Parallel Performance Engineering using Score-P and Vampir

William Williams
william.williams@mailbox.tu-dresden.de
GWT-TUD GmbH
Dresden, Germany

Holger Brunst
holger.brunst@tu-dresden.de
Zentrum für Informationsdienste und
Hochleistungsrechnen, Technische Universität Dresden
Dresden, Germany

ABSTRACT

This tutorial will introduce participants to the Score-P measurement system and the Vampir trace visualization tool for performance analysis. We will provide examples and hands-on exercises covering the full performance engineering workflow cycle on applications that include MPI, OpenMP, and GPU parallelism. Users will learn the following concepts:

(1) How to collect an initial profile of their code with Score-P
(2) Evaluation of that profile and its associated measurement overhead
(3) The concepts of scoring and filtering a profile and measurement respectively
(4) How to control the Score-P measurement system via environment variables
(5) How to collect useful traces with acceptable overhead
(6) How to understand trace visualization in Vampir

CCS CONCEPTS

• **General and reference** → **Measurement**; **Performance**; • **Software and its engineering** → **Software performance**.

KEYWORDS

measurement, performance engineering, high performance computing, visualization

ACM Reference Format:
William Williams and Holger Brunst. 2023. Parallel Performance Engineering using Score-P and Vampir. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3583715

1 INTRODUCTION

High performance computing presents its own unique challenges to performance engineering. An application's performance characteristics may change substantially as it is scaled up, or as new hardware is introduced. Since 2012, the Score-P[5] measurement system has focused on the data collection part of this problem; namely, how can users collect performance data that captures information from multiple forms of parallelism in a single tool and tune what data is collected based on the needs of their current performance experiment? Furthermore, how can users effectively predict whether their measurement overhead will remain within acceptable boundaries, and how can they adjust their experiments accordingly?

The Vampir[9] visualization tool has even older origins; it was among the earliest MPI-oriented performance tools, dating back to 1996. Vampir has been extended over the years to visualize trace data representing current parallel paradigms. In particular, visualization of GPU utilization and I/O behavior have been areas under active development in recent years.

2 METHODOLOGY

The performance engineering workflow for parallel applications is presented as a cyclic flow, with preparation, measurement, analysis, and optimization phases. The tutorial focuses on preparation, measurement, and analysis, especially in its half-day format. We introduce users to the concept of parallel performance engineering as a part of a larger performance engineering process: it is, generally speaking, wasteful to optimize the multi-node scalability of code that has not yet received any serial or node-level optimizations. Users are provided with an HPC or HPC-like environment to perform the various hands-on tasks in the tutorial. We have had great success with JupyterHub as a platform for remote access to HPC systems that allows for performant X11-in-browser use of visualization tools. The E4S[4] virtual appliance/container initiative is also helpful in situations where there is no specific target system associated with the tutorial. This allows participants to perform the hands-on steps directly on their personal laptops. Of course, there are tradeoffs in each case: JupyterHub is still somewhat bandwidth-dependent, running E4S in the cloud without JupyterHub all but requires measurement data to be transferred to a user's local system for visualization, and running E4S locally requires participants to prepare in advance by downloading and installing the VM or container.

2.1 Preparation

Preparation has two important aspects when using Score-P as a measurement system: building the application with the correct instrumentation hooks, and designing a reasonable performance experiment. We provide a pre-modified version of a standard benchmark to users (generally either BT-MZ[2] or TeaLeaf[8]), with the build system pre-configured to allow easy addition and removal of Score-P instrumentation and to generate instrumented binaries in a separate binary directory. This provides a useful demonstration of

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal.

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3583715

good practice in data management, as well as an easy introduction to the concept of instrumenting an application with Score-P.

We introduce the concept of performance models from the beginning of the tutorial. Users should have at least a rough idea of what behavior they expect from their application, and what data they need to collect in order to verify whether those expectations are met. We introduce the concept of dividing performance factors into serial and parallel factors: serial factors can be expected to be present at any scale, whereas parallel factors are consequences of increasing scale. We provide a brief overview of strong[1] and weak[10] scaling, the roofline[11] model, load balancing analysis, and critical path analysis as models and analyses that would dictate different types and granularities of data collection.

2.2 Measurement

We begin discussion of measurement with an overview of the types of measurement systems that are typically available for HPC applications. We divide these along two major axes: instrumentation vs. sampling as the data collection method, and profiles (in particular, call-path profiles) vs. execution traces as the output produced. We discuss some of the tradeoffs present in each of these: for instance, sampling may be applied to applications with no special preparation, and may for some purposes be more broadly applicable than instrumentation, it also can require a monitoring process and will be generally less precise than instrumentation. Traces, likewise, provide greater precision at greater cost. As we introduce these topics, we are already looking ahead to preparing participants to learn how to mitigate the drawbacks of using instrumentation to generate execution traces for HPC applications.

A measurement with Score-P can involve configuration in multiple dimensions: a user may add instrumentation markers manually to their source code, there is considerable latitude for compile- and link-time configuration of the possible measurement subsystems, and finally at run-time, the measurement may also be tuned. The run-time tuning allows, in principle, the same executable to be used for the most coarse- and fine-grained measurements a user might desire. We introduce participants to the `scorep-info` tool, which allows them to check which environment variables are active in their build of Score-P, and explain how to use these variables to select what is measured, what resources are available to the measurement system, and what output is subsequently produced.

For the initial hands-on with a benchmark, as for most use cases involving modern Score-P, users need no special configuration–the tool's automatic detection of MPI, OpenMP, and if TeaLeaf is used as the benchmark of the day, CUDA are typically sufficient on most HPC systems and within the E4S virtual environments. CUDA in particular is a useful teaching tool here, as it requires sensible run-time configuration via environment variables. In practical environments, configuration of which MPI function groups are enabled can also be critical to managing overhead, and this is important to emphasize to users–it is typically uninteresting, in comparison to the overhead involved, to instrument every `MPI_Test` or `MPI_Request_GetStatus` performed by an application.

```
% scorep-score scorep-bt-mz_4x6_profile/profile.cubex

Estimated aggregate size of event trace:                   160GB
Estimated requirements for largest trace buffer (max_buf): 41GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       41GB
(warning: The memory requirements cannot be satisfied by Score-P to avoid
 intermediate flushes when tracing. Set SCOREP_TOTAL_MEMORY=4G to get the
 maximum supported memory or reduce requirements using USR regions filters.)

flt     type     max_buf[B]          visits time[s] time[%] time/visit[us]  region
        ALL 43,161,138,617 6,591,877,701 2574.27   100.0           0.39  ALL
        USR 42,988,632,934 6,574,788,217 1061.84    41.2           0.16  USR
        OMP    167,683,712    16,359,424 1485.15    57.7          90.78  OMP
        COM      4,697,810       722,740    2.85     0.1           3.94  COM
        MPI        124,120         7,316   24.43     0.9        3338.67  MPI
    SCOREP             41             4    0.00     0.0          43.70  SCOREP
```

Figure 1: Example output from `scorep-score`. The BT-MZ benchmark has been profiled without applying any filters to the measurement.

```
% scorep-score -r scorep-bt-mz_4x6_profile/profile.cubex


flt     type     max_buf[B]          visits time[s] time[%] time/visit[us]  region
        ALL 43,161,138,617 6,591,877,701 2574.27   100.0           0.39  ALL
        USR 42,988,632,934 6,574,788,217 1061.84    41.2           0.16  USR
        OMP    167,683,712    16,359,424 1485.15    57.7          90.78  OMP
        COM      4,697,810       722,740    2.85     0.1           3.94  COM
        MPI        124,120         7,316   24.43     0.9        3338.67  MPI
    SCOREP             41             4    0.00     0.0          43.70  SCOREP

        USR 13,812,365,034 2,110,313,472  446.77    17.4           0.21  binvcrhs
        USR 13,812,365,034 2,110,313,472  352.08    13.7           0.17  matmul_sub
        USR 13,812,365,034 2,110,313,472  225.19     8.7           0.11  matvec_sub
        USR    596,197,758    87,475,200   19.29     0.7           0.22  lhsinit
        USR    596,197,758    87,475,200   11.49     0.4           0.13  binvrhs
        USR    447,869,968    68,892,672    7.02     0.3           0.10  exact_...
```

Figure 2: Detailed per-region from scorep-score, from the same run as in Figure 1

2.3 Scoring and Filtering Hands-On

We present two major analysis topics in this tutorial: understanding an overview trace of a parallel program's behavior, and analysing the quality of a measurement itself. In the tutorial we treat the second of these topics first: in order to produce a useful overview trace file, it is almost always necessary to analyze the program's behavior and determine what regions are, and are not, of interest for such a trace. This leads us to two fundamental concepts: scoring and filtering a measurement.

Scoring a measurement we define as the process by which we take a summary profile measurement and determine the relative costs of instrumenting (and in particular of tracing) the various code regions represented in that measurement. To do this, we provide the `scorep-score` tool, which summarizes the frequency of visits to each region, the average duration of a visit, which regions are associated with particular parallel paradigms, and which regions are on call paths leading to those (see Figure 1).

Given the scoring summary of a measurement, one can then build a list of regions that are of interest, and regions that are not of interest (see Figure 2). For a first overview trace, we typically advise users to focus on the regions involving parallel behavior of some sort, and the regions leading to those, with other regions included to the precise extent that they induce minimal additional measurement overhead and trace file size. Since version 7.0 of Score-P, the `scorep-score` tool has supported automatic generation of filter files according to user-provided parameters, with defaults that reflect this best practice. In our experience so far, this has been a great help to users and makes the scoring and filtering process simple in many cases.

Once a proper filter has been created, participants are then guided through a verification process: apply the filter prospectively

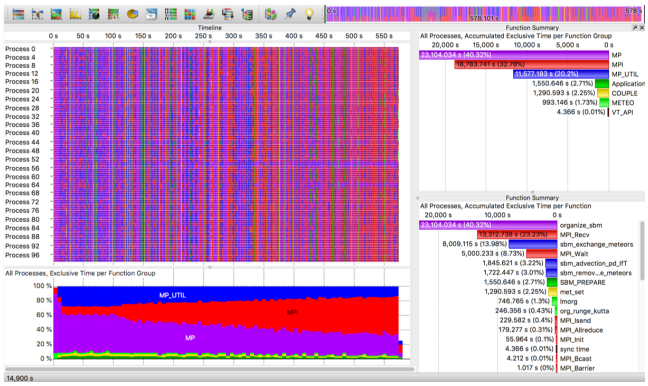

Figure 3: A Vampir visualization of COSMO-SPECS with a growing dynamic load imbalance. At the top right, we can see that the MPI time share over the course of the simulation is around a third of execution time. In the bottom left, the growing red wedge shows that the MPI share is in fact increasing, in aggregate, during this run.

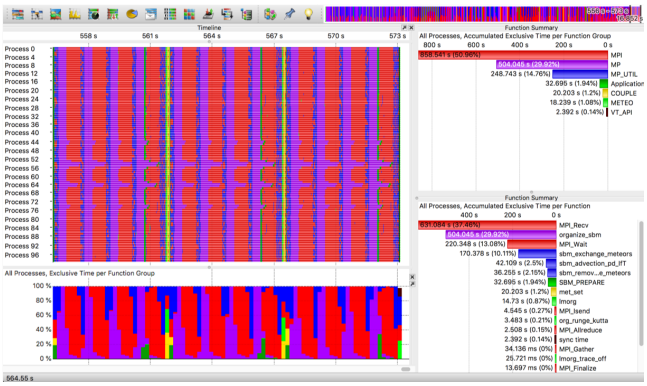

Figure 4: Detail of an imbalanced iteration in COSMO-SPECS. Again, MPI waiting time is shown in red and is clearly imbalanced across ranks in `MPI_COMM_WORLD`. Cloud microphysics (MP group), in purple, is responsible for this imbalance.

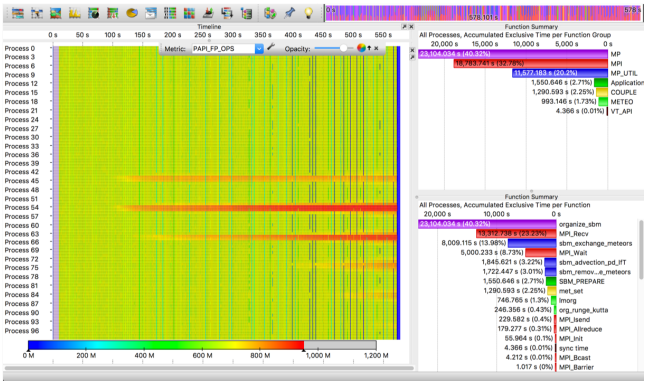

Figure 5: The Performance Radar view of the load imbalance in COSMO-SPECS. The ranks with an increased workload, as measured by total FLOPS, are visible in red.

to the existing measurement, analyse the projected new behavior of the application, and measure with the new filter and an appropriate amount of trace buffer. This will lead to the participants' first overview trace, and from there we will turn our attention to actual trace analysis with Vampir.

2.4 Trace Analysis with Vampir: the COSMO-SPECS Case Study

Vampir presents trace data using two major types of visualization: timelines, and summary charts. The timeline windows display program behavior for some selected time interval, including among others the active region (top of the call stack) for all locations in the Master Timeline view, the call stack for a given location in the Process Timeline view, and the value of a selected metric across all locations in the Performance Radar view. The summary charts dynamically aggregate performance data for the selected time interval; in this way, users can see the distribution of execution time, communication bandwidth, I/O load, and more via these summaries.

One of the primary case studies we present with Vampir is an example of a dynamically-growing load imbalance in a coupled COSMO-SPECS[7] weather simulation run. In this example, the simulation initially proceeded with a fixed grid and no load balancing.

As clouds formed, some ranks became computationally overburdened, leading to a dramatic increase in MPI waiting time on all other ranks. In Vampir, we show users multiple steps in identifying this problem. First, we show that the amount of MPI waiting time is, in aggregate, quite high, and that this suggests a performance problem exists. Then, we show that the fraction of time spent waiting in MPI calls is not static, but growing over time: this provides some hint that users should investigate the difference between early and late iterations in this particular measurement. See Figure 3 for an example of these first visualizations.

From there, users should be able to see that the computation time for cloud microphysics, while it is balanced early on, is imbalanced later. Figure 4 is what users will see upon zooming in to examine an imbalanced iteration later in the simulation. They can then confirm that this involves real computational effort via the Performance Radar's view (Figure 5 of total FLOPS per rank: the ranks simulating locations where clouds have formed have noticeably higher FLOPS than the others.

For comparison, we then present an optimized trace[6] with the FD4 dynamic load balancing system integrated into COSMO-SPECS. This shows the same number of iterations of the same simulation being completed in well under half the wall time, with the load remaining evenly distributed across processes over time (Figure 6).

Users are encouraged to work out the costs and benefits of the FD4 algorithm based on the function summary data. The two COSMO-SPECS traces also highlight for participants the benefits of combining manually-selected function groups with those that Vampir can automatically generate from a trace file. When functions are grouped sensibly for the purpose of a particular analysis, these groups may then be used in various Vampir views to understand

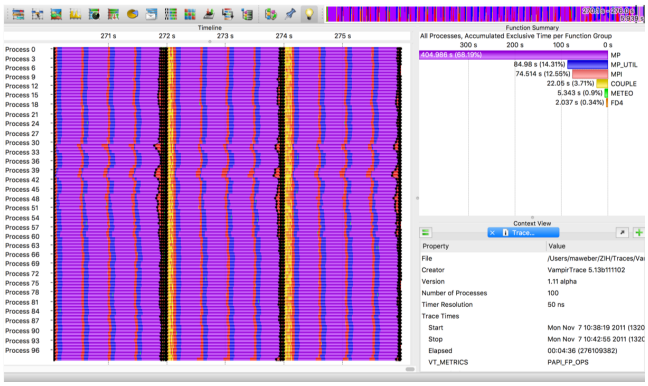

Figure 6: A Vampir visualization of COSMO-SPECS with the FD4 dynamic load balancing algorithm. On the left, the Master Timeline shows three late iterations of the simulation. At the top right, we can see that the MPI time share during the iterations being examined has dropped to 12%, while the FD4 algorithm itself adds negligible overhead (less than 1%).

the application's behavior at a more abstract level than simply the level of individual regions.

2.5 Hands-On and Exercises

Our exercises for users fall into two main categories. First, there will be a chosen benchmark application that they will profile, score, filter, and trace. This leads them through all the steps necessary to produce an initial visualization of an application in Vampir, while working with an application with known, generally stable performance characteristics, a short build cycle, and an easy-to-modify build system. This is, therefore, an ideal-world scenario for users. Second, when time allows, we present one or more toy applications with deliberately introduced performance problems (or in some cases, applications whose performance on the tutorial system is simply bad out-of-the-box). The participants will then analyse the application with appropriate tools (coarse wall time measurements as well as Score-P and Vampir) and attempt to determine what optimizations are possible.

Our selection of exercises typically focuses on common mistakes in HPC codes, namely failures of load balancing and failure to control the overhead associated with parallelization as shown via a sparse matrix/vector multiplication in OpenMP. In each of these cases, we ask users to start with a simple analysis based on a reasonable expectation of strong scaling and an application where this expectation, via wall time, clearly fails to be met. We then invite them to apply Score-P and Vampir to see a more detailed view of where the application spends its time, and ask for possibilities for what may have gone wrong. Finally, we point users to the offending OpenMP parallel region and ask them to use `schedule=dynamic` and appropriate run-time control to tune it for better behavior.

3 VARIATIONS OF THIS TUTORIAL

Versions of this material may be adapted to as short as a half-hour lecture (omitting all hands-on material and all performance engineering background material) or as long as a full-week workshop with bring-your-own-code sessions for approximately half the workshop time. One notable variant even stretched the material over a six-month period, with the week-long workshop material presented one day a month and project teams working independently on their own code between sessions. In the past, we have found that longer, on-site versions with the opportunity for users to work with tool developers on their own code lead to longer-term collaborations and more successful ongoing use of the tools.

4 FUTURE WORK

Several topics have been growing in importance in the HPC community over the past decade to the point where we will need to incorporate them into future versions of this tutorial. Performance engineering of Python code has become a topic of particular interest in connection with ML/Big Data frameworks, but also in regular scientific computing. Machine learning and big data frameworks also introduce new analysis problems, both due to their common implementation as MPMD applications and due to the multiple layers of abstraction between what the user/developer controls and what the hardware executes. Use of Score-P's Python bindings[3] could easily be worked into a full-day version of this tutorial. This would entail providing at least one example/hands-on exercise that uses the Python bindings for a typical HPC-oriented use case. Using Score-P and Vampir for performance analysis of these heterogeneous frameworks, in contrast, would likely require either a separate tutorial or a second day dedicated to the specialized topic here. Both the collection of measurement data and its analysis become much more complex, and even our basic tour of performance models most likely needs to be revised and expanded to ensure that participants can translate from the behavior of a machine learning framework to a specific performance model.

ACKNOWLEDGMENTS

This tutorial draws on the work of many colleagues over a span of many years. Everyone who has presented any previous version of this tutorial has helped refine it into its present form. In particular, the authors would like to thank Brian Wylie, Bernd Mohr, Christian Feld, Markus Geimer, Anke Visser, Luis DeRose, Christina Mühlbach, Bert Wesarg, Frank Winkler, Ronny Tschüter, Matthias Weber, Johannes Ziegenbalg, Robert Dietrich, Andreas Knüpfer, Marc Schlütter, Ronny Brendel, Jens Lukaschkowitz, Tobias Hilbrich, Thomas William, Dirk Schmidl, and Michael Knobloch for their contributions over the years to past versions of this tutorial.

The authors would also like to thank Matthias Lieber for allowing the use of his dissertation work as a performance engineering case study, and Bert Wesarg for his comments and suggestions on this paper.

The authors are grateful to the Center for Information Services and High Performance Computing at TU Dresden for providing its facilities for high throughput calculations.

REFERENCES

[1] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[2] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 158–165.

[3] Andreas Gocht, Robert Schöne, and Jan Frenzel. 2020. Advanced Python Performance Monitoring with Score-P. *Tools for High Performance Computing 2018 / 2019* (Oct. 2020), 261–270. https://doi.org/10.1007/978-3-030-66057-4_14 arXiv:2010.15444 [cs.DC]

[4] M Heroux, J Willenbring, S Shende, C Coti, W Spear, J Peyralans, J Skutnik, and E Keever. 2020. E4S: Extreme-scale Scientific Software Stack. In *2020 Collegeville Workshop on Scientific Software Whitepapers*.

[5] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. 2012. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*. Springer, 79–91.

[6] Matthias Lieber, Verena Grützun, Ralf Wolke, Matthias S Müller, and Wolfgang E Nagel. 2012. Highly scalable dynamic load balancing in the atmospheric modeling system COSMO-SPECS+ FD4. In *Applied Parallel and Scientific Computing: 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part I 10*. Springer, 131–141.

[7] Matthias Lieber, Ralf Wolke, Verena Grützun, Matthias S Müller, and Wolfgang E Nagel. 2010. A framework for detailed multiphase cloud modeling on HPC systems. In *Parallel Computing: From Multicores and GPU's to Petascale*. IOS Press, 281–288.

[8] Simon McIntosh-Smith, Matthew Martineau, Tom Deakin, Grzegorz Pawelczak, Wayne Gaudin, Paul Garrett, Wei Liu, Richard Smedley-Stevenson, and David Beckingsale. 2017. TeaLeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 842–849.

[9] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. 1996. VAMPIR: Visualization and analysis of MPI resources. (1996).

[10] Xian-He Sun and John L Gustafson. 1991. Toward a better parallel performance metric. *Parallel Comput.* 17, 10-11 (1991), 1093–1109.

[11] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785