare traces using alignment techniques. These techniques highlight performance differences between different application runs. At the same time, the approach fails to highlight differences between processes inside a single run. An extension of the Paraver tools suite [7] targets a tool that characterizes computation phases. It clusters these periods with a common clustering algorithm. The result is a classification of phases that differ in instructions-per-second rates. While this approach provides an overview of the different performance characteristics of computation phases, it does not highlight individual variations within processes.

A range of performance tools also employ automatic analysis techniques for performance problems. Scalasca [21] automatically searches trace data for a range of inefficiency patterns. Located patterns are ranked by their severity and impact on the application performance. This approach saves the analyst from manually analyzing the trace data, but it is also restricted to a limited set of performance problems. The tool Periscope [6] follows an online monitoring approach. Application performance is analyzed while the application executes. The tool iteratively applies tests for performance critical behavior. Since the tool does not check for all performance problems at the same time, it may miss detectable problems in the application. Both approaches, Scalasca and Periscope, do not visualize runtime imbalances over time in order to detect performance bottlenecks.

To the best of our knowledge no tool provides detection and visualization of fine-grained performance variations throughout the application execution.

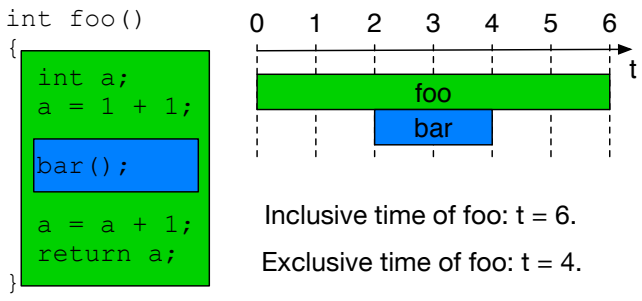


Figure 1. Inclusive vs. exclusive time of a function invocation.

III. Methodology Overview

We apply our approach to traces of a parallel application. Profiles do not suffice since we require information on runtime variations of application functions. Additionally, our visualization targets timeline visualizations that also apply to traces only. For our prototype implementation we use traces from Score-P [11] and VampirTrace [15]. We require no modifications for their measurement and our technique also applies to traces from other parallel performance analysis suites, such as Extrae [4].

We apply our technique after an application run successfully created a trace. In-situ analysis while the target application is still running is feasible as well, but the performance analysis suite that we use for our prototype does not support such a workflow. Using the trace as input we perform three steps for our technique:

1) Identification of time-dominant functions that are used to partition the complete run into small segments[1],
2) Computation of performance variations between these function invocations (segments), and
3) An intuitive visualization to present the overall result.

[1]As we use invocations of the time-dominant function as segments, the inclusive time of the dominant function invocation equals the respective segment duration.

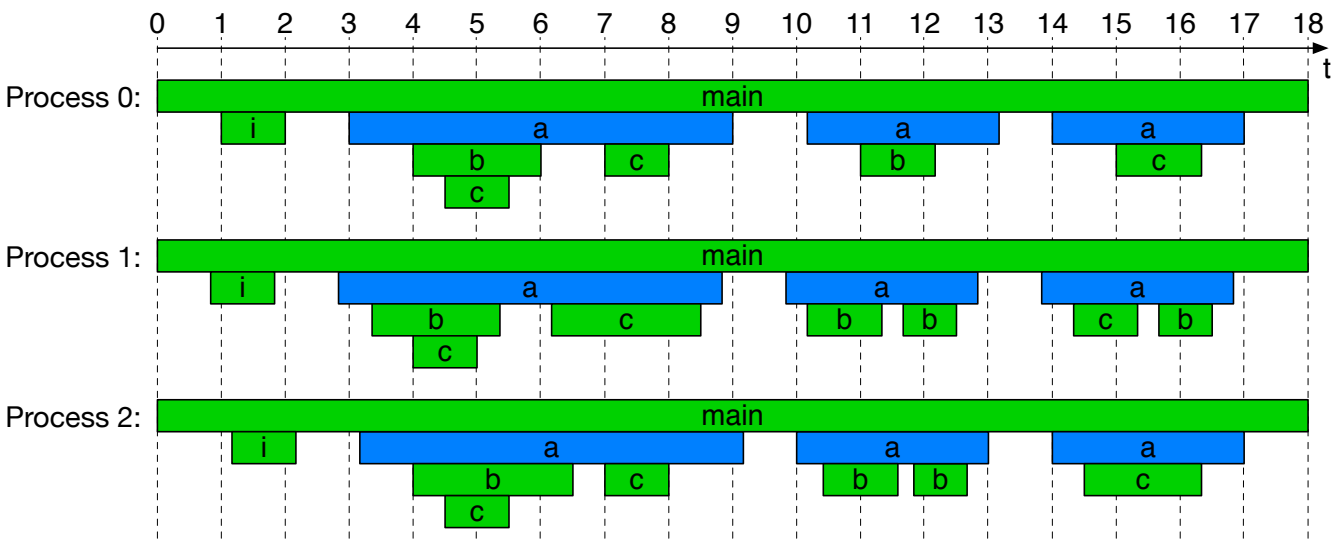


Figure 2. For our performance variation analysis we select one time-dominant function. The selected function needs to have a possibly high inclusive time and an invocation count higher than the number of processes. In this example function a fulfills these criteria.

IV. Identification of Time-Dominant Functions

We need to identify reoccurring parts of an application in order to detect performance variations during the application run. Reoccurring parts allow us to separate an overall execution into multiple segments. Afterwards, we can compare runtimes between the segments. Thus, the first step of our analysis is a selection of suitable segments to highlight performance variations. Since parallel applications usually execute functions repeatedly, as they are being called in loops, we select such a function for our segments. Note that the measurement systems that we use also support instrumentation of loop bodies. If such an instrumentation is in use, we can use loop bodies as well. For simplicity we use the term *function* in the following. To decide which particular function we select as segments, we consider their *inclusive* times.

When measuring a function's invocation, there are two options to report the function's duration: *inclusive* and *exclusive* time. Figure 1 depicts the difference between inclusive and exclusive time. Inclusive time represents the complete duration of a function's invocation, from initially entering to finally leaving the function call. This time also includes the time spent in sub-functions. The inclusive time for function `foo` in Figure 1 starts with entering `foo` ($t = 0$) and stops when `foo` is left ($t = 6$). The inclusive time includes the sub-call to function `bar` and is 6 in the example. The exclusive time, on the contrary, represents only the amount of time spent directly inside the respective function's invocation, excluding sub-functions. The exclusive time of function `foo` in Figure 1 starts with entering `foo` at $t = 0$, excludes the sub-call of function `bar` ($t = 2$ to $t = 4$), and ends with leaving of `foo` at $t = 6$, i.e., it is 4 in the example.

We consider inclusive time to detect dominant functions, since it includes the overall performance impact of a function. Figure 2 illustrates an example that uses three processes and the functions `main`, `i`, `a`, `b`, as well as `c`. A time-dominant function should have a considerable impact on the total application runtime. For that we consider the aggregated inclusive time of each function. Selecting the function with the highest aggregated inclusive time, however, is not a good choice for a dominant function. In the example, this would be the function `main` (54 time steps). Such top call-level functions may be suitable to compare the runtime between processes, but they are not suited to analyze variations over the runtime. Additionally, they provide no segmentation of the overall runtime. Thus, just maximum aggregated inclusive time is not a good selection criteria alone.

As a consequence, for a time-dominant function we select a function with high aggregated inclusive time, but which also features a higher number of invocations. In the example of Figure 2, the function with the highest inclusive time share is `main`. The function `main` is called three times on the three processes in total. Thus, top call-level functions like `main` have exactly as many invocations as there are parallel processing elements. Thus, we define a time-dominant function f as:

- For p processing elements, f is invoked at least $2p$ times and there exists no other function that satisfies this condition and has higher aggregated inclusive time.

In the example, the function with the second highest inclusive time share is `a` (36 time steps). Function `a` is called nine times on three processes, i.e., it satisfies the invocation count restriction. Hence, `a` is the time-dominant function for the example.

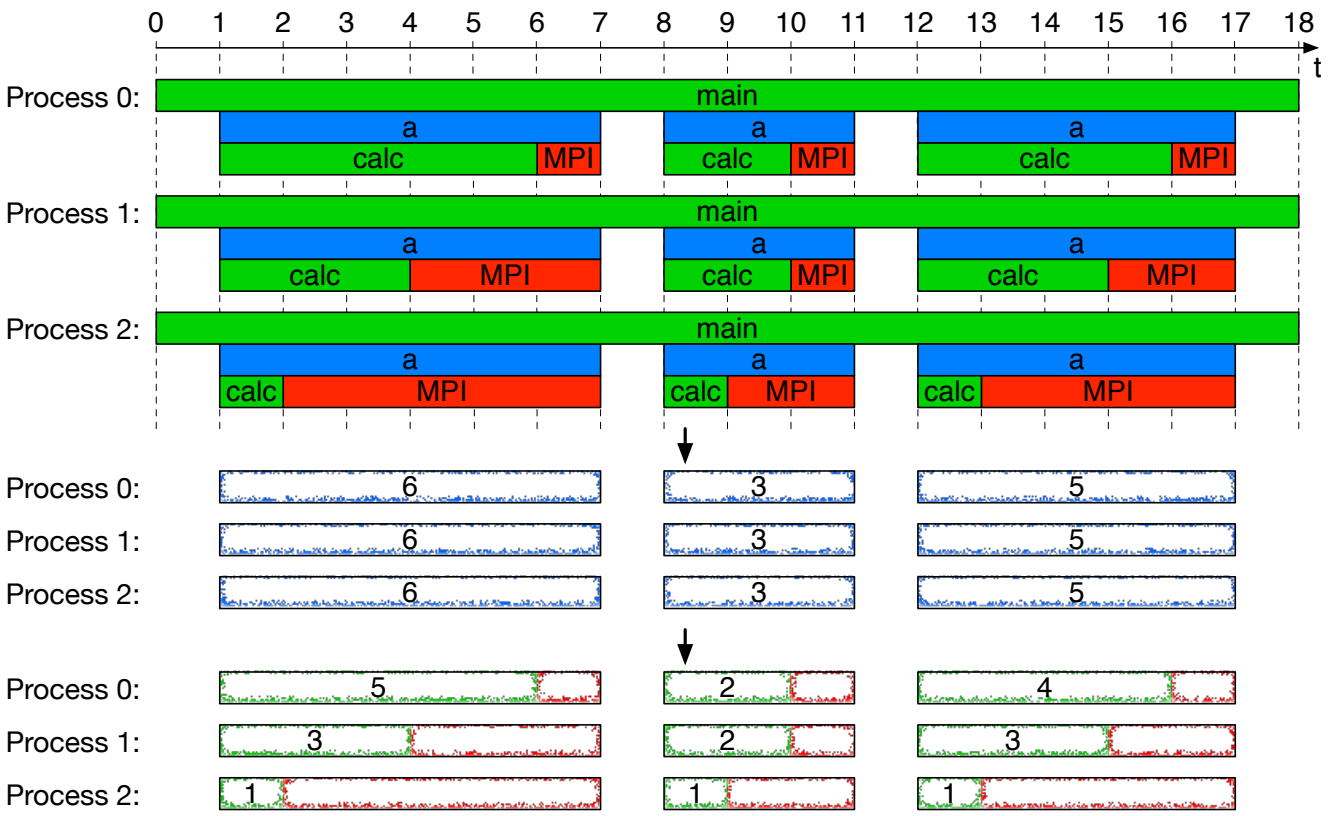


Figure 3. Calculation of performance variations. First, shown in the middle, we calculate segment durations (inclusive time of function a in this example). Then, shown at the bottom, we subtract synchronization time from the segment durations to compute our synchronization-oblivious segment time (SOS-time).

While our selection criterion is heuristic, the subsequent use cases demonstrate that it provides functions that represent iterative application behavior well in practice. Comparing the runtime of such functions, and thus, analyzing the durations of iterations provides a solid foundation to detect a wide range of performance hotspots.

V. Analysis of Runtime Imbalances

With a selected time-dominant function we can compare the runtimes of its individual invocations. This comparison highlights shifts in runtime behavior over time and identifies runtime imbalances. For instance, if an application runs gradually slower, the inclusive time of a good dominant function will usually increase as well over time. Also, outlier-iterations with exceptionally long runtime will impact the inclusive time of a dominant function.

The comparison of inclusive times of dominant functions alone has a shortcoming: In many applications, synchronization calls are also included in iterative function behavior. Figure 3 illustrates this behavior with the communication function named `MPI`. As we previously highlighted, our heuristic selects `a` as the dominant function for this example. If we directly compare the inclusive time of `a`'s invocations, we receive the results shown in the middle of Figure 3. The iterations in the middle (duration of 3) are twice as fast as the first iteration (duration of 6). This analysis already detects performance variation across iterations. However, if some iterations show a differing behavior from others, there are often only a few processing elements that cause this behavior. With the direct comparison of dominant function durations, we cannot identify the processes that cause the differences. The reason is often, that synchronization between processes is included in the iteration code. For instance, consider the example in Figure 3. In each iteration all processes first run a calculation (function `calc`) and then call an MPI [14] synchronization operation, e.g., `MPI_Barrier` (indicated as function `MPI`). *Process 2* in Figure 3 completes its calculations faster than *Process 0*. As a consequence, the MPI synchronization call in *Process 2* runs longer, as this process is waiting for *Process 0* to finish. The difference of the calculation part between the processes remains hidden, since the synchronization wait time is included in the inclusive time for the dominant function. Therefore, we adjust our calculation method to cover performance variations between segments. Instead of directly using each segment's duration, we subtract any synchronization time from its inclusive time. The synchronization time can be easily detected if the application uses common parallelization

libraries like MPI [14], OpenMP [16], or similar. In such case, we check each segment for synchronization operations, e.g., `MPI_Wait`, `MPI_Reduce`, or `omp barrier`, and subtract their runtime from the inclusive time of our dominant functions. We refer to this adapted segment time as *synchronization-oblivious segment time* (SOS-time) and use it as measure for runtime imbalances. Figure 3 (bottom) depicts this process for the example. Our SOS-times correctly reflect the performance differences between the processes. For instance, for the first iteration in Figure 3 the SOS-time of *Process 2* shows 1 compared to a SOS-time of 5 for *Process 0*, i.e., it highlights the computational load imbalance in the first iteration.

VI. Visualization of Runtime Imbalances

As the last step of our approach, we visualize our SOS-times to the analyst. Therefore, we implemented our analysis methods in the Vampir performance analysis framework [3]. To achieve an intuitive visualization, we overlay commonly used timeline views of Vampir. We use the SOS-times as values for a new metric counter. For our visualization we encode the metric with a color-coded scale. Blue—cold—colors indicate short durations, whereas red—hot—colors indicate long durations. Figures 4(b), 5(b), 5(c), and 6(b) in the subsequent section present examples of our visualization.

VII. Case Study

In this section we demonstrate the applicability of our approach with three use cases. To initially measure application performance data we use traces from the Score-P [11] and VampirTrace [15] measurement frameworks. As we illustrated in the methodology overview, our approach requires no modifications at measurement time. We implement our analysis as part of the Vampir [3] analysis and visualization toolkit. In the following, we analyze trace files with known performance problems to demonstrate the capabilities of our technique.

A. Load Imbalance - COSMO-SPECS

The first case study is an analysis of the execution of a weather forecast code [8]. The code couples two models, *COSMO* and *SPECS*, for a more accurate simulation of cloud and precipitation processes. COSMO is the regional weather forecast model originally developed at the German Weather Service (DWD). SPECS is a detailed cloud microphysics model developed at the Leibniz Institute for Tropospheric Research (IfT). SPECS computes detailed interactions between aerosols, clouds, and precipitation.

Figure 4(a) shows the Vampir timeline visualization of the overall application run. The execution under study uses 100 MPI processes that Vampir each represents with a horizontal bar. The colors then identify the currently active functions across the overall execution time. Red identifies MPI activities, purple SPECS activities, green COSMO activities, and yellow highlights the coupling between the two models. Compared to COSMO, the SPECS calculations are significantly more compute intensive. Therefore, purple areas—SPECS code—dominate the application run. The execution of COSMO code—green areas—is barely visible in Figure 4(a). This behavior is caused by the computational demand of the underlying physics. However, Figure 4(a) also shows another trend. Throughout the execution, the fraction of MPI—red areas—increases, up to a point where MPI activities are dominating towards the end of the run. Our heuristic selects a dominant function whose occurrences represent individual iterations. If we compare the plain inclusive time of this function (segment durations), we observe gradually increased durations towards the end of the application run.

To find the cause of the degrading performance, we use our synchronization-oblivious segment time (SOS-time) from Section III. Figure 4(b) presents this metric and highlights that only a few processes (*Process 44*, *45*, *54*, *55*, *64*, *65*) exhibit increases in this metric. Particularly *Process 54* needs more time than any other process for its calculations.

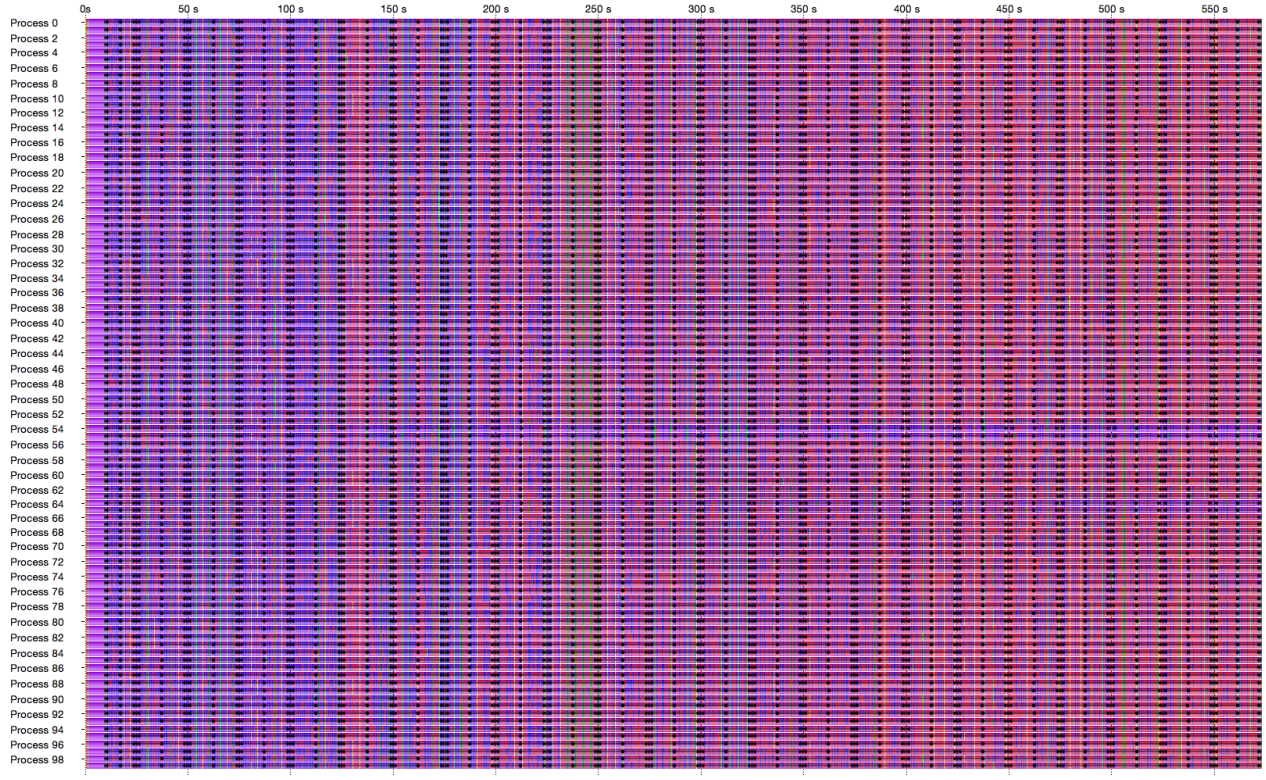
The reason for this behavior is a static decomposition of the computational grid. COSMO employs a two dimensional (horizontal) decomposition into $M \times N$ domains and applies no dynamic load balancing. SPECS uses the same data structures and decomposition as the COSMO model, but instead computes cloud microphysics. During the execution, SPECS introduces large load imbalances, since its computational cost heavily depends on the presence and size distribution of various cloud particle types in the grid cell [12]. Thus, the layout of clouds in the application domain determines the local work. In other words, while *Process 54* still performs cloud microphysics calculations, the other processes idle while waiting for it to finish. A solution to this performance problem is to introduce dynamic load balancing for the SPECS model.

Our analysis and visualization correctly represents the performance situation of the application. By following the high—red—values the analyst is pointed directly to the cause of the performance bottleneck.

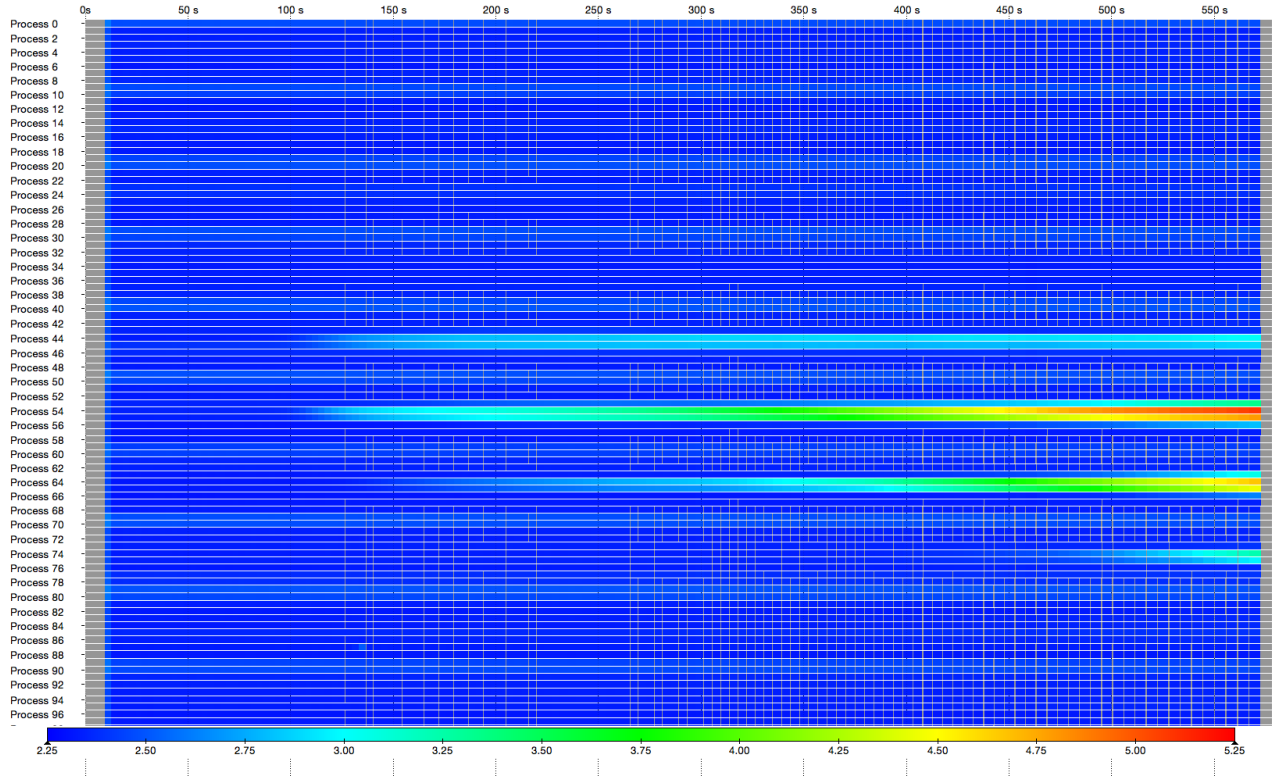
B. Process Interruption - COSMO-SPECS+FD4

In this case study we analyze an extended version of the previous weather forecast code [8]. In this version the developer has added a dynamic load balancing mechanism, called FD4 [12], to the SPECS model. As described in the first case study, the high computational demand of the SPECS code, combined with its high dependence on local workload—presence of cloud particles in the domain—demand a dynamic load balancing for efficient computation.

The application run under study uses 200 MPI processes. Our initial analysis—not shown—detected that only a few iterations behaved differently and exhibited larger durations than other iterations. The goal of this study is to detect the

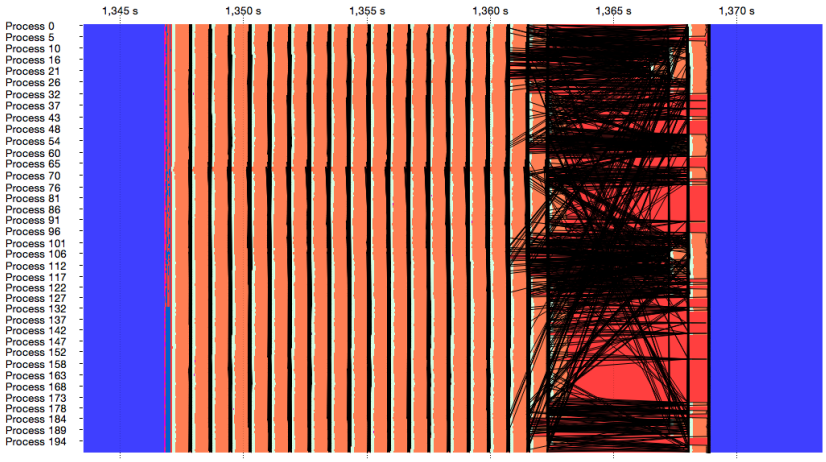


(a) Timeline visualization of COSMO-SPECS running on 100 processes, showing increasing MPI durations (red areas) over time.

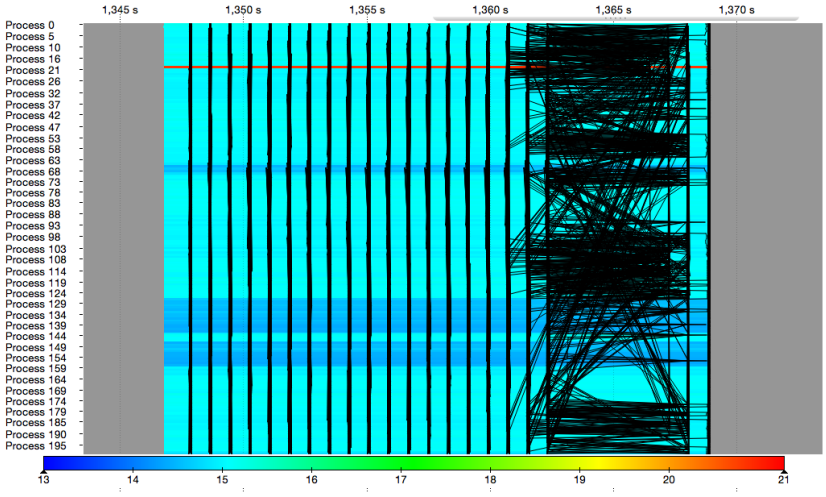


(b) Runtime variation analysis result. Several processes (middle) exhibit higher runtimes (SOS-time) in their dominant function.

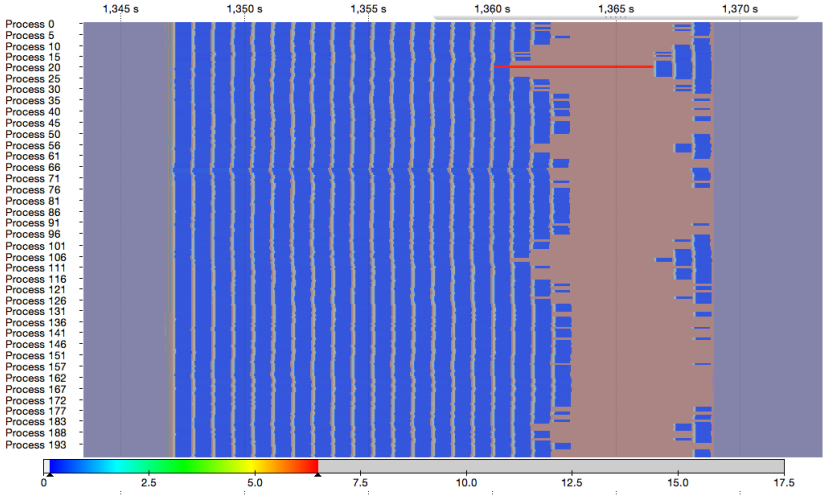
Figure 4. Analysis of the COSMO-SPECS weather forecast code. (a) shows the timeline visualization. (b) shows our analysis results.



(a) Timeline visualization of COSMO-SPECS+FD4 running on 200 processes.

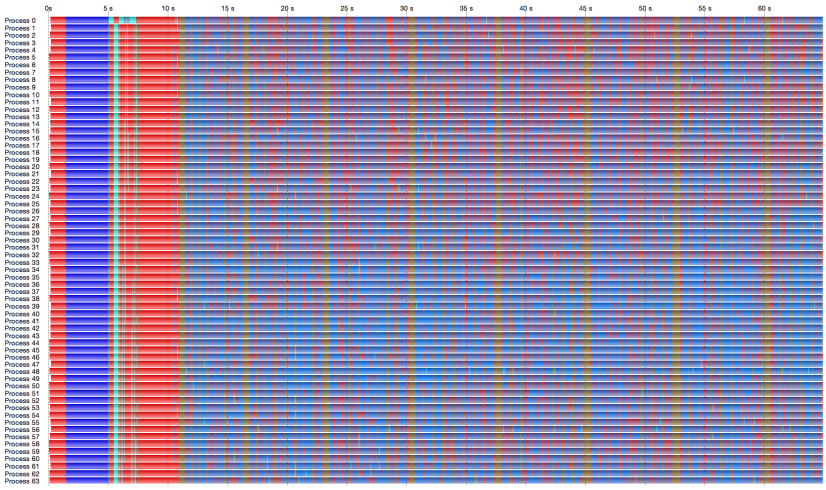


(b) Coarser runtime variation analysis result (SOS-time). Especially *Process 20* exhibits a high duration in its dominant function.

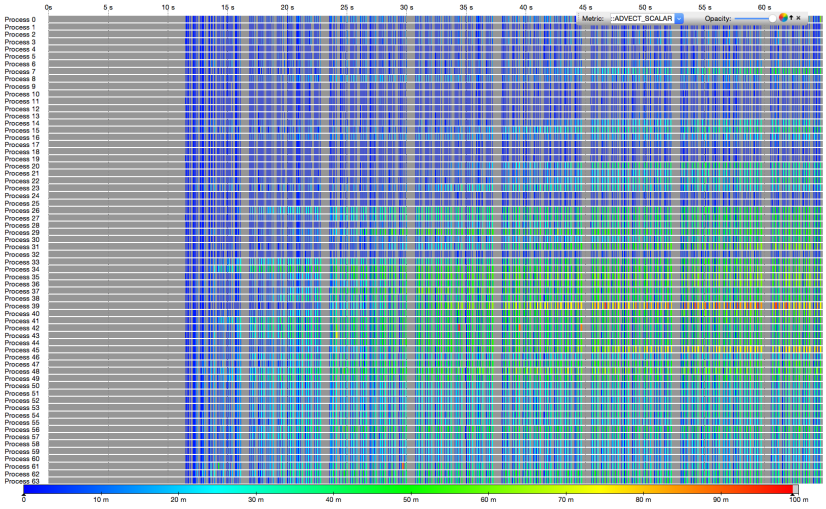


(c) Finer runtime variation analysis result (SOS-time). Using smaller segments sizes allows direct identification of the one function invocation that causes the performance degradation.

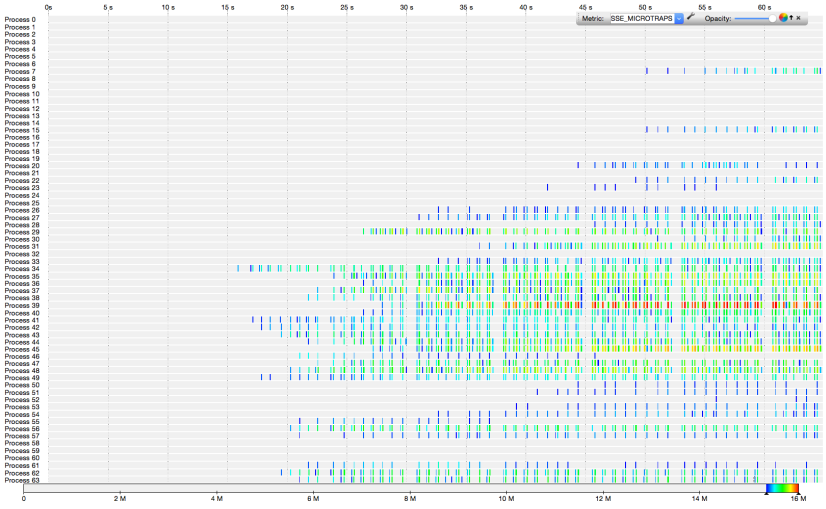
Figure 5. Analysis of a COSMO-SPECS+FD4 application run. Displayed is just one iteration. (a) shows the timeline visualization. (b) (coarser segments) and (c) (finer segments) show our variation analysis results using SOS-time.



(a) Timeline visualization of WRF running on 64 processes.



(b) Runtime variation analysis. Especially *Process 39* exhibits high SOS-times in its segments.



(c) Values of the counter `FR_FPU_EXCEPTIONS_SSE_MICROTRAPS`. *Process 39* shows a high number of floating-point exceptions.

Figure 6. Analysis of a WRF application run. (a) shows the timeline visualization. (b) shows our variation analysis results and (c) shows values the floating-point exceptions counter.

reason for these slow iterations. Therefore, the analyst used a second measurement run to only record slow iterations. For normal iterations the analyst discarded the tracing data. We show the timeline visualization of one slow iteration in Figure 5(a). Again, different colors represent different activity types. Red relates to MPI code, blue indicates areas where performance data was dropped, while orange and white areas relate to SPECS activities. The black lines indicate MPI messages sent from one process to another. The runtimes of COSMO and FD4 are so short compared to SPECS that these areas are not directly visible in Figure 5(a). Looking at the behavior in Figure 5(a), we see that one SPECS timestep near the end of this iteration takes significantly longer than the others. Especially, increased MPI wait time—more red areas—and higher message transfer times—longer black lines—indicate this behavior. However, the reason causing the slower timestep in this iteration is not immediately visible.

By using our runtime variation analysis, we can guide the analyst to the cause of this performance problem. We show the result of our analysis in Figure 5(b). The red line in the figure highlights a high SOS-time for *Process 20*. Thus, the performance problem is caused by longer computation time of *Process 20*. To find the exact place of the performance problem, we can refine granularity by adapting the dominant function. By choosing a function with a smaller inclusive time we achieve a more fine-grained segmentation. This option is beneficial to track the origin of a performance problem. We show the result of the finer segmentation in Figure 5(c). This figure clearly shows a single function call—red line—that runs significantly longer than all other invocations—blue lines—of this function. A closer inspection of *Process 20* shows, that this single function call exhibits a low number of total assigned CPU cycles (measured with the PAPI counter PAPI_TOT_CYC [19]). Subsequently, *Process 20* has been interrupted exactly during the execution of this function's invocation. The cause for the interruption is assumed to be an influence from the operating system.

Using our runtime variation analysis, the analyst is directly pointed to the performance bottleneck. Without an extended search, the subsequent analysis can be focused directly on the hotspot and quickly reveal the cause of the performance problem.

C. Floating-Point Exceptions - WRF

In this case study we analyze an application run of the *Weather Research and Forecasting model* (WRF), with a standard benchmark case (12km CONUS) [18]. The application under study uses 64 MPI processes. Figure 6(a) presents Vampir's basic timeline visualization. Red areas relate to MPI activities. Blue areas relate to computations of the *dynamical core* of WRF. These parts of the application compute for instance density, temperature, pressure, and winds in the atmosphere. Brown areas relate to the *physical parameterization* calculations of WRF. For instance clouds, rain, and radiation are computed in these parts.

In the early parts of the run (left of Figure 6(a)) the application executes model initialization and I/O activities that take about 11 seconds. Afterwards, the actual iterations begin. Basic Vampir statistics for the iterations show a 25% fraction of MPI activities, which highlights a noticeable parallelization overhead. The timeline view in Figure 6(a) does not present an immediate cause for this overhead.

In Figure 6(b) we visualize the SOS-time of the dominant function of the application run. The segments located in the lower right part in the figure highlight increased durations. Particularly *Process 39* exhibits higher durations than the other processes. A closer inspection supports this immediate result: *Process 39* computes slower and causes the other processes to wait. Based on hints that floating-point intensive functions compute slower, the analyst found that a high number of floating-point exceptions slows down *Process 39*. For validation we show the values of the counter `FR_FPU_EXCEPTIONS_SSE_MICROTRAPS` color-coded in Figure 6(c). As shown in the figure, *Process 39* exhibits an exceptional high number—red areas—of floating-point exceptions. Moreover, comparing Figure 6(b) and Figure 6(c) we see, that the results of the counter `FR_FPU_EXCEPTIONS_SSE_MICROTRAPS` perfectly match our runtime variation analysis.

This shows, that our approach correctly depicts the application performance behavior. By following our visualization, the analyst is guided closely to the performance issue. If necessary, focused subsequent analyses then reveal the root cause of the performance problem.

VIII. Conclusions

We present an effective and lightweight approach to facilitate visual analysis of performance data. Our approach guides the analyst directly to performance bottlenecks. We identify functions that are reoccurring and have a substantial impact on the overall runtime of an application first. Then, we calculate an implicit runtime for these functions that excludes communication and synchronization costs. Our visualization of this synchronization-oblivious implicit time highlights performance variations. For parallel applications that must strive to achieve good load balance, this metric efficiently highlights a wide range of load balancing problems. Since we rely on timestamped traces of performance data, we can also efficiently highlight behavior that changes over time. We compute and visualize our metric as part of the Vampir performance framework.

Our analysis method—performance variations during an application's execution—proves to be a viable approach for the detection of performance hotspots. In three analysis case studies we demonstrate the effectiveness of our approach by locating performance bottlenecks in the application runs. Since our methods directly identify the location of the

performance problem, we enable focused subsequent analysis to find the underlying root-cause of the problem.

Effectively, our methods support the performance analyst in helping him to focus on performance problems faster. We save the analyst from long analysis sessions, manually searching for performance problems.

REFERENCES

[1] 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.

[2] R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer Berlin Heidelberg, 2003.

[3] H. Brunst and M. Weber. Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite. In *Proceedings of the 6th International Parallel Tools Workshop*, pages 95–114. Springer Berlin Heidelberg, September 2012.

[4] Extrae User Guide. https://www.bsc.es/computer-sciences/performance-tools/trace-generation/extrae/extrae-user-guide, Jan. 2016.

[5] M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, and B. J. N. Wylie. Further improving the scalability of the Scalasca toolset. In *Proc. of PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable tools for High Performance Computing, Reykjavik, Iceland, June 6–9 2010*, volume 7134 of *Lecture Notes in Computer Science*, pages 463–474. Springer, 2012.

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

[7] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic Detection of Parallel Applications Computation Phases. In *Parallel & Distributed Processing. IPDPS 2009. IEEE International Symposium on*, pages 1–11, 2009.

[8] V. Grützun, O. Knoth, and M. Simmel. Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM-SPECS: Model description and first results. *Atmospheric Research*, 90(24):233–242, 2008.

[9] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole. Enabling Event Tracing at Leadership-class Scale Through I/O Forwarding Middleware. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 49–60, New York, NY, USA, 2012. ACM.

[10] Intel Trace Analyzer and Collector. http://software.intel.com/en-us/articles/intel-trace-analyzer/, Nov. 2015.

[11] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer Berlin Heidelberg, 2012.

[12] M. Lieber, V. Grützun, R. Wolke, M. S. Müller, and W. E. Nagel. Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4. In *Proc. PARA 2010*, volume 7133 of *LNCS*, pages 131–141, 2012.

[13] K. Mohror, K. L. Karavanic, and A. Snavely. Scalable Event Trace Visualization. In *Proceedings of the 2009 international conference on Parallel processing*, Euro-Par'09, pages 228–237, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] MPI: A Message-Passing Interface Standard, Version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[15] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September 2007*, pages 637–644, 2007.

[16] OpenMP. http://openmp.org/wp/openmp-specifications, Nov 2015.

[17] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, pages 17–31, March 1995.

[18] G. Shainer, T. Liu, J. Michalakes, J. Liberman, J. Layton, O. Celebioglu, S. A. Schultz, J. Mora, and D. Cownie. Weather Research and Forecast (WRF) Model Performance and Profiling Analysis on Advanced Multi-core HPC Clusters. In *10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[19] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer Berlin Heidelberg, 2010.

[20] M. Weber, K. Mohror, M. Schulz, B. R. de Supinski, H. Brunst, and W. E. Nagel. Alignment-Based Metrics for Trace Comparison. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 29–40. Springer-Verlag, Berlin, Heidelberg, 2013.

[21] F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore, M. Pfeifer, and Z. Szebenyi. Usage of the SCALASCA Toolset for Scalable Performance Analysis of Large-Scale Parallel Applications. In *Proceedings of the 2nd Parallel Tools Workshop, Stuttgart, Germany*, pages 157–167. Springer, July 2008.