frontiers | Frontiers in High Performance Computing

TYPE Technology and Code
PUBLISHED 24 March 2026
DOI 10.3389/fhpcp.2025.1709051

The Score-P performance tools ecosystem

Christian Feld[1]*, Alexandru Calotoiu[2], Gregor Corbin[1], Markus Geimer[1], Marc-André Hermanns[3], Maximilian Knespel[4], Bernd Mohr[1], Jan André Reuter[1], Maximilian Sander[4], Pavel Saviankou[1], Marc Schlütter[1], Robert Schöne[4], Sameer S. Shende[5], Anke Visser[1], Bert Wesarg[4,6], William R. Williams[4], Felix Wolf[7], Brian J. N. Wylie[1] and Mikhail Zarubin[4]

[1]Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Jülich, Germany, [2]Scalable Parallel Computing Laboratory, Department of Computer Science, ETH Zürich, Zürich, Switzerland, [3]IT Center, RWTH Aachen University, Aachen, Germany, [4]Information Services and High Performance Computing, Center for Interdisciplinary Digital Sciences, TUD Dresden University of Technology, Dresden, Germany, [5]Performance Research Laboratory, Oregon Advanced Computing Institute for Science and Society, University of Oregon, Eugene, OR, United States, [6]GWT-TUD GmbH, Dresden, Germany, [7]Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany

OPEN ACCESS

EDITED BY
Ming Zhao,
Arizona State University, United States

REVIEWED BY
Yoonho Park,
IBM Research, United States
Onur Cankur,
University of Maryland, College Park, MD, United States

*CORRESPONDENCE
Christian Feld
c.feld@fz-juelich.de

RECEIVED 19 September 2025
REVISED 14 November 2025
ACCEPTED 26 November 2025
PUBLISHED 24 March 2026

CITATION
Feld C, Calotoiu A, Corbin G, Geimer M, Hermanns M-A, Knespel M, Mohr B, Reuter JA, Sander M, Saviankou P, Schlütter M, Schöne R, Shende SS, Visser A, Wesarg B, Williams WR, Wolf F, Wylie BJN and Zarubin M (2026) The Score-P performance tools ecosystem. *Front. High Perform. Comput.* 3:1709051. doi: 10.3389/fhpcp.2025.1709051

COPYRIGHT
© 2026 Feld, Calotoiu, Corbin, Geimer, Hermanns, Knespel, Mohr, Reuter, Sander, Saviankou, Schlütter, Schöne, Shende, Visser, Wesarg, Williams, Wolf, Wylie and Zarubin. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner(s) are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

With the first exascale computing systems in production, tuning and scaling HPC applications to fully utilize the available hardware resources has become more important than ever. Thus, there is a strong need for software tools that assist application developers with this task. The Score-P instrumentation and measurement infrastructure plays a major role in filling this gap. Score-P is a community-driven, highly scalable tool suite for profiling and event tracing of massively parallel HPC application codes, and aimed to be easy to use. It provides measurement data via common data formats and runtime interfaces for a variety of complementary analysis tools developed by multiple institutions and companies, allowing users to gain insights into the communication, synchronization, input/output, and scaling behavior of their applications, pinpointing performance bottlenecks and their causes. In this article, we provide an overview of the current state of the Score-P infrastructure and its related tools ecosystem Cube, Extra-P, TAU, Scalasca, and Vampir. In particular, we detail Score-P's current design and architecture, both of which are highly flexible and extensible. Moreover, we describe how Score-P interacts with the analysis tools mentioned above and highlight the major extensions implemented over the past 10+ years to keep pace with the rapidly changing landscape of HPC hardware and parallel application programming interfaces. Furthermore, we discuss emerging challenges, particularly with respect to the ever-growing heterogeneity in both hardware and software, for collecting and analyzing performance data from applications running on future top-tier computing systems.

KEYWORDS

instrumentation, application execution measurement, profiling, event tracing, parallel performance analysis

1 Introduction

The computational power of High-Performance Computing (HPC) systems is continually increasing, with the first exascale systems now in production use. This immense compute performance comes at the cost of increased hardware complexity. Contemporary HPC systems combine CPUs, accelerators such as GPUs, and multi-layer memory and network hierarchies. In addition, the number of available parallel application programming interfaces is growing, ranging from vendor-specific low-level APIs that expose every hardware feature to high-level abstractions that promise portability and convenience. This complexity in both hardware and software poses significant challenges for HPC application developers in tuning and scaling their codes. Thus, there is a strong need for software tools that assist with these tasks. To address this gap, both industry and academia have developed many tools over the past decades. However, the performance tools community is obviously confronted with similar challenges as their target audience, requiring significant resources to keep their software up-to-date.

In 2009, research teams from six organizations developing performance analysis tools recognized that maintaining and extending their proprietary instrumentation and measurement infrastructures to support new features of HPC systems and programming interfaces had become increasingly burdensome. Precious development time had to be spent by each team to implement similar functionality in their own solution, thus limiting the resources available to improve the data analysis features. In addition, the use of different output data formats prevented interoperability. That is, interested application developers were required to learn about and repeatedly apply several measurement systems to their code—with subtle differences in usage—so they could analyze it with multiple tools that provide complementary analysis functionality. This was reported as a major inconvenience, particularly during joint training sessions organized within the context of the Helmholtz Virtual Institute — High Productivity Supercomputing (VI-HPS, 2025). To address the issues caused by a fragmented tools landscape, these teams joined forces. They began collaborating on the unified instrumentation and measurement infrastructure *Score-P* (Knüpfer et al., 2012), with initial funding from the German Federal Ministry of Education and Research (BMBF) and the U.S. Department of Energy (DOE). Since then, Score-P has grown into a major player in the performance tools community. Score-P has been widely adopted by many HPC sites around the world and demonstrated its strength in a multitude of performance assessments of production HPC codes. In particular, Score-P is used as one of the key data acquisition tools by the POP Centre of Excellence (POP-CoE Partners, 2025). For over a decade, the use of Score-P has been taught at numerous multi-day, bring-your-own-code VI-HPS Tuning Workshops; tutorials at major HPC conferences such as SC and ISC; and international HPC summer schools (Wylie et al., 2025), training over 1,000 actual and potential tool users.

In this article, we present an up-to-date overview of the Score-P infrastructure and its related analysis tools ecosystem. In particular, we provide the following contributions:

- We detail Score-P's current design and architecture, and describe how it interacts with the various analysis tools based on top of Score-P's common output formats.
- We highlight the major enhancements implemented over the past 10+ years to keep pace with the rapidly changing landscape of HPC hardware and parallel application programming interfaces.
- We discuss emerging challenges in collecting and analyzing performance data from applications running on future top-tier computing systems, particularly given the ever-growing heterogeneity in both hardware and software.

The remainder of this article is organized as follows. Section 2 provides an overview of the Score-P performance tools ecosystem and briefly explains instrumentation and measurement with Score-P. Subsequent subsections then introduce several established complementary analysis tools that leverage Score-P's open, efficient, and scalable measurement output formats. These tools form the Score-P ecosystem. Section 3 then provides an in-depth description of Score-P's internal architecture and key components, data model, and data sources. This is followed by a description of the continuous integration/continuous deployment (CI/CD) and testing infrastructures used to guarantee the product's portability and quality in production. Next, Section 4 discusses challenges emerging from current HPC trends and provides an outlook on future research. Finally, after providing an overview of related work in Section 5, we conclude the paper in Section 6.

2 The Score-P performance tools ecosystem

This section provides a high-level overview of the Score-P instrumentation and measurement framework, along with its related ecosystem of performance analysis tools. It describes the typical usage workflow and outlines Score-P's overall design, core components, and basic data flow before introducing various complementary analysis tools that comprise the ecosystem.

2.1 Score-P overview and workflow

Score-P is a highly scalable (Wylie, 2018) open-source tool suite intended to be easy to use. The latest release can be obtained from its project website (Score-P Development Team, 2025c) and from Zenodo (Score-P Development Team, 2025a), where previous releases are also available. Score-P is available under the 3-clause BSD license (Regents of the University of California, 1999). It supports both *profiling*—providing an aggregated summary of the program execution over time at more or less constant memory costs—and *event tracing*—chronologically capturing individual execution events, with memory cost proportional to duration—of HPC applications. Although tracing is more memory-intensive, it enables in-depth analyses by reconstructing dynamic application behavior. Analyzing the collected data allows the Score-P tools ecosystem to provide deep insights into the performance behavior of massively

parallel applications. This helps users pinpoint performance bottlenecks in their code and identify their root causes related to computation, communication, synchronization, I/O, memory management, and scalability.

The Score-P framework comprises a collection of libraries and command-line tools. To collect performance data for an application, it must first be *instrumented*. That is, measurement hooks must be injected into the application code at relevant points to capture execution behavior. Moreover, the Score-P libraries that implement the hooks and the general data collection infrastructure must be linked into the executable. For C, C++, and Fortran codes, this is achieved by prefixing the compiler with the `scorep` instrumenter command for both compilation and linking. Alternatively, one of the provided compiler wrappers, such as `scorep-mpicc`, can be used to work around limitations of common build systems. By default, the instrumenter leverages the compiler to intercept the entry and exit points of functions/subroutines in the user's source code. However, users can also manually annotate more coarse- or fine-grained code regions via an API. The instrumenter also applies various heuristics to identify which parallel APIs are being used. These heuristics and any user overrides are used to select the necessary measurement libraries during link-time. There is also experimental support for measuring unmodified application binaries via the `LD_PRELOAD` dynamic interposition mechanism. In this case, the call path context for parallel sections and user code regions is obtained via unwinding and sampling, respectively (see Section 3.2.2). The Score-P bindings for Python (Gocht et al., 2021) use the same preloading mechanism.

Figure 1 shows the basic data flow within an instrumented application during execution and how the resulting measurement data is consumed by analysis tools. The instrumentation hooks—depicted by the dark green boxes at the bottom—are implemented in *adapter* libraries. There is a separate adapter for each supported API, as well as for manual and compiler-based instrumentation (see Section 3.2), and for interfacing with TAU (see Section 2.4). Currently, only one process-level and one thread-level parallelism adapter can be enabled at a time. All other adapters may be combined freely. Whenever an instrumentation hook is triggered, the corresponding adapter maps the respective activity to an abstract event model and passes the *event* to the Score-P measurement core (see Section 3.1). There, the event-specific data is augmented with high-accuracy timestamps and, optionally, hardware counter information (see Section 3.1.5). Score-P provides interfaces to commonly used counter sources and an API for writing plug-ins to inject custom counter data. The augmented events are then passed on to all enabled event consumers, the so-called *substrates*. Substrates are provided for both call path profiling and event tracing; custom substrates can be implemented as plug-ins again. Both plug-in options have been described by Schöne et al. (2017). The two built-in substrates store measurement data in CUBE4 format (Saviankou et al., 2015) for profiles and OTF2 trace archives (Eschweiler et al., 2012) for traces. This data can be used by one or more of the established complementary analysis tools from the Score-P tools ecosystem.

All measurement artifacts are stored in a distinct measurement archive directory. This directory contains metadata describing the Score-P measurement configuration for reference and reproducibility.[1] The kind of output produced and all other measurement options are specified using environment variables. The available options and their defaults can be determined using the `scorep-info` command-line tool.[2]

Score-P defaults to generating a call path profile—the recommended starting point for every performance analysis cycle. Using the `scorep-score` tool on a profile (see Figure 2) provides initial insights without requiring a sophisticated analysis tool. `scorep-score` provides a flat profile for the instrumented regions—categorized by the adapter from which they originate—with data such as duration, number of visits, estimated memory requirements for future tracing measurements, and derived metrics. These metrics offer valuable information to help users reconfigure measurements. For instance, regions irrelevant to the intended analysis or inducing high overhead can be omitted using a measurement filter. `scorep-score` can assist in generating a filter file based on (configurable) built-in heuristics. Additionally, the effect of a filter file on a future measurement can be predicted. Once a suitable measurement configuration has been identified—potentially after iteration—further profiles and/or traces can be generated for use with the various analysis tools in the Score-P performance tools ecosystem. These tools are described in more detail in the following subsections.

2.2 Cube: interactive profile analysis

Cube is an open-source toolset for analyzing and visualizing performance data stored in CUBE4 profiles. It consists of three subcomponents: a highly optimized C writer library (CubeW) tailored for use by Score-P and Scalasca (see Section 2.5), a C++ library and command-line tools for reading, writing, and manipulating analysis reports (CubeLib), and a graphical user interface (CubeGUI).

The CUBE4 data model provides a structured view of program behavior with three dimensions: performance metrics, program call paths, and system resources. Each dimension can be organized hierarchically. The graphical user interface, CubeGUI (see Figure 3), allows users to navigate these dimensions using coupled tree browsers, enabling detailed examination of where and how computational resources are utilized. Data can be viewed in various display modes, for example, as absolute or relative values. Additionally, tree nodes are color-coded to make hotspots easy to identify. There are several alternative views for the system resource dimension, such as a topological view or a box/violin plot showing the statistical distribution across system resources. These views use a plug-in architecture (Knobloch et al., 2021), which also allows for custom plug-ins to facilitate tailored analyses (see Figure 4). Cube provides its own specialized scripting language, CubePL (Saviankou et al., 2015). It enables users to create derived metrics, thus enhancing analytical depth by combining and reinterpreting raw measurement data. CubeGUI can operate in client-server

1 Here, reproducibility refers to repeating a measurement using the same setup. Obviously, the measurement data itself will vary from run to run.

2 See `scorep-info config-vars --full` for a documented list of all configuration variables supported by the Score-P installation.

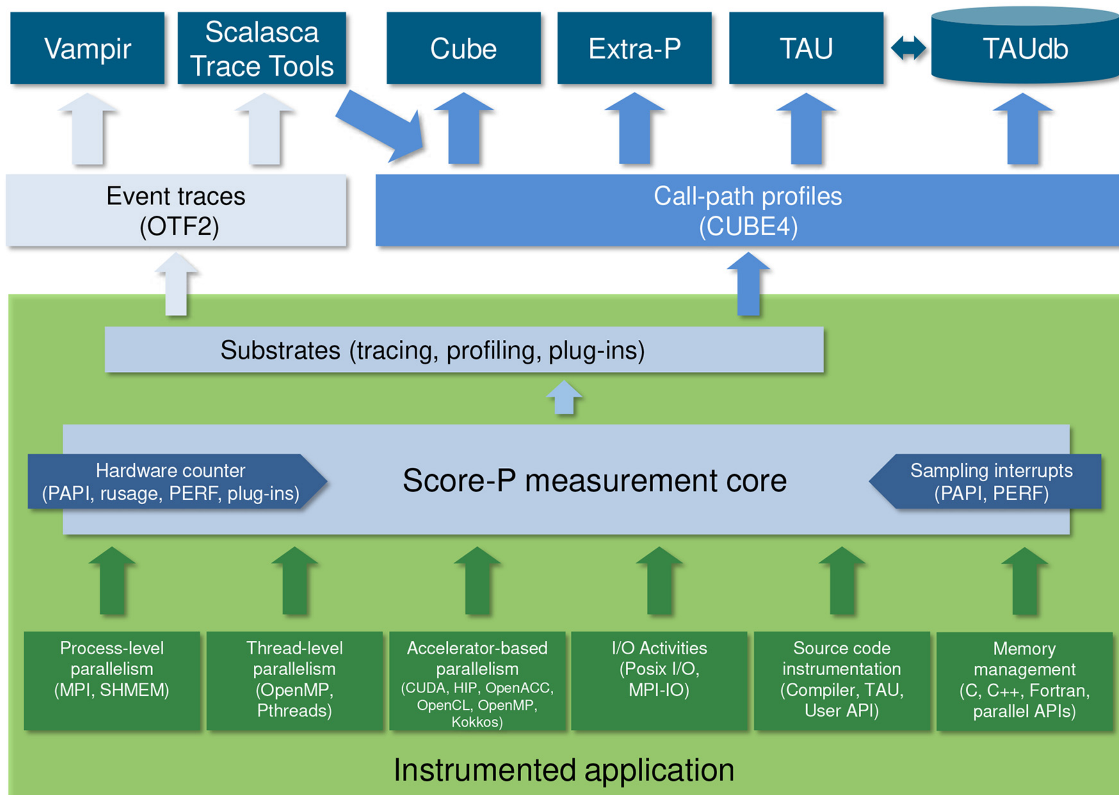

FIGURE 1
The Score-P performance tools ecosystem overview illustrates the basic data flow among the essential Score-P components and their interactions with various analysis tools via common data formats.

mode to analyze reports remotely or if they exceed the capabilities of a user's computer. In this mode, a server provided by the CubeLib package processes queries on the HPC system and sends only aggregated data to the connected CubeGUI for display. The CubeLib package comes with a set of command-line tools for advanced data manipulation. These include algebra tools for subtracting, merging, and averaging data from multiple runs, as well as other tools for post-processing reports (Song et al., 2004). Cube is available on its website (Cube Development Team, 2025) under the 3-clause BSD license.

2.3 Extra-P: empirical performance modeling

Extra-P is an empirical performance modeling tool intended to identify scalability bottlenecks (Calotoiu et al., 2013). When scaling up a code to a larger allocation or machine, predicting the program's run time can prevent foreseeable scalability issues and save valuable computing resources.

From a series of measurements with varied parameters, Extra-P builds models in the *performance model normal form* (PMNF), which restricts the search space to a combination of n polynomial and logarithmic terms.

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot log_2^{j_k}(p)$$

This example illustrates how varying the process count p impacts the run time $f(p)$, with the coefficients c_k computed using linear regression. A default search space with potential values for i_k and j_k is provided as a starting point, covering the most commonly encountered application behavior. Alternatively, the search space can be automatically refined as described in Reisert et al. (2017, sec. 3.3). Building models for multidimensional parameter spaces is also supported (Calotoiu et al., 2016).

Extra-P uses Score-P profile measurements as input to generate individual performance models for each metric and each kernel measured. The workflow requires the user to gather Score-P measurements for each parameter value included in the study. A minimum of five distinct values is required.[3] Extra-P then compares the call trees across experiments, identifies the common elements (kernels present in all experiments), and creates models for all metrics across all kernels. Users can explore these models in a GUI, shown in Figure 5, that offers different analysis options, such as changing the highlighted metric or ranking the models either asymptotically or at a given parameter value, thereby highlighting scalability issues. Extra-P is available from GitHub (Extra-P Development Team, 2025) under the 3-clause BSD license.

3 The multi-run feature of the `scan` command from the Scalasca Trace Tools can automate this process if measurement parameters can be specified via environment variables.

```
$ scorep-score -r profile.cubex

Estimated aggregate size of event trace:                   159GB
Estimated requirements for largest trace buffer (max_buf): 20GB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       20GB
(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 21,388,528,061 6,551,935,849 2896.19   100.0           0.44  ALL
         USR 21,309,225,312 6,537,020,537 1449.20    50.0           0.22  USR
         OMP     76,659,904    14,156,800 1430.34    49.4         101.04  OMP
         COM      2,355,080       724,640    3.14     0.1           4.33  COM
         MPI        287,724        33,864   13.51     0.5         398.85  MPI
      SCOREP             41             8    0.00     0.0          96.79  SCOREP

         USR  6,883,222,086 2,110,313,472  617.62    21.3           0.29  binvcrhs
         USR  6,883,222,086 2,110,313,472  426.24    14.7           0.20  matmul_sub
         USR  6,883,222,086 2,110,313,472  362.94    12.5           0.17  matvec_sub
         USR    293,617,584    87,475,200   20.17     0.7           0.23  lhsinit
         USR    293,617,584    87,475,200   16.10     0.6           0.18  binvrhs
         USR    101,320,128    31,129,600    5.14     0.2           0.16  exact_solution
         OMP      6,715,008       617,472    0.22     0.0           0.36  !$omp parallel @exch_qbc.f90:206
         OMP      6,715,008       617,472    0.22     0.0           0.36  !$omp parallel @exch_qbc.f90:217
         OMP      6,715,008       617,472    0.22     0.0           0.36  !$omp parallel @exch_qbc.f90:245
         OMP      6,715,008       617,472    0.22     0.0           0.36  !$omp parallel @exch_qbc.f90:256
         OMP      3,374,208       310,272    0.71     0.0           2.28  !$omp parallel @rhs.f90:29
         ...
```

FIGURE 2
When applied to a CUBE4 profile, `scorep-score` estimates the buffer requirements for a trace experiment that uses the same measurement configuration **(top)**. It also provides a high-level breakdown of how much time is spent in the computational parts (USR) of the code vs. the different APIs captured (here, OpenMP and MPI), expressed in absolute and relative terms to the overall run time, among others **(middle)**. Finally, it provides a flat profile, here sorted by trace buffer requirements **(bottom)**. When a filter file is applied, `scorep-score` adjusts the trace buffer estimation accordingly (not shown).

2.4 TAU: interactive profile analysis and cross-experiment data mining

The TAU Performance System® (Shende and Malony, 2006) is a versatile performance evaluation toolkit that supports both profiling and event tracing. TAU can be used as a standalone tool suite or integrated with Score-P. It provides an instrumentation layer, the measurement launcher `tau_exec`, and the analysis tools *ParaProf*, *PerfExplorer*, and *TAUdb*. ParaProf is a Java-based 3D profile browser, while PerfExplorer is a cross-experiment data analysis toolkit that uses performance data stored in TAUdb, a database that stores and organizes experiments. Integration with Score-P can occur on both the measurement and analysis sides. Measurement-side integration requires TAU to be configured with Score-P. This enables TAU's instrumentation layer to interface with Score-P's more efficient measurement core via an adapter and generate performance reports in OTF2 and CUBE4 formats. It allows uninstrumented application binaries to be launched with `tau_exec` to measure application performance using TAU and Score-P. Analysis-side integration occurs through TAU's analysis tools. These tools can process CUBE4 profiles generated by Score-P or Scalasca. For example, Figure 6 shows the cross-experiment analysis capabilities of TAU's PerfExplorer using CUBE4 data. Figure 7 shows the integration on both the measurement and analysis sides. An uninstrumented PETSc ex50 application was launched with `tau_exec` and created a CUBE4 profile via the Score-P measurement layer. ParaProf is used to display this profile. Finally, TAU profiles recorded in its native format can be converted to CUBE4 format and processed by Extra-P or CubeGUI.

A distinctive feature of TAU's MPI support is the ability to track idle time spent in implicit barriers within MPI collective operations. This feature enables the identification of synchronization problems in profiling mode (Huck et al., 2025). TAU is available in E4S containers (Willenbring et al., 2024) and through the Spack (Gamblin et al., 2015) and EasyBuild (Hoste et al., 2012) package managers. It is also available via source downloads (University of Oregon, 2024) and distributed under a BSD-style license.

2.5 Scalasca Trace Tools: automatic event trace analysis

The Scalasca Trace Tools (Geimer et al., 2006, 2009) provide a set of trace-based performance analysis tools with a focus on MPI and (host-side) OpenMP. Its core component—the automatic trace analyzer—is implemented as a parallel program using a trace-replay approach. It is able to identify wait states in communication and synchronization operations, pinpoint delays (i.e., imbalances) as their root causes (Böhme et al., 2010), and identify the activities on the critical path (Böhme et al., 2012). Analysis results are provided as additional metrics in an enhanced profile report in CUBE4 format (see Figure 8), highlighting candidate regions for optimization and parallel execution bottlenecks.

The original Scalasca toolset included its own instrumentation and measurement system, which generated trace data in a custom format, along with the Cube libraries and GUI. With the advent of Score-P, however, the Scalasca trace analysis components have

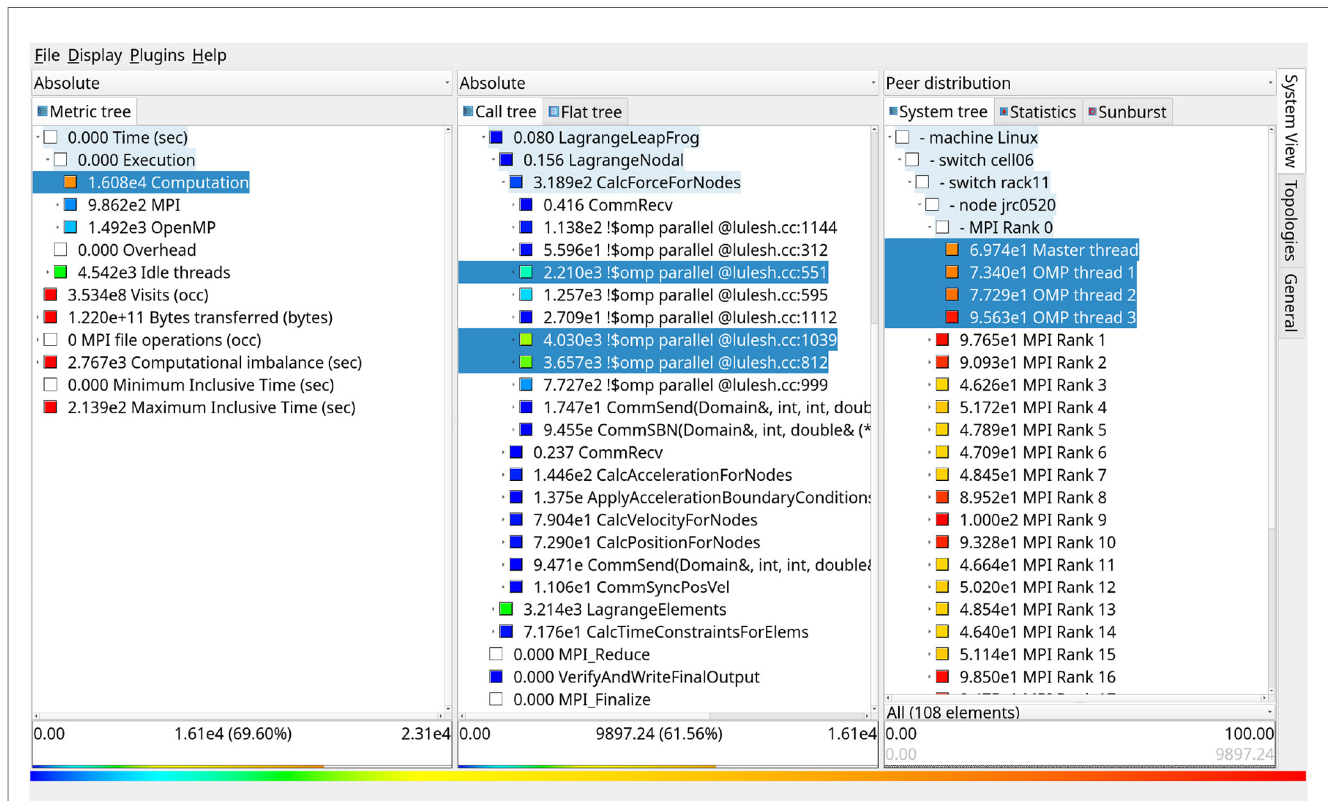

FIGURE 3
Screenshot of CubeGUI showing a post-processed Score-P profile report. Performance data is presented using three coupled tree browsers: the metric tree, the call tree, and the system tree (from left to right). Each column shows the distribution of values across the corresponding dimension for the selected nodes in columns to the left. Collapsed tree nodes show inclusive metric values (i.e., including child nodes), and expanded nodes show exclusive values.

been factored out into a separate package and based on the common OTF2 and CUBE4 data formats (Zhukov et al., 2015). In addition, the Scalasca convenience commands to simplify trace collection and subsequent analysis have been updated to properly handle the instrumented executables, configuration variables, and measurement archive directories produced and consumed by Score-P. They also leverage commands provided by Score-P and the Cube packages where possible. The Scalasca Trace Tools have been continuously updated to stay synchronized with Score-P's latest releases. In particular, leveraging the more comprehensive measurement capabilities of Score-P enabled Scalasca to further refine and extend its analyses. The Scalasca Trace Tools are available on their website (Scalasca Development Team, 2025) under the 3-clause BSD license.

2.6 Vampir: manual/visual event trace analysis

Vampir (Nagel et al., 1996) is a versatile and scalable event trace visualization and exploration tool. It consumes OTF2 traces produced by Score-P, without requiring any intermediate processing. It also supports the Chrome Trace Event Format (Duca and Sinclair, 2016). The central view is the *master timeline* (see Figure 9), which plots the event's origin on the y-axis and its time on the x-axis. The master timeline provides a holistic visualization of an application's entire activities, including active functions for threads or kernels in offloading devices, communication between MPI ranks, data transfers to and from offloading devices, I/O to the parallel file system, and measured or derived performance counters. Performance counters are color-coded by value and overlaid on activities. In addition to the master timeline, Vampir provides performance counter plots and specialized timelines for activities on shared resources. A multitude of summary charts, a communication matrix, and a call tree provide statistical summaries of the current zoom level. Finally, *summary timelines* combine equidistant summaries into a timeline plotted against the y-axis. Figure 10 shows a selection of these additional features. All timelines and statistical charts are synchronized and automatically apply the requested zoom level. Vampir can run on a local workstation or operate in a client-server model, which is particularly useful for large-scale analysis. In the latter model, the server runs as an MPI application on dedicated HPC resources and provides access to Score-P's measurement results. The client connects to the server and submits analysis queries.

Recently, Vampir extended its visualization capabilities to include complex scientific workflow executions (Tschüter et al., 2019), finally settling on the community-driven *WfFormat* (Coleman et al., 2022) execution trace file format. Workflow task execution is visualized by direct or indirect dependencies; the latter are derived from the tasks' input and output files. If an

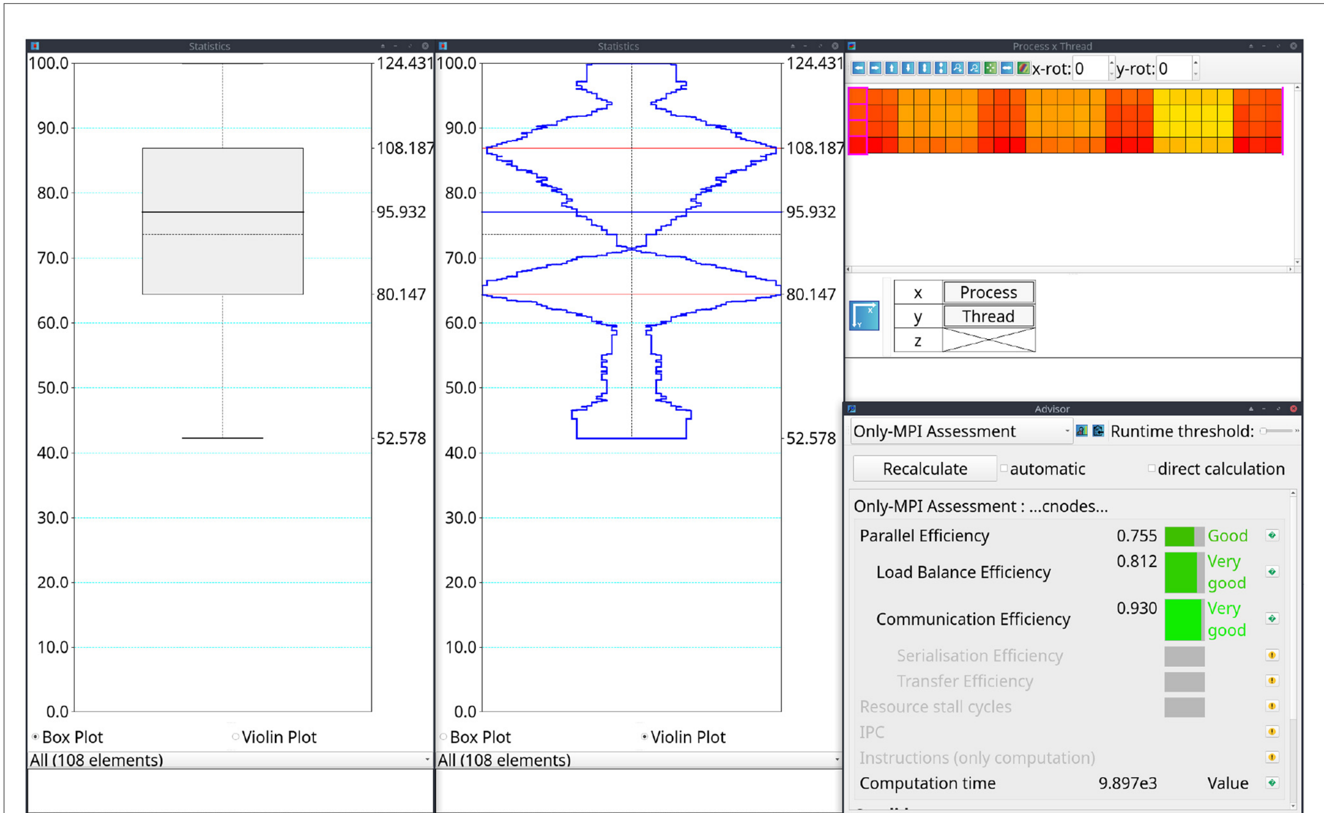

FIGURE 4
CubeGUI data visualization plug-ins. **(Left)** Box plot of time distribution across the system tree. **(Middle)** Violin plot of the same data. **(Top right)** Process × thread topology view of the same data. **(Bottom right)** Advisor plugin showing POP CoE efficiency metrics (POP-CoE Partners, 2022).

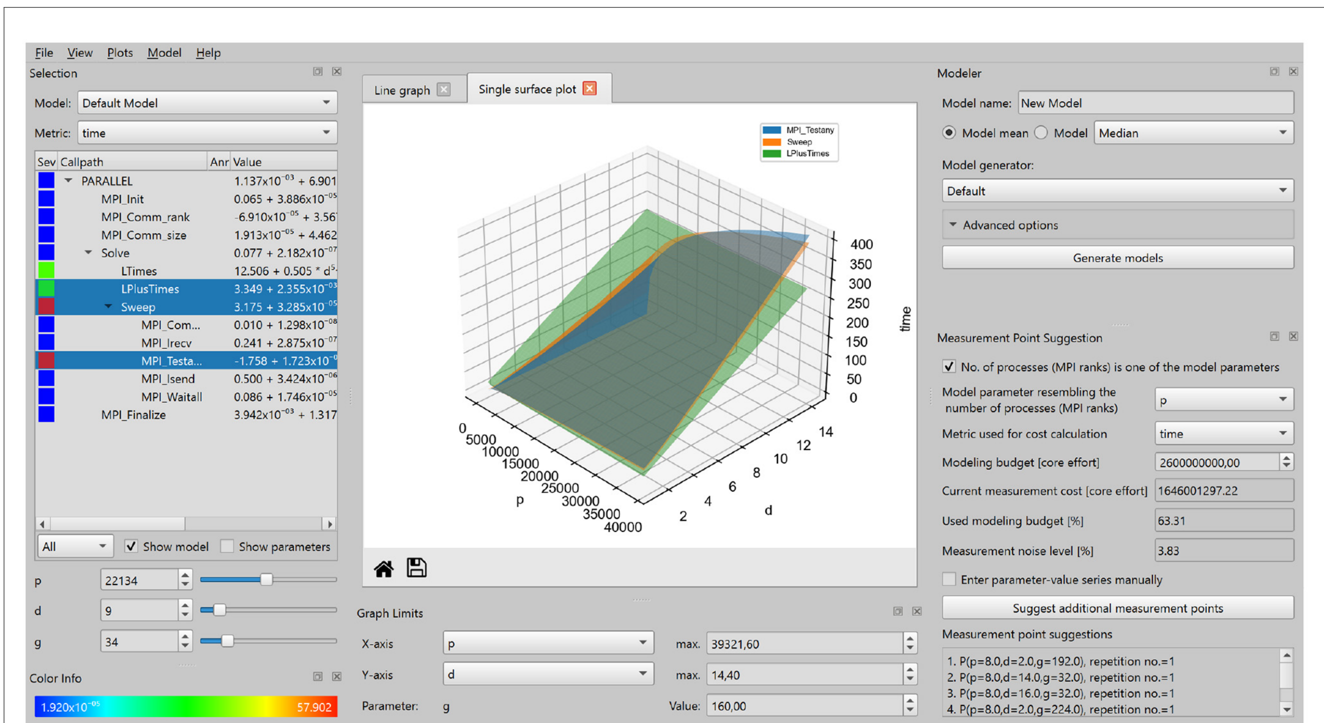

FIGURE 5
Screenshot of the Extra-P GUI.

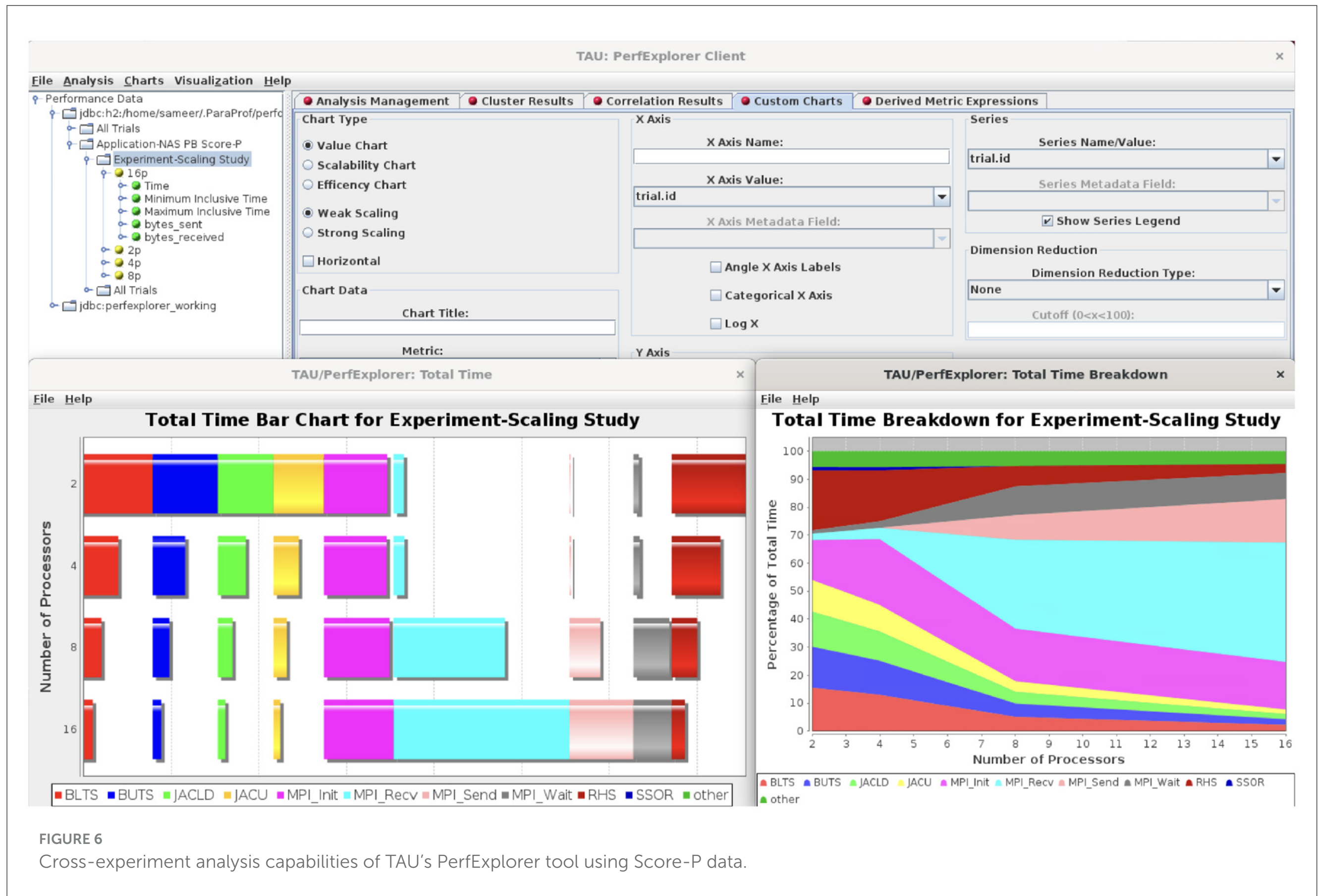

FIGURE 6
Cross-experiment analysis capabilities of TAU's PerfExplorer tool using Score-P data.

individual task has results from a Score-P measurement, the task can be analyzed immediately in Vampir's timeline visualization. Vampir is available (Vampir Development Team, 2025) under a commercial license.

3 Score-P architecture and key components

As described in Knüpfer et al. (2012), the first releases of Score-P supported only four application types via four monolithic measurement libraries: serial, OpenMP, MPI, and MPI with OpenMP. However, this approach proved to be too restrictive. To support additional data sources without introducing a combinatorial explosion in the number of installed libraries, we made the event and data sources, as well as the corresponding measurement core components, freestanding and thus flexibly composable. That is, we install independent event and data source libraries that are combined as needed when the instrumented application is linked. These sources communicate with the measurement core via well-defined interfaces. Similarly, the event consumers were refactored and decoupled from the measurement core, allowing improved extensibility via plug-ins as mentioned in Section 2.1.

The following subsections detail the current state of Score-P's architecture, the measurement core entities, the event and data sources, and their interactions. We also explain how code quality and portability are ensured in Score-P's development process.

3.1 Measurement core infrastructure

The *measurement core*, depicted in Figure 1, defines a *data model* to attribute measurement events to event sources and the execution environment. This also includes contextual data called *definitions*. Data sources communicate this information to the measurement core via the event and definition interfaces. Similarly, the *substrates* interface decouples the event sources from event consumers. Finally, the *subsystems* interface dispatches measurement-related information to all interested parties. These components and interfaces are described in more detail below.

3.1.1 Data model

Score-P represents performance data using events and definitions. Definitions are abstractions that represent static entities and provide context for events, such as source code regions, operating system and API abstractions (e.g., processes or accelerator streams), or communication contexts (e.g., MPI communicators). Definitions may also refer to other definitions. These definitions live in a process-local address space and are represented as numerical identifiers inside events. To analyze a multi-process experiment with n processes, a global set of

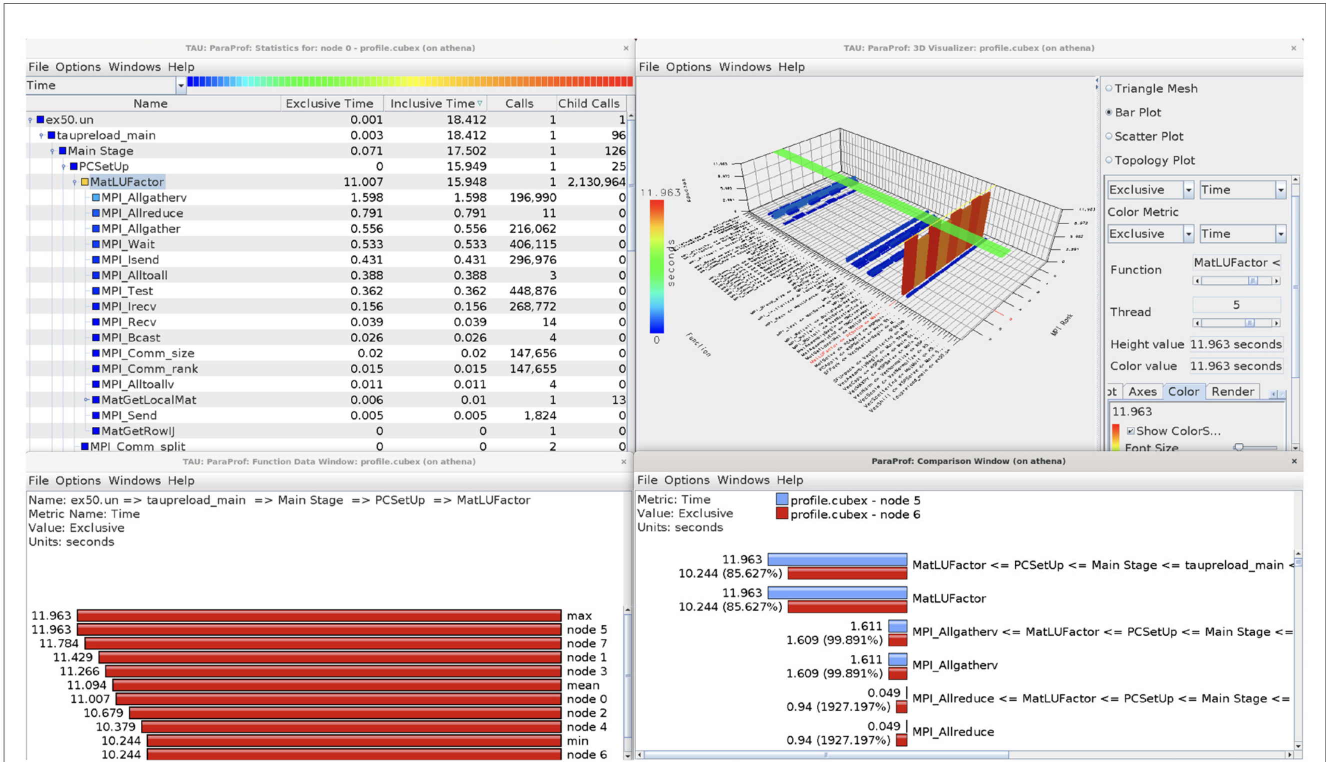

FIGURE 7
TAU's ParaProf visualizing a CUBE4 format profile from an uninstrumented PETSc ex50 application launched using `tau_exec` using the Score-P measurement layer integrated with TAU.

FIGURE 8
Screenshots of a Scalasca Trace Tools analysis result in the CubeGUI, showcasing the root cause analysis. **(Left)** MPI communication times and related wait states are negligible (0.52% of the overall run time; see *MPI* sub-metric in the *Time* metric tree at the top of the left pane). However, a significant proportion of the OpenMP thread idleness (selected node in the *Delay costs* metric tree in the left pane) is caused by data exchanges, that is, the MPI communication calls highlighted in the call tree (right pane). Thus, even minimal improvements can have a significant impact. **(Right)** The delay costs of MPI wait states (selected nodes in left pane) are caused by computation and load imbalance in the OpenMP-parallelized solver loops (highlighted call paths in the right pane), delaying subsequent communication operations. That is, inefficiencies in the solver loops cause cascading effects, amplifying their importance on overall performance.

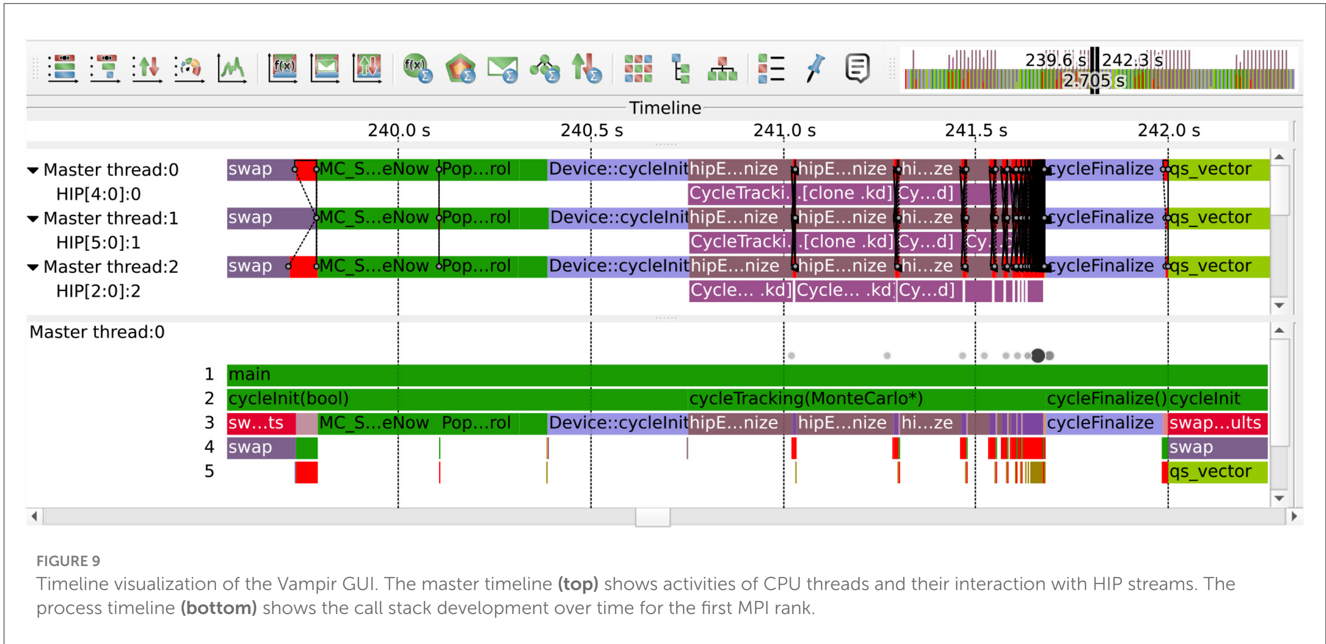

FIGURE 9
Timeline visualization of the Vampir GUI. The master timeline **(top)** shows activities of CPU threads and their interaction with HIP streams. The process timeline **(bottom)** shows the call stack development over time for the first MPI rank.

FIGURE 10
Selected features of the Vampir GUI. Two summary timelines **(top left)**, one for the peer-to-peer communication and one for the function profile across all locations. Two process summaries at the **top right** show clusterings based on the function profiles for the MPI ranks and HIP streams, respectively. A function summary shows a high-level flat profile in a pie chart in the **bottom left**. A communication matrix for all MPI ranks is in the **bottom right**.

definitions must be created (see Geimer et al., 2010, sec. 4.3) together with tables that map process-local to global definition identifiers. This step is called *unification* and is a tree-based, $O(\log\ n)$ parallel algorithm run after the measurement phase has finished. It is described in Geimer et al. (2012, sec. 2).

The *location* and *location group* definitions represent, respectively, thread-like and process-like entities in a program. Locations record events in uniform, chronological order as they happen. A location group contains a set of locations and generally represents a single address space. In contrast to these software-level abstractions, the *system tree* represents the underlying hardware: cores, sockets, devices, and even network hardware (see Figure 11). Cartesian *topologies* and their visual representations (see Figure 4) are an attempt to make large-scale system trees more intuitive (see Knobloch et al., 2021, sec. 4.3). These topologies can be generated from available system tree data as well as from network, MPI, and user-instrumentation APIs. Within the program itself, *region* definitions describe function-like entities. This means that a region is entered and exited in a manner consistent with maintaining a call stack. Region definitions relate to a *source file* definition and a *string* definition holding the region's name, and are associated with *region roles* and *paradigms*. Paradigms associate a region with a programming paradigm, such as MPI or OpenMP. The *compiler* and *user* paradigms represent compiler- and user-instrumented regions in user code, respectively. Roles represent concepts that cut across paradigms; for example, accelerator task submission is represented by a region role. This task submission role, and its corresponding task execution role, can be applied across all paradigms that support offloading.

Finally, events form the core of any performance tool's data model. Events occur on a given location at a specific time, and are typically associated with a region and a paradigm. They may also refer to other contextual data, such as communication contexts or additional metrics. Many events in Score-P are intrinsically paired on a given location (*Enter*/*Leave*), while some are paired across locations or location groups (*Send*/*Receive*).

3.1.2 Subsystems

During measurement, the event data gathered by the event and data sources (see Section 3.2) is passed on to the measurement core, then to the substrates, and finally to the file system (see Figure 1). Information about the measurement's state or the creation of new locations via the threading adapters (see Section 3.2.6) must be distributed to all interested parties. For example, the measurement initialization consists of several ordered steps. In one of these steps, the adapters register the environment variables they need for configuration. Another step consistently communicates the start of the measurement. Only then do events reach the measurement core. Yet another example is the pre-unification, where process-local definitions are amended with information that is only available once the measurement phase is complete but which is needed before the multi-process unification outlined in the previous section begins. To address these communication needs, we provide the subsystem abstraction used by all Score-P components. Subsystems communicate measurement-related events and location-lifetime events (as shown in Figure 12) to interested parties without creating direct couplings. Components only register for events of interest. Communication is triggered by the measurement core and the creators of locations. The subsystem abstraction also allows for per-location subsystem-specific data storage.

3.1.3 Measurement core

The instrumentation inserts hooks into the application, which trigger events in the adapters. The adapters then process these events, generate or obtain event-related definitions, and pass this information to the measurement core. In addition to tracking all definitions, the measurement core decouples the adapters from the event consumers—the substrates—by providing an API that the substrates can selectively subscribe to. It performs consistency checks on the received data and, for CPU-triggered events, obtains the corresponding location. The measurement core also augments the events with configurable, high-resolution timestamps, and optionally, with hardware counter metrics, if requested via subsystem-registered environment variables (see Section 3.1.5). The augmented event data is then passed to the registered event consumers.

3.1.4 Substrates

In Score-P, event consumers are called substrates. Score-P comes with two built-in event consumers, the profiling and the tracing substrates. A plug-in mechanism also allows for customized event consumers (Schöne et al., 2017). A substrate subscribes to a subset of events offered by the measurement core by registering a callback function for each event type of interest. To stay informed of measurement- and location-related events, substrates also participate in subsystem communication.

3.1.4.1 Profiling substrate

Profiling—also known as run-time summarization—involves accumulating statistics during the measurement process, rather than storing metrics alongside events for later analysis, as described by Geimer et al. (2010, sec. 5.1). The profiling substrate creates location-specific trees of nodes (to avoid thread synchronization), where each node represents a region within a *call tree*, that is, a sequence of nested regions as they were entered. Upon entry, nodes increment a visit count and store metric data, such as the current timestamp. When leaving the region again, this timestamp is used to calculate the inclusive duration spent in the region, which is then added to the node's accumulated duration. Other metrics, such as hardware counters, follow this approach. Some regions selectively attach data such as bytes sent or received. This statistical approach has the benefit of reduced memory requirements. Once a call tree has been fully visited, no further memory is required. To support OpenMP tasks, which disrupt the classic sequence of nested region *Enter* and *Leave* events, the measurement core introduced a *task* abstraction to which the profiling substrate subscribes (Lorenz et al., 2012). Creating a task creates an additional tree of nodes. Upon completion of the measurement, each process merges its available trees and creates call tree definitions, which are then

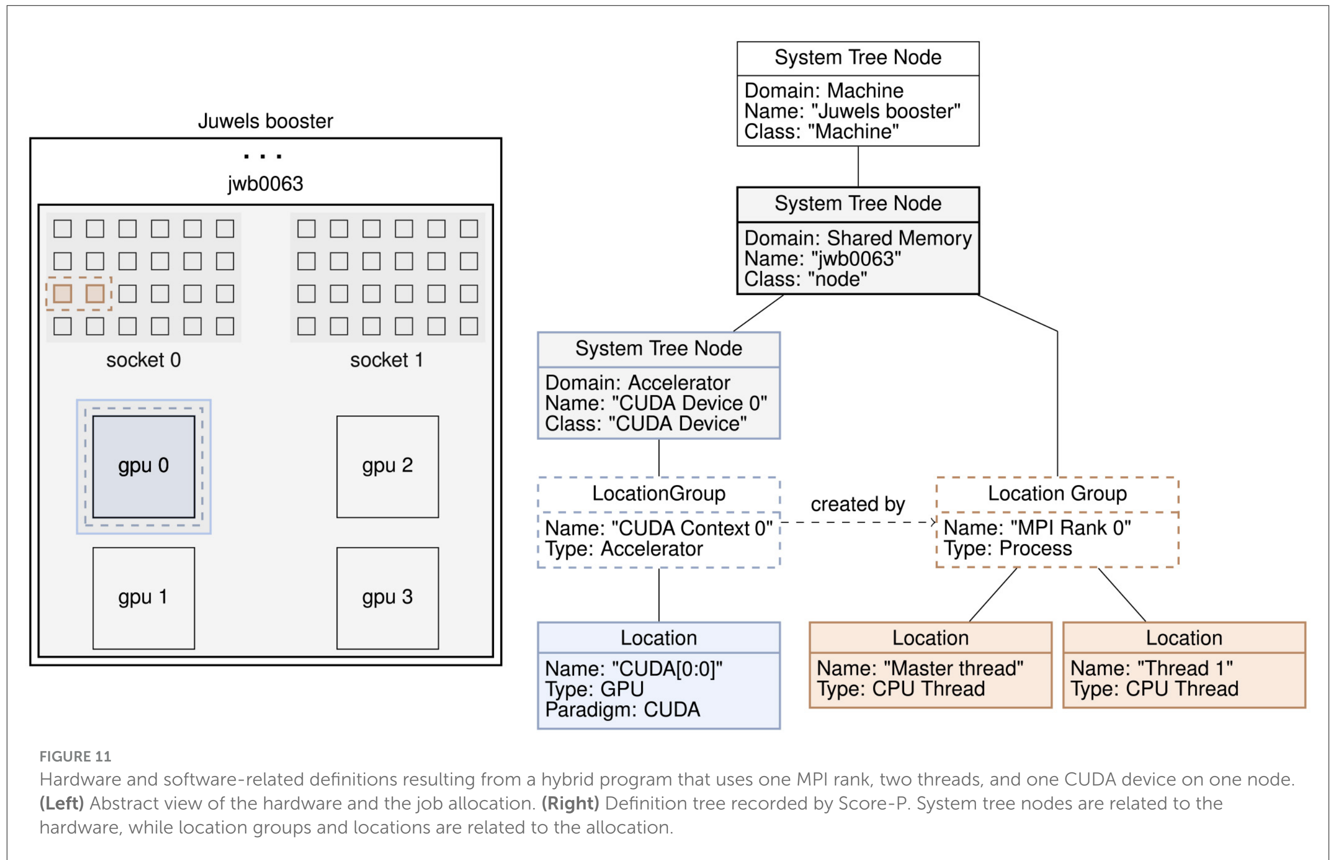

FIGURE 11
Hardware and software-related definitions resulting from a hybrid program that uses one MPI rank, two threads, and one CUDA device on one node. **(Left)** Abstract view of the hardware and the job allocation. **(Right)** Definition tree recorded by Score-P. System tree nodes are related to the hardware, while location groups and locations are related to the allocation.

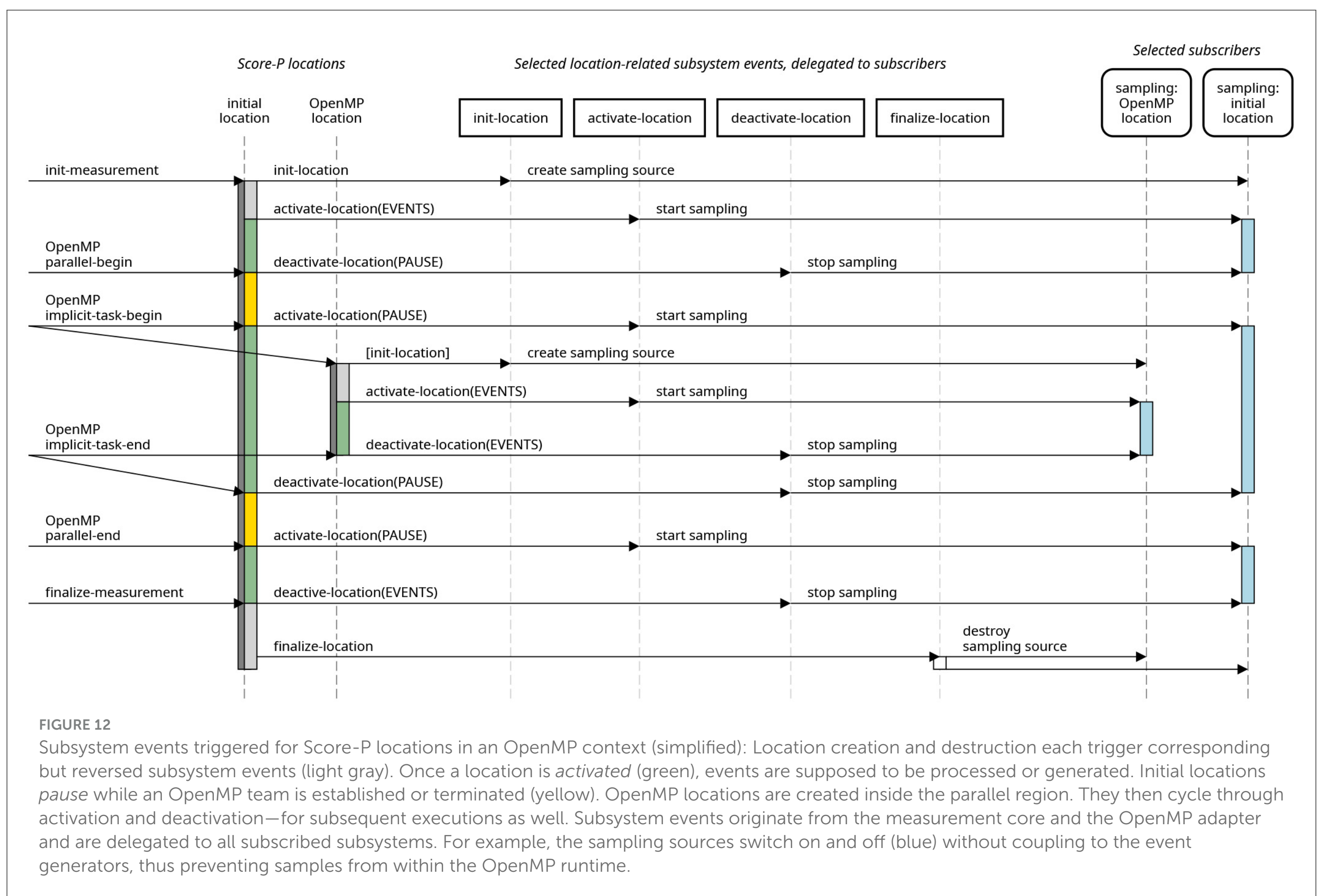

FIGURE 12
Subsystem events triggered for Score-P locations in an OpenMP context (simplified): Location creation and destruction each trigger corresponding but reversed subsystem events (light gray). Once a location is *activated* (green), events are supposed to be processed or generated. Initial locations *pause* while an OpenMP team is established or terminated (yellow). OpenMP locations are created inside the parallel region. They then cycle through activation and deactivation—for subsequent executions as well. Subsystem events originate from the measurement core and the OpenMP adapter and are delegated to all subscribed subsystems. For example, the sampling sources switch on and off (blue) without coupling to the event generators, thus preventing samples from within the OpenMP runtime.

subject to unification. Finally, process-local call tree data is gathered to create the `profile.cubex` CUBE4 summary analysis report.

3.1.4.2 Tracing substrate

The second built-in substrate is the tracing substrate. Its primary purpose is to translate measurement core events and definitions into the OTF2 event model (Eschweiler et al., 2012). Often, this simply involves delegating the core events to the OTF2 archive writer. The writer has a per-process buffer of configurable size. It stores event records chronologically per location. Locations obtain chunks of the buffer as needed, thus distributing the available memory. It is important to ensure that the buffer is large enough to hold all events of a measurement run. Buffers that are too small are flushed at run-time. The I/O caused by this flushing introduces delays that distort the measurement and make postmortem trace analysis less reliable. The Score-P command-line tool `scorep-score` assists in setting the buffer size based on an existing CUBE4 analysis report,[4] see Figure 2. Unlike profile experiments, which measure process-local durations, trace experiments correlate timestamps obtained from multiple high-resolution clocks within a distributed system. On contemporary HPC systems, these clocks are generally not synchronized, which can lead to inaccurate relative event timings that may misrepresent the logical event order. Score-P uses linear interpolation to synchronize clocks across processes at the beginning and end of a measurement, thereby mitigating this problem. Additionally, postmortem timestamp adjustments that address clock frequency drifts within a measurement can be applied to OTF2 archives by exploiting *happened-before* relationships encoded in Score-P's events (Becker et al., 2013). The OTF2 archive that is written to the file system consists of several files. One is the `traces.otf2` metadata file, the *anchor* file of the measurement. This file relates to the global definitions file, which contains the unified definitions captured at run-time, including all locations created during the measurement. For each location, there are local-to-global definition mappings, as well as a file containing event records for that location.

3.1.5 Further components

Upon request, the measurement core acquires additional *metrics* from sources such as the Performance Application Programming Interface (PAPI) (Jagode et al., 2025), Resource Usage (`getrusage`) (Free Software Foundation, 2023), and Linux' `perf` event infrastructure (Linux perf wiki Contributors, 2024). Users can also create metric sources via a plug-in mechanism, as described by Schöne et al. (2017).[5] Metrics to be added to the measurement are specified via environment variables and are accessed through a uniform API from the measurement core. They can be configured to either augment *Enter* and *Leave* events or to be collected asynchronously. Asynchronous metrics are recorded in dedicated metric locations to ensure the correct temporal order of events within one location. These locations also store metrics collected for entities in the system's hardware and software hierarchies. The granularity at which metrics are captured is defined during measurement initialization and determines the location(s) responsible for recording. Currently, metrics can be recorded in four ways: (1) on each location (e.g., PAPI events); (2) on the initial location of a location group (e.g., the memory allocated by a process); (3) on only one location per compute node (e.g., the node's memory temperature); or (4) once for the entire measurement (e.g., the energy consumption of the entire HPC system).

The measurement core infrastructure includes additional components that enable flexible adaptation to measurement requirements. For example, there are several implementations of a multi-process communication abstraction that allow our algorithms, such as the hierarchical unification, to be implemented regardless of the communication API used for a specific measurement. Moreover, an *address-to-line* lookup is made available for all Score-P components. This lookup selectively searches shared objects to provide the filename, line number, and function name of an address. It uses our *atomics* abstraction, which provides access to GNU built-in atomic operations (Free Software Foundation, 2025b) for all compilers supported by Score-P. The measurement core also provides a *filtering* component to determine which regions should be included or excluded from the measurement. These features, among others, are used in the event and data acquisition adapters that are described in the following section.

3.2 Event/Data sources

The Score-P instrumenter must decide which data sources to enable during compilation and linking. Here we will go into more depth on these data sources themselves and on the general techniques we apply across multiple data sources to control what ends up in a measurement. Each data source corresponds to an *adapter* in our architecture; the instrumenter's fundamental task is to select the adapters relevant to a given instrumentation phase and integrate them into the corresponding build step. Adapters are implemented as libraries, contain an adapter-specific subsystem, and translate events from data sources to our internal event model. There are essentially three points where instrumentation may be inserted into an application, with corresponding consequences for adapter design: compile-time, link-time, and run-time. A compile-time adapter allows the use of source code transformation (e.g., via the preprocessor) or interposition in the compilation process but is correspondingly tightly coupled to the source code and can require recompilation of the application for some changes. A link-time adapter may be implemented using various forms of symbol interception; typically, this will be either through weak symbols and a library designed for link-time instrumentation or through a more general form of symbol interception and rewriting. A run-time adapter typically implements a set of callback functions. These functions are registered when the corresponding runtime is initialized, which enables Score-P to obtain events from the runtime. Each event generated by these adapters may be augmented with metric data (see Section 3.1.5).

4 The CubeGUI plug-in *ScorePion* can be used as a graphical alternative.

5 Score-P provides a low-level C API. A convenience C++ wrapper for that interface and a collection of plug-ins are available at https://github.com/score-p/

3.2.1 Compiler instrumentation

Most compilers offer a method of instrumenting applications at compile-time without requiring source code modifications. Since its inception, Score-P has supported automatic instrumentation of applications via compiler flags. This allows Score-P to record fine-grained information of all called (non-inlined) user functions without the need for manual instrumentation by the user. However, such instrumentation can pose a high overhead, especially if inlined functions are instrumented as well. Therefore, filtering of the measurements might be necessary. Some frameworks, such as `tcollect` in the classic Intel compilers, provided a native way to filter functions. However, most interfaces lack this functionality and thus rely on run-time filtering, which naturally induces additional overhead. To enable both compile-time filtering and finer control over instrumentation, we turned to compiler plug-ins. Score-P v1.4 introduced a compiler plug-in for GCC, based on the GIMPLE intermediate representation (IR) produced by the compiler (Ziegenbalg and Wesarg, 2012). This plug-in supports compile-time filtering, allowing the user to specify which functions to instrument and which to exclude, thereby reducing the instrumented application's overhead. Furthermore, it allows us to implement static filters that exclude categories of functions that should never be instrumented: internals of tools interfaces and Score-P, inlined functions, artificial functions, and more. Score-P currently does not perform additional static analysis beyond what is available from the IR function's name and already available properties. However, arbitrary static analysis is possible during this phase. A similar plug-in for LLVM-based compilers was introduced in Score-P v9.0, based on the LLVM intermediate representation (Tschüter et al., 2017). This study also quantified the overhead difference between compile-time and run-time filtering with LLVM. The LLVM plug-in yields a proper event sequence when C++ exceptions are thrown, which the GCC plug-in does not. Both plug-ins must address changes to the respective GCC and LLVM plug-in APIs introduced by new versions. This makes Score-P's continuous build infrastructure described in Section 3.3 even more essential. Jobs use development snapshots of the upstream compilers to be able to detect API changes early in the development cycle.

The compiler instrumentation infrastructure is designed to be extensible, allowing us to easily add new instrumentation features in the future. This was recently demonstrated with the prototype development of a new compiler instrumentation adapter for Score-P (Kreuzer et al., 2025), based on the function call tracing system XRay (Berris et al., 2016) (see also Section 4).

3.2.2 Stack unwinding and sampling

Given the potentially significant overhead of automatic compiler instrumentation (Zarubin and Wesarg, 2024), Score-P sought an alternative that could still provide the calling context of the APIs of interest. Collecting stack backtraces at instrumentation points appeared as a natural solution and was introduced as an experimental recording mode in Score-P v2.0. This work is based on Szebenyi et al. (2011) but has been considerably adapted. With the stack backtrace infrastructure in place, Score-P also introduced a sampling mode that produces stack backtraces outside of instrumented API calls. This results in a hybrid recording mode that combines instrumentation events from the APIs with stack backtraces from sampled interrupts into a single consistent event model.

Multiple monitoring systems, such as HPCToolkit (Tallent et al., 2008), use a context-sensitive call graph to reduce redundancy in the event model for pure sampling events. These graphs are commonly referred to as *calling context trees*. Each node in such a tree not only holds the function name from the stack backtrace but also the current instruction or source code location inside that function. The parent of each node is the call site of this function. To support the hybrid recording mode, new Score-P sampling events named *CallingContextEnter*, *CallingContextLeave*, and *CallingContextSample* have been introduced, referencing nodes inside this calling context tree. The first two replace the traditional *Enter* and *Leave* events referring to a region. In general, Score-P collects the stack backtrace at each *CallingContextEnter* event, extends the existing calling context tree structure, and passes the event to the active substrates. A *CallingContextLeave* event pops the calling context node of the corresponding *CallingContextEnter* event and continues with the event consumption.

We can divide Score-P's instrumentation-based regions into three categories: *functions* that are a product of automatic compiler instrumentation, *wrapped regions* that represent calls into an otherwise uninstrumented library (such as MPI), and *artificial regions* that are neither of these (such as manual user instrumentation). In this hybrid recording mode, events from compiler instrumentation are suppressed. Thus, a Score-P-instrumented binary can be used for both compiler instrumentation and sampling-based measurements without recompilation. The distribution of *CallingContextSample* events allows inference of how events from compiler instrumentation would be distributed with respect to the sampling trigger. This trigger can be based on any counter provided by PAPI or `perf`, though it is time-based by default. Thus, it provides a drop-in replacement for compiler instrumentation. In the absence of sampling interrupts, the remaining two cases differ only in whether an *Enter* event is augmented with a stack backtrace. According to Score-P's philosophy, implementation details of external libraries, such as MPI or I/O calls, should be hidden from users. Wrapped regions implement this idea. A stack backtrace is collected only for the *Enter* event of a wrapped region. Subsequent sampling events avoid a stack backtrace so as not to show the internal call stack while inside the wrapped region. In contrast, artificial regions are always augmented with a stack backtrace because they are considered user code. The same occurs when an artificial region is entered while inside a wrapped region. This situation usually indicates that the user passed a callback function into a library.

In addition, sampling events can also be triggered by an interrupt generator. Samples triggered outside of any wrapped region are augmented with a stack backtrace. Inside wrapped regions, however, they are only recorded as *hits* without the library-internal call stack.

3.2.3 Manual instrumentation

Score-P also allows users to manually annotate their source code using an extensive API for instrumentation and measurement

control. It can be used in conjunction with compiler-based instrumentation to manually define additional regions of interest, such as coarse-grained program phases or more fine-grained code sections within subroutines, but also as an alternative to explicitly control the level of detail. Additionally, the API allows users to attach parameter values to code regions. These are stored in the trace data using specific events; in profiling mode, each distinct parameter value creates a separate child node in a call path profile. This allows users to drill down on performance data based on run-time inputs. As a special case, dynamic region profiling uses the visit count as a parameter, allowing for the capture of time-dependent behavior even with profiling. To reduce the memory footprint, an online clustering algorithm is used (Szebenyi et al., 2009). Finally, the API allows capturing of user-defined Cartesian topologies (see Section 3.1.1) and the manual pausing and resuming of measurements. The latter is achieved by switching off/on all nonessential substrate event callbacks.

3.2.4 Library wrapping

Library wrapping describes a family of methods that intercept calls from an application to a library. This allows a measurement system to interpose its own instrumentation. This is commonly used when programming paradigms lack a dedicated performance tools interface. Library wrapping enables a measurement system to generate *Enter* and *Leave* events, as well as more meaningful semantic events based on function parameters, around calls to a library. Traditionally, the wrapping is applied at link-time or at load-time (when the program is loaded into memory and before execution begins), as described in Dietrich et al. (2010).[6] As an alternative, Poliakoff and LeGendre (2019) introduced GOTCHA, a mechanism that dynamically wraps functions in shared libraries at run-time. This allows fine-tuning which functions are wrapped, if any, thereby eliminating any source of overhead for omitted functions. In Score-P, traditional library wrapping was used for several adapters (see the sections below). However, GOTCHA wrapping in Score-P can be controlled solely via environment variables, thus providing additional flexibility and streamlining the instrumentation process. Therefore, since Score-P v9.0, we support GOTCHA as our only library wrapping method. In addition to the provided adapters based on library wrapping, users can create wrappers for arbitrary C/C++ libraries. Brendel et al. (2019) describe the interactive workflow for building and using such *user library wrappers*. This workflow produces libraries for use with Score-P's instrumenter. However, the introduction of GOTCHA made these user library wrappers run-time loadable plug-ins.

3.2.5 MPI adapter

The Message Passing Interface (MPI) (Message Passing Interface Forum, 2023) is the *de facto* standard for distributed memory parallelism. It enables the development of portable parallel programs across different computer architectures and network protocols. The MPI API provides abstractions for many common parallel operations, for example, point-to-point communication, collective communication, and collective file I/O. Since the majority of HPC applications rely on MPI for multi-node parallelization, we strive to support this essential API in Score-P as broadly as possible. Therefore, members of the Score-P development team are actively participating in MPI's standardization body, the MPI Forum, especially in the tools working group. This involvement goes back to the beginning of the Score-P project and MPI 3.1.

The MPI specification included a profiling interface right from the start. Third parties/tools can easily interpose themselves between MPI and the user application. That is, a tool library can provide `MPI_` symbols that override the corresponding symbols from the MPI library. These tool-provided wrappers then perform the necessary work on the tool's side and delegate the user-requested MPI call to a matching `PMPI_` procedure that implements the actual functionality in the MPI library. Score-P also uses this standard mechanism. Consequently, its MPI wrappers rely on the semantics of the API calls as defined by the MPI specification.

3.2.5.1 Language bindings and wrapper generation

MPI defines three sets of language bindings: a C interface, a Fortran interface designed around the capabilities of Fortran 90, and a Fortran 2008 interface. Score-P provides wrappers for most functions in all three sets of bindings. To cover the entire MPI 4.0 API, a total of 1,200 wrapper functions are needed.[7] Therefore, it is important to automatically generate wrappers for the different language bindings, as demonstrated by Gamblin (2010) among others. As part of its initial release, Score-P already implemented a wrapper generator for the C and Fortran 90 bindings based on a manually maintained XML specification. Rasmussen et al. (2016) discussed the fundamental problems of writing correct Fortran 2008 wrappers and presented an implementation where, in principle, all tool code could still be written in C. However, their approach proved cumbersome to extend to complete coverage of MPI. Generating tool code in Fortran proved to be a better approach, as described in Corbin (2025). This new generator relies on a machine-readable API specification file provided by MPI since version 4.0, which is further interpreted by the Python module `pympistandard` (Ruefenacht, 2024). The wrappers for the Fortran 2008 bindings are separate from the established C wrappers but mirror their implementation. Thus, we ensure that the Fortran 2008 wrappers call the corresponding PMPI routine in the same language with identical arguments. This avoids a number of issues related to mixed-language MPI programs and Fortran-to-C argument conversions. In addition, various Score-P measurement core interfaces provide Fortran 2008 wrappers in order to allow Fortran code to call into the measurement system safely.

3.2.5.2 Feature overview

Given the continuous evolution and extension of the MPI standard—which, in its current version 5.0, describes over 500 procedures on 1,100 pages—and the multitude of combinations of MPI implementations[8] and compilers, maintaining MPI support

6 Library wrapping is typically applied to *shared* libraries. However, link-time wrapping using the GNU linker's `--wrap` option can wrap symbols in *static* libraries.

7 Not counting the large-count interfaces separately.

8 The two major libraries are Open MPI and MPICH, which also form the basis of several vendor implementations.

is an ongoing effort. As of Score-P v9.0, all MPI 4.0 functions—except for the MPI tools (MPI_T) interface—are recorded with at least *Enter* and *Leave* events.

Beyond the entry and exit of MPI functions, Score-P records additional events to enable subsequent analyses. For example, to support a point-to-point late-sender/late-receiver analysis in Scalasca, a trace must include all necessary information to replay point-to-point communication. Therefore, Score-P records events for sent and received messages, which, in addition to the number of bytes transferred, also contain the message envelope, that is, the communicator, tag, and peer rank. Likewise, on all participating processes, collective events record their kind, the communicator, and the root rank of the collective (if applicable). To interpret this information correctly, events for communicator creation also have to be recorded. Note that regardless of what is *recorded* in a measurement, Score-P will internally *track* a potentially broader set of events to ensure consistency. For example, the creation and deletion of communicators, as well as the creation and completion of requests, are always tracked.

We strive to support the MPI features that are commonly used in HPC. Broadly speaking, this includes the recording of detailed information for point-to-point, collective, and one-sided communication, as outlined above, as well as for synchronization and I/O operations. In addition, we also collect metadata for groups and communicators, as well as rank-to-coordinate mappings for Cartesian topologies. Recently, Score-P has become capable of recording internally consistent measurements of applications that use the `MPI_THREAD_MULTIPLE` threading model. However, this still presents challenges for visualization and analysis tools; see Hünich et al. (2025) for a discussion of how these are addressed in Vampir. Not yet covered are persistent collectives, process spawning, or the sessions model. Current efforts focus on supporting matched messages (Haus, 2024) and partitioned communication (Thärigen et al., 2023).

Score-P categorizes MPI wrappers into groups of functions that can be enabled or disabled at measurement time. Groups are selected such that any valid combination of enabled groups captures a set of events that produce an internally consistent measurement. In other words, for any given profiling or tracing record, either all or none of the relevant events are recorded. The events and their information are generally similar within each group.

3.2.6 Threading adapters

There are two types of threading execution models in HPC, in Score-P called *create-wait* model and *fork-join* model. The create-wait model is used by, for instance, the low-level POSIX Threads (Pthreads) API. It allows the *creation* of individual threads: a different, concurrent flow of work is created in our event model that overlaps with existing ones. Once a thread's work is complete, the thread either goes out of scope or is *waited* for by another thread. In the *fork-join* model, an existing thread *forks* a *team of threads* or *parallel region*—a set of threads of which the forking thread becomes a part. Once this team finishes its concurrent execution, the threads synchronize and *join*, meaning all but the forking thread stop execution. The most prominent example of this model is OpenMP (OpenMP Architecture Review Board, 2024). To detect and analyze thread management overhead, as well as synchronization and waiting times caused by imbalances, Score-P at least needs to intercept thread creation and synchronization events.

For Pthreads, library wrapping (see Section 3.2.4) is used to intercept the API functions responsible for thread management and synchronization. Pthread wrapping also applies to threading abstractions for which a specific instrumentation method is lacking, such as TBB, Cilk, and C++ threads. Arguments to Pthread functions must also be wrapped to capture created-by relationships in the new thread's location and to attribute joining events accordingly. The OpenMP fork-join model groups the threads of parallel regions. This allows analysis only within the group. By capturing OpenMP barrier events, analysis tools can detect waiting times caused by imbalances, for example. Recording OpenMP locks and critical sections, as well as Pthread mutexes and condition variables, enables synchronization time reporting. We augment these events with *happened-before* relationships (as per Lamport, 1978), which also enable contention analysis.

While the Pthreads API is stable, the OpenMP specification has undergone several extensions during Score-P's lifetime. Prior to version 5.0, the OpenMP specification did not define a tools interface. To instrument OpenMP directives and API functions, we therefore initially relied on source-to-source preprocessing using OPARI (Mohr et al., 2002), and later OPARI2, as described in an Mey et al. (2012, sec. 8). OPARI2 augments OpenMP directives found in the source code with calls to the POMP2 API.[9] Unlike the more general approach presented in Liao et al. (2008), we attempted to limit the complexity of OPARI2's source code modification by first preprocessing compilation units, then injecting POMP2 calls, before performing the actual compilation. The introduction of the `task` construct in OpenMP 3.0 posed significant challenges for instrumentation, processing, and analysis. Apart from tasking, per-location events are always properly nested. However, OpenMP tasks disrupt the classic sequence of nested region *Enter* and *Leave* events. This makes it impossible to reconstruct dynamic call paths or correctly attribute performance metrics to individual task regions without further measures. We addressed this issue by extending OPARI2 (Lorenz et al., 2010) and Score-P (Lorenz et al., 2012) to distinguish and track individual task instances, as well as their suspension and resumption. OpenMP 4.0 then introduced device constructs that target accelerators. These constructs were examined by Dietrich et al. (2014) in the context of Intel MIC devices and OPARI2. Any source-code-level approach to recording OpenMP device construct events is fundamentally limited. Not only is the device-side activity inaccessible, but source code transformations may also introduce unnecessary synchronization compared to the uninstrumented application. Therefore, device constructs are not supported by OPARI2. These shortcomings do not exist in OMPT, the preferred OpenMP instrumentation method since Score-P v9.0.

OMPT, introduced with OpenMP 5.0, is an API designed by the OpenMP Language Committee to support the construction of portable, efficient, and vendor-neutral performance tools. An initial draft was described by Eichenberger et al. (2013) and

9 The OPARI2 parser can be extended to instrument other `pragma`-based language extensions (Jiang et al., 2014).

compared to OPARI2 in Lorenz et al. (2014). OMPT, as specified, is a callback-based API that allows registration of callbacks of interest at program start. It provides a view of a vendor-specific OpenMP runtime. OMPT events generally contain an address, which allows tools to map these events to the corresponding source code location. When an address is available, we use our address-to-line lookup service (see Section 3.1.5) for this mapping. The same OpenMP program, when instrumented by OMPT and OPARI2, generates a similar, but not identical, series of OpenMP events. Code that relied on the OPARI2 event sequence, therefore, needed to be adapted. Feld et al. (2019) described the adaptations required in Score-P and the analysis tools, and examined the overhead associated with introducing either OPARI2 or OMPT instrumentation to applications. However, the barrier and worksharing constructs exhibited higher overhead than before due to contended address lookups. A new, atomics-based hash table implementation solved this problem. Measurement overhead was reexamined by Reuter et al. (2025), who also addressed a primary challenge of providing the initial OMPT implementation in Score-P v8.0: detecting runtimes that deviate from the specification. Such deviations are now detected early during the installation of Score-P and addressed by implementing workarounds, deactivating a specific feature, or even removing OMPT support from Score-P. Our OpenMP test suite, especially the `ompt-printf` tool (see Section 3.3.2), helped us detect these deviations and file bug reports with the runtime vendors. OPARI2 and OMPT coexist in a Score-P installation, with only one being active at a time. Since Score-P v9.0, OMPT is the default for OpenMP instrumentation if available. Clang-based runtimes usually provide the necessary headers and libraries, whereas GNU compilers lack support so far.

Unlike OPARI2, OMPT also supports the recording of device construct events. Hence, Score-P v9.0 extended its OMPT implementation to support those as well. This is described in more detail in the following section.

3.2.7 Accelerator adapters

With the significant adoption of accelerators and their respective programming models in HPC, supporting both is crucial for performance analysis. The CUPTI (NVIDIA Corporation, 2025a) library for CUDA, a collaboration between NVIDIA and the performance tools community (Malony et al., 2011), was the first to provide a dedicated interface for capturing performance data. Most accelerator programming models have followed suit, with OpenCL being the only exception. Score-P has supported CUDA since its inception and added support for OpenCL in version 1.4, as well as for other paradigms in subsequent years. As of release 9.0, Score-P covers CUDA, HIP, OpenMP Target, OpenACC, OpenCL, and Kokkos as accelerator-based paradigms.

In our data model, accelerators and their corresponding API abstractions are represented similarly to host-side entities, where threads are modeled as locations and processes as location groups. That is, we model streams as locations and contexts as location groups.[10] The adapters may additionally record lower-level information such as devices as elements of the system tree as appropriate, for example, see Figure 11. OTF2's requirement to record events per location in chronological order imposes certain restrictions on how we use these tools interfaces. Host-side events must be recorded via callbacks or function interception; otherwise, the event stream for that location may have moved on. Device-side events, on the other hand, are naturally received via a notification callback, which provides a buffer of event records for processing. For a variety of reasons, these events may not arrive in perfect chronological order when using buffer-based tools interfaces. There are three major cases we are concerned with: *overlap from imprecision*, where conversion to our unified clock results in some nanoseconds of overlap between two actually-sequential events; *out-of-order delivery in a single buffer*, which can be handled by sorting, truncating, or dropping the out-of-order events; and *out-of-order delivery across multiple buffers*, which can only be handled by truncating or dropping unless we perform a global sort of all GPU events at program exit. Currently, we truncate events when possible and drop them entirely otherwise. In practice, very few events are delivered out-of-order in a single buffer, regardless of the tool's interface, and out-of-order delivery across multiple buffers is still a rare edge case. This allows us to avoid expensive sorting operations, albeit at the cost of occasional dropped or truncated events. This general architecture is common across CUDA, HIP, and OpenMP's OMPT interface for target offload. The events and metrics we seek to collect are also broadly similar across these adapters. Score-P records when kernels are launched by the host, when and where each launch is executed, and when the host is waiting for their completion. Similarly, Score-P captures when and where data transfers start and end and which participants are waiting for the transfer, where possible. The adapters track memory allocation and deallocation of the various memory kinds available in these programming models. To record events in line with our model, Score-P must track the creation and deletion of streams and threads. Score-P models data transfers to and from accelerators using the same set of RMA events used by the MPI adapter. Kernel executions are equivalent to function calls on a device location, with appropriate region roles associated. Since tracking all cases of synchronization is known to be challenging, we focus on recording explicit synchronizations triggered by users. For most of these adapters, it is necessary—or at least desirable—to correlate data about host-side events with the corresponding device-side events. Based on the current call path, we create a unique hash as a correlation identifier between a host-side call and a device-side event. We store this information for the host-side event and retrieve it during device-side event handling.

OpenACC, OpenCL, and Kokkos only provide host-side events, and thus their adapters are purely callback-based. OpenACC and Kokkos rely on their tools interfaces, whereas OpenCL relies on library wrapping; see Section 3.2.4. To get a full picture, these interfaces must be combined with a native accelerator adapter, such as CUDA or HIP. In this case, correlation between host-side events and device-side activity is possible only via lower-level driver calls. The same holds true if the OMPT interface does not support device-side event buffers.

10 If contexts are not applicable for a paradigm, we create a single artificial context.

3.2.8 Additional adapters

There are two major areas of performance measurement covered by Score-P that rely on adapters not yet discussed. These are memory allocation tracking and I/O measurements. Score-P provides an adapter for memory allocation that tracks C and C++ language-level allocation and deallocation calls, as well as the high-bandwidth memory allocator from the `memkind` library (pmem.io, 2025). In addition to exposing the call regions for allocation and deallocation, this adapter provides per-process allocation metrics. To accomplish this, it must track the allocation size associated with each pointer. This allocation metric mechanism was then extended to track per-device metrics for use in various accelerator adapters.

In addition to recording MPI file I/O as part of the MPI adapter, Score-P also records POSIX and ISO C I/O events. The Score-P event model for I/O maps the various I/O calls to a generic set of file operations and, where relevant, records the size of data transfers, which is described in detail in Tschüter et al. (2018). Both adapters rely on library wrapping (see Section 3.2.4) for their instrumentation.

3.3 Ensuring code quality and portability

The landscape of HPC systems is diverse, constantly changing, and often custom. Thus, the distributed development team invested early on in software quality, automated builds, tests, and deployment. Our goal is to support relevant technologies that evolve alongside the TOP500 list (TOP500.org, 2025), including the machines hosted by the authors' institutions. Currently, the most relevant CPU architectures are x86-64 and ARM, typically combined with accelerators from NVIDIA, AMD, and Intel. Linux is used as the operating system, and various software toolchains—that is, version-dependent combinations of compilers, MPI implementations, accelerator programming environments, and support libraries—are used on top. To ensure compatibility with this vast number of combinations in both shared and static library modes, we leverage an extensive CI/CD infrastructure (Feld et al., 2021). It encompasses real-world HPC systems as well as customized container setups.[11] A successful CI/CD pipeline run results in a public, self-contained tarball[12] that installs flawlessly on user systems.

3.3.1 Development model

Score-P development is built around two Git branches: the main development branch and the current release branch. The main development branch contains pre-production code that eventually becomes the next *feature release*. New features are developed in separate branches, which are thoroughly reviewed and discussed via GitLab merge requests (GitLab Inc., 2025) targeting the main development branch. Each push to a branch corresponding to a merge request triggers a *merge request pipeline*. This pipeline performs prerequisite checks on the code to guarantee a consistent coding style and adherence to conventions before building a ready-to-use tarball in a container environment. To build the tarball, we use GNU Automake's `distcheck` target (Free Software Foundation, 2025a), which ensures that the generated tarball builds successfully and that the provided checks pass under the given configuration. These checks test the Score-P library internals and possible combinations of instrumentation, using the installed product (`installcheck` target). If these checks pass, we have confidence that the instrumentation process inserts hooks and libraries as expected. Failures are communicated in the merge request discussion and via email. The development tarballs generated by the pipeline are publicly available,[13] making it easy to share them for further testing.

Once a feature merge request has been approved by senior developers, it is merged. Then, a different CI/CD pipeline—the *full pipeline*—is invoked. Here, the prerequisite checks are omitted, while the tarball is built using the same setup as during the merge request. Additionally, this tarball is deployed to several real-world HPC systems. There, it is built and tested using available toolchains, which increases the feature coverage compared to the restricted container environment. However, depending on system load and file system technology, this process can take several hours. Ideally, a successful full pipeline would be a prerequisite for a merge. Therefore, we are investigating using GitLab *webhooks* to invoke the full pipeline before the merge, rather than after. Currently, this pipeline can be manually triggered at any time to help identify and address portability challenges early on. Successful pipelines leave tarball installations on the systems for future use. Alternative installation methods via package management systems, such as EasyBuild (Hoste et al., 2012), Spack (Gamblin et al., 2015), and OpenHPC (Schulz et al., 2016), have grown in popularity. We therefore also provide corresponding recipes for our official releases that use the successfully tested tarballs. These recipes can easily be integrated into scientific software stacks, such as EESSI (Dröge et al., 2023) built on EasyBuild and E4S (Willenbring et al., 2024) built on Spack.

At some point, the main development branch becomes the root of the next release branch. It uses the same *full pipeline* process after incorporating merge requests, which are allowed here only to include bug fixes. Of course, developers are obliged to also port bug fixes to the main development branch. With the full pipeline in place for feature and bug fix development, we automatically provide the latest changes as ready-to-use tarballs and installations on HPC systems. The first tag on a new release branch then becomes the next *feature release*; subsequent tags denote *bug-fix releases* that do not contain any new features. Tagging also triggers a full pipeline; the only difference is that the installation prefix on HPC systems changes from temporary to persistent.

3.3.2 Test suites

The tests outlined above guarantee the successful installation of Score-P, functional library internals, and the instrumentation of

11 See, for example, the Score-P development Docker containers (Score-P Development Team, 2025b) and the instrumentation methods depicted in Figure 1.

12 https://perftools.pages.jsc.fz-juelich.de/cicd/scorep/

13 https://perftools.pages.jsc.fz-juelich.de/cicd/scorep/development.html

small test programs for specific toolchains. However, since these programs are not executed as part of the CI/CD pipelines, run-time behavior remains untested.

We intentionally outsourced run-time tests to existing, feature-specific test suites. One reason is to prevent licensing issues that might arise from including the test suite codebase in Score-P. Additionally, we would need to support all launcher environments of the systems that users or we build Score-P on. Instead, we execute feature-specific test suites in a limited number of toolchain environments under our control. Each test suite is a separate GitLab project that provides a pipeline recipe using a given Score-P tarball. They can be used to improve our library implementations and to customize `configure` time checks, for example, to introduce workarounds for situations where runtime implementations deviate from given standards or specifications. In the worst case, a feature can be disabled. The test suites aim at ensuring that correctly written programs still produce the correct results when instrumented. However, they are less suitable for verifying the correct Score-P event count or event order, which is likely to be nondeterministic in parallel programs, rendering comparisons to a reference run useless.

One of the test suites we use is the MPICH test suite (Argonne National Laboratory, 2025), which contains over 1000 tests designed to test API usage and stress the implementation. We modified the test suite for use with Score-P to address two areas where our needs differ from those of the MPICH developers. First, we excluded tests of erroneous MPI programs, as Score-P's behavior on incorrect programs is undefined. Second, we added additional tests to check for regressions when Score-P issues were fixed but not yet covered by existing tests.

In addition, we use the collection of examples provided with the OpenMP specification (OpenMP Architecture Review Board, 2025). This covers a large portion of the possible uses of OpenMP. Again, the collection of examples needed to be adapted for use with Score-P, with some erroneous tests being excluded. Additionally, the test suite has been extended to include known issues, for example, specific OpenMP patterns not provided by the examples. For OpenMP instrumentation, we transition from the source-to-source instrumenter OPARI2 to the OMPT interface. For the latter, we developed `ompt-printf` (Reuter et al., 2025) to examine how runtimes across different vendors and versions interpret the specification. With the help of this tool, we have filed approximately 80 bug reports between December 2022 and May 2025, which the compiler vendors have addressed. `ompt-printf` is run in addition to non-instrumented and Score-P-instrumented versions of the examples.

These test suites proved essential for implementing recent features in Score-P v9.0, such as support for *OpenMP target events* and *MPI Fortran 2008 bindings*. Similar test suites are used for CUDA and OpenACC.

4 Challenges and emerging directions

Keeping up with the evolution of the various parallel paradigms supported by Score-P is an ongoing effort, and tools interfaces provide the necessary connection between these two. While the MPI specification has provided a usable tools interface from the very beginning, tool support is, unfortunately, often not a first-class citizen for many APIs—and sometimes not even an afterthought. Attempting to aggregate data from essentially all APIs used in an application and trying to present performance data at the level of the corresponding API, Score-P is particularly strongly affected by this. That is, collecting data from sources ranging from system libraries (e.g., I/O and threads), where tools interfaces are essentially nonexistent, to GPU offloading, where a proper tools interface is essential, can be quite challenging. In our experience, co-designing an interface between API and tools developers leads to better results. This can ensure that an interface can be implemented with reasonable effort and overhead, while summarizing an ever-changing set of API functions into a fixed set of events appropriate for a broad range of tools. However, even when a tools interface is needed and well-specified, implementations may lag significantly behind in making tools-related functionality available. This suggests that raising awareness of the tools community's needs is generally required.

In the MPI area, supporting matched probes and receives is an ongoing effort (Haus, 2024). Support for partitioned communication is also work in progress (Thärigen et al., 2023). We are also undertaking an initial exploration of recording the layout of MPI datatypes and integrating this information with our existing MPI I/O recording to better understand the I/O access patterns associated with user-defined types. Integrating MPI sessions into Score-P, in particular adopting the idiom of "one session per tool"—as suggested by the MPI Forum—but also tracking the various active sessions and allowing control of which sessions are part of the measurement, presents an interesting challenge for future development. Finally, the MPI_T interface in principle allows us to augment the logical view of data, such as bytes transferred, with a more accurate physical view based on the work an implementation performs after optimization. In practice, this will require cooperation with the MPI Forum and implementers to define consistent semantics for MPI_T variables, followed by implementation in Score-P to take advantage of this.

With respect to accelerators, we also identified some potential for improvements and enhancements in Score-P. For example, we are actively developing a generic event model for accelerator offloading that focuses on recording the host-device correlation of activities. We plan to leverage additional kernel properties, such as queues, available in newer tools interfaces, to augment measurement data. Moreover, we are in the process of adding support for accelerator-based collective communication libraries such as NCCL and RCCL. Finally, both CUDA and ROCm provide interfaces to sample device-side instructions, enabling deeper insights into kernel execution beyond what is already captured in a Score-P measurement. Other GPU metrics can be sampled asynchronously via PAPI.

With our transition to the GOTCHA-based library wrapping (see Section 3.2.4), it has become more urgent to extend Score-P's filtering capabilities to ignore whole threads and support multiple threading adapters concurrently. In particular, the combination of Pthread and OMPT measurement has great practical significance, as many third-party libraries use POSIX threads, and these calls are more readily visible now. We are developing a method to

detect, at Pthread creation time, the threading model to which a thread actually belongs. This will enable us to delay the creation of locations for OpenMP threads until the parallel region is encountered, which is a prerequisite for supporting Pthread and OpenMP adapters concurrently.

Regarding compiler instrumentation, we are evaluating the XRay adapter prototype mentioned in Section 3.2.1 for inclusion in Score-P. XRay relies on compilers inserting code at function entries and exits, as well as recording those locations in tables. It strikes a balance between our sophisticated compiler plug-ins and flag-based compiler instrumentation, enabling (almost) zero-overhead run-time filtering. The prototype adapter is designed around and tested against LLVM's XRay implementation.

The increasing number and complexity of supported programming paradigms are also posing challenges for testing and CI. First, developing and maintaining test suites with decent coverage is already a major effort. This is further aggravated by the need to validate measurement results to ensure correct behavior. Also, early access to vendor-specific compilers, runtimes, and appropriate hardware is essential but often not readily available. This has resulted in severe regressions in compilers and runtimes in the past, causing significant effort to work around them or requiring the deactivation of entire features within Score-P. Similar to tools interfaces, increased collaborative efforts could help to alleviate these issues.

Another area that requires further research is how to improve the presentation of measurement data from asynchronous executions, in particular with respect to profiles. This applies to all task-based parallel programming models, including OpenMP host-side tasking and accelerator offloading of any kind. To the best of our knowledge, all existing performance tools present tasks and/or accelerator kernels as flat lists—including Score-P profiles viewed in CubeGUI or TAU's ParaProf. However, for complex codes with many tasks and/or kernels, these can get cumbersome to analyze. In addition, properly handling task dependencies exacerbates the situation. Here, the tools community needs to develop novel ideas for structuring such data and making it more accessible to users.

While Score-P currently records measurement configurations, it lacks the capabilities to capture metadata about the application itself. Furthermore, Score-P does not provide infrastructure to correlate generated outputs with application-specific context, leaving performance results detached from their execution environment and provenance. To address these limitations, we have proposed enhancing Score-P by integrating metadata management capabilities based on the Resource Description Framework (RDF) (Sander et al., 2025). This RDF-based extension will enable comprehensive provenance tracking of compilation and linking processes, run-time dependencies, and user-specified environment variables. Expressing performance metadata as RDF helps implement the FAIR Guiding Principles—natively delivering interoperability, and, when published with persistent identifiers, open web protocols, indexing, licenses, and extensive provenance, improving findability, accessibility, and reusability (Wilkinson et al., 2016).

5 Related work

Originally, Score-P also served as measurement infrastructure for the online performance analysis tool Periscope (Benedict et al., 2010). Over time, this tool was extended into the Periscope Tuning Framework (Mijaković et al., 2016) for run-time auto-tuning applications. The necessary connection between Score-P and Periscope's agent network was initially tightly integrated into Score-P's measurement core but has later been factored out into a custom substrate plug-in. Meanwhile, however, the Periscope projects have been discontinued.

Due to the importance of performance analysis in HPC application development, many other tools have been developed to address various aspects. A detailed overview of typical HPC tools can be found in Mohr (2014) and Knobloch and Mohr (2020). Vendor-specific tools offer the most comprehensive features for the corresponding vendors' architectural frameworks. For instance, the recently discontinued Intel Trace Analyzer and Collector (ITAC) (Wrinn and Asbury, 2004) was specifically designed for MPI-based codes. ITAC allowed users to visualize and examine the MPI communication to identify potential bottlenecks in their applications. It was accompanied by the Intel Advisor, which provides insights into node-level application performance, including threading, vectorization, and memory use. While neither tool was exclusively designed for Intel hardware, both were optimized for it. NVIDIA offers Nsight Systems (NVIDIA Corporation, 2025c) and Nsight Compute (NVIDIA Corporation, 2025b) to gain insights into applications using NVIDIA GPUs. Nsight Systems grants a system-wide view of the application, including CPU and GPU interactions, while Nsight Compute focuses on kernel-level performance analysis. Using sampling techniques and collecting low-level hardware events, both tools provide a comprehensive understanding of individual-kernel and overall-application performance. Similarly, AMD provides several tools built on top of the ROCm software ecosystem, such as ROCprofiler (AMD, 2025) and Omnitrace (AMD, 2024). All of these vendor-specific tools are generally very powerful and provide more fine-grained information than Score-P. However, they are limited to the hardware or framework for which they were designed, thus delivering only a partial view of application performance across different programming models. Furthermore, most of these tools are designed exclusively for analyzing single-node performance. They may lack the capacity to effectively scale to large HPC systems and tend to become more complex when interpreting multi-node results. On the other hand, Score-P and its tool ecosystem have been designed with scalability in mind and have been successfully used with more than 1.8 million concurrent threads (Wylie, 2018).

By contrast, other open-source tools widely used in the HPC community are not tied to a specific hardware architecture. HPCToolkit (Tallent et al., 2008) is a tool suite using an asynchronous sampling approach. Unlike Score-P's hybrid recording mode (see Section 3.2.2), HPCToolkit does provide insights into the internals of parallel APIs. Data can be collected as both aggregated profiles and traces (sample-based time series) and interactively examined in the graphical user interface HPCViewer. Entirely based on event traces is the Barcelona Supercomputing

Center (BSC) tools suite. Its core components are the Extrae measurement system and the Paraver and Dimemas tools (Pillet et al., 1995). While Paraver is a highly configurable trace visualizer that can also calculate statistics, Dimemas can simulate the impact of varying network latency and bandwidth on application performance. This suite has meanwhile been complemented by tools for performance analytics such as clustering (Gonzalez et al., 2009), tracking (Llort et al., 2013) and folding (Servat et al., 2012), as well as the multi-level simulator MUSA (Grass et al., 2016). In this sense, the BSC tools suite also forms an ecosystem that addresses various aspects of performance analysis, similar to Score-P.

Finally, there are also a number of trace-based tools that have adopted OTF2 as their trace file format, forming their own OTF2 ecosystem. One notable example is the lo2s monitoring tool (Ilsche et al., 2017), a lightweight node-level performance monitoring tool used to analyze applications, the operating system, and hardware. The OTF2 calling context event model (see Section 3.2.2) was co-developed with the lo2s developers. Other tools that use OTF2 include Ravel (Isaacs et al., 2014), EZtrace (Trahay et al., 2011), ViTE (Coulomb et al., 2012), and TALP (Lopez et al., 2021). Babeltrace (Babeltrace Development Team, 2025) is an established interoperability tool that allows conversion of many other trace data formats to and from OTF2. It is widely used both to visualize Score-P-generated data in other tools and to visualize results from other measurement systems in Vampir. Additionally, several tools provide converters from OTF2 to their native format—either as part of the product or via third-party plug-ins—including the Intel Trace Analyzer and Collector and Eclipse Trace Compass (Eclipse Foundation, 2025).

6 Conclusion

The Score-P instrumentation and measurement infrastructure has emerged as a major vehicle in the HPC performance tools community and has been widely adopted. Designed to address the critical need to tune and scale HPC applications, Score-P is a community-driven, highly scalable, and easy-to-use tool suite that supports both profiling and event tracing of massively parallel codes. Its architecture has evolved into a flexible and extensible design, enabling the composable integration of various event and data sources and supporting complementary analysis tools via well-defined interfaces and common data formats.

Score-P provides users with comprehensive measurement data and an established ecosystem of diverse analysis tools that enable deep insight into application communication, synchronization, I/O, and scaling behavior. This allows users to pinpoint performance bottlenecks and their root causes. The ecosystem includes tools such as Cube for interactive profile analysis, Extra-P for empirical performance modeling, TAU for cross-experiment data mining,[14] Scalasca for automatic trace analysis, and Vampir for visual trace analysis. Over the past decade, Score-P has continually adapted to the rapidly changing landscape of HPC hardware and parallel APIs. Significant enhancements have been implemented, including support for Fortran 2008 MPI language bindings, the OpenMP tools interface OMPT, and a wide array of accelerator paradigms, such as CUDA, HIP, OpenMP Target, OpenACC, OpenCL, and Kokkos.

Despite these achievements, the performance tools community—Score-P in particular—faces ongoing challenges due to the growing complexity and heterogeneity of HPC systems, both in hardware and software. Therefore, we have identified and outlined several key areas for future research and development. Through continual design evolution, adoption of new technologies, and active engagement with the HPC community and API developers, the Score-P developer community remains dedicated to delivering essential, scalable, and insightful performance measurement capabilities for the next generation of top-tier computing systems.

14 Unlike the other tools, TAU can also be used as a standalone tool suite.

Data availability statement

The software covered in the article is available from the following locations: Score-P, OTF2, OPARI2: https://score-p.org; Cube, Scalasca: https://www.scalasca.org; Extra-P: https://github.com/extra-p/extrap; TAU: https://www.cs.uoregon.edu/research/tau; Vampir: https://vampir.eu.

Author contributions

CF: Writing – review & editing, Funding acquisition, Visualization, Resources, Software, Writing – original draft, Project administration, Conceptualization. AC: Writing – original draft, Visualization. GC: Software, Writing – review & editing, Writing – original draft, Visualization. MG: Conceptualization, Visualization, Software, Writing – review & editing, Writing – original draft, Funding acquisition, Resources, Project administration. M-AH: Visualization, Software, Writing – review & editing. MK: Resources, Funding acquisition, Writing – original draft, Visualization, Software, Writing – review & editing. BM: Resources, Writing – review & editing, Funding acquisition, Supervision, Conceptualization. JR: Writing – original draft, Software, Writing – review & editing. MSa: Software, Writing – original draft. PS: Visualization, Software, Writing – original draft. MSc: Writing – original draft, Visualization, Software, Funding acquisition, Writing – review & editing. RS: Software, Funding acquisition, Conceptualization, Writing – review & editing, Writing – original draft. SS: Resources, Conceptualization, Visualization, Funding acquisition, Writing – original draft, Software. AV: Writing – review & editing, Software. BW: Visualization, Conceptualization, Resources, Project administration, Funding acquisition, Writing – original draft, Software, Writing – review & editing. WW: Writing – review & editing, Software, Conceptualization, Writing – original draft, Funding acquisition, Visualization, Project administration. FW: Conceptualization, Supervision, Writing – review & editing, Funding acquisition. BJNW: Funding acquisition, Resources, Writing – review & editing. MZ: Writing – original draft, Software.

Funding

The author(s) declared that financial support was received for this work and/or its publication. This study was supported by

funding from the European Union's Horizon Europe research and innovation programme via the European HPC Joint Undertaking (EuroHPC JU), Grant 101143931 (The Performance Optimisation and Productivity Centre of Excellence, POP CoE, in HPC), and Grant 101033975 (European Pilot for Exascale, EUPEX), the Free State of Saxony and the Federal Government of Germany, Grant NHR2025SN (NHR@TUD), the State of North Rhine-Westphalia of Germany (HPC.NRW), the German Federal Ministry of Research, Technology and Space (BMFTR), Grant 16ME0630 (Energy-Efficient HPC Through Optimized Simulation Methods, ENSIMA), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), Project No. 545608190 (EnergyDoldrums), the German Federal Ministry of Research, Technology and Space (BMFTR), Funding No. NHR2021HE, and the Hessian Ministry of Higher Education, Research and the Arts (HMWK), Kapitel 1502, Förderprodukt 19 NHR4CES, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), Grant 449683531 (ExtraNoise), and the U.S. National Science Foundation, Award OAC-2311831. The following organizations have previously provided grants to support the development of Score-P: the European Union's Horizon 2020 programme, the European Union's 7th Framework Programme for Research (FP7), the European Union's ITEA programme, the German Federal Ministry of Education and Research (BMBF), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), and the U.S. Department of Energy (DOE). The authors declare that the development of Score-P received funding from Advanced Micro Devices, Inc. (AMD), Intel Corporation, Nvidia Corporation, and Siemens AG. These funders were not involved in the study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Conflict of interest

The author(s) declared that this work was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declared that generative AI was not used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

AMD (2024). *Omnitrace: Application Profiling, Tracing, and Analysis*. Available online at: https://rocm.docs.amd.com/projects/omnitrace/en/latest/doxygen/html (Accessed July 28, 2025).

AMD (2025). *Using rocprofv3 - ROCprofiler-SDK*. Available online at: https://github.com/ROCm/rocprofiler-sdk (Accessed July 24, 2025).

an Mey, D., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., et al. (2012). "Score-P - a unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010, (CiHPC-2010), Proceedings of an International Conference on Competence in High Performance Computing, June 2010, Schloss Schwetzingen*, Germany, eds. C. Bischof, H. G. Hegering, W. E. Nagel, and G. Wittum (Cham: Springer), 85–97. doi: 10.1007/978-3-642-24025-6_8

Argonne National Laboratory (2025). *MPICH: High-Performance Portable MPI – Downloads*. Available online at: https://www.mpich.org/downloads/ (Accessed July 24, 2025).

Babeltrace Development Team (2025). *Babeltrace, An Open-Source Trace Manipulation Toolkit*. Available online at: https://github.com/efficios/babeltrace (Accessed September 5, 2025).

Becker, D., Geimer, M., Rabenseifner, R., and Wolf, F. (2013). Extending the scope of the controlled logical clock. *Cluster Comput.* 16, 171–189. doi: 10.1007/s10586-011-0181-8

Benedict, S., Petkov, V., and Gerndt, M. (2010). "PERISCOPE: an online-based distributed performance analysis tool," in *Tools for High Performance Computing 2009*, eds. M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel (Berlin: Springer), 1–16. doi: 10.1007/978-3-642-11261-4_1

Berris, D. M., Veitch, A., Heintze, N., Anderson, E., and Wang, N. (2016). *Xray: A function call Tracing System*. Mountain View, CA: Technical report, Google.

Böhme, D., de Supinski, B. R., Geimer, M., Schulz, M., and Wolf, F. (2012). "Scalable critical-path based performance analysis," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China* (Shanghai: IEEE), 1330–1340. doi: 10.1109/IPDPS.2012.120

Böhme, D., Geimer, M., Wolf, F., and Arnold, L. (2010). "Identifying the root causes of wait states in large-scale parallel applications," in *Procedings of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA* (San Diego, CA: IEEE), 90–100. doi: 10.1109/ICPP.2010.18

Brendel, R., Wesarg, B., Tschüter, R., Weber, M., Ilsche, T., and Oeste, S. (2019). "Generic library interception for improved performance measurement and insight," in *Programming and Performance Visualization Tools*, eds. A. Bhatele, D. Boehme, J. A. Levine, A. D. Malony, and M. Schulz (Cham: Springer International Publishing), 21–37. doi: 10.1007/978-3-030-17872-7_2

Calotoiu, A., Beckingsale, D., Earl, C. W., Hoefler, T., Karlin, I., Schulz, M., et al. (2016). "Fast multi-parameter performance modeling," in *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER)* (Taipei: IEEE Computer Society), 1–10. doi: 10.1109/CLUSTER.2016.57

Calotoiu, A., Hoefler, T., Poke, M., and Wolf, F. (2013). "Using automated performance modeling to find scalability bugs in complex codes," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA* (New York, NY: ACM), 1–12. doi: 10.1145/2503210.2503277

Coleman, T., Casanova, H., Pottier, L., Kaushik, M., Deelman, E., Ferreira da Silva, R. et al. (2022). WfCommons: a framework for enabling scientific workflow research and development. *Future Gener. Comput. Syst.* 128, 16–27. doi: 10.1016/j.future.2021.09.043

Corbin, G. (2025). "Performance measurements of modern Fortran MPI applications with Score-P," in *Tools for High Performance Computing 2024 - Proceedings of the 15th International Workshop on Parallel Tools for High Performance Computing*, eds. H. Mix, C. Niethammer, B. Wesarg, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing).

Coulomb, K., Degomme, A., Faverge, M., and Trahay, F. (2012). "An open-source tool-chain for performance analysis," in *Tools for High Performance Computing 2011*,

eds. H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch (Berlin: Springer), 37–48. doi: 10.1007/978-3-642-31476-6_4

Cube Development Team (2025). *Cube 4.x Download*. Available online at: https://www.scalasca.org/scalasca/software/cube-4.x/ (Accessed July 29, 2025).

Dietrich, R., Ilsche, T., and Juckeland, G. (2010). "Non-intrusive performance analysis of parallel hardware accelerated applications on hybrid architectures," in *2010 39th International Conference on Parallel Processing Workshops* (San Diego, CA: IEEE), 135–143. doi: 10.1109/ICPPW.2010.30

Dietrich, R., Schmitt, F., Grund, A., and Schmidl, D. (2014). "Performance measurement for the OpenMP 4.0 offloading model," in *Euro-Par 2014: Parallel Processing Workshops*, eds. L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, et al. (Cham: Springer International Publishing), 291–301.

Dröge, B., Holanda Rusu, V., Hoste, K., van Leeuwen, C., O'Cais, A., Röblitz, T., et al. (2023). EESSI: a cross-platform ready-to-use optimised scientific software stack. *Softw. Pract. Exp*. 53, 176–210. doi: 10.1002/spe.3075

Duca, N., and Sinclair, D. (2016). *Trace Event Format*. Technical report, Google. Available online at: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview (Accessed July 24, 2025).

Eclipse Foundation (2025). *Trace Compass*. Available online at: https://eclipse.dev/tracecompass (Accessed July 24, 2025).

Eichenberger, A. E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., et al. (2013). "OMPT: an OpenMP tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators, Vol. 8122 of LNCS. 9th International Workshop on OpenMP, Canberra (Australia), 16 Sep 2013 - 18 Sep 2013* (Berlin: Springer), 171–185. doi: 10.1007/978-3-642-40698-0_13

Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W. E., Wolf, F., et al. (2012). "Open trace format 2: the next generation of scalable trace formats and support libraries," in *Applications, Tools and Techniques on the Road to Exascale Computing, 2012, Advances in Parallel Computing, Vol. 22*, eds. K. De Bosschere, E. H. D'Hollander, G. R. Joubert, D. Padua, F. Peters, and M. Sawyer (Amsterdam: IOS Press), 481–490.

Extra-P Development Team (2025). *Extra-P, Automated Performance Modeling for HPC Applications*. Available online at: https://github.com/extra-p/extrap (Accessed July 28, 2025).

Feld, C., Convent, S., Hermanns, M.-A., Protze, J., Geimer, M., Mohr, B., et al. (2019). "Score-P and OMPT: navigating the perils of callback-driven parallel runtime introspection," in *OpenMP: Conquering the Full Hardware Spectrum: 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings 15* (Cham: Springer), 21–35. doi: 10.1007/978-3-030-28596-8_2

Feld, C., Geimer, M., Hermanns, M.-A., Saviankou, P., Visser, A., Mohr, B., et al. (2021). "Detecting disaster before it strikes: on the challenges of automated building and testing in HPC environments," in *Tools for High Performance Computing 2018/2019/Mix, Hartmut (Editor)*; *Cham: Springer International Publishing, 2021, Chapter 1. 12th International Parallel Tools Workshop, Stuttgart (Germany), 17 Sep 2018 - 18 Sep 2018* (Cham: Springer International Publishing), 3–26. doi: 10.1007/978-3-030-66057-4

Free Software Foundation (2023). *The GNU C Library - Resource Usage*. Available online at: https://www.gnu.org/software/libc/manual/html_node/Resource-Usage.html (Accessed July 28, 2025).

Free Software Foundation (2025a). *GNU Automake Manual: Checking the Distribution*. Available online at: https://www.gnu.org/software/automake/manual/html_node/Checking-the-Distribution.html (Accessed July 24, 2025).

Free Software Foundation (2025b). *Using the GNU Compiler Collection: Built-in Functions for Memory Model Aware Atomic Operations*. Available online at: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html (Accessed July 27, 2025).

Gamblin, T. (2010). *wrap.py: A PMPI Wrapper*. Available online at: https://github.com/LLNL/wrap (Accessed July 21, 2025).

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., de Supinski, B. R., et al. (2015). *The Spack Package Manager: Bringing Order to HPC Software Chaos. Supercomputing 2015 (SC–15)* (Austin, TX). doi: 10.1145/2807591.2807623

Geimer, M., Saviankou, P., Strube, A., Szebenyi, Z., Wolf, F., Wylie, B. J. N., et al. (2012). "Further improving the scalability of the scalasca toolset," in *Applied Parallel and Scientific Computing*, eds. K. Jónasson (Cham: Springer Berlin Heidelberg), 463–473. doi: 10.1007/978-3-642-28145-7_45

Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., Mohr, B., et al. (2010). The Scalasca performance toolset architecture. *Concurr. Comput. Pract. Exp*. 22, 702–719. doi: 10.1002/cpe.1556

Geimer, M., Wolf, F., Wylie, B. J. N., and Mohr, B. (2006). "Scalable parallel trace-based performance analysis," in *Proceedings of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Bonn, Germany, Vol. 4192 of LNCS* (Cham: Springer), 303–312. doi: 10.1007/11846802_43

Geimer, M., Wolf, F., Wylie, B. J. N., and Mohr, B. (2009). A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Comput*. 35, 375–388. doi: 10.1016/j.parco.2009.02.003

GitLab Inc. (2025). *GitLab Documentation: Merge Requests*. Available online at: https://docs.gitlab.com/user/project/merge_requests/ (Accessed July 24, 2025).

Gocht, A., Schöne, R., and Frenzel, J. (2021). "*Advanced Python Performance Monitoring with Score-P*," in *Tools for High Performance Computing 2018/2019 - Proceedings of the 12th and of the 13th International Workshop on Parallel Tools for High Performance Computing, Stuttgart, Germany, September 2018, and Dresden, Germany, September 2019*, eds. H. Mix, C. Niethammer, H. Zhou, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing), 261–270. doi: 10.1007/978-3-030-66057-4_14

Gonzalez, J., Gimenez, J., and Labarta, J. (2009). "Automatic detection of parallel applications computation phases," in *2009 IEEE International Symposium on Parallel Distributed Processing* (Rome: IEEE), 1–11. doi: 10.1109/IPDPS.2009.5161027

Grass, T., Allande, C., Armejach, A., Rico, A., Ayguadé, E., Labarta, J., et al. (2016). "MUSA: a multi-level simulation approach for next-generation HPC machines," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, UT: IEEE), 526–537. doi: 10.1109/SC.2016.44

Haus, K. (2024). *Extending a Performance Analysis Tool to Handle MPI Message Probing*. (Bachelor's thesis). Aachen: FH Aachen - University of Applied Sciences.

Hoste, K., Timmerman, J., Georges, A., and De Weirdt, S. (2012). "EasyBuild: building software with ease," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (Salt Lake City, UT: IEEE), 572–582. doi: 10.1109/SC.Companion.2012.81

Huck, K. A., Shende, S., Malony, A. D., Coti, C., Spear, W., Alcaraz, J., et al. (2025). Preparing the TAU performance system for exascale and beyond. *Int. J. High Perform. Comput. Appl*. 39, 532–552. doi: 10.1177/10943420251334456

Hünich, D., Wesarg, B., and Tschüter, R. (2025). "Exploring multi-threaded communication behavior of a large-scale CFD solver with Vampir," in *Tools for High Performance Computing 2024 - Proceedings of the 15th International Workshop on Parallel Tools for High Performance Computing*, eds. H. Mix, C. Niethammer, B. Wesarg, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing).

Ilsche, T., Schöne, R., Bielert, M., Gocht, A., and Hackenberg, D. (2017). "lo2s–multi-core system and application performance analysis for Linux," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)* (Honolulu, HI: IEEE), 801–804. doi: 10.1109/CLUSTER.2017.116

Isaacs, K. E., Bremer, P.-T., Jusufi, I., Gamblin, T., Bhatele, A., Schulz, M., et al. (2014). Combing the communication hairball: visualizing parallel execution traces using logical time. *IEEE Trans. Vis. Comput. Graph*. 20, 2349–2358. doi: 10.1109/TVCG.2014.2346456

Jagode, H., Danalis, A., Congiu, G., Barry, D., Castaldo, A., Dongarra, J., et al. (2025). Advancements of PAPI for the exascale generation. *Int. J. High Perform. Comput. Appl*. 39, 251–268. doi: 10.1177/10943420241303884

Jiang, J., Philippen, P., Knobloch, M., and Mohr, B. (2014). *Performance Measurement and Analysis of Transactional Memory and Speculative Execution on IBM Blue Gene/Q, Vol. 8632 of Lecture Notes in Computer Science* (Berlin: Springer), 26–37. doi: 10.1007/978-3-319-09873-9_3

Knobloch, M., and Mohr, B. (2020). Tools for GPU computing-debugging and performance analysis of heterogenous HPC applications. *Supercomput. Front. Innovat*. 7, 91–111. doi: 10.14529/jsfi200105

Knobloch, M., Saviankou, P., Schlütter, M., Visser, A., and Mohr, B. (2021). "A picture is worth a thousand numbers–enhancing cube's analysis capabilities with plugins," in *Tools for High Performance Computing 2018/2019 - Proceedings of the 12th and of the 13th International Workshop on Parallel Tools for High Performance Computing, Stuttgart, Germany, September 2018, and Dresden, Germany, September 2019*, eds. H. Mix, C. Niethammer, H. Zhou, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing), 237–259. doi: 10.1007/978-3-030-66057-4_13

Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., 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*, eds. H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch (Berlin: Springer), 79–91. doi: 10.1007/978-3-642-31476-6_7

Kreuzer, S., Adelmann, P., and Bischof, C. (2025). "A runtime-adaptable instrumentation back-end for Score-P," in *Tools for High Performance Computing 2024 - Proceedings of the 15th International Workshop on Parallel Tools for High Performance Computing*, eds. H. Mix, C. Niethammer, B. Wesarg, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing).

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 558–565. doi: 10.1145/359545.359563

Liao, C., Quinlan, D. J., Willcock, J. J., and Panas, T. (2008). *Automatic Parallelization Using OpenMP Based on STL Semantics*. Livermore, CA : Lawrence Livermore National Laboratory (LLNL).

Linux Perf Wiki Contributors (2024). *perf: Linux Profiling with Performance Counters*. Available online at: https://perfwiki.github.io/main/ (Accessed July 28, 2025).

Llort, G., Servat, H., González, J., Giménez, J., and Labarta, J. (2013). "On the usefulness of object tracking techniques in performance analysis," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 29* (New York, NY: ACM), 11. doi: 10.1145/2503210.2503267

Lopez, V., Ramirez Miranda, G., and Garcia-Gasulla, M. (2021). "TALP: a lightweight tool to unveil parallel efficiency of large-scale executions," in *Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy, PERMAVOST '21* (New York, NY: Association for Computing Machinery), 3–10. doi: 10.1145/3452412.3462753

Lorenz, D., Dietrich, R., Tschüter, R., and Wolf, F. (2014). "A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P," in *Using and Improving OpenMP for Devices, Tasks, and More*, eds. L. DeRose, B. R. de Supinski, S. L. Olivier, B. M. Chapman, and M. S. Müller (Cham: Springer International Publishing), 161–172. doi: 10.1007/978-3-319-11454-5_12

Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., and Wolf, F. (2010). "How to reconcile event-based performance analysis with tasking in OpenMP," in *Proceedings of 6th International Workshop of OpenMP (IWOMP), Tsukuba, Japan, Vol. 6132 of Lecture Notes in Computer Science* (Cham: Springer), 109–121. doi: 10.1007/978-3-642-13217-9_9

Lorenz, D., Philippen, P., Schmidl, D., and Wolf, F. (2012). "Profiling of OpenMP tasks with Score-P," in *2012 41st International Conference on Parallel Processing Workshops (ICPPW)*, (Pittsburgh, PA: IEEE), 10. doi: 10.1109/ICPPW.2012.62

Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., et al. (2011). "Parallel performance measurement of heterogeneous parallel systems with GPUs," in *2011 International Conference on Parallel Processing* (Taipei: IEEE), 176–185. doi: 10.1109/ICPP.2011.71

Message Passing Interface Forum (2023). *MPI: A Message-Passing Interface Standard Version 4, 1*. Available online at: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (Accessed July 01, 2025).

Mijaković, R., Firbach, M., and Gerndt, M. (2016). "An architecture for flexible auto-tuning: the Periscope Tuning Framework 2.0," in *2016 2nd International Conference on Green High Performance Computing (ICGHPC)* (Nagercoil: IEEE), 1–9. doi: 10.1109/ICGHPC.2016.7508066

Mohr, B. (2014). Scalable parallel performance measurement and analysis tools - state-of-the-art and future challenges. *Supercomput. Front. Innovat.* 1, 108–123. doi: 10.14529/jsfi140207

Mohr, B., Malony, A. D., Shende, S., and Wolf, F. (2002). Design and prototype of a performance tool interface for OpenMP. *J. Supercomput.* 23, 105–128. doi: 10.1023/A:1015741304337

Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C., and Solchenbach, K. (1996). *Vampir: Visualization and Analysis of MPI resources* (Jülich).

NVIDIA Corporation (2025a). *NVIDIA CUDA Profiling Tools Interface*. Available online at: https://developer.nvidia.com/cupti (Accessed July 29, 2025).

NVIDIA Corporation (2025b). *NVIDIA Nsight Compute Documentation*. Available online at: https://docs.nvidia.com/nsight-compute/ (Accessed June 2, 2025).

NVIDIA Corporation (2025c). *NVIDIA Nsight Systems User Guide*. Available online at: https://docs.nvidia.com/nsight-systems/UserGuide/ (Accessed June 2, 2025).

OpenMP Architecture Review Board (2024). *The OpenMP API Specification*. Available online at: https://www.openmp.org/ (Accessed July 24, 2025).

OpenMP Architecture Review Board (2025). *OpenMP Examples Repository*. Available online at: https://github.com/OpenMP/Examples (Accessed July 24, 2025).

Pillet, V., Labarta, J., Cortes, T., and Girona, S. (1995). "Paraver: a tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and Occam developments, Vol. 44* (Manchester), 17–31.

pmem.io (2025). *Memkind*. Available online at: https://pmem.io/memkind/ (Accessed July 29, 2025).

Poliakoff, D., and LeGendre, M. (2019). "Gotcha: an function-wrapping interface for HPC tools," in *Programming and Performance Visualization Tools*, eds. A. Bhatele, D. Boehme, J. A. Levine, A. D. Malony, and M. Schulz (Cham: Springer International Publishing), 185–197. doi: 10.1007/978-3-030-17872-7_11

POP-CoE Partners (2022). *Multiplicative Performance Metrics for Hybrid Parallel Codes*. Available online at: https://pop-coe.eu/sites/default/files/pop_files/pop_hybrid_metrics_multiplicative_handout_v2.pdf (Accessed September 17, 2025).

POP-CoE Partners (2025). *Performance Optimisation and Productivity - A Centre of Excellence in HPC*. Available online at: https://pop-coe.eu (Accessed July 29, 2025).

Rasmussen, S., Schulz, M., and Mohror, K. (2016). "Allowing MPI tools builders to forget about Fortran," in *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI '16* (New York, NY: Association for Computing Machinery), 208–211. doi: 10.1145/2966884.2966889

Regents of the University of California (1999). *The 3-Clause BSD License*. Available online at: https://opensource.org/license/BSD-3-Clause (Accessed July 25, 2025).

Reisert, P., Calotoiu, A., Shudler, S., and Wolf, F. (2017). "Following the blind seer-creating better performance models using less information," in *Proceedings of the 23rd International Conference on Parallel Processing (Euro-Par)* (Cham: Springer), 106–118. doi: 10.1007/978-3-319-64203-1_8

Reuter, J. A., Feld, C., and Mohr, B. (2025). "Score-P and OMPT: smoothing the bumpy road to OpenMP performance measurement," in *Tools for High Performance Computing 2024 - Proceedings of the 15th International Workshop on Parallel Tools for High Performance Computing*, eds. H. Mix, C. Niethammer, B. Wesarg, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing).

Ruefenacht, M. (2024). *pympistandard*. Available online at: https://github.com/mpi-forum/pympistandard (Accessed July 1, 2025).

Sander, M., Williams, W. R., and Wesarg, B. (2025). "Working towards FAIRness in performance data," in *Proceedings of the 35th Parallel CFD International Conference 2024, volume 69 of Schriften des Forschungszentrums Jülich IAS Series. 35th Parallel CFD International Conference 2024, Bonn (Germany), 2 Sep 2024–4 Sep 2024* (Jülich: Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag), 192–196.

Saviankou, P., Knobloch, M., Visser, A., and Mohr, B. (2015). *Cube v4: From Performance Report Explorer to Performance Analysis Tool, volume 51 of Procedia Computer Science* (Amsterdam: Elsevier), 1343–1352. doi: 10.1016/j.procs.2015.05.320

Scalasca Development Team (2025). *Scalasca 2.x Download*. Available online at: https://www.scalasca.org/scalasca/software/scalasca-2.x/ (Accessed July 29, 2025).

Schöne, R., Tschüter, R., Ilsche, T., Schuchart, J., Hackenberg, D., and Nagel, W. E. (2017). "Extending the functionality of Score-P through plugins: interfaces and use cases," in *Tools for High Performance Computing 2016: Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing, October 2016, Stuttgart, Germany* (Cham: Springer), 59–82. doi: 10.1007/978-3-319-56702-0_4

Schulz, K. W., Baird, C. R., Brayford, D., Georgiou, Y., Kurtzer, G. M., Simmel, D., et al. (2016). "Cluster computing with OpenHPC," in *HPC Systems Professionals Workshop 2016* (Salt Lake City, UT).

Score-P Development Team (2025a). *Scalable Performance Measurement Infrastructure for Parallel Codes (Score-P)*. Available online at: https://doi.org/10.5281/zenodo.1240731 (Accessed October 31, 2025).

Score-P Development Team (2025b). *Score-P on Docker Hub*. Available online at: https://hub.docker.com/u/scorep (Accessed July 24, 2025).

Score-P Development Team (2025c). *Score-P Website*. Available online at: https://score-p.org (Accessed September 12, 2025).

Servat, H., Llort, G., Giménez, J., Huck, K., and Labarta, J. (2012). "Folding: detailed analysis with coarse sampling," in *Tools for High Performance Computing 2011*, eds. H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch (Berlin: Springer), 105–118. doi: 10.1007/978-3-642-31476-6_9

Shende, S., and Malony, A. D. (2006). The TAU parallel performance system, SAGE publications. *Int. J. High Perform. Comput. Appl.* 20, 287–331. doi: 10.1177/1094342006064482

Song, F., Wolf, F., Bhatia, N., Dongarra, J., and Moore, S. (2004). "An algebra for cross-experiment performance analysis," in *Proceedings of the International Conference on Parallel Processing (ICPP), Montreal, Canada* (Montreal, QC: IEEE Society), 63–72. doi: 10.1109/ICPP.2004.1327905

Szebenyi, Z., Gamblin, T., Schulz, M., Supinski, B. R., Wolf, F., and Wylie, B. J. (2011). "Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs," in *2011 IEEE International Parallel and Distributed Processing Symposium* (Anchorage, AK: IEEE), 640–651. doi: 10.1109/IPDPS.2011.67

Szebenyi, Z., Wolf, F., and Wylie, B. J. N. (2009). "Space-efficient time-series call-path profiling of parallel applications," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA* (New York, NY: ACM). doi: 10.1145/1654059.1654097

Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., and Krentel, M. (2008). HPCToolkit: performance tools for scientific computing. *J. Phys. Conf. Ser.* 125:012088. doi: 10.1088/1742-6596/125/1/012088

Thärigen, I., Hermanns, M.-A., and Geimer, M. (2023). "An event model for trace-based performance analysis of MPI partitioned point-to-point communication," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. Workshop on Programming and Performance Visualization Tools, Denver, CO (USA), 12 Nov 2023 - 12 Nov 2023* (New York, NY: ACM), 1357–1367. doi: 10.1145/3624062.3624205

TOP500.org (2025). *TOP500: The List*. Available online at: https://top500.org (Accessed July 24, 2025).

Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., Dongarra, J., et al. (2011). "EZTrace: a generic framework for performance analysis," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (New York, NY: ACM), 618–619. doi: 10.1109/CCGrid.2011.83

Tschüter, R., Herold, C., Wesarg, B., and Weber, M. (2018). "A methodology for performance analysis of applications using multi-layer I/O," in *Euro-Par 2018: Parallel Processing*, eds. M. Aldinucci, L. Padovani, and M. Torquati (Cham: Springer International Publishing), 16–30. doi: 10.1007/978-3-319-96983-1_2

Tschüter, R., Herold, C., Williams, W., Knespel, M., and Weber, M. (2019). "A top-down performance analysis methodology for workflows: tracking performance issues from overview to individual operations," in *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)* (Denver, CO: IEEE), 21–30. doi: 10.1109/WORKS49585.2019.00008

Tschüter, R., Ziegenbalg, J., Wesarg, B., Weber, M., Herold, C., Döbel, S., et al. (2017). "An LLVM instrumentation plug-in for Score-P," in *Proceedings of the Fourth*

Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC'17 (New York, NY: Association for Computing Machinery). doi: 10.1145/3148173.3148187

University of Oregon (2024). *TAU - Tuning and Analysis Utilities: Software Download Requests*. Available online at: https://www.cs.uoregon.edu/research/tau/downloads.php (Accessed July 29, 2025).

Vampir Development Team (2025). *Vampir - Performance Optimization*. Available online at: https://vampir.eu (Accessed September 10, 2025).

VI-HPS (2025). *Virtual Institute - High Productivity Supercomputing (VI-HPS)*. Available online at: https://www.vi-hps.org (Accessed July 29, 2025).

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., et al. (2016). The FAIR guiding principles for scientific data management and stewardship. *Sci. Data* 3, 1–9. doi: 10.1038/sdata.2016.18

Willenbring, J. M., Shende, S. S., and Gamblin, T. (2024). Providing a flexible and comprehensive software stack via spack, an extreme-scale scientific software stack, and software development kits. *Comput. Sci. Eng.* 26, 20–30. doi: 10.1109/MCSE.2024.3395016

Wrinn, M., and Asbury, R. (2004). "MPI tuning with Intel®trace analyzer and Intel®trace collector," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)* (Los Alamitos, CA: IEEE Computer Society), 4.

Wylie, B. (2018). *Nekbone MPI+OMP on JUQUEEN Performance Audit (POP_AR_112)*. doi: 10.5281/zenodo.17142583

Wylie, B. J. N., Feld, C., Geimer, M., Llort, G., Mendez, S., Mercadal, E., et al. (2025). 15+ years of joint parallel application performance analysis/tools training with Scalasca/Score-P and Paraver/Extrae toolsets. *Future Gener. Comput. Syst*. 162:107472. doi: 10.1016/j.future.2024.07.050

Zarubin, M., and Wesarg, B. (2024). "Reasoning the runtime overhead of profiling tools," in *Tools for High Performance Computing 2023 - Proceedings of the 14th International Workshop on Parallel Tools for High Performance Computing*, eds. C. Niethammer, W. E. Nagel, and M. M. Resch (Cham: Springer International Publishing).

Zhukov, I., Feld, C., Geimer, M., Knobloch, M., Mohr, B., Saviankou, P., et al. (2015). "Scalasca v2: back to the future," in *Proceedings of Tools for High Performance Computing 2014* (Cham: Springer), 1–24. doi: 10.1007/978-3-319-16012-2_1

Ziegenbalg, J., and Wesarg, B. (2012). *The Benefit of GCC's Open Structure on Instrumentation in the HPC Area*. Talk at GNU Tools Cauldron, July 2012. Available online at: https://gcc.gnu.org/wiki/cauldron2012#The_Benefit_of_GCC.27s_open_structure_on_instrumentation_in_the_HPC_area (Accessed June 06, 2025).