Performance Tuning in the Petascale Era

Felix Wolf[1,2], **David Böhme**[1,2], **Markus Geimer**[1], **Marc-André Hermanns**[1], **Bernd Mohr**[1], **Zoltan Szebenyi**[1,2], **and Brian J. N. Wylie**[1]

[1] Forschungszentrum Jülich,
Jülich Supercomputing Centre, 52425 Jülich, Germany
E-mail: {f.wolf, d.boehme, m.geimer, m.a.hermanns, b.mohr, z.szebenyi, b.wylie}@fz-juelich.de

[2] RWTH Aachen University,
Graduate School AICES, 52056 Aachen, Germany

Driven by application requirements and accelerated by current trends in microprocessor design, the number of processor cores on modern supercomputers grows from generation to generation. As a consequence, supercomputing applications are required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. However, writing code that runs efficiently on large numbers of processors remains a significant challenge. The situation is exacerbated by the fact that the rising number of cores imposes scalability demands not only on applications but also on the software tools needed for their development.

To address this challenge, the Helmholtz University Young Investigators Group *Performance Analysis of Parallel Programs* at Jülich Supercomputing Centre (JSC) in cooperation with the JSC Division *Application Support* 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 tens of thousands of processors. In this article, we highlight the research activities of our group during the past two years and give an outlook on future work.

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 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.

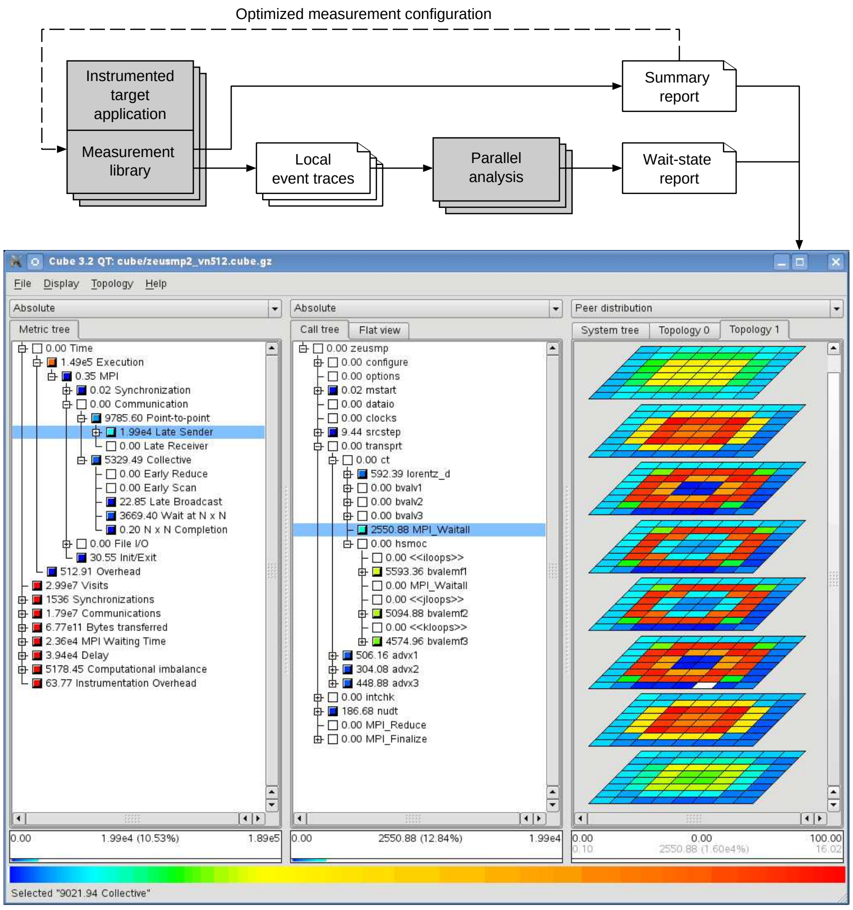
As the current trend in microprocessor development continues, this need will become even stronger in the future. Facing increasing power dissipation and little instruction-level parallelism left to exploit, computer architects are realizing further performance gains by using larger numbers of moderately fast processor cores rather than by further increasing the speed of uni-processors. As a consequence, supercomputer applications are being required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. With an exponentially rising number of cores, the often substantial gap between peak performance and the performance 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 displays, or limited I/O performance).

To overcome this challenge, the Helmholtz University Young Investigators Group *Performance Analysis of Parallel Programs* at Jülich Supercomputing Centre (JSC) in cooperation with the JSC Division *Application Support* creates software technologies aimed at improving the performance of applications running on leadership-class systems with tens of thousands of cores. At the center of our activities lies the development of Scalasca[1], 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 very large processor configurations.

In this article, we give an overview of Scalasca and highlight research accomplishments of our group during the past two years, focusing on the analysis of wait states and of time-dependent behavior, as these two examples address the scalability of Scalasca regarding both the number of processes and the length of execution, respectively.

2 Scalasca Overview

The current version of Scalasca supports measurement and analysis of MPI applications written in C,C++ and Fortran on a wide range of current HPC platforms. Hybrid codes making use of basic OpenMP features in addition to passing messages are supported as well. Figure 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 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[2], 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 processors 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 function call path and process/thread which can be interactively examined in the provided analysis report explorer.



```
0.00 Time
  1.49e5 Execution
    0.35 MPI
      0.02 Synchronization
      0.00 Communication
        9785.60 Point-to-point
          1.99e4 Late Sender
          0.00 Late Receiver
        5329.49 Collective
          0.00 Early Reduce
          0.00 Early Scan
          22.85 Late Broadcast
          3669.40 Wait at N x N
          0.20 N x N Completion
      0.00 File I/O
      30.55 Init/Exit
    512.91 Overhead
  2.99e7 Visits
  1536 Synchronizations
  1.79e7 Communications
  6.77e11 Bytes transferred
  2.36e4 MPI Waiting Time
  3.94e4 Delay
  5178.45 Computational imbalance
  63.77 Instrumentation Overhead
```

```
0.00          1.99e4 (10.53%)          1.89e5
```

```
0.00 zeusmp
  0.00 configure
  0.00 options
  0.02 mstart
  0.00 dataio
  0.00 clocks
  9.44 srcstep
  0.00 transprt
    0.00 ct
      592.39 lorentz_d
      0.00 bvalv1
      0.00 bvalv2
      0.00 bvalv3
      2550.88 MPI_Waitall
      0.00 hsmoc
        0.00 <<iloops>>
        5593.36 bvalemf1
        0.00 MPI_Waitall
        0.00 <<jloops>>
        5094.88 bvalemf2
        0.00 <<kloops>>
        4574.96 bvalemf3
    506.16 advx1
    304.08 advx2
    448.88 advx3
  0.00 intchk
  186.68 nudt
  0.00 MPI_Reduce
  0.00 MPI_Finalize
```

```
0.00          2550.88 (12.84%)          1.99e4
```

```
0.00          0.00                     100.00
0.10          2550.88 (1.60e4%)        16.02
```

```
Selected "9021.94 Collective"
```

Figure 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).

3 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 communication-intensive applications to large process counts, such wait states can present severe challenges to achieving good performance.

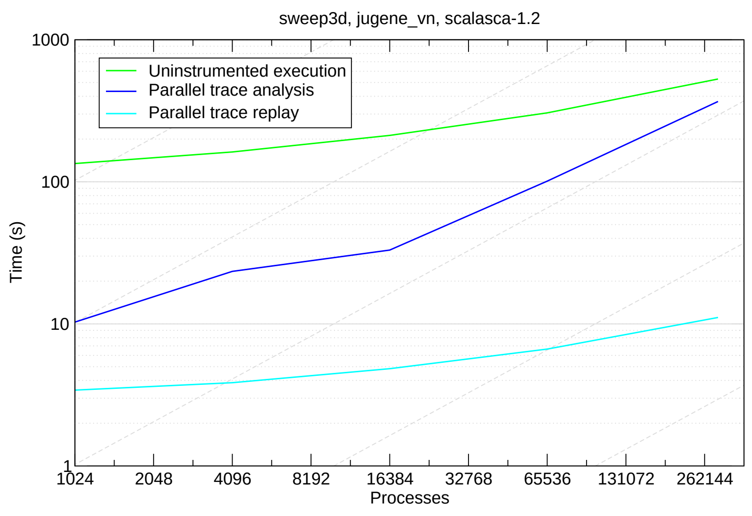


Figure 2. Scalability of wait-state search for the benchmark application SWEEP3D on the full JUGENE system. The graph charts wall-clock execution times for the uninstrumented application and the analyses of trace files generated by instrumented versions with a range of process numbers. The figure shows the total time needed for the parallel analysis including loading the traces and collating the results and the time needed for the replay in isolation. It can be seen that the total analysis takes less than 400 seconds and – for this particular example – is still faster than the application itself.

3.1 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 available on system such as Blue Gene/P 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[3]. During the replay, the analyzer identifies wait states 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 complete trace analyses of runs with up to 294,912 cores on the full IBM Blue Gene/P system JUGENE (Figure 2).

3.2 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. Building on earlier work by Meira, Jr. et al.[4], 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.

3.3 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 we can realistically hope 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.[5,6]

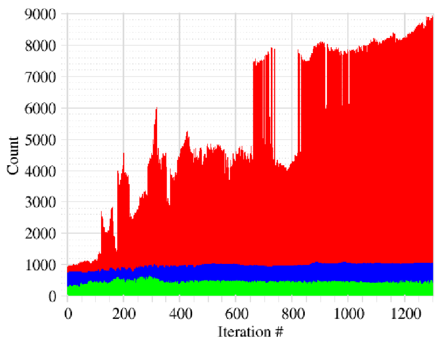
4 Analysis of Time-Dependent Behavior

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[7] or when the state of the computation adjusts to new conditions in so-called adaptive codes[8].

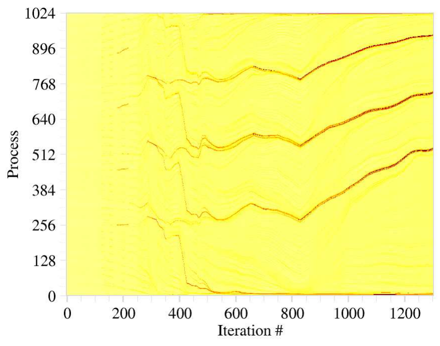
4.1 Observing Individual Iterations

To study the time-dependent behavior, Scalasca was equipped with iteration instrumentation capabilities corresponding to dynamic timers in TAU[9] that allow the distinction of individual iterations both in runtime summaries and in event traces. Moreover, to simplify the understanding of the resulting temporal data, we implemented several display tools including iteration graphs with minimum, median, and maximum representation (Figure 3(a)) as well as heat maps to cover the full (process, iteration) space for a given performance metric (Figure 3(b)).

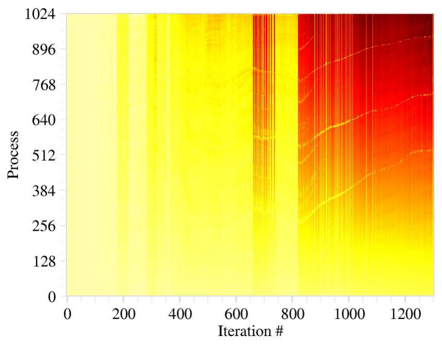
With this new toolbox at our disposal, we evaluated the performance behavior of the SPEC MPI2007 benchmark suite on the IBM SP p690 cluster JUMP, observing a large variety of complex temporal characteristics ranging from gradual changes and sudden transitions of the base-line behavior to both periodically and irregularly occurring peaks, including possible noise[10]. Moreover, problems with several benchmarks that limited their



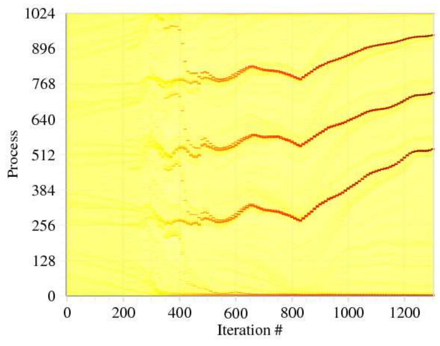
(a) Minimum (green), median (blue), and maximum (red) number of point-to-point messages sent or received by a process in an iteration.



(b) Number of messages sent.



(c) Late Sender waiting time.



(d) Number of particles owned by a process.

Figure 3. Gradual development of a performance problem along 1,300 timesteps of PEPC on 1,024 processors.

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[11] particle simulation code developed at Jülich Supercomputing Centre and subject to an application liaison between our group and the PEPC developer team. 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[12] revealed a severe and gradually increasing communication imbalance (Figure 3(a)). 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 3(b)) in rank order, introducing Late Sender waiting times at processes with higher ranks (Figure 3(c)). Interestingly, the communication imbalance correlated very well with the number of particles "owned" by a process (Figure 3(d)), 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.

4.2 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[13] 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 offers 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.

5 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 share development costs, we will integrate Scalasca closer with related tools including Periscope[14], TAU[15], and Vampir[16].

6 Acknowledgment

Financial support from the Helmholtz Association of German Research Centers through Grants VH-NG-118 and VH-VI-228 and the Deutsche Forschungsgemeinschaft (German Research Association) through Grant GSC 111 is gratefully acknowledged. Moreover, the authors would like to thank the John von Neumann Institute for Computing for the opportunity to use their high-end computer resources.

References

1. Jülich Supercomputing Centre, *Scalasca*, `http://www.scalasca.org/`.
2. Felix Wolf, Felix Freitag, Bernd Mohr, Shirley Moore, and Brian J. N. Wylie, *Large Event Traces in Parallel Performance Analysis*, in: Proc. of the 8th Workshop on Parallel Systems and Algorithms (PASA, Frankfurt/Main, Germany), Lecture Notes in Informatics, pp. 264–273, Gesellschaft für Informatik. March 2006.

3. Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr, *A scalable tool architecture for diagnosing wait states in massively-parallel applications*, Parallel Computing, **35**, no. 7, 375–388, 2009.
4. Wagner Meira, Jr., Thomas J. LeBlanc, and Alexandros Poulos, *Waiting time analysis and performance visualization in Carnival*, in: Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'96), pp. 1–10, Philadelphia, PA, 1996.
5. Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie, *Verifying Causality Between Distant Performance Phenomena in Large-Scale MPI Applications*, in: Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 78–84, IEEE Computer Society, Weimar, Germany, February 2009.
6. David Böhme, Marc-Andre Hermanns, Markus Geimer, and Felix Wolf, *Performance Simulation of Non-Blocking Communication in Message-Passing Applications*, in: Proc. of the 2nd Workshop on Productivity and Performance (PROPER 2009), August 2009, (to appear).
7. Darren J. Kerbyson, Kevin J. Barker, and Kei Davis, *Analysis of the Weather Research and Forecasting (WRF) Model on Large-Scale Systems*, in: Proc. of the Conference on Parallel Computing (ParCo, Aachen/Jülich, Germany), vol. 15 of *Advances in Parallel Computing*, pp. 89–98, IOS Press. September 2007.
8. Sameer S. Shende, Allen D. Malony, Alan Morris, Steven Parker, and J. Davison de St. Germain, *Performance Evaluation of Adaptive Scientific Applications using TAU*, in: Proc. of the International Conference on Parallel Computational Fluid Dynamics (Washington DC, USA), May 2005.
9. Allen D. Malony, Sameer S. Shende, and Alan Morris, *Phase-Based Parallel Performance Profiling*, in: Proc. of the Conference on Parallel Computing (ParCo, Malaga, Spain), vol. 33 of *NIC Series*, pp. 203–210, John von Neumann Institute for Computing. September 2005.
10. Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf, *SCALASCA Parallel Performance Analyses of SPEC MPI2007 Applications*, in: Proc. of the 1st SPEC Int'l Performance Evaluation Workshop (SIPEW, Darmstadt, Germany), vol. 5119 of *Lecture Notes in Computer Science*, pp. 99–123, Springer. June 2008.
11. Paul Gibbon, Wolfgang Frings, Sonja Dominiczak, and Bernd Mohr, *Performance Analysis and Visualization of the N-Body Tree Code PEPC on Massively Parallel Computers*, in: Proc. of the Conference on Parallel Computing (ParCo, Málaga, Spain), vol. 33 of *NIC Series*, pp. 367–374, October 2005.
12. Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf, *Scalasca Parallel Performance Analyses of PEPC*, in: Proc. of the EuroPar Workshop on Productivity and Performance (PROPER 2008, Las Palmas de Gran Canaria, Spain), vol. 5415 of *Lecture Notes in Computer Science*, pp. 305–314, Springer. August 2008.
13. Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie, *Space-Efficient Time-Series Call-Path Profiling of Parallel Applications*, in: Proc. of the ACM/IEEE conference on Supercomputing (SC09, Portland, OR), November 2009.
14. Technical University of Munich, *Periscope*, `http://www.lrr.in.tum.de/~gerndt/home/Research/PERISCOPE/Periscope.htm`.
15. University of Oregon, *TAU*, `http://www.cs.uoregon.edu/research/tau/`.
16. Technische Universität Dresden, *Vampir*, `http://www.vampir.eu/`.