Performance Analysis Tools for MPI Applications and their Use in Programming Education

Anna-Lena Roth
anna-lena.roth@informatik.hs-fulda.de
Hochschule Fulda, University of Applied Sciences
Fulda, Germany

Tim Süß
tim.suess@informatik.hs-fulda.de
Hochschule Fulda, University of Applied Sciences
Fulda, Germany

ABSTRACT

Performance analysis tools are frequently used to support the development of parallel MPI applications. They facilitate the detection of errors, bottlenecks, or inefficiencies but differ substantially in their instrumentation, measurement, and type of feedback. Especially, tools that provide visual feedback are helpful for educational purposes. They provide a visual abstraction of program behavior, supporting learners to identify and understand performance issues and write more efficient code. However, existing professional tools for performance analysis are very complex, and their use in beginner courses can be very demanding. Foremost, their instrumentation and measurement require deep knowledge and take a long time. Immediate, as well as straightforward feedback, is essential to motivate learners. This paper provides an extensive overview of performance analysis tools for parallel MPI applications, which experienced developers broadly use today. It also gives an overview of existing educational tools for parallel programming with MPI and shows their shortcomings compared to professional tools. Using tools for performance analysis of MPI programs in educational scenarios can promote the understanding of program behavior in large HPC systems and support learning parallel programming. At the same time, the complexity of the programs and the lack of infrastructure in educational institutions are barriers. These aspects will be considered and discussed in detail.

CCS CONCEPTS

• **General and reference → Surveys and overviews**; **Performance**; • **Applied computing → Interactive learning environments**; • **Computing methodologies → Parallel algorithms**.

KEYWORDS

performance analysis; parallel programming; MPI; HPC; education

ACM Reference Format:
Anna-Lena Roth and Tim Süß. 2023. Performance Analysis Tools for MPI Applications and their Use in Programming Education. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578245.3584358

1 INTRODUCTION

Scientific calculations, simulations, graphics calculations, or artificial intelligence calculations often cannot be performed in an acceptable time on individual computers. The amount of data to be processed increases over time, and the computational models are becoming more and more accurate. Therefore, reasonable computing power is required in the form of High-Performance Computing (HPC) clusters. In addition, industry also increasingly requires computing resources through HPC clusters (e.g., in developing artificial intelligence for autonomous driving). That is why parallel programming is becoming more important in programming education. Over the years, it has been expected that every software developer has certain basic knowledge of distributed and parallel programming. The 2013 Computer Science Curricula reports that parallel and distributed programming should become a core part of the computer science curriculum and recommends that any courses be distributed throughout the curriculum [16]. The resulting guidelines were revised and amended in 2020 by Prasad et. al. [53]. Message Passing Interface (MPI) is a de facto standard for teaching parallel programming. Many beginners learn parallel programming with MPI, which is also widely used in industry and research. Developers are expected to write correct and efficient code that uses fast, problem-based algorithms that exploit a cluster's full computational power. However, codes that perform well on one system can cause problems in other environments. Particularly novice programmers, but also experienced developers, have difficulties here. Neither the code nor the compiler can tell in advance whether a program is efficient or whether there are various performance problems, program errors, or bottlenecks.
Performance analysis tools are often used to find and eliminate such performance issues. For example, tools based on the instrumentation framework Valgrind [2] and the GNU profiler gprof [1] are very well-known and widely used for profiling and debugging applications to find program issues and analyze program behavior. However, they are not specifically designed to analyze parallel MPI programs. A problem that can frequently occur in MPI applications is that a process has to wait too long for an event of another process (e.g., *LateSender* problem), and during this time, computing resources are wasted. For identifying such problems, special analysis tools are necessary, on which this paper focuses. General profiling, debugging, and analysis tools developed for serial applications or not for MPI applications are not considered further.
This paper provides an overview of existing tools for analyzing MPI applications. First, the relevant steps for conducting a performance analysis are explained. Then, existing tools for performance analysis with MPI, widely used in industry and research today, are presented. These are categorized into "debugging tools",

"instrumentation and measurement tools", "performance analysis, visualization, and tuning tools", and "all-in-one tools". Many of the tools presented already existed in the 90s and still have a high relevance today. It can be assumed that the tools will also gain in importance with the increasing significance of parallel computing. In the second part, the current state of existing tools to support programming education with MPI is surveyed, and the use of the described professional tools in educational scenarios is discussed based on field reports.

2 RELATED WORK

A first overview of the visualization of parallel systems was published in 1993 by Kraemer and Stasko [37]. Their focus is on instrumentation and measurement. However, the tools described in the paper are now obsolete. With their survey from 2001, Moore et al. wanted to advance the development and advancement of the presented tools and libraries for performance analysis of MPI applications [48]. The "Survey of software environments for parallel distributed processing" by Delistavrou and Margaritis gives an extensive overview of different environments for parallel programming [18]. Also, various tools for analyzing parallel programs are mentioned but not explained in detail.

Two papers from the 2014 and 2020 Supercomputing Frontiers and Innovations conferences present various tools for analyzing parallel applications. In his paper from 2014, Mohr presents the state-of-the-art tools that are most commonly used for analysis and performance measurement of parallel applications [47]. He limits himself to the best-known tools, whose functionality is explained in detail. In 2020, Knobloch and Mohr presented performance analysis tools specifically for GPU computing [35]. Although the paper includes tools that can be used to analyze parallel MPI programs, the focus is on the general analysis of parallel programs concerning GPU computing.

The use of professional tools in programming education has also been discussed in the literature. Delistavrou and Margaritis integrated Eclipse PTP [20], TAU [50], ISP [59], and GEM [58] into a programming environment for teaching parallel programming with MPI [19]. Zhang et al. developed an online program through which students can upload their programs and submit them for execution on the Tianhe-2 supercomputer [78]. Through a user interface, learners can configure the execution and choose whether to collect profiling data using mpiP [3] or TAU during the execution. Yazici et al. also mention the use of GEM and TAU within their concept of using real-life applications when teaching parallel computing [76]. Malakar reports on his experiences in teaching, in which he used TAU, and HPCToolkit [28], among others, to explain MPI profiling [43].

This paper does not only present and categorize various professional performance analysis tools for MPI applications. It also discusses opportunities and problems in using them in educational scenarios. Here, perspectives and barriers arising from the general use of professional tools for programming education are described. The analysis is not limited to a specific tool.

3 PARTS AND TECHNIQUES TO FIND INEFFICIENT CODE

In order to be able to improve the performance of programs, an analysis of the code and various measurements during a program's run-time must be carried out. The results must then be provided to users. There are various possibilities for this, which are presented in the following sections.

3.1 Instrumentation and measurement

Software instrumentation means adding extra code to an application for monitoring a program's behavior [34]. There are two different types of instrumentation. First, in *static instrumentation*, extra code is added to the application statically. That can be done manually (e.g., by marking the source code by the user), automatically (by the compiler), or by linking against pre-instrumented libraries [31]. After static instrumentation, the application is executed with the added code for collecting corresponding data. This step is called measurement. Second, in *dynamic instrumentation*, instrumentation and measurement occur during run-time. So that the application does not have to be recompiled, the instrumentation is always based on the binary code. Therefore, dynamic instrumentation is also called *binary instrumentation*, although static instrumentation can also be based on the binary. Most professional performance analysis tools support both static instrumentation and dynamic instrumentation capabilities. For dynamic analysis, most tools use the Dyninst API [69].

Performance data can be collected through the instrumented code, either by *tracing* or *profiling*. Profiling collects summary statistics about a program during execution by retrieving performance metrics for specific events [62]. In MPI applications, profiling can be used to measure the number of MPI function calls and their total duration. Data collection is then triggered by an MPI routine's start and end. Profiling can also be based on sampling, where a hardware interval timer periodically interrupts the execution of the program to collect data [62]. Tracing is more detailed and represents temporal aspects of the program. A trace is a log of events within the program [75]. In contrast to profiling, tracing can show when and where an error or performance problem occurred in the source code. Tracing can be done by manually instrumenting the source code, it can also be relying on hardware such as the branch trace store in recent Intel CPUs, and it can be generated by an instrumentation set generator [75]. In general, tracing collects larger amounts of data than profiling, which means that tracing takes more time and thus can significantly affect the application execution time.

3.2 Debugging

Debugging is the identification of program errors in computer programs. In parallel applications, debugging is often used to identify race conditions or deadlocks and find their reasons. In complex HPC environments and when using low-level programming paradigms, where the users are responsible for memory management, debugging becomes very challenging [35]. Besides general programming errors (e.g., access to an array's non-existing index), debuggers must also find MPI-specific errors. Such MPI-specific errors, like starting a non-blocking communication, which cannot be completed before `MPI_Finalize` is called, are harder to find by debugging [27].

3.3 Analysis

Program analysis often includes the search for inefficient code sections, performance problems, oversubscribed cores, imbalances, or bottlenecks. Data collected through instrumentation and measurement is stored in trace or profiling files. These can be read and processed by an analysis process to provide feedback to users subsequently. Here, a distinction is made between run-time analysis and so-called *post-mortem* analysis. Most tools are based on post-mortem analysis of collected data. After the complete execution of a program, trace and profile files are read and analyzed in a separate step. Finally, the results are presented to users as feedback. Post-mortem analysis can take a long time, especially for large programs. As soon as changes are made to the code or manual instrumentation annotations, the analysis must be performed again, which has consequently re-compiling and re-executing the program until all performance problems are solved.
Performance analysis during run-time in available tools usually takes place based on dynamic instrumentation (e.g., Paradyn [70], Dyninst API [69]). Code can be modified at run-time, eliminating the need for multiple compilations and executions. An analysis during run-time involves looking for specific properties (e.g., bottlenecks) in programs that can hurt performance. Furthermore, an automated application tuning can make concrete suggestions for improvements (e.g., Periscope Tuning Framework [55]). However, run-time analysis can also take place independently of dynamic instrumentation by making the analysis results available while the program is still running.

3.4 Feedback

To best support users in improving their applications or codes, feedback must be provided from which users can deduce which problems occur, why they occur, and how they can be solved. This feedback can be provided in various ways and is visual in nature. For example, some tools mark parts of the source code where problems could have occurred (e.g., HPCToolkit [28], ARM DDT [41]). Then, either users have to find ways to improve the flagged code fragments themselves, or the tools provide suggestions. Other tools show problems within a graphical representation in human-readable files (e.g., ARM Performance Reports [42], MAQAO [51]) or Graphical User Interfaces (GUIs) (e.g., TAU [50], Scalasca [33], Vampir [24]). Often these files or user interfaces include diagrams, charts, and timelines represented in two or three dimensions.

4 DEBUGGING TOOLS FOR MPI APPLICATIONS

The following section presents and differentiates commercial and open-source tools for debugging parallel MPI programs.
ARM DDT [41] is a commercial debugging tool as part of the ARM FORGE debug and profile suite. The focus is on tracking memory errors. Memory accesses are closely tracked, and allocations are checked for possible errors. For simplifying the debugging process, array visualization is also provided so that users can track the current contents of an array at any time during run-time.
A very similar alternative is **TotalView** [52]. These two commercial tools support GPU debugging, and they are very elaborate. Alternatively, there are open-source solutions for debugging parallel applications, which do not support GPU debugging and are not specialized for debugging memory accesses.
Marmot Umpire Scalable Tool (MUST) [68] extends and scales the functionality of the run-time checker Marmot [38] and the correctness checker Umpire [74], and extends its functions for MPI-specific deadlock analysis [27]. Marmot's main functions are verifying the correct usage of MPI resources, and the time-out-based search for deadlocks [26]. Umpire can be used mainly for graph-based deadlock searches. MUST is an open-source run-time error detection tool for MPI applications, which was developed to ensure efficient debugging of parallel programs even within large systems. The scalability of MUST depends on the attached application's scalability, and it observes all MPI communication operations, focusing on detecting deadlocks, datatype matching violations, and incorrect communication buffer use [54, 68].
Stack Trace Analysis Tool (STAT) [39] is a highly scalable, open-source debugging tool for parallel MPI applications. Debugging "extreme-scale applications" is often time-consuming, so STAT reduces the *problem exploration space* from thousands of processes to a few by sampling stack traces before debugging [6]. It collects stack traces from all processes and merges them into a compact form [39]. Processes that exhibit similar behavior are grouped into equivalence classes. In the STAT GUI, the resulting merged stack trace is visualized, and based on it, debugging is performed by users. For example, deadlocks can be detected by visually displaying call stacks of different processes when all processes are waiting simultaneously.
In-Situ Partial (ISP) [59] is an open-source MPI debugger or dynamic verifier that, unlike the other tools presented, is based on the idea of model checking [71]. The entire state space of a program is verified as a model and checked using safety properties so that, for example, no deadlock can occur. In contrast to conventional model checkers, ISP works with the existing program code and not with special verification languages. GEM [58] can be used as a graphical user interface for ISP integrated into Eclipse PTP [29]. ISP also provides a Java GUI for controlling the debugger and analyzing the results.

5 INSTRUMENTATION AND MEASUREMENT TOOLS

This section describes tools that can be used independently for a program's instrumentation and measurement. These tools produce files that can be evaluated by analysis and visualization tools.
mpiP [3] is a lightweight profiling library for MPI which collects statistical information about MPI functions [73]. It generates very little overhead, as only superficial data is collected and summarized as a report. This report contains information about the run-time, the time spent in MPI functions, and various call site statistics.
Dyninst [69] is an API for dynamic binary instrumentation used by many popular tools, such as Open|SpeedShop [4], TAU [50], Extrae [65] and STAT [39]. It is based on and uses the functionality of the tool Paradyn [70], which had its beginnings in 1993 and has been continuously developed since then. The peculiarity is that Paradyn has been the first tool that uses dynamic instrumentation and measurement [46] to find only current performance problems

in extensive programs. Therefore, it searches for possible bottlenecks based on typical programming and performance errors. With the help of the W^3 Search Model, a performance consultant guides the placement and modification of the instrumentation during runtime[70]. The advantage of dynamic instrumentation is that it does not introduce a large overhead associated with the typical tracing process.
Extrae [65] is a pure instrumentation tool for the generation of trace files for the visualization tool Paraver [13]. It offers different types of instrumentation. Dynamic instrumentation during runtime is possible by Dyninst. Mainly, `LD_Preload` is used to intercept binaries during load time. Tracing is event-based, e.g., by calling an MPI function.
Score-P [32, 36] was developed to provide a unified instrumentation framework whose output can be used by different evaluation or analysis tools. Developers often have to analyze their programs with different tools, as each tool offers different functions and views. This means a high effort if the steps of instrumentation and measurement have to be performed separately for each tool, since each visualization tool requires different formats for trace and profiling files. Periscope can store the data collected by instrumentation and measurement in Open Trace File 2 (OTF2) (provided by Scalasca [33] and Vampir [24]), CUBE4 (provided by Scalasca and TAU [50]), or TAU snapshot formats. Additionally, the data can be queried by an online interface [36]. This allows users to combine several well-known and comprehensive performance analysis tools with only one instrumentation and measurement step. Score-P provides, for example, compiler instrumentation, MPI library interposition, source code instrumentation via TAU instrumenter, and user instrumentation [36].

6 TOOLS FOR PERFORMANCE ANALYSIS, VISUALIZATION, AND TUNING

The performance analysis, visualization, and tuning tools described previously do not have their own instrumentation and measurement techniques and analyze existing trace or profiling files. Scalasca and Vampir used to have their own mechanisms for instrumentation and measurement. However, these mechanisms have not been further developed for both. Instead, Score-P is used as a unified infrastructure. **Scalasca** [33] is a trace-based performance analysis software specially designed for large-scale systems. It is the official successor to KOJAK [7]. A special feature of Scalasca is the detection of wait states in MPI applications, which can be caused, for example, by an uneven load distribution [21]. Scalasca requires as input a so-called summary report in OTF2 format. This is typically created by the instrumentation framework Score-P. After instrumentation, Scalasca's *measurement and analysis nexus* configures and manages the collection of the application's performance experiments. [79]. Users can choose between a summary analysis report or event traces automatically given to Scalasca's trace analyzer to identify bottlenecks and *wait states*. So the report generated by the Scalasca trace analysis is similar to the summary report but includes identified inefficiency metrics [79]. Analysis reports are created in CUBE4 format and can be explored with the Cube GUI. The Cube GUI contains three views of the data, which are called dimensions of the performance space: *metric dimension*, *program dimension*, and *system dimension*. Within the metric dimension, users can select which metric should be analyzed in more detail (e.g., synchronizations, communications, bytes transferred). The program dimension shows a call tree of the corresponding executed functions. The system dimension provides information about involved processes and threads in the form of diagrams.
Vampir [24] is a commercial tool for detailed, visual analysis of trace files. It uses Score-P's instrumentation and generates and visualizes three different types of charts from a trace file. In the *Timeline Charts*, events are displayed according to their temporal occurrence on a horizontal axis. The individual processes are viewed separately, and their function calls are displayed visually. *Statistical Charts* show a summary of the occurrence of functions, processes, communication, I/O, and the call tree. They are suitable for gaining a quick overview of the program's behavior. *Informational Charts* provide a more detailed insight into individual function calls and their contexts. Functions are assigned to groups by color and displayed in a tree structure.
Periscope Tuning Framework (PTF) [55] works on Score-P's online access interface. It enables automatic search for performance problems through iterative online analysis. Measurements are configured, obtained, and evaluated on the fly so that no trace or profiling files have to be stored [23]. Periscope is operated via a graphical user interface, which is integrated into Eclipse PTP [20]. Users write their code in Eclipse and can search for properties that have a negative impact on performance. For example, users can search for the *LateSender* property in the program [45]. Score-P's monitor looks for scenarios where an `MPI_Receive` occurs before the corresponding `MPI_Send` so that the receiver has to wait a long time to receive a message, and corresponding places are marked in the code. Another special feature for analyzing MPI communication is that PTF can automatically detect wait patterns.
Paraver [13] uses trace files created by Extrae. It is a flexible parallel program visualization and analysis tool based on an easy-to-use GUI. The GUI has two main views: The *timeline view* visualizes the behavior of an application over time, and the *statistics view* (histogram, profiles) extends the analysis with the distribution of metrics. A special feature that distinguishes Paraver from other presented tools is the ability to split trace files, save the resulting individual parts separately, and compare them.
Extra-P [17] is an automatic performance-modeling tool that supports finding scalability errors. A scalability error means that scaling in a part of the program becomes unexpectedly bad. Extra-P visualizes them in the form of performance patterns. A performance model expresses a performance metric of interest (e.g., execution time, energy consumption) as a function of execution parameters (e.g., size of the input, number of processors) [17]. It also creates human-readable models for performance metrics (e.g., floating-point operations, MPI calls), and these metrics can be assigned as possible reasons for the scalability of program errors.

7 ALL-IN-ONE TOOLS

The following section presents all-in-one performance analysis solutions that perform instrumentation and measurement, as well as analyze and evaluate them and provide feedback to users.

ARM Performance Reports [42] is a commercial tool for producing HTML reports that summarize and characterize MPI application performance. It belongs to the ARM Forge Suite and is based on the underlying profiling mechanism. The feedback report is divided into three parts: *Compute* gives information about how much time the application took in total and how much of that time was spent in libraries, *MPI* summarizes the times of the MPI calls, and *I/O* shows the times spent in file system I/O. Furthermore, different *Breakdown* submenus are listed, where the spent time for CPU, MPI, or I/O, is more exactly subdivided and visualized. This way, bottlenecks, and performance issues can be detected and identified.
Intel Trace Analyzer and Collector (ITAC) [15] is part of the Intel oneApi Base & HPC Toolkit and analyzes MPI programs regarding their correctness and possible bottlenecks, buffer overlaps, and deadlocks. In addition, the program behavior can be visualized within a GUI to identify communication hot spots and improve efficiency [15]. The diagrams display how much time applications spend within MPI functions, OpenMP functions, and serial code. All these parts can be analyzed separately. Typical MPI *LateSender* or *wait at barrier* problems are automatically identified and visualized in the GUI, and their reasons can be analyzed within an event timeline.
Open|SpeedShop [4] consists of a plugin organized infrastructure. Each program component is implemented as a plugin so that each functionality can be used independently and coherently. Moreover, Open|SpeedShop can be flexibly extended by new plugins. Programmers can include all plugins of Open|Speedshop when developing new tools for performance analysis. That simplifies the development of new analysis tools or provides alternative instrumentation options. A collector plugin coordinates binary instrumentation based on DPCL/Dyninst and collects data. This is then passed to a data abstraction layer for storage in a SQL database. A view plugin queries the data there, and the panel plugin represents the data in a GUI [60]. The analysis of programs takes place as so-called "experiments". The type of experiment chosen decides in what way the instrumentation will be performed (e.g., different types of tracing and sampling). Open|Speedshop provides two different mechanisms to introduce performance instrumentation into application binaries: offline (data collector is loaded at link time) or online (insert data collection into running binary) in combination with a tree-based aggregation network based on MRNet [61]. Typically, offline instrumented data is displayed post-mortem. Online data can be displayed and analyzed in Open|Speedshop's GUI during run-time. The GUI provides a *stats panel* where users can filter for all executed functions, statements, linked objects, and loops. These elements are displayed, including the CPU time they took, and can be sorted, for example, by the percent of CPU time taken. At the same time, users can display the area of code within a source panel that has taken up a lot of CPU time. A small graphical representation can generate different charts of the CPU time used. Further information, e.g., which process had the most MPI library time, where the most expensive call to `MPI_Wait` occurred or information about load balancing can be retrieved as well.
Tuning and Analysis Utilities (TAU) [50] is one of the most comprehensive tools for analyzing parallel applications. TAU offers many ways to instrument applications within one API. Instrumentation is possibly source-based through manual source code annotations, preprocessor-based through automatically created annotations, and compiler-based by optimizing and re-instrumenting the source code. In addition, TAU supports wrapper library-based instrumentation for tracking MPI calls, binary instrumentation via DyninstAPI, interpreter-based instrumentation, component-based instrumentation, virtual-machine-based instrumentation, multi-level instrumentation, and selective instrumentation [63]. As an alternative to TAU's own instrumentation API, files can be instrumented by Score-P too. TAU supports both, various tracing and profiling methods that users can consciously control [63]. TAU's visualization tool is called "ParaProf". In addition to 2D visualizations that visualize the called functions of the individual threads or processes in a timeline, ParaProf offers a 3D environment that users can operate and control intuitively. A special feature is that within the 3D environment, all cluster nodes can be viewed as a model. The utilization of the nodes is encoded in color. In addition, different 3D models show the communication load of the nodes to each other. With *TAUoverSupermon* and *TAUmon*, research approaches exist for the implementation of an online version of the TAU analysis tool. They are focused on the problem that the amount of data increases immeasurably, especially through I/O, due to the ever-increasing level of parallelization in large-scale systems [40, 49]. These online features have not yet been integrated into the official version of TAU.
Modular Assembly Quality Analyzer and Optimizer (MAQAO) [51] combines static and dynamic analysis based on binary files and focuses on core performance optimization. MAQAO has three main modules. First, *LProf*, a sampling-based profiler that collects a list of loops and functions during execution. Second, *CQA*, a static analyzer that assesses the quality of code. Third, *ONE View*, which aggregates the results from LProf and CQA and makes a report as HTML-File [72]. MAQAO is well suited to identify time-consuming functions and loops and detect load balancing issues.
HPCToolkit [28] avoids the instrumentation step and combines a profile execution ("hpcrun") with a binary analysis ("hpcstruct") instead. "hpcrun" is an asynchronous sampling method using hardware performance counters. This type of profile execution has a low overhead of 1%-5% [28]. In order to find the reasons for bottlenecks more quickly, *calling-contect-sensitive* measurements are associated with source code structures in a binary analysis ("hpcstruct"). To combine performance data and structure information, "hpcprof" overlays call path profiles and traces with program structure and correlates the result with source code. The results are written into a database that can be explored via "hpcviewer", a GUI that presents profiles or traces. The GUI is divided into three sections. First, the *source pane* with the source code. Second, the *navigation pane*, which represents the application as a hierarchical tree-based structure for displaying the performance data. Third, the *metric pane*, in which the concrete metrics of individual samples are displayed [44]. A line of code is associated with a part of the tree structure and several sample metrics. By clicking on a line of code, the associated structures are highlighted, so that code fragments with poor performance become apparent, and users can identify their reasons more quickly. Moreover, HPCToolkit highlights code areas where inefficiencies and anomalies occurred automatically during sampling.
Caliper [9, 12] differs from the other tools mentioned because it

can integrate performance profiling capabilities directly into an HPC application and make them available whenever the application is executed [10]. The tool is based on the idea of focusing not only on the performance data of an individual program run but on the comparison between different executions and on different HPC systems by profiling and collecting data at each execution of the program. For this purpose, the performance profiling library is integrated into the application, and measurement is controlled, for example, by command line options [11]. Developers must manually extend the application's code through commands to enable profiling at desired locations. The data can be saved as a human-readable report, a .hatched file for the Python library Hatched, or a .cali file. The .cali files can be stored locally or online in a SQL database and analyzed with the web-based visualization tool *SPOT*, which mainly provides users with diagrams to compare different program runs.

8 EDUCATIONAL TOOLS FOR PARALLEL PROGRAMMING WITH MPI

At many educational institutions, the theory of parallel programming is taught as a priority, while practical work with real high-performance systems is addressed little. Students program, for example, with MPI on a single computer or even their laptop. This is either due to the lack of resources or the complexity of using computing clusters. This approach does not allow learners to see benefits and advantages of parallel programming, nor does it allow them to gain experience with HPC environments and computer clusters. Experience reports and studies conclude that the context of realistic scenarios is essential for learning parallel programming and understanding its benefits [19, 30, 77]. To reduce this gap between theory and practice, simulation tools are increasingly being implemented to simulate or virtualize different infrastructures of clusters [14, 56, 64, 66, 77]. Gusev et al. presented a prototype of a cloud-based e-learning and benchmarking platform for students to try out predefined algorithms with different predefined implementations and compare them in terms of speed, speedups, and efficiency [25]. **ParaLib** [67] is a library of parallel algorithms that compares the computational complexity of different algorithms and parallel programming languages with respect to a standard programming problem [22]. The execution time and the speedup of computational efficiency are displayed for each experiment. **System for AUtomated Code Evaluation (SAUCE)** [5] is a web-based tool with the main function to assess parallel programming tasks in an automated way so that learners get immediate feedback about whether the task was solved correctly.

9 USE OF PROFESSIONAL PERFORMANCE ANALYSIS TOOLS IN EDUCATIONAL SCENARIOS

Although some tools have already been developed for specific use in programming education with MPI, none focus on a similar performance analysis as the presented professional tools. Educational tools are often limited to standard problems, such as matrix multiplications or sorting algorithms, which students can analyze [57, 66, 67]. Several researches pointed out that it is essential for students' understanding and acceptance of parallel programming to show them authentic scenarios of HPC with large problems on large machines and not to limit themselves exclusively to small standard problems [30, 43]. Joiner et al. also highlight the importance of performance analysis in this context [30]. The professional tools presented can analyze any application, whether a matrix multiplication or a large simulation. The authors have also found out that it is important to teach how speedup can be achieved by parallel programming at the beginning of a parallel programming course. This is particularly effective if the speedup is visually presented to students. The described professional tools for the visualization of the performance of MPI applications can not only visualize the speedup but also offer visualizations of process communication, processor memory, I/O, and performance issues or bottlenecks. On the one hand, this can contribute to a far-reaching understanding, but on the other hand, the professional tools are aimed at experienced developers and can quickly overwhelm beginners. In practice, professional tools are mainly used for quick analysis of large applications and allow developers to spend less time tuning their application and more time focusing on underlying science [21]. In educational scenarios, the focus is on learners recognizing, understanding, and avoiding their errors in the future.

In the past, much research has been done about the use of professional tools in programming education [19, 43, 76, 78]. The tool TAU was used most frequently for this purpose. However, in all scenarios described, the tools were not used until the end of a course, after the learners had been given a comprehensive knowledge of instrumentation and measurement. This contradicts the idea of Joiner et al. to integrate visual feedback of applications at the beginning of a course.

The tools presented in this paper often require instrumentation of code. The manual code instrumentation is associated with a high effort that grows proportionally with the size of the application. Deep knowledge is required, as potentially inefficient code areas must be manually annotated. Automated instrumentation requires only user execution and not manual development of instrumentation code. However, the process can still take a long time, and users must perform the instrumentation still by themselves, especially for the instrumentation-only tools (mpiP, Caliper, Extrea, and Score-P). Decisions about various sampling, tracing, and profiling methods must be made by users and require appropriate knowledge. Even in tools where a binary analysis takes place, and the measurement can be controlled entirely in a GUI (e.g., Open|Speedshop), various procedures have to be selected manually [4]. Depending on the size of the application's code, instrumentation and measurement can take several hours or even days. Programming lectures are limited in time, and long waiting times negatively affect learners' motivation. Malakar used TAU and HPCToolkit within the teaching of parallel computing and reports long waiting times explicitly when loading and displaying profiling data from the network, as the university did not provide a proper computing cluster but only used various computer labs as architecture [43]. In the course evaluation, the learners criticized these waiting times. Joiner et al. also emphasize that it is particularly effective if the speedup of an application is apparent to students during run-time rather than post-mortem [30]. To ensure immediate as well as straightforward feedback, online tools that provide immediate visual feedback at run-time should be used for educational scenarios. In the literature, the term "online tool" is often used when tools use dynamic/binary instrumentation so that users can intervene in the process while the program is running or retrieve data during this time [8, 21]. Tools like Caliper or HPCToolkit support storing the collected data in an online database. Score-P provides an online interface through which requests can be sent. Just Open|SpeedShop does not only write the data collected by the binary analysis to a database, but can also visualize it within a real-time GUI, allowing users to track in real-time what is happening in their running program. However, the tool provides only minimal diagrams and plots of a program's behavior and performance and focuses more on flagging code that leads to performance problems. An online tool that visualizes the utilization of the nodes in real-time, for example, can be very suitable for obtaining simple and rapid feedback. Users observe the program's behavior immediately after they have started their program. This does not only contribute significantly to a better understanding. Users also see at first glance if, for example, one of the nodes is completely overloaded while all the others have only a low workload. They can stop the execution of their application at that point and improve the code.

10 CONCLUSION AND FUTURE WORK

There is a wide range of performance analysis tools for MPI applications. They support different methods for instrumentation and measurement and offer partly different possibilities for the visualization of a program's behavior and the performance of an application in two or three dimensions. Using these tools in programming education could significantly contribute to a better understanding and help learners write, from the beginning, efficient code that exploits the full potential of large computational clusters. However, it turned out that the presented professional tools are too complex, and all require prior knowledge in instrumentation and measurement. In addition, most tools do not provide immediate online feedback during run-time, which is essential for maintaining learner motivation. In order to support the education of parallel programming with MPI in the future, an online tool can be developed that supports the performance analysis of MPI applications but does not require any instrumentation or measurement steps, which must be carried out by learners, and can be used intuitively even by beginners.

REFERENCES

[1] 1998. GNU gprof - The GNU profiler. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html. Accessed: 2022-11-16.

[2] 2000-2022. Valgrind. https://valgrind.org. Accessed: 2022-11-16.

[3] 2020. mpiP 3.5. https://github.com/LLNL/mpiP. Accessed: 2022-11-04.

[4] 2022. Open|Speedshop. https://openspeedshop.org. Accessed: 2022-10-26.

[5] 2022. SAUCE - System for AUtomated Code Evaluation. https://github.com/moschlar/SAUCE. Accessed: 2022-11-02.

[6] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. 2007. Stack Trace Analysis for Large Scale Debugging. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*. IEEE, 1–10. https://doi.org/10.1109/IPDPS.2007.370254

[7] Juelich Supercomputing Centre at Forschungszentrum Juelich and Innovative Computing Laboratory at the University of Tennessee. 2022. KOJAK. https://icl.utk.edu/kojak/index.html. Accessed: 2022-10-24.

[8] Jean-Baptiste Besnard, Marc Pérache, and William Jalby. 2013. Event Streaming for Online Performance Measurements Reduction. In *42nd International Conference on Parallel Processing (ICPP 2013)*. IEEE Computer Society, 985–994. https://doi.org/10.1109/ICPP.2013.117

[9] David Boehme. 2015-2021. Caliper: A Performance Analysis Toolbox in a Library. http://software.llnl.gov/Caliper/. Accessed: 2022-10-20.

[10] David Boehme. 2020. Tool Time: Caliper - A Performance Analysis Toolbox in a Library. https://pop-coe.eu/blog/tool-time-caliper-a-performance-analysis-toolbox-in-a-library.

[11] David Böhme, Pascal Aschwanden, Olga Pearce, Kenneth Weiss, and Matthew P. LeGendre. 2021. Ubiquitous Performance Analysis. In *High Performance Computing - 36th International Conference (ISC High Performance 2021) (Lecture Notes in Computer Science, Vol. 12728)*. Springer, 431–449. https://doi.org/10.1007/978-3-030-78713-4_23

[12] David Böhme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Giménez, Matthew P. LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*. IEEE Computer Society, 550–560. https://doi.org/10.1109/SC.2016.46

[13] BSC. 2022. Paraver. https://tools.bsc.es/paraver. Accessed: 2022-10-24.

[14] Henri Casanova, Arnaud Legrand, Martin Quinson, and Frédéric Suter. 2018. SMPI Courseware: Teaching Distributed-Memory Computing with MPI in Simulation. In *2018 IEEE/ACM Workshop on Education for High- Performance Computing (EduHPC@SC 2018)*. IEEE, 21–30. https://doi.org/10.1109/EduHPC.2018.00006

[15] Intel Corporation. [n.d.]. Intel Trace Analyzer and Collector (ITAC). https://www.intel.com/content/www/us/en/developer/tools/oneapi/trace-analyzer.html#gs.ijzdvr. Accessed: 2022-11-18.

[16] Association Curricula. 2013. Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. (2013). https://doi.org/10.1145/2534860

[17] Technische Universtiaet Darmstadt and ETH Zurich. 2020. Extra-P. https://github.com/extra-p/extrap. Accessed: 2022-10-24.

[18] Constantinos T. Delistavrou and Konstantinos G. Margaritis. 2010. Survey of Software Environments for Parallel Distributed Processing: Parallel Programming Education on Real Life Target Systems Using Production Oriented Software Tools. In *14th Panhellenic Conference on Informatics (PCI 2010)*. IEEE Computer Society, 231–236. https://doi.org/10.1109/PCI.2010.26

[19] Constantinos T. Delistavrou and Konstantinos G. Margaritis. 2011. Towards an Integrated Teaching Environment for Parallel Programming. In *15th Panhellenic Conference on Informatics (PCI 2011)*. IEEE Computer Society, 3–7. https://doi.org/10.1109/PCI.2011.16

[20] Eclipse Foundation. 2022. Eclipse Parallel Tools Platform (PTP). https://www.eclipse.org/ptp/. Accessed: 2022-11-16.

[21] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca performance toolset architecture. *Concurr. Comput. Pract. Exp.* 22, 6 (2010), 702–719. https://doi.org/10.1002/cpe.1556

[22] Victor Gergel, Evgeny Kozinov, Alexey Linev, and Anton Shtanyuk. 2016. Educational and Research Systems for Evaluating the Efficiency of Parallel Computations. In *Algorithms and Architectures for Parallel Processing (ICA3PP 2016) (Lecture Notes in Computer Science, Vol. 10049)*. Springer, 278–290. https://doi.org/10.1007/978-3-319-49956-7_22

[23] Michael Gerndt, Ventsislav Petkov, and Yuri Oleynik. 2010. Performance analysis with Periscope. https://www.vi-hps.org/cms/upload/material/tw10/vi-hps-tw10-Periscope_Overview.pdf. Accessed: 2022-10-24.

[24] GWT-TUD GmbH. 2022. Vampir. https://vampir.eu. Accessed: 2022-10-24.

[25] Marjan Gusev, Sasko Ristov, Goran Velkoski, and Bisera Ivanovska. 2014. E-learning and Benchmarking Platform for Parallel and Distributed Computing. *Int. J. Emerg. Technol. Learn.* 9, 2 (2014), 17–21. https://doi.org/10.3991/ijet.v9i2.3215

[26] Tobias Hilbrich. 2014. *Runtime MPI Correctness Checking with a Scalable Tools Infrastructure*. Ph. D. Dissertation. Dresden University of Technology. https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-175472

[27] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI runtime error detection with MUST: advances in deadlock detection. In *SC Conference on High Performance Computing Networking, Storage and Analysis (SC 2012)*. IEEE/ACM, 30. https://doi.org/10.1109/SC.2012.79

[28] Rice University Houston. 2000-2022. HPCToolkit. http://hpctoolkit.org/index.html. Accessed: 2022-10-24.

[29] Alan Humphrey, Christopher Derrick, Ganesh Gopalakrishnan, and Beth Tibbitts. 2010. GEM: Graphical Explorer of MPI Programs. In *39th International Conference on Parallel Processing (ICPP Workshops 2010)*. IEEE Computer Society, 161–168. https://doi.org/10.1109/ICPPW.2010.33

[30] David A. Joiner, Paul Gray, Thomas Murphy, and Charles Peck. 2006. Teaching parallel computing to science faculty: best practices and common pitfalls. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2006)*. ACM, 239–246. https://doi.org/10.1145/1122971.1123007

[31] Forschungszentrum Juelich. [n.d.]. Score-P, scalable performance measurement infrastructure for parallel codes. https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/index.html. Accessed: 2022-10-16.

[32] Forschungszentrum Juelich. [n.d.]. Score-P, Scalable performance measurement infrastructure for parallel codes. https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/index.html. Accessed: 2022-10-20.

[33] Forschungszentrum Juelich and Technische Universitaet Darmstadt. 2022. Scalasca. https://www.scalasca.orgl. Accessed: 2022-10-24.

[34] Torsten Kempf, Kingshuk Karuri, and Lei Gao. 2008. Software Instrumentation. In *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc. https://doi.org/10.1002/9780470050118.ecse386

[35] Michael Knobloch and Bernd Mohr. 2020. Tools for GPU Computing - Debugging and Performance Analysis of Heterogenous HPC Applications. *Supercomput. Front. Innov.* 7, 1 (2020), 91–111. https://doi.org/10.14529/jsfi200105

[36] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2011. 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*. Springer, 79–91. https://doi.org/10.1007/978-3-642-31476-6_7

[37] Eileen T. Kraemer and John T. Stasko. 1993. The Visualization of Parallel Systems: An Overview. *J. Parallel Distributed Comput.* 18, 2 (1993), 105–117. https://doi.org/10.1006/jpdc.1993.1050

[38] B. Krammer, K. Bidmon, M.S. Müller, and M.M. Resch. 2004. MARMOT: An MPI analysis and checking tool. In *Parallel Computing*. Advances in Parallel Computing, Vol. 13. North-Holland, 493–500. https://doi.org/10.1016/S0927-5452(04)80063-7

[39] Lawrence Livermore National Laboratory. [n.d.]. STAT: Stack Trace Analysis Tool. https://hpc.llnl.gov/software/development-environment-software/stat-stack-trace-analysis-tool. Accessed: 2022-10-20.

[40] Chee Wai Lee, Allen D. Malony, and Alan Morris. 2010. TAUmon: Scalable Online Performance Data Analysis in TAU. In *Euro-Par 2010 Parallel Processing Workshops - HeteroPar, HPCC, HiBB, CoreGrid, UCHPC, HPCF, PROPER, CCPI, VHPC (Lecture Notes in Computer Science, Vol. 6586)*. Springer, 493–499. https://doi.org/10.1007/978-3-642-21878-1_61

[41] Arm Limited. 2022. ARM DDT, The Number One Debugger for C, C++ and Fortran, Threaded and Parallel Code. https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt. Accessed: 2022-10-20.

[42] Arm Limited. 2022. ARM Performance Reports. https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-performance-reports. Accessed: 2022-10-20.

[43] Preeti Malakar. 2019. Experiences of Teaching Parallel Computing to Undergraduates and Post-Graduates. In *26th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW 2019)*. IEEE, 40–47. https://doi.org/10.1109/HiPCW.2019.00020

[44] John Mellor-Crummey, Nathan R. Tallent, Mike Fagan, and Jan Odegard. 2007. Application performance profiling on the Cray XD1 using HPCToolkit. In *Proc. of the Cray User's Group*.

[45] Robert Mijakovic, Michael Firbach, and Michael Gerndt. 2016. An architecture for flexible auto-tuning: The Periscope Tuning Framework 2.0. In *2nd International Conference on Green High Performance Computing (ICGHPC 2016)*. IEEE, 1–9. https://doi.org/10.1109/ICGHPC.2016.7508066

[46] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. 1995. The Paradyn Parallel Performance Measurement Tool. *Computer* 28, 11 (1995), 37–46. https://doi.org/10.1109/2.471178

[47] Bernd Mohr. 2014. Scalable parallel performance measurement and analysis tools - state-of-the-art and future challenges. *Supercomput. Front. Innov.* 1, 2 (2014), 108–123. https://doi.org/10.14529/jsfi140207

[48] Shirley Moore, David Cronk, Kevin S. London, and Jack J. Dongarra. 2001. Review of Performance Analysis Tools for MPI Parallel Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 2131)*. Springer, 241–248. https://doi.org/10.1007/3-540-45417-9_34

[49] Aroon Nataraj, Matthew J. Sottile, Alan Morris, Allen D. Malony, and Sameer Shende. 2007. *TAUoverSupermon* : Low-Overhead Online Parallel Performance Monitoring. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference (Lecture Notes in Computer Science, Vol. 4641)*. Springer, 85–96. https://doi.org/10.1007/978-3-540-74466-5_11

[50] Department of Computer and Information Science University of Oregon. 1997-2020. TAU, Tuning and Analysis Utilities. http://www.tau.uoregon.edu. Accessed: 2022-10-24.

[51] University of Versailles St Quentin. 2004-2021. Maqao (Modular Assembly Quality Analyzer and Optimizer). http://http://www.maqao.org. Accessed: 2022-10-26.

[52] Inc. Perforce Software. 2022. TotalView HPC Debugging Software. https://totalview.io/products/totalview. Accessed: 2022-10-20.

[53] Sushil K. Prasad, Almadena Yu. Chtchelkanova, Sajal K. Das, Frank Dehne, Mohamed G. Gouda, Anshul Gupta, Joseph F. JáJá, Krishna Kant, Anita La Salle, Richard LeBlanc, Manish Lumsdaine, David A. Padua, Manish Parashar, Viktor K. Prasanna, Yves Robert, Arnold L. Rosenberg, Sartaj Sahni, Behrooz A. Shirazi, Alan Sussman, Charles C. Weems, and Jie Wu. 2011. NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing: core topics for undergraduates. In *Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE 2011)*. ACM, 617–618. https://doi.org/10.1145/1953163.1953336

[54] Joachim Protze, Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, Wolfgang E. Nagel, and Matthias S. Müller. 2014. MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach. In *43rd International Conference on Parallel Processing Workshops, (ICPPW 2014)*. IEEE Computer Society, 206–215. https://doi.org/10.1109/ICPPW.2014.37

[55] Readex. 2020. Periscope Tuning Framework. https://www.readex.eu/index.php/periscope-tuning-framework/p. Accessed: 2022-10-24.

[56] Sasko Ristov, Marjan Gusev, Blagoj Atanasovski, and Nenad Anchev. 2013. Using EDUCache Simulator for the Computer Architecture and Organization Course. *Int. J. Eng. Pedagog.* 3, 3 (2013), 47–56. https://doi.org/10.3991/ijep.v3i3.2784

[57] Sasko Ristov, Marjan Gusev, and Goran Velkoski. 2014. Cloud E-learning and Benchmarking Platform for the Parallel and Distributed Computing Course. In *2014 IEEE Global Engineering Education Conference (EDUCON 2014)*. IEEE, 645–651. https://doi.org/10.1109/EDUCON.2014.6826161

[58] Utah School of Computing. [n.d.]. GEM - Graphical Explorer of MPI Programs. http://formalverification.cs.utah.edu/GEM/. Accessed: 2022-11-04.

[59] Utah School of Computing. [n.d.]. ISP (In-situ Partial Order): a dynamic verifier for MPI Programs. http://formalverification.cs.utah.edu/ISP-release/. Accessed: 2022-11-04.

[60] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. 2008. Open | SpeedShop: An open source infrastructure for parallel performance analysis. *Sci. Program.* 16, 2-3 (2008), 105–121. https://doi.org/10.3233/SPR-2008-0256

[61] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. 2009. Analyzing the performance of Scientific Applications with Open|SpeedShop. In *Parallel Computational Fluid Dynamics*. 151–159.

[62] Sameer Shende. 1999. Profiling and tracing in linux. In *In Proceedings of Extreme Linux Workshop*.

[63] Sameer Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (2006), 287–311. https://doi.org/10.1177/1094342006064482

[64] Elizabeth Shoop, Richard A. Brown, Eric Biggers, Malcolm Kane, Devry Lin, and Maura Warner. 2012. Virtual clusters for parallel and distributed education. In *Proceedings of the 43rd ACM technical symposium on Computer science education (SIGCSE 2012)*. ACM, 517–522. https://doi.org/10.1145/2157136.2157287

[65] BSC Tools. 2022. Extrae. https://tools.bsc.es/extrae. Accessed: 2022-10-20.

[66] Lobachevsky University. 2022. ParaLab. https://hpc-education.unn.ru/en/trainings/teachware/paralab. Accessed: 2022-11-02.

[67] Lobachevsky University. 2022. ParaLib – Parallel Computational Methods Library. https://hpc-education.unn.ru/en/trainings/teachware/paralib. Accessed: 2022-11-02.

[68] RTWH Aachen University. 2022. MUST - MPI Runtime Correctness Analysis. https://itc.rwth-aachen.de/must/. Accessed: 2022-10-20.

[69] University of Wisconsin University of Maryland. 2019. Dyninst. https://www.dyninst.org. Accessed: 2022-10-20.

[70] Computer Sciences Department University of Wisconsin. 2020. Paradyn. http://www.paradyn.org/overview/screen-shots.html. Accessed: 2022-10-20.

[71] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. ISP: a tool for model checking MPI programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2008)*. ACM, 285–286. https://doi.org/10.1145/1345206.1345258

[72] Cédric Valensi, William Jalby, Mathieu Tribalat, Emmanuel Oseret, Salah Ibnamar, and Kevin Camus. 2019. Using MAQAO to Analyse and Optimise an Application. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2019)*. 423–424. https://doi.org/10.1109/MASCOTS.2019.00052

[73] Jeffrey Vetter and Chris Chambreau. 2014. mpiP: Lightweight, Scalable MPI Profiling. http://gec.di.uminho.pt/Discip/MInf/cpd1415/PCP/MPI/mpiP_%20Lightweight,%20Scalable%20MPI%20Profiling.pdf. Accessed: 2022-11-04.

[74] Jeffrey S. Vetter and Bronis R. de Supinski. 2000. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings Supercomputing 2000*. IEEE Computer Society, 51. https://doi.org/10.1109/SC.2000.10055

[75] Jack Whitham. 2016. Profiling versus Tracing. https://www.jwhitham.org/2016/02/profiling-versus-tracing.html. Accessed: 2022-10-17.

[76] Ali Yazici, Alok Mishra, and Ziya Karakaya. 2016. Teaching Parallel Computing Concepts Using Real-Life Applications*. *International Journal of Engineering Education* 32 (03 2016), 772–781.

[77] Gonzalo Zarza, Diego Lugones, Daniel Franco, and Emilio Luque. 2012. An Innovative Teaching Strategy to Understand High-Performance Systems through Performance Evaluation. In *Proceedings of the International Conference on Computational Science (ICCS 2012) (Procedia Computer Science, Vol. 9)*. Elsevier, 1733–1742. https://doi.org/10.1016/j.procs.2012.04.191

[78] Yuxiao Zhang, Jiang Li, Di Wu, and Yunfei Du. 2018. Improving Student Skills on Parallel Programming via Code Evaluation and Feedback Debugging. In *IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE 2018)*. IEEE, 1069–1073. https://doi.org/10.1109/TALE.2018.8615351

[79] Ilya Zhukov, Christian Feld, Markus Geimer, Bernd Mohr, Michael Knobloch, and Pavel Saviankou. 2015. Scalasca v2: Back to the Future. In *Tools for High Performance Computing 2014*. Springer International Publishing, 1–24. https://doi.org/10.1007/978-3-319-16012-2_1