2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC) | 979-8-3315-0909-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/HIPC62374.2024.00013

Retrospection on the Performance Analysis Tools for Large-Scale HPC Programs

Zhibo Xuan[‡*], Xin You[‡*], Hailong Yang[‡], Mingzhen Li[†], Zhongzhi Luan[‡], Yi Liu[‡], and Depei Qian[‡]
School of Computer Science and Engineering, Beihang University, China[‡],
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy, China[†]
{xzb,youxin2015,hailong.yang,07680,yi.liu,depeiq}@buaa.edu.cn[‡], limingzhen@ict.ac.cn[†]

***Abstract*—As the performance gap between hardware and software widens, performance analysis tools are essential for understanding the behavior of large-scale High-Performance Computing (HPC) programs. These tools provide insights into the performance bottlenecks and help in optimizing the performance of the programs. In this paper, we present a comprehensive study of performance analysis tools for large-scale HPC systems including both sampling-based and instrumentation-based tools that are commonly adopted in the HPC community. We investigate the abundance and overheads of data collection as well as the analysis capabilities of HPCToolkit, TAU, and Scalasca with representative programs at scale. Our study shows that different performance analysis tools have distinct strengths and weaknesses, and the choice of a performance analysis tool depends on the specific requirements of the user. We also discuss the challenges and future directions in the field of performance analysis tools for large-scale HPC systems.**

I. Introduction

Performance is critical for both scientific and industrial applications on various domains, including molecular dynamics [1], computational fluid dynamics [2], climate modeling [3], and large language models [4]. However, due to the end of Moore's law, the performance improvements of general-purpose processer are bogged down, which achieves only 3% per year for single core performance [5]. Software is becoming increasingly difficult to achieve notable performance improvements through hardware updates. To make matters worse, recent years of Golden Bell Prize winners have shown that the cutting-edge performance of application performance only achieves 1.78% of peak performance of Fugaku Supercomputer [6]. The performance gap between the peak performance of the hardware and the actual performance of the software is widening.

As large-scale high-performance programs are becoming increasingly complex, it is unrealistic to manually analyze and understand the performance of target programs for large-scale high-performance computing (HPC) systems. To better understand the gap between the attainable software and hardware performance, various performance tools are proposed for performance analysis, including HPCToolkit [7], TAU [8], and Scalasca [9]. These tools can identify the performance hotspots of target programs and help in optimizing the program performance by providing insights of various common performance issues, including poor scalability and performance variance. However, performance analysis tools have different features and capabilities, and the choice of a performance analysis tool depends on the specific analysis requirements of the user.

Different performance analysis tools have different data collection methodologies as well as analysis capabilities according to their attention to the functional requirements. Specifically, one common concern of the performance analysis tools is the overhead of data collection. The data collection overhead can significantly affect the performance of the target program, and thus it is essential to minimize it. Another concern is the accuracy and abundance of data collection. The performance analysis tools should collect accurate and abundant performance data to provide insights into the performance bottlenecks of the target program. The performance analysis tools should also provide the ability to analyze the collected performance data to identify the performance bottlenecks with intuitive presentations for developers to guide further program optimizations.

For data collection methods, the performance analysis tools can roughly be divided into two categories: sampling-based and instrumentation-based tools. Sampling-based tools, such as HPCToolkit [7], collect performance data by sampling the target program at regular time intervals. These tools are generally lightweight with acceptable overheads, but they may miss some performance data (e.g., function parameters). Instrumentation-based tools, such as TAU [8] and Scalasca [9], collect performance data by instrumenting the target program with probes. These tools often exhibit higher overhead than sample-based tools when collected function calls are triggered at significantly high frequency, but they can collect more detailed performance data with notable accuracy.

For analyzing the performance of large-scale HPC programs, developers are commonly concerned about the bottlenecks of the target program, which are often represented as hotspot functions (i.e., the most time-consuming or resource-consuming code regions), scalability issues (i.e., performance loss compared to the ideal linear speedup), and performance variance (i.e., the significant performance difference between different runs). The performance analysis tools should provide insights into these common performance issues to help developers optimize the performance of the target program. However, the performance analysis tools have different strengths

* Both authors contributed equally to this paper.

2640-0316/24/$31.00 ©2024 IEEE
DOI 10.1109/HiPC62374.2024.00013

and weaknesses in analyzing these performance issues. The choice of a performance analysis tool depends on the specific requirements of the user, including the type of application, the target platform, and the analysis criteria. However, there is no empirical study that evaluates the commonly adopted performance analysis tools on large-scale HPC systems to provide guidance on choices of performance analysis tools as well as discuss common shortbacks of existing tools on large-scale HPC systems for future direction.

In this paper, we present a comprehensive study of performance analysis tools for large-scale HPC systems. We identify the key features of performance analysis tools and then evaluate the tools based on these features. We provide a detailed comparison of the performance analysis tools regarding their features and capabilities. We also discuss the challenges and future directions in the field of performance analysis tools for large-scale HPC systems. Specifically, this paper makes the following contributions:

- We present a comprehensive study of performance analysis tools for large-scale HPC systems, including data collection and analysis capabilities in common concerns.
- We identify the strengths and pitfalls of the existing performance analysis tools from the key feature aspects, including data collection, trace analysis, hotspot analysis, scalability, and performance variance.
- We provide several future directions of performance analysis tools at scale according to the above comparison of the representative tools.

The rest of this paper is organized as follows. Section II provides the background of performance analysis tools. Section III presents the comparison methodology used in this study. Section IV evaluates the performance analysis tools based on the methodology and discusses the comparison results. Section V concludes the paper and discusses future directions.

II. Background

In this section, we briefly summarize the key concepts and terminologies related to performance analysis tools for large-scale HPC systems. For analyzing the performance of large-scale HPC programs, developers first need to collect the performance data from the specific execution of the target program and then analyze the collected performance data to identify the performance bottlenecks for further optimization. The following subsections provide the background of performance data collection and large-scale performance analysis tools in detail.

A. Performance Data Collection

For effective performance analysis, several performance data need to be collected from the target program, including CPU performance counters, elapsed time, MPI communication, function events, and function parameters. The performance data can be collected by two methods: sampling-based and instrumentation-based.

Sampling-based performance data collection - Generally, the sampling-based performance data collection method is based on the idea of sampling the target program at regular time intervals, where the sample distribution can provide a statistically accurate profile of the target program execution. The sampling rate is often represented as the number of samples per second (e.g., 1000Hz indicates 1000 samples per second). For each sample, tools can obtain the current state of the sampled program execution, including its call stack, timestamp, and the value of CPU performance counters. However, the sampling-based data collection approaches are non-invasive, and detailed value-related data (e.g., function parameters) is hard to collect. This leads to the incapability of providing detailed performance analysis for some specific performance issues, including the MPI communication patterns [10], late sender/receiver [9], [11], and fine-grained analysis of performance variance [12], [13]. Some hardware platforms provide hardware precise event-based sampling (e.g., PEBS in Intel X86) that can sample the program states, including register files and control states, with low overheads at high accuracy [14], which can support some detailed value-related data collection and fine-grained performance analysis [15]–[17]. However, the hardware-based sampling methods are often limited by the hardware platforms and may not be available on all platforms.

Instrumentation-based performance data collection - Generally, instrumentation-based performance data collection methods are based on the idea of instrumenting the target program with probes to collect the performance data at the function entry and exit points, MPI communication points, and other specific code regions. Specifically, the probes can be inserted into the target program with library interception, binary instrumentation, compiler instrumentation, and source-code instrumentation. Library interception can intercept the library functions (e.g., PMPI profiling interface in MPI standard) by pre-loading specific library wrappers via LD_PRELOAD [7], [8], [13], [18]. Binary instrumentation can insert the probes into the binary code with static or dynamic binary rewriting [19]–[22]. Compiler instrumentation can insert the probes into the target program by integrated instrumentation pass at the compile time [8], [9], [23], while source-code instrumentation requires program developers to insert the corresponding probe API calls into the interested code regions [24], [25]. Regardless of the specific instrumentation approaches, the instrumentation-based performance data collection methods can provide a wide range of performance data, including the function parameters, accurate function event traces, and even fine-grained instruction or operand values. However, the instrumentation-based performance data collection methods often exhibit higher overhead than sample-based tools when inserted probes are triggered at significantly high frequency.

For performance analysis of large-scale HPC programs, performance tools often need to balance the trade-off between the abundance and overhead of data collection to provide effective performance analysis [7]–[9]. However, collecting the entire trace including all computation and communication events with detailed performance data is often infeasible due to the high overheads (details in Section IV-A). To mitigate

this issue, large-scale performance analysis tools either collect performance data via sampling [7], [26] or selectively obtain the most significant traces (e.g., MPI communication traces) via instrumentation [8], [9] for further performance analysis.

B. Goals of Large-Scale Performance Analysis

For performance analysis of large-scale HPC programs, the common concern of the program developers is how to better understand the performance bottlenecks of the target program and optimize the performance of the program through intuitive guidance from the performance analysis tools. Although there are varieties of performance issues that can be analyzed, in this paper, we focus on the most common performance issues at a large scale, including the hotspots, poor scalability, and performance variance.

Trace - The trace analysis is the common ability of performance analysis tools to provide detailed performance data of the target program execution, including the function events, function parameters, and MPI communication patterns. Several performance issues, such as late sender and receiver [11], require MPI communication traces for analysis. The trace analysis is often presented in a timeline view to help developers understand the performance bottlenecks of the target program execution. For more accurate trace analysis, the existing performance analysis tools already adopt timeline alignments to mitigate the potential time skewness of the collected traces [27], [28].

Hotspot - The hotspot analysis is a common performance analysis technique that identifies the functions that consume significant execution time or resources. The hotspot functions are often the performance bottlenecks of the target program, and optimizing the hotspot functions can significantly improve the performance of the program. For presenting the hotspots of the target program execution, some commonly adopted performance analysis tools, such as HPCToolkit [7], can provide top-down (i.e., tree view with the top of the call stack as root), bottom-up (i.e., tree view with the bottom of the call stack as root), and flat (i.e., functions) view of the hotspots with the detailed performance data, including elapsed times and collected metrics from hardware performance counters.

Scalability - The scalability loss is another common performance analysis goal to diagnose the performance bottlenecks of the target program when numbers of processes increase. The scalability analysis helps developers understand the performance loss of the target program compared to the ideal linear speedup. Almost all commonly adopted performance analysis tools, such as HPCToolkit [7], TAU [8], and Scalasca [9], can provide the scalability analysis of the target program with different numbers of processes. Some advanced performance analysis techniques [26], [29] can further locate the root cause of specific scalability issues for better optimization guidance. The scalability analysis can help developers identify the performance bottlenecks of the target program and optimize the performance of the program for large-scale HPC systems.

Performance Variance - Performance variance indicates the significant performance difference between different runs of the target program. The performance variance can be caused by various factors, including system noise, hardware failure, and the misconfigured runtime environment [30]. The performance variance analysis can help the developers figure out the sources of the specific performance variance and provide rich information to avoid or alleviate such unexpected fail-slows at scale via hardware re-configuration or software optimization [12], [13].

TABLE I: The hardware and software specifications.

Processor	Intel Golden 6240@2.60GHz
Nodes	32
Cores	36
Memory	384 GB
Network	100 Gbps
Storage	160Gbps
Software	OpenMPI 4.0.7, gcc 9.4.0 -O3
HPCToolkit	Version 2022.10.01
TAU	Version 2.32
Scalasca	Version 2.6

III. Methodology

Although there are widely adopted performance tools in large-scale HPC systems, there is no existing work that provides a comprehensive study of the existing performance analysis tools for performance analysis of large-scale HPC programs according to our knowledge. In this section, we describe our testbed of the performance analysis tools and the corresponding comparable metrics on the common concerns for performance analysis.

A. Experimental Setup

We evaluate a homebuilt cluster with hardware and software configuration as shown in Table I. Specifically, the cluster consists of 32 computing nodes. Each node is equipped with one 36-core Intel Golden 6240 processor running at 2.60 GHz frequency. There is 384 GB of memory for each node. All nodes are connected with a 100 Gbps network. The storage has over 160 Gbps I/O bandwidth. All programs are compiled with GCC 9.4.0 with `-O3` compiler optimizations. We use OpenMPI 4.0.7 for MPI communication.

For a comprehensive comparison of the existing large-scale performance tools, we evaluate the following performance tools that are well-known for providing rich performance guidance for large-scale HPC programs:

- **HPCToolkit [7]** is a sampling-based performance analysis tool that provides insights into the performance bottlenecks of the target programs. HPCToolkit can collect traces and generate profiles from the collected data. In our evaluation, we leverage the default sampling rate at 300Hz per event type for HPCToolkit.
- **TAU [8]** is an instrumentation-based performance analysis tool that provides the ability to collect wide ranges of performance data. TAU requires re-compilation with its own compilation toolchains (e.g., tau_cc) to obtain the detailed function traces. For trading off the overhead and abundance of data collection, TAU provides both profile

run (denoted as `TAU-P`) that do not collect detailed function traces and trace run (denoted as `TAU-T`) that collect detailed function traces. In our evaluation, we enable the auto event throttling with default settings ($numcalls > 100,000$ && $usecs/call < 10$) for the `TAU-P` run and only collects MPI functions for `TAU-T`.

- **Scalasca [9]** is an instrumentation-based performance analysis tool that targets scalable performance tracing and analysis. Scalasca requires re-compilation with instrumentation compiler toolchains (e.g., scorep [31]) to obtain the detailed function traces. Scalasca requires one profile run (denoted as `Scalasca-P`) before collecting the full trace (denoted as `Scalasca-T`) for the target programs to obtain reasonable configurations as well as function filters for lower overhead. In our evaluation, we collect without any filters for `Scalasca-P` and tracing with the provided configuration as well as filters that only collect MPI communication traces.

The aforementioned performance tools are representative of state-of-the-art performance analysis tools and are widely adopted in the HPC community. We evaluate these tools based on the following criteria: abundance and overhead of data collection, trace analysis, hotspots analysis, scalability, and performance variance. Note that although primitive diagnosing of the above performance issues does not require full event traces, MPI communication traces are still required for several advanced root cause analysis of specific performance issues in state-of-the-art research [13]. We evaluate these tools with the NAS Parallel Benchmarks (NPB) [32] and the real-world application LULESH [33].

Specifically, we use class B input for NPB-16 processors, class D for NPB-1,024 processors, and *-s 40 -i 400* for LULESH in our evaluation according to the evaluation scale. For HPCToolkit, we use *-t -e REALTIME -e PAPI_TOT_INS* to enable the trace function and performance metric collection function. For TAU, we enable the auto throttling function, callpath, and communication matrix collection. For Scalasca, to generate the filter file and trace configuration (e.g., max buffer size), we first use *scalasca -analyze* and *scalasca -examine* to profile the target benchmarks and applications. We use the same input datasets for all the performance analysis tools to ensure a fair comparison.

B. Compariable Criteria

The performance analysis tools can be evaluated with two aspects: data collection and analysis capabilities. We evaluate the performance analysis tools based on the following criteria:

1) Data Collection: For both primitive and advanced performance analysis, developers first need to collect enough performance data from the target program execution. In this paper, we call the ability to collect different types of performance data as *abundance* of data collection. The abundance of data collection is essential for supporting various useful and important performance analysis tasks. For large-scale homogeneous clusters, we investigate whether the performance analysis tools can collect the following performance data:

TABLE II: The collected performance data types of the evaluated performance analysis tools.

Data Type/Metrics	Sampling-based	Instrumentation-based	
	HPCToolkit	TAU	Scalasca
CPU Performance Counter	✓	✓*	✓*
Function Elapsed Time	Statistics	Accurate*	Accurate*
Function Parameters		✓*	✓*
MPI Communication	✓	✓	✓
Function Trace	✓	✓*	✓*

*** TAU and Scalasca requires re-compilation with its own compilation toolchains (e.g., tau_cc, scorep) to obtain the detailed function traces.**

- **CPU Performance Counter:** The ability to collect CPU performance counter data, often represented as PAPI [34] or Linux perf [35] events. It can provide deep insights into the hardware performance bottlenecks of the target program (e.g., top-down microarchitecture analysis [36]).
- **Function Elapsed Time:** The ability to collect the elapsed time of the target program execution. This is the most basic performance data that can provide insights into the overall performance of the target program.
- **Function Parameters:** The ability to collect runtime values of function parameters, including computational and communication (e.g., MPI) function parameters. Some advanced performance analysis tasks, such as communication patterns and performance variance analysis, require the ability to collect function parameters.
- **MPI Communication:** The ability to collect MPI communication events. As MPI is the de facto standard for parallel programming in HPC, the ability to collect MPI communication events is essential for understanding the communication bottlenecks of the target program.
- **Function Trace:** The ability to collect function event trace with corresponding start and end timestamps. This provides a detailed execution view of the target program.

Besides, the time and storage overhead of data collection is another important aspect for evaluating the performance analysis tools. Specifically, time overhead is measured as the execution time of the target program with and without the performance analysis tools. Storage overhead is measured as the storage space required to store the collected performance data. The overhead of data collection is essential for minimizing the impact of the performance analysis tools on the target program execution. The higher time overhead leads to significant time and commercial costs for performance analysis and limits the applicability of the performance analysis tools at a large scale. The higher storage overhead leads to the difficulty of storing and analyzing the collected performance data, which may further result in unexpected fails due to exceeding the storage capacity (e.g., maximum 1 TB storage budgets adopted in our evaluated homebuilt HPC cluster).

2) Analysis Capabilities: For performance analysis capabilities of the evaluated performance analysis tools, it is difficult to provide a quantitative metric for comparison. Instead, we provide the pros and cons of the evaluated performance analysis tools based on the following common performance analysis tasks for large-scale HPC programs:

TABLE III: The overhead of the evaluated state-of-the-art tools.

Program	Scale	Time (s)						Storage (KB)				
		native	HPCToolkit	TAU-P	TAU-T	Scalasca-P	Scalasca-T	HPCToolkit	TAU-P	TAU-T	Scalasca-P	Scalasca-T
BT	16	24.99	61.55	389.75	17.70	183.27	27.92	74,752	8,192	73,728	512	26,112
	1024	39.28	131.86	531.00	56.93	291.46	56.97	4,823,450	508,416	33,554,944	59,392	13,631,488
CG	16	6.01	8.80	28.93	17.76	13.80	9.00	74,752	8,192	155,648	1,024	51,200
	1024	32.31	75.99	454.59	36.82	249.41	45.88	4,823,450	520,704	27,787,776	20,480	8,493,466
EP	16	3.95	7.49	6.29	4.32	5.19	6.89	74,752	8,192	24,576	512	1,536
	1024	5.73	34.66	6.51	9.50	8.08	13.07	4,718,592	508,416	1,573,376	12,288	20,480
FT	16	6.59	11.31	8.06	8.46	7.57	8.42	74,752	8,192	24,576	512	1,536
	1024	19.96	62.81	20.43	19.10	18.28	26.25	4,823,450	508,416	1,573,376	29,184	547,840
IS	16	2.65	3.80	25.76	2.36	13.25	3.57	50,688	8,192	24,576	512	1,536
	1024	7.43	37.20	31.87	7.40	17.09	13.07	4,823,450	508,416	1,573,376	17,920	31,744
LU	16	16.47	42.10	28.77	20.70	23.58	22.97	74,752	8,192	532,480	512	159,232
	1024	35.21	104.46	62.32	135.42	43.46	123.80	4,823,450	508,416	206,766,592	39,936	20,971,520
MG	16	2.85	4.62	3.53	4.16	2.82	3.66	71,168	8,192	49,152	512	12,288
	1024	6.53	41.85	7.49	8.87	6.82	16.84	4,823,450	508,416	5,870,080	45,568	62,914,560
SP	16	27.08	47.58	31.92	30.78	29.94	30.01	74,752	8,192	114,688	512	34,300
	1024	45.26	111.19	44.62	58.59	41.14	63.58	4,823,450	508,416	61,342,208	37,376	1,677,722
LULESH	27	56.64	105.61	142.79	57.51	102.38	>1h	443,392	13,824	186,368	3,584	117,600
	1000	61.84	159.92	140.15	60.80	98.35	>1h	4,610,560	496,128	10,738,176	86,016	2,137,088

- **Trace Analysis -** Trace analysis provides the detailed execution view of the target program with trace visualization or profiles generated from the collected function traces. We evaluate the performance analysis tools with the trace analysis capabilities with their built-in visualization GUI interface for intuitive comparison.
- **Hotspot Analysis -** Hotspots indicate functions or code regions that consume the most significant time or resources. For a fair comparison, we run the default hotspot analysis within each evaluated performance tool and provide the top few functions reported by these tools. Apparently, for the same program execution with the same input at the same scale, the analysis results of the top 10 hotspots should be similar, which gains the most attention for developers to further investigate the performance optimization opportunities.
- **Scalability Analysis -** Scalability analysis aims to identify the causes of poor performance scalability of target program execution at different scales. Poor scalability can lead to low utilization of large-scale computation resources and even the inability to obtain higher performance even running with more nodes. We evaluate the performance analysis tools with 16 and 1024 processes to evaluate the tool's ability to analyze the scalability issues.
- **Performance Variance Analysis -** Performance variance indicates the significant performance slowdown of the different execution instances of the same program. For a fair comparison, we run the program with and without injected disturbances to evaluate their ability to identify the performance variance.

For each evaluated analysis capability, we qualitatively investigate the intuitiveness, accuracy, and completeness of the analysis results provided by the performance analysis tools. Specifically, the analysis results should be intuitive for developers to understand the performance issues of the target program. The analysis results should be accurate to provide reliable guidance for optimizing the performance of the target program. Besides, the analysis results should also be actionable to provide optimization guidance for the target program, such as providing problematic code locations, calling contexts, and comprehensive diagnosis of root causes.

IV. Results

In this section, we present the results of each evaluated performance analysis tool based on the methodology presented in Section III. We evaluate the performance analysis tools based on the following key features: data collection, hotspot analysis, scalability, and performance variance.

A. Abundance and Overhead of Data Collection

For the data collection, we evaluate the abundance and overhead of the performance data collected by the evaluated performance analysis tools. The abundance of performance data indicates the types of performance data collected by the tools, including CPU performance counters, elapsed time, function parameters, MPI communication, and function traces, as mentioned in Section III-B. The results are shown in Table II. Generally, the abundance of the data collection is tightly correlated to the basic data collection methods, regardless of the specific implementation details of each tool (e.g., both TAU and Scalasca are instrumentation-based tools). For sampling-based tools, the collected performance data is limited to CPU performance counters, elapsed time, and MPI communication, while instrumentation-based tools can collect more detailed performance data, including function parameters and function traces. Note that the sampling-based tools obtain function elapsed time via the distribution of the collected samples, which is statistically accurate based on the assumption that the samples are uniformly distributed along time and time-consuming functions will result in more samples. In contrast, by instrumenting the target program with probes, the instrumentation-based tools can collect the accurate function elapsed time and function parameters with notable accuracy.

The overhead of data collection indicates the time and storage overheads of the performance data collection. The time overhead is the additional time consumed by the performance analysis tools for data collection, and the storage overhead is the additional storage space consumed by the performance

data collected by the tools. The time and storage overheads are measured in seconds and kilobytes (KB), respectively. The results are shown in Table III. Specifically, for time overheads of all evaluated tools, the sampling-based tools (i.e., HPCToolkit) generally have lower overheads than the instrumentation-based tools (i.e., TAU-P and Scalasca-P) when collecting all kinds of function events without considering function parameters. For TAU-T and Scalasca-T, which are configured to collect only MPI communication events, the time overheads are significantly lower and even negotiable at a small scale. For TAU-P and Scalasca-P, which are configured to collect all kinds of function events, they both incur significant time overheads at both scales and are even not able to finish the data collection for real-world applications. Especially for Scalasca, LULESH at both 27 and 1000 ranks fails to collect for tracing due to job timeout, as shown in Table III. According to the slurm output files, we figure out that the Scalasca is blocked by the tightly coupled implementation of postmortem analysis of tracing data, which can not be disabled or executed separately. Therefore, for collecting computational events for large-scale HPC programs, sampling-based approaches are more suitable than instrumentation-based approaches, while communication events are more suitable for instrumentation-based approaches due to the rich parameter information collected for further analysis.

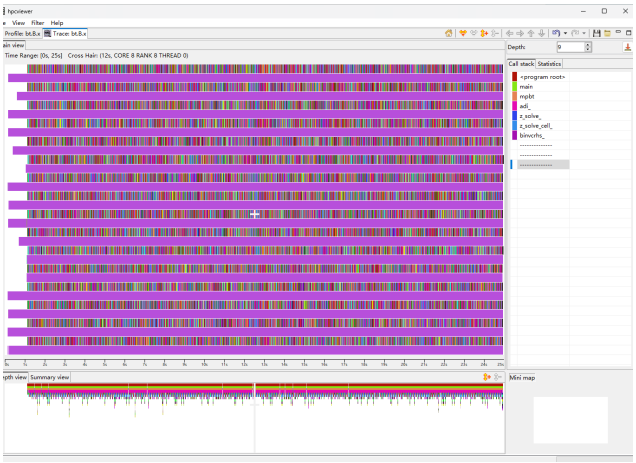
For storage overheads, the instrumentation-based tools have higher overheads than the sampling-based tools due to the detailed performance data collected at higher event frequency. The storage overheads of the instrumentation-based tools are often proportional to the number of function calls, while the storage overheads of the sampling-based tools are often proportional to the number of samples collected by the tools.

***Pitfall 1**: Leveraging only one type of data collection method may not be sufficient for comprehensive performance analysis with acceptable overheads.*

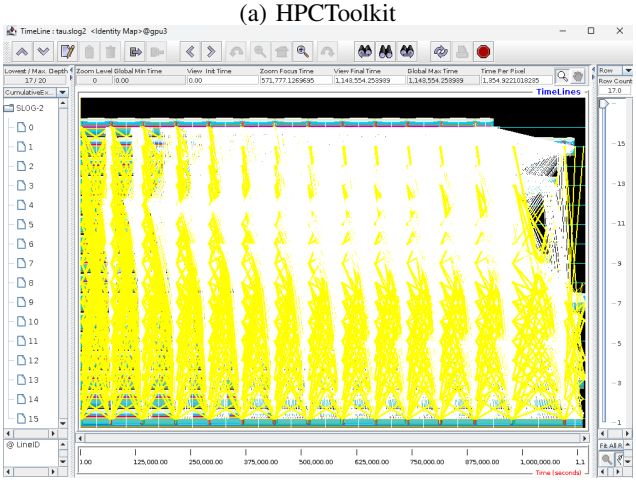
As demonstrated in the above results, the sampling-based tools generally have lower overheads than the instrumentation-based tools when collecting all kinds of function events without considering function parameters, while instrumentation-based tools can obtain runtime values of function parameters. For performance analysis of large-scale HPC programs, the parameters of MPI functions often play an important role in the detailed diagnosis of inefficient communications and scalability issues [11]–[13], while function parameters are rarely exploited by the existing performance analysis techniques. Therefore, for future development of cutting-edge performance analysis tools, it is essential to combine the advantages of both sampling-based and instrumentation-based tools to provide comprehensive performance data collection with abundant data types at low overheads.

B. Trace Analysis

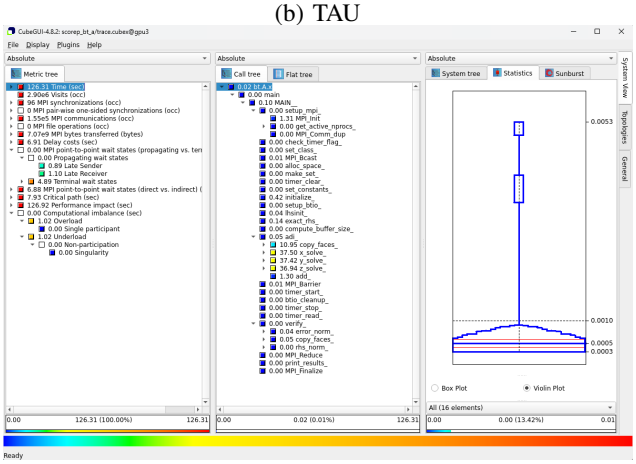
The visualization of the collected performance traces of BT with 16 MPI ranks by HPCToolkit, TAU, and Scalasca



(a) HPCToolkit



(b) TAU



(c) Scalasca

Fig. 1: The visualization of trace analysis results of the evaluated performance analysis tools.

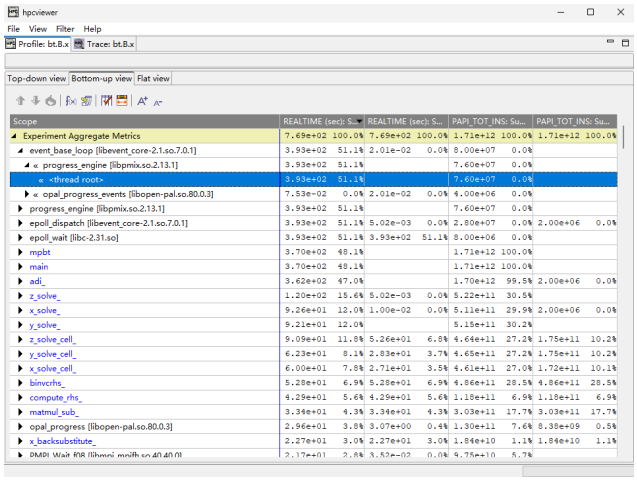
are demonstrated in Figure 1. Specifically, for HPCToolkit as shown in Figure 1(a), its sampling-based trace can represent the distribution of the samples by leaf functions and corresponding calling contexts. For TAU as shown in Figure 1(b), its visualized traces additionally provide message-passing linkage between communication events that present the triangular communication patterns of BT to some extent. However, it is still hard to figure out the performance issues buried beneath the visualized traces. Generally, for both HPCToolkit and TAU, even though the scale of the collected traces is limited (only 16 processes), the trace visualization is already complex, and hard to interpret any actionable performance guidance from the messy visualized traces as shown in Figure 1(a) and (b). For Scalasca as shown in Figure 1(c), it does not provide such timeline visualization of collected traces. Instead, it extracts several useful metrics that can be representative of some kinds of inefficient communications, such as the late sender and receiver issues [9], [11]. Such extracted profiles from the massive amounts of tracing data are more intuitive and easier for developers to understand the potential performance issues of the target program execution.

***Pitfall 2**: Direct trace visualization is too messy and meaningless for actionable performance guidance at scale.*

In sum, for trace analysis of large-scale HPC programs, the existing performance analysis tools still lack effective visualization techniques to present the collected traces in an intuitive way. The visualized traces are often complex and hard to interpret, which requires developers to have a deep understanding of the target program execution to figure out the potential performance bottlenecks. Effective performance analysis tools can provide useful metrics extracted from the collected traces for better optimization guidance. For future development directions, trace analysis tools can also provide intuitive highlights of the abnormal performance patterns in the visualized traces to help developers quickly locate the potential performance bottlenecks.

C. Hotspot Analysis

The top 10 hotspots of the evaluated HPC programs with HPCToolkit, TAU, and Scalasca are demonstrated in Figure 2. Specifically, for HPCToolkit as shown in Figure 2(a), its bottom-up view of the hotspots can provide the tree view of the call stack with the leaf function call as root. For TAU as shown in Figure 2(b), its top 10 hotspots are presented in a flat view with detailed performance statistics, including elapsed times and accumulated number of function calls. For Scalasca as shown in Figure 2(c), it provides function elapsed times in the memory buffer size (a.k.a., relative to the number of function calls) order, which is not intuitive enough for identifying the hotspots. As demonstrated in Figure 2(c), such poor presentation can lead to difficulty in identifying the most significant code regions for further performance investigation as the function on the top apparently is not the most time-consuming function.



(a) HPCToolkit

```
FUNCTION SUMMARY (total):
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs Count/Call Name
              counts total counts
---------------------------------------------------------------------------------------
100.0   6.423E+06   1.693E+13           16          16 1058099810264 .TAU application
100.0     1.3E+07   1.693E+13           16        4099 1058099408816 MPBT
 99.4   4.848E+07   1.682E+13         3216       16080 5231394157 ADI
 34.1   3.304E+08   5.777E+12         3216      122208 1796215536 Z_SOLVE
 33.9   5.747E+12   5.747E+12       115776           0   49641149 MPI_Wait()
 32.2    3.32E+08   5.448E+12         3216      122208 1693901300 Y_SOLVE
 28.3    3.22E+08   4.786E+12         3216      122208 1488183453 X_SOLVE
 20.1   1.718E+12     3.4E+12        12864 6.11053E+08  264287217 Y_SOLVE_CELL
 20.1   1.718E+12     3.4E+12        12864 6.11053E+08  264286961 Z_SOLVE_CELL
 20.1   1.716E+12   3.399E+12        12864 6.11053E+08  264206761 X_SOLVE_CELL
 11.2    1.89E+12    1.89E+12 6.0903E+08           0       3102 BINVCRHS
 10.0    1.69E+12    1.69E+12 6.0903E+08           0       2774 MATMUL_SUB
  8.6   1.452E+12   1.452E+12 6.0903E+08           0       2383 MATVEC_SUB
  4.8   1.764E+09   8.104E+11         3232       45248  250746637 COPY_FACES
  4.1   6.907E+11   6.907E+11         3232           0  213707035 MPI_Waitall()
  0.7   1.177E+11   1.177E+11         3232           0   36423207 COMPUTE_RHS
  0.4   3.281E+10    6.39E+10           32 1.28593E+07 1996894524 INITIALIZE
  0.2   4.192E+10   4.192E+10 1.73406E+07           0       2418 EXACT_SOLUTION
  0.1   1.992E+10   1.992E+10        12864           0    1548694 Z_BACKSUBSTITUTE
  0.1   1.991E+10   1.991E+10        12864           0    1547594 Y_BACKSUBSTITUTE
  0.1   1.932E+10   1.932E+10        12864           0    1501783 X_BACKSUBSTITUTE
  0.1   9.561E+09   1.783E+10           16   3.42E+06 1114346652 EXACT_RHS
  0.1    1.56E+10    1.56E+10 6.03E+06           0        2586 BINVRHS
  0.1   1.084E+10   1.084E+10           32           0  338751537 MPI_Barrier()
  0.1    6.56E+05   1.079E+10           16          48  674336827 VERIFY
  0.1   2.739E+09   1.004E+10           16 1.06126E+06  627379370 ERROR_NORM
```

(b) TAU

```
Estimated aggregate size of event trace:                   45GB
Estimated requirements for largest trace buffer (max_buf): 2874MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       2876MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=2876MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type    max_buf[B]         visits time[s] time[%] time/visit[us]  region
         ALL 3,013,330,717 1,851,047,227 1380.16   100.0          0.75  ALL
         USR 3,012,062,170 1,850,641,387 1221.51    88.5          0.66  USR
         MPI     1,054,032       273,840  153.92    11.2        562.09  MPI
         COM       214,474       131,984    4.72     0.3         35.79  COM
     SCOREP            41            16    0.00     0.0        236.59  SCOREP

         USR   992,025,450   609,030,000  233.11    16.9          0.38  binvcrhs_
         USR   991,257,228   609,030,000  174.52    12.6          0.29  matvec_sub_
         USR   991,257,228   609,030,000  192.28    13.9          0.32  matmul_sub_
         USR    28,207,114    17,340,552    4.28     0.3          0.25  exact_solution_
         USR    10,060,050     6,030,000    1.80     0.1          0.30  binvrhs_
         MPI       429,870        77,280    0.31     0.0          4.05  MPI_Irecv
         MPI       429,870        77,280    0.66     0.0          8.51  MPI_Isend
         MPI       188,136       115,776  131.38     9.5       1134.80  MPI_Wait
         USR        62,712        38,592    0.08     0.0          2.01  lhsabinit_
         USR        20,904        12,864  171.48    12.4      13329.98  x_solve_cell_
         USR        20,904        12,864    9.48     0.7        736.66  x_backsubstitute_
         USR        20,904        12,864  180.62    13.1      14040.36  y_solve_cell_
         USR        20,904        12,864    9.90     0.7        769.24  y_backsubstitute_
         USR        20,904        12,864  204.27    14.8      15879.23  z_solve_cell_
         USR        20,904        12,864    8.18     0.6        636.18  z_backsubstitute_
```

(c) Scalasca

Fig. 2: The top 10 hotspots of the evaluated HPC programs with HPCToolkit, TAU, and Scalasca, respectively.

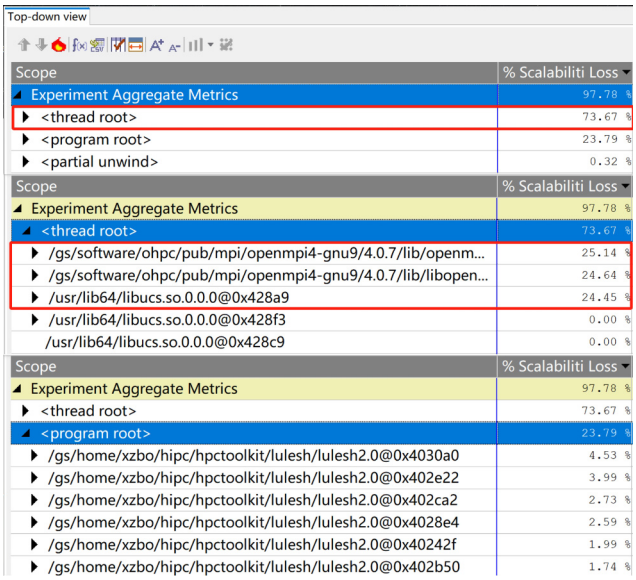
Generally, all evaluated performance analysis tools have the ability to identify the hotspots of the target program execution. However, the presentation of the hotspots is quite essential for intuitive and actionable optimization guidance. According to our opinion, the hotspot presentation of HPCToolkit (Figure 2(a)) is the most intuitive with rich optimization insights due to the tree view of the call stack. The flat view of the TAU is also acceptable for identifying the most time-consuming functions with detailed performance statistics. However, the presentation of the hotspots of Scalasca is not intuitive enough to identify the most significant code regions for further performance investigation. All evaluated performance analysis tools only provide the performance statistics of the hotspots, which requires further investigation to identify the performance opportunities. Although these tools provide basic hotspot analysis capabilities, they still reveal little actionable performance guidance for developers to optimize the target large-scale HPC program.

***Pitfall 3**: Presenting the performance statistics of hotspots is not sufficient for actionable optimization guidance.*

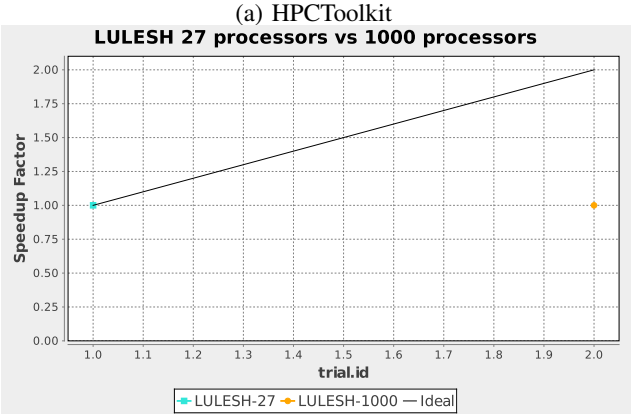
D. Scalability

The scalability analysis results of the LULESH with HPCToolkit and TAU are demonstrated in Figure 3. Note that Scalasca does not provide any description of the scalability analysis capabilities in their user manuals [9]. Specifically, for HPCToolkit as shown in Figure 3(a), its scalability analysis results are presented as the custom scalability loss metrics defined in its user manual [7] in the form of top-down tree views of the target program execution. Such scalability loss metrics with calling context attributions can provide intuitive insights into which functions are suffering from significant scalability loss. However, as it has merged with all processes and threads in the tree view, it is hard to figure out the root causes of the specific scalability issues of the target program execution and still requires non-trivial efforts to figure out actionable optimization insights from the given profiles. For TAU as shown in Figure 3(b), its weak scalability analysis results are presented in the form of the measured and ideal speedup of the target program execution with different numbers of processes. Compared to HPCToolkit, TAU can provide visualized plots for an intuitive understanding of the gap between measured and ideal scalability of the target HPC programs, but it cannot provide any actionable guidance on how to achieve the ideal speedups. Although there are research works that can be tailored to accurately localize the root causes of specific scalability issues [26], [29], the existing performance analysis tools are still limited to small program binaries to provide guidance with acceptable overheads.

***Pitfall 4**: Large-scale performance analysis tools are lack of actionable optimization guidance with accurate localization of scalability root causes within acceptable overheads.*



(a) HPCToolkit

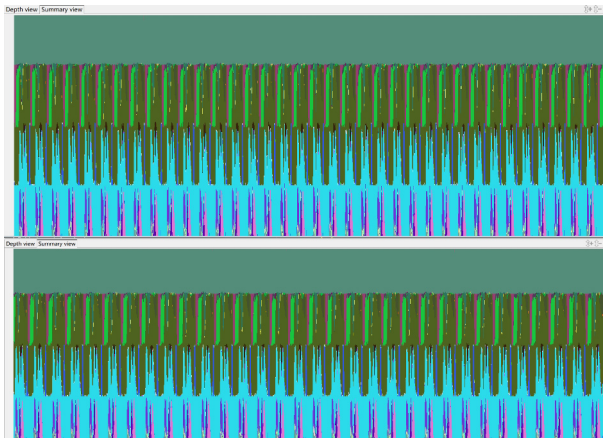


(b) TAU

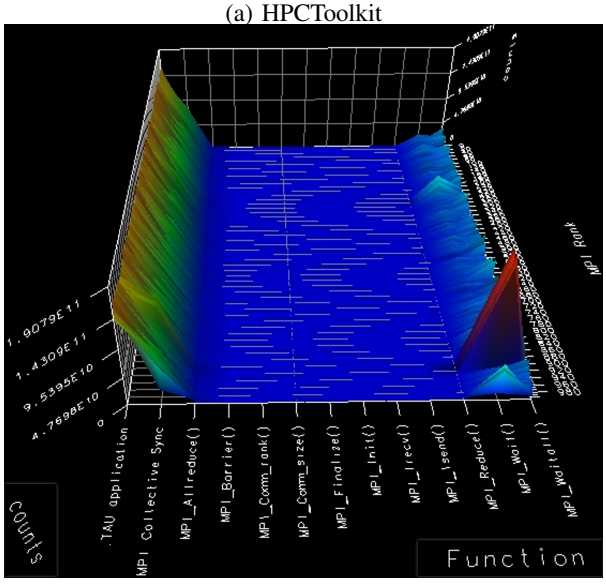
Fig. 3: The scalability analysis results of the evaluated HPC programs with HPCToolkit and TAU, respectively.

E. Performance Variance

The performance variance analysis results of the LULESH with HPCToolkit and TAU are demonstrated in Figure 4. Specifically, HPCToolkit does not actually provide specific analysis capabilities to identify the performance variance. As shown in Figure 4(a), when visualizing the function traces of the target program execution with and without injected disturbances by HPCToolkit, we cannot identify the injected performance variance from the visualized traces. For TAU, its performance variance analysis results are presented with ParaProf as the distribution of the elapsed times of the target program execution with different runs. As shown in Figure 4(b), we can intuitively identify the abnormal MPI_Wait performance metrics of several MPI ranks (i.e., red peaks in Figure 4), which are the exact nodes with memory noise injection. However, the significant performance anomaly is identified as MPI functions that are waiting for the asyn-



(a) HPCToolkit



(b) TAU

Fig. 4: The performance variance analysis results of the evaluated HPC programs with HPCToolkit and TAU, respectively.

chronous communications, the developers will be misled that such anomaly comes from the network systems, which leads to non-trivial debugging efforts in the wrong direction.

Pitfall 5*: The existing large-scale performance analysis tools still lack accurate localization of the root causes of the performance variance.*

F. Discussion: Future Directions

According to the aforementioned evaluation results of the commonly adopted performance analysis tools at scale, we can obtain the following insights for future development of performance analysis tools for large-scale HPC programs:

1) Combining sampling-based and instrumentation-based data collection approaches. According to *Pitfall 1*, computational events are more suitable for sampling-based approaches, while communication events are more suitable for instrumentation-based approaches. Therefore, for comprehensive performance analysis of large-scale HPC programs, it is essential to combine the advantages of both sampling-based and instrumentation-based tools to provide comprehensive performance data collection with abundant data types at low overheads.

2) Trace visualization of large-scale execution should highlight the problematic regions with human-friendly annotations. According to *Pitfall 2*, tools should provide a more intuitive focus on the abnormal or suspicious events or regions by highlighting or other human-friendly annotations of the massive amount of events and processes in the collected traces of large-scale HPC program execution. In addition, such a massive amount of tracing data brings a further exploration of combining machine-learning techniques to automatically identify the potential performance bottlenecks of the target program execution.

3) Multi-dimensional hotspot analysis with actionable optimization guidance. According to *Pitfall 3*, tools should provide multi-dimensional hotspot analysis with actionable optimization guidance for developers to quickly locate the potential performance bottlenecks of the target large-scale HPC program execution, including inefficient instruction or data structures, time or resource-consuming, and so on.

4) Accurate and fast root cause analysis of scalability loss. According to *Pitfall 4*, tools should provide accurate localization of scalability root causes with actionable optimization guidance for developers to achieve the ideal speedups of the target large-scale HPC program execution. We can combine several cutting-edge machine-learning techniques (e.g., graph neural networks [37]) to automatically identify the root causes of the scalability issues of the target program execution within acceptable overheads for large-scale HPC programs.

5) Leveraging rich attributions of function traces with deep-learning-based anomaly detection for performance variance analysis. According to *Pitfall 5*, tools should provide accurate localization of the root causes of the performance variance for developers to avoid or alleviate such unexpected fail-slows at scale via hardware re-configuration or software optimization. We can combine cutting-edge deep-learning-based anomaly detection techniques for time-series data with function event traces that are attributed to rich parameters and performance counter metrics to automatically identify the root causes of the performance variance of the target program execution within acceptable overheads for large-scale HPC programs.

V. Conclusion

In this paper, we presented a comprehensive study of performance analysis tools for large-scale HPC systems. We identified the key features of performance analysis tools and evaluated the performance analysis tools based on these features. We provided a detailed comparison of the performance analysis tools based on their features and capabilities. We found that the performance analysis tools have different features and capabilities, and they are suitable for different types

of applications. We also found that the performance analysis tools have different levels of support for large-scale HPC systems. We believe that our study will help researchers and practitioners to choose the right performance analysis tools for their applications.

ACKNOWLEDGEMENTS

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB3001801), National Natural Science Foundation of China (No. 62322201, 62072018, U23B2020 and U22A2028), and the Fundamental Research Funds for the Central Universities (YWF-23-L-1121, JKF-20240198 and JK2024-58). Hailong Yang is the corresponding author.

REFERENCES

[1] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.

[2] E. Merzari, S. Hamilton, T. Evans, M. Min, P. Fischer, S. Kerkemeier, J. Fang, P. Romano, Y.-H. Lan, M. Phillips, E. Biondo, K. Royston, T. Warburton, N. Chalmers, and T. Rathnayake, "Exascale multiphysics nuclear reactor simulations for advanced designs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3627038

[3] M. Taylor, P. M. Caldwell, L. Bertagna, C. Clevenger, A. Donahue, J. Foucar, O. Guba, B. Hillman, N. Keen, J. Krishna, M. Norman, S. Sreepathi, C. Terai, J. B. White, A. G. Salinger, R. B. McCoy, L.-y. R. Leung, D. C. Bader, and D. Wu, "The simple cloud-resolving e3sm atmosphere model running on the frontier exascale system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3627044

[4] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[5] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[6] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaïm, W. Zhang, J.-L. Vay, and H. Vincenti, "Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[8] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[9] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and computation: Practice and experience*, vol. 22, no. 6, pp. 702–719, 2010.

[10] D. G. Chester, S. A. Wright, and S. A. Jarvis, "Understanding communication patterns in hpcg," *Electronic Notes in Theoretical Computer Science*, vol. 340, pp. 55–65, 2018.

[11] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf, "Identifying the root causes of wait states in large-scale parallel applications," *ACM Trans. Parallel Comput.*, vol. 3, no. 2, jul 2016. [Online]. Available: https://doi.org/10.1145/2934661

[12] X. Tang, J. Zhai, X. Qian, B. He, W. Xue, and W. Chen, "vsensor: leveraging fixed-workload snippets of programs for performance variance detection," in *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, 2018, pp. 124–136.

[13] L. Zheng, J. Zhai, X. Tang, H. Wang, T. Yu, Y. Jin, S. L. Song, and W. Chen, "Vapro: Performance variance detection and diagnosis for production-run parallel applications," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 150–162.

[14] M. A. Sasongko, M. Chabbi, P. H. J. Kelly, and D. Unat, "Precise event sampling on amd versus intel: Quantitative and qualitative comparison," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 5, pp. 1594–1608, 2023.

[15] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 332–347.

[16] M. Chabbi, S. Wen, and X. Liu, "Featherlight on-the-fly false-sharing detection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 152–167.

[17] M. A. Sasongko, M. Chabbi, M. B. Marzijarani, and D. Unat, "Reusetracker: Fast yet accurate multicore reuse distance analyzer," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 1, pp. 1–25, 2021.

[18] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," 2005.

[19] "Dyinst, homepage: https://www.dyninst.org," 2022. [Online]. Available: https://www.dyninst.org

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[21] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[22] X. You, H. Yang, K. Lei, Z. Luan, and D. Qian, "Vclinic: A portable and efficient framework for fine-grained value profilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 892–904.

[23] X. You, H. Yang, Z. Xuan, Z. Luan, and D. Qian, "Powerspector: Towards energy efficiency with calling-context-aware profiling," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1272–1282.

[24] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: performance introspection for hpc software stacks," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 550–560.

[25] D. Yokelson, O. Lappi, S. Ramesh, M. S. Väisälä, K. Huck, T. Puro, B. Norris, M. Korpi-Lagg, K. Heljanko, and A. D. Malony, "Soma: Observability, monitoring, and in situ analytics for exascale applications," *Concurrency and Computation: Practice and Experience*, p. e8141, 2024.

[26] Y. Jin, H. Wang, R. Zhong, C. Zhang, and J. Zhai, "Perflow: A domain specific framework for automatic performance analysis of parallel applications," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, 2022. [Online]. Available: https://doi.org/10.1145/3503221.3508405

[27] D. Becker, R. Rabenseifner, F. Wolf, and J. C. Linford, "Scalable timestamp synchronization for event traces of message-passing applications," *Parallel Computing*, vol. 35, no. 12, pp. 595–607, 2009, selected papers from the 14th European PVM/MPI Users Group Meeting. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819109000155

[28] L. Wei and J. Mellor-Crummey, "Using sample-based time series data for automated diagnosis of scalability losses in parallel programs," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 144–159. [Online]. Available: https://doi.org/10.1145/3332466.3374538

[29] Y. Jin, H. Wang, T. Yu, X. Tang, T. Hoefler, X. Liu, and J. Zhai, "Scalana: Automating scaling loss detection with graph analysis," in *SC20: In-*

ternational Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2020, pp. 1–14.

[30] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey *et al.*, "Fail-slow at scale: Evidence of hardware performance faults in large production systems," *ACM Transactions on Storage (TOS)*, vol. 14, no. 3, pp. 1–26, 2018.

[31] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*. Springer, 2012, pp. 79–91.

[32] D. H. Bailey, "Nas parallel benchmarks," *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.

[33] I. Karlin, J. Keasler, and J. R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[34] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "Papi software-defined events for in-depth performance analysis," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1113–1127, 2019.

[35] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[36] Intel, "Top-down microarchitecture analysis method," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html

[37] P. Li, Y. Guo, Y. Luo, X. Wang, Z. Wang, and X. Liu, "Graph neural networks based memory inefficiency detection using selective sampling," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–14.