An Overhead Analysis of MPI Profiling and Tracing Tools

Sascha Hunold
hunold@par.tuwien.ac.at
Faculty of Informatics
Research Group for Parallel Computing
TU Wien
Vienna, Austria

Jordy I. Ajanohoun*
ajanohoun@par.tuwien.ac.at
Faculty of Informatics
Research Group for Parallel Computing
TU Wien
Vienna, Austria

Ioannis Vardas†
vardas@par.tuwien.ac.at
Faculty of Informatics
Research Group for Parallel Computing
TU Wien
Vienna, Austria

Jesper Larsson Träff
traff@par.tuwien.ac.at
Faculty of Informatics
Research Group for Parallel Computing
TU Wien
Vienna, Austria

ABSTRACT

MPI performance analysis tools are important instruments for finding performance bottlenecks in large-scale MPI applications. These tools commonly support either the profiling or the tracing of parallel applications. Depending on the type of analysis, the use of such a performance analysis tool may entail a significant runtime overhead on the monitored parallel application. However, overheads can occur in different stages of the performance analysis with varying severity, e.g., the overhead when initializing an MPI context is typically less problematic than when monitoring a high number of short-lived MPI function calls.

In this work, we precisely define the different types of overheads that performance engineers may encounter when applying performance analysis tools. In the context of performance tuning, it is crucial to avoid delaying individual events (e.g., function calls) when monitoring MPI applications, as otherwise performance bottlenecks may not show up in the same spot as when running the applications without applying a performance analysis tool. We empirically examine the different types of overheads associated with popular performance analysis tools for a set of well-known proxy applications and categorize the tools according to our findings. Our study shows that although the investigated MPI profiling and tracing tools exhibit a rather unique overhead footprint, they hardly influence the net time of an MPI application, which is the time between the Init and Finalize calls. Performance engineers should be aware of all types of overheads associated with each tool to avoid very costly batch jobs.

*This work was partially supported by the Austrian Science Fund (FWF): project P 33884-N.
†This work was partially supported by the Austrian Science Fund (FWF): project P 31763-N31.

PERMAVOST '22, June 30, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9314-0/22/06…$15.00
https://doi.org/10.1145/3526063.3535353

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

MPI, performance analysis, profiling, tracing, overhead

ACM Reference Format:
Sascha Hunold, Jordy I. Ajanohoun, Ioannis Vardas, and Jesper Larsson Träff. 2022. An Overhead Analysis of MPI Profiling and Tracing Tools. In *Proceedings of the 2nd Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn Strategy (PERMAVOST '22), June 30, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3526063.3535353

1 INTRODUCTION

Mitigating performance bottlenecks in large-scale codes that run on supercomputers is a challenging task, as one needs to detect these bottlenecks first. Several performance analysis tools have been developed for solving this task, which mainly fall into two categories: (1) profiling and (2) tracing tools. Profiling tools provide a short, condensed performance fingerprint of an application. The output of profiling tools typically comprises the information of how much time is spent in different parts of the code, usually at function level. Tracing tools, in comparison, record the start and the end timestamp of certain parts of the code (events), which will eventually allow performance engineers to get an overview of the entire application run. These overviews are usually presented as Gantt charts. In addition to profiling and tracing tools, performance engineers may also use tools for reading out Hardware Performance Counters (HPCs), such as PAPI [20] or Likwid [21]. Such HPCs can also be used in combination with profiling or tracing tools to reason about the cause of performance penalties.

One of the biggest challenges for tool developers is to create performance tools that incur a small overhead. The first reason is that the behavior of the application under investigation should not be impacted by the use of a performance tool. Otherwise, identified, potential root causes of performance bottlenecks may simply turn out to be a tool artifact instead of a real performance issue. The second and also very important reason is that large-scale jobs on thousands of cores are costly (in terms of compute-hours). Thus,

performance analysis tools that significantly increase the job duration will not be used in practice.

In the present work, we investigate the overheads associated with various performance analysis tools in the context of the Message Passing Interface (MPI). First, we distinguish the different types of overheads that may occur during the execution of MPI programs when performance analysis tools are used for monitoring these applications. Second, we introduce our experimental setup. In particular, we discuss the choices of both the investigated proxy applications and the size of the input instances. Afterwards, we examine which overheads the various performance tools entail when monitoring applications in practice. In particular, we make the following contributions:

(1) We define the different types of overheads that can typically occur when applying performance analysis tool to large-scale MPI codes.
(2) We present the results of an in-depth comparative experimental study of the different overheads of performance analysis tools, which points out potential pitfalls when applying certain performance tools.

In the remainder of the paper, we first give a deeper introduction to the different performance analysis tools in Section 2. We define the different types of overheads that we can associate with performance tools in Section 3, where we also introduce the experimental setup of our study. We present experimental results in Section 4 and discuss the findings in Section 5.

2 RELATED WORK

Performance analysis tools allow us to associate hardware performance counters (HPCs), e.g., the number of level 2 data cache misses, with events (e.g., function calls) in order to pinpoint the root causes of performance bottlenecks. For solving this task, we can utilize tools that read out HPCs at different points time, e.g., PAPI [20] and Likwid [21]. On top of these HPCs, there are numerous other tools for analyzing the performance of parallel programs on parallel architectures. We will discuss these tools and their methods in more details in Section 3.

Chung et al. [7] analyzed the impact of performance tools on MPI applications specifically for the Blue Gene/L supercomputer. Their study included the tools IBM HPCT, Paraver, KOJAK, Tau, and mpiP. The authors examined the overall execution time of applications with and without using the tools, as well as the volume of the collected performance data. Chung et al. performed a weak scaling analysis with four MPI applications (e.g., SWEEP3D), and examined the overhead of each tool. They reported that profiling tools (like mpiP) had a very small overhead of about 3%, but the overhead of the tracing tools was super-linearly increasing with the number of processes, quickly reaching an overhead of 100% and more. Chung et al. then proposed ideas for compressing traces, which are similar to the ones used by ScalaTrace [16] and Pilgrim [24].

Since we analyze the impact of performance tools on proxy applications, we built upon work done previously. Klenk and Fröning [13] gave an in-depth analysis of typical MPI proxy applications, in which they characterize the computational pattern used by each of the MPI codes. They also provided statistics of the overall time spent in MPI communication operation when running each application with a different process count. The MPI profiles were obtained with IPM [18].

The work of Sultana et al. [19] is orthogonal to the paper by Klenk and Fröning [13], as they provide a detailed analysis of the types of MPI functions used in typical proxy applications, e.g., whether non-blocking collectives are used or how much time is spent in point-to-point communication. In the study of Sultana et al. [19], all applications were executed with 256 processes, which were profiled with an in-house MPI profiler that leveraged the PMPI interface.

Chunduri et al. [6] presented a study in which they characterize typical MPI usage strategies across a large set of MPI applications that were executed on two supercomputers at Argonne National Laboratory. In order to characterize the usage profile of MPI applications, the authors created their own MPI profiler called Autoperf, which also uses the PMPI infrastructure.

All previously mentioned approaches had in common that they relied on the PMPI interface to intercept MPI calls for profiling or tracing. When tools rely on the PMPI interface, they cannot easily be chained, which would often be beneficial. The QMPI [8] layer goes one step further by proposing a novel tools interface that allows the simultaneous use of different monitoring tools. From a software perspective, the approach follows a decorator pattern: At the innermost level, the actual MPI function is called. This function call is wrapped by a tool, which becomes the innermost tool. This wrapped function call is wrapped again by another tool, and so on. Thus, the runtime statistics of the outermost tool also include the time (and possibly other metrics) of the calls to the inner tools.

Last, we mention two other tools, Caliper [3] and the MicroBench Maker [10], that can be used for analyzing the MPI application performance. Caliper is a library that lets developers mark specific code regions of interest. These annotations can then be used to create performance profiles or traces for application runs. This is especially useful for the introspection of HPC libraries, as most tools gather profiling data only until a certain level in the software stack, e.g., the MPI layer. Thus, Caliper can be used to correlate performance data of events in the application layer with events below the MPI layer, which would not be possible with traditional tools.

MicroBench Maker [10] is an another approach to gain insights into the performance of MPI applications. Although, the tool is mostly targeted towards micro-benchmarking, e.g., collecting a sequence of measurements of `MPI_Allgather` calls, it can also be used for benchmarking certain code regions. The fundamental idea is that a code region is marked, similar to Caliper, and then executed a predefined number of times in order to obtain a statistical sample of its execution time. With that approach, it is possible to tune the performance of a given code region.

3 OVERHEAD ANALYSIS: THE SETUP

We start by defining the individual times that we can examine when monitoring MPI applications with different performance analysis tools.

3.1 Definitions: MPI Performance Times

We decompose the time for analyzing the performance of an MPI application into three different components, which are the *net*, the

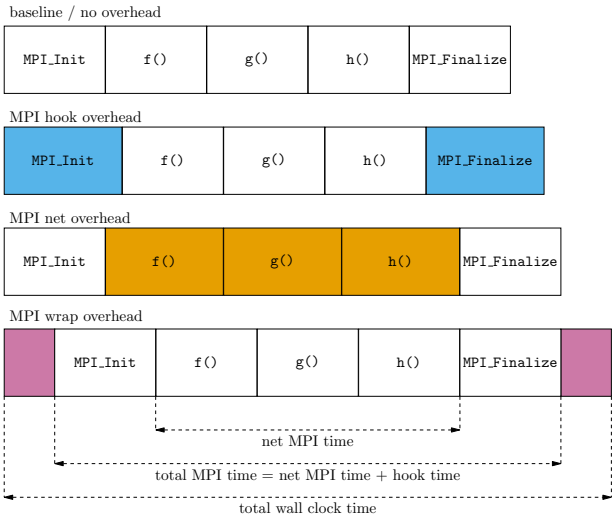

Figure 1: Types of overheads in an MPI application, e.g., an application consisting of three parallel functions `f()`, `g()`, and `h()`. ■ denotes the overhead caused by tools inside the MPI application (net). ■ denotes the overhead incurred due to tools inside `MPI_Init` and `MPI_Finalize` (hook). ■ denotes the additional wrap overhead of tools (wrap).

hook, and the *wrap* time. We formally define these times below and show an illustration of the individual times in Figure 1.

Definition 3.1 (MPI net time). The *MPI net time* t^i_{net} of process p_i denotes the time that process p_i spends executing instructions right after exiting `MPI_Init` and before entering `MPI_Finalize`.

Definition 3.2 (MPI hook time). The *MPI hook time* t^i_{hook} of process p_i denotes the total time that process p_i spends in the functions `MPI_Init` and `MPI_Finalize`.

Definition 3.3 (MPI wrap time). The *MPI wrap time* t^i_{wrap} of process p_i is defined as the commulative time that consists of (1) the time from starting the monitoring tool for process p_i until the process enters `MPI_Init` and (2) the time after `MPI_Finalize` until the monitoring tool finishes its execution.

Thus, the total time of monitoring an MPI application using a performance analysis tool for process p_i is defined as the sum of its net, hook, and wrap times, i.e., $t^i_{\text{overall}} = t^i_{\text{net}} + t^i_{\text{hook}} + t^i_{\text{wrap}}$.

Figure 1 shows an example of the three types of overheads when monitoring MPI applications. The most crucial overhead is the *MPI net overhead*, which occurs if performance tools significantly delay the execution of the function calls within the main application. In scenarios where each MPI call is intercepted, the amount of time spent for storing performance data can have a huge impact on the overhead. It may also happen that buffers need to be flushed from time to time, primarily if tracing tools run out of buffer space, which can also increase the MPI net overhead. Another important overhead to consider is the *MPI hook overhead*, which refers to the additional time that is spent in `MPI_Init` to initialize the monitoring process and the time that is required to collect and write the final performance data in `MPI_Finalize`. The hook overhead is important for estimating the cost of a batch job in a queuing system, as a significant delay of the wallclock time can quickly become very costly, especially if only one core writes out the performance data while possibly thousands of cores are idling. The third overhead considered in our work is the *MPI wrap overhead*. This overhead refers to the additional time that is required to start an MPI program from a performance analysis tool. It specifically addresses tools which take the MPI application as an argument and which write the performance data after the monitored MPI application has finished its execution. The HPCToolkit [2] is such a tool that initializes itself before the MPI application starts and that writes the performance data after the application has already been completed.

3.2 MPI Performance Analysis Tools

Our study covers a number of profiling and tracing tools, although several tools are able to do both. We emphasize the fact that we are primarily interested in monitoring MPI events. For that reason, we compare the overheads introduced by performance tools while tracking MPI events, and in particular, MPI communication events, such as Send-Recv or Allreduce.

We note that we cannot cover the full feature set of all tools described herein. Most tracing tools are very feature-rich and provide a variety of methods to record performance events (e.g., function wrapping, dynamic instrumentation, stack sampling, etc.). For that reason, we described each tool in, what we believe, is the most common use case for that tool.

mpiP and IPM. Let us start with introducing the profiling tools examined in our study. mpiP [23] is one such MPI profiler, which is centered around the idea of presenting the MPI usage statistics per call site. A call site is a specific place in the program from which MPI events originate, i.e., calls to the same MPI operation but from different parts of the code can be distinguished. The profiler IPM [18] produces similar results as mpiP, as both tools report on the accumulated time spent in different MPI communication operations. However, IPM uses a rather buffer-centric presentation of the MPI statistics compared to mpiP. While mpiP presents the performance data by call site, IPM aggregates the data by buffer size, e.g., IPM distinguishes between an Allreduce on ten bytes and an Allreduce on 100 bytes.

While mpiP and IPM use the PMPI interface of MPI for recording profiling events, two other performance tools use different techniques to extract application profiles, which are Score-P [14] and HPCToolkit [2].

Score-P. It is hard to compare the functionality of Score-P to mpiP and IPM, as Score-P provides multiple methods for performance analysis. Similar to IPM and mpiP, Score-P can make use of the PMPI interface for recording timing statistics of MPI operations. However, in addition to profiling, Score-P also supports the tracing of MPI applications, where the start and the end time of each MPI event is recorded for each process. Yet, Score-P can do even more, for example, it supports the automatic instrumentation of functions. This way, it is possible to obtain fine-grained statistics of individual events. Moreover, it also supports stack sampling for generating

Table 1: MPI performance analysis tools. Tools marked with '*' are evaluated in our study.

name	ref	type	main method
PAPI	[20]	HPCs	special-purpose registers
Likwid	[21]	HPCs	special-purpose registers
*IPM	[18]	profiling	PMPI interface
*mpiP	[23]	profiling	PMPI interface
*HPCToolkit	[2]	profiling/ tracing	stack sampling
*Score-P	[14]	profiling/ tracing	PMPI interface instrumentation, sampling
*Extrae	[4]	tracing	PMPI interface instrumentation, sampling
ScalaTrace	[16]	tracing	trace compression
*Pilgrim	[24]	tracing	trace compression

performance profiles and traces. Score-P is the measurement infrastructure that lays the foundation for other performance analysis tools such as Scalasca [9], Tau [17], or Vampir [15].

Extrae. The functionality of the tracing part of Score-P is comparable to the Extrae/Paraver [5] toolchain. Paraver is a sophisticated viewer for traces that were recorded with Extrae, and Extrae supports instrumentation and sampling techniques.

HPCToolkit. HPCToolkit [2] is a performance analysis tool that supports profiling and tracing. It relies on sampling-based measurements of the current program stack and of hardware performance counters. After the data collection has been completed, HPCToolkit can generate a trace or a profile from the samples recorded for each process.

ScalaTrace and Pilgrim. Last, we mention two MPI tracing libraries that are primarily concerned with trace compression, which are ScalaTrace [16] and Pilgrim [24]. ScalaTrace distinguishes between intra- and inter-node compression. The intra-node compression is performed on the fly, i.e., each process compresses its local trace while executing the MPI program. In a second step, these profiles are merged into a global profile, while trying to identify repetitive patterns that can be exploited for further compression. The trace library Pilgrim uses a different compression method, as it builds a context free grammar during the execution of the program. To that end, it stores MPI calls and their parameters in a signature table to be used as terminal symbols in the grammar. Both libraries, ScalaTrace and Pilgrim, significantly reduce the size of recorded traces compared to traces obtained with Extrae or Score-P.

Table 1 presents an overview of the described performance analysis tools and their implemented methods. The table also marks these tools with an asterisk, whose runtime overheads are investigated in the present work.

3.3 Inspected MPI Applications

A study on the impact of performance analysis tools on MPI application clearly depends on the type of the examined applications. In our work, we wanted to build upon the studies presented by Sultana et al. [19] and by Klenk and Fröning [13]. Thus, we also focused on the ECP Proxy Applications [1]. Sultana et al. excluded Ember, miniQMC, and XSBench from their analysis, which we also did. The reason is that miniQMC and XSBench were reported to lack MPI calls for solving a computational problem, and Ember is a collection of common communication patterns. In addition, we excluded MACSio, as it is primarily targeted towards I/O performance. We also excluded Fortran-based applications such as NEKbone.

We examined the following five ECP Proxy Applications [1]:

(1) AMG (commit 3ada8a1),
(2) ExaMiniMD (commit 7a31e3b),
(3) miniAMR (commit ff07856),
(4) miniVite (commit 2324e20), and
(5) SW4Lite (commit 06b888c).

They use a mix of point-to-point and collective communication operations and do not depend on external libraries such as Laghos and SWFFT, as each external library could potentially introduce another experimental factor into our overhead study, which we wanted to avoid.

While examining the overheads of different MPI performance tools, we noticed that we should also consider a worst-case scenario for each tool, which consists of continuously calling MPI functions. For that purpose, we used our own ReproMPI benchmark [11, 12], which can provide this functionality.

Table 2: Performance analysis tools and their build details.

Software	Version	Package origin
MPICH	3.4.2	spack
Extrae	3.8.3	spack
HPCToolkit	2022.01.15	spack
IPM	02f0cdc (sha1)	GitHub
mpiP	3.5	spack
Pilgrim	e389398 (sha1)	GitHub
Score-P	7.0	spack

3.4 Hardware Configuration

We run all experiments on a 36-node cluster at TU Wien called *Hydra*. Each compute node is comprised of two 16-core Intel Xeon Gold 6130F processors, leading to 32 physical cores per node. The compute nodes are interconnect with a dual-rail Intel Omni-Path network, where each processor (socket) has its own Omni-Path interface. Profiles and traces are written to HOME directories, which are imported from an NFS server via a 10-Gbit-Ethernet link.

3.5 Selection of Performance Analysis Tools

Table 2 contains a list of all MPI performance analysis tools and their respective version examined in our study. Compared to Table 1, we omit results for PAPI, Likwid, and ScalaTrace from the experimental results. One reason is that PAPI and Likwid logically work at a different, lower level than the other tools, which often themselves use PAPI to monitor HPCs. For ScalaTrace, the situation was different. ScalaTrace worked in many scenarios, and the observed overheads were comparable to the ones of Pilgrim. Yet, ScalaTrace offers two types of trace compression, one is the node-only compression while

the other is the global compression. When enabling global compression, we should obtain a single compressed trace. Unfortunately, we had problems getting the global compression method to work, which was primarily caused by our more recent set of developer tools (gcc, libunwind). Although the node-compression worked, we decided to omit these results, as it would be less fair to compare it to tools that gather the performance data and write one final file (e.g., mpiP, Pilgrim, Extrae, etc.).

3.6 Objectives and Limitations

3.6.1 Main Focus: MPI. Our main focus of the overhead study of the various performance analysis tools is on MPI, since all of the tools support MPI. For that reason, in each tool, we only enabled the features that are needed to monitor MPI events. This is especially important for Score-P and Extrae, as they allow users to monitor many other types of functions (e.g., OpenMP or CUDA) as well.

Score-P. All proxy applications were compiled using the Score-P compiler wrapper with the `--nocompiler` and `--noopenmp` flags enabled. This way we can ensure that no other function except MPI functions will be instrumented.

Extrae. When configuring Extrae, we disabled all additional modules except MPI, e.g., OpenMP, Pthreads, hardware performance counters, etc. We use the merge option (enabled by default) to eventually create a single merged trace in the Paraver trace format.

HPCToolkit. We use the tracing option (`hpcrun -t`) when measuring the overheads of HPCToolkit. The main reason was that we almost exclusively use this option for our own research, as it provides a large amount of useful statistics. For the experiments, we have used the default sampling frequency. We have also analyzed the impact of the sampling frequency on the overhead, but we are unable to show the results in this paper due to space limitations.

Pilgrim. We comment out the `MPI_Wait` and `MPI_Test` wrapper functions in Pilgrim, as they led to unreasonable large runtimes for AMG. The proxy application AMG makes heavy use of `MPI_Test`, which seemed to cause performance issues with Pilgrim. For the other proxy applications, no noticeable difference was observed whether or not Pilgrim wrapped these functions.

3.6.2 Time Measurements. Figure 2 depicts our measurement setup. We wrap calls to `MPI_Init` and `MPI_Finalize` in our own small functions called `TIME_MPI_Init` and `TIME_MPI_Finalize`. Thus, we modify the source of each application by replacing the original MPI init and finalize functions. Inside each wrapper, we record timestamps using the POSIX function `clock_gettime`, since we cannot rely on `MPI_Wtime` because the MPI environment may not be initialized. This way, we can measure the time spent inside `MPI_Init` and `MPI_Finalize`, and we then also know how much time is spent in the main MPI application. To measure the wrap time, we record the time that is spent in an `srun/mpirun` call. This will eventually reflect the total amount of compute minutes that we have to pay for each MPI job in our batch scheduling system.

3.6.3 Input Sizes. We also had to decide on the size of the inputs given to each ECP proxy application. Strong scaling was quickly dismissed, as the running time is either too small for a larger number

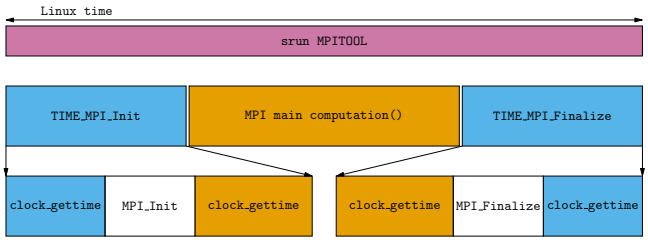

Figure 2: Time measurements for quantifying the different tool overheads (net, hook, wrap).

of processes or too large for a small number of processes. We also attempted to perform weak scaling experiments, but some tools simply did not scale well, which rendered obtaining a statistically sound number of experiments infeasible.

Eventually, we select the input sizes by the runtime of the baseline run, i.e., how long the overall execution takes without applying a performance analysis tool. In particular, we chose the parameters in such a way that the input leads to a runtime of roughly 10 s to 20 s. This allows to not only compare the tools per application, but also to compare the application characteristics for a similar runtime. The input sizes used for the experiments shown in the present article are summarized in Table 3.

3.6.4 Experimental Factors and Limitations. We are aware that our study will only capture a relatively small fraction of possible performance analysis experiments, as the experimental parameter space is huge. Our goal was to get a first overview of the individual overheads introduced by each tool for two process configurations: 1×32 and 32×32 processes. With the smaller process configuration, we test how well the tools work on one compute node only. By examining the results for 32×32 processes, we get insights in how well the tools scale.

In contrast to previous studies [7, 24], the resulting file size of profiles and trace is not one of our main concerns. Our major focus is on examining if the performance analysis tools perturb the actual MPI application, i.e., altering the net time of an application.

Our study is currently limited to one machine only. We also planned to present experimental from other machines. However, it turned out to be more problematic than we had initially expected. First, in a multi-user batch system, jobs may wait for several days in a queue before the requested number of resources is granted, which limits the number of experiments that can be done in a certain amount of time. Second, and more severe were the software compilation and building problems. We settled for only five applications and six performance tools. However, getting all of them to run on other systems was extremely time consuming. For example, we could not get Pilgrim to work with MPI libraries other than MPICH. In another case, we were unable to compile libunwind with the pre-installed compiler, which is required by several performance tools, such as mpiP. Third, several tracing tools may take very long to merge and write out performance data. These runs quickly consume the users' allocated compute hours.

Table 3: MPI applications used in this study and associated input sizes.

name	input 1 × 32 procs	input 32 × 32 procs
AMG	-problem 1 -n 80 80 80 -P 4 4 2	-problem 1 -n 80 80 80 -P 16 8 8
ExaMiniMD	run=1000; region=0 40 0 40 0 40	run=1200; region=0 120 0 120 0 120
miniAMR	npx 4 npy 4 npz 2 nx 32 ny 16 nz 16 num_tsteps 120	npx 16 npy 8 npz 8 nx 32 ny 16 nz 16 num_tsteps 120
miniVite	-n 1966208	-n 62918656
SW4Lite	pointsource.in modified	pointsource.in modified
	grid x=10 y=10 z=6 h=0.04; time t=1.0	grid x=160 y=20 z=6 h=0.04; time t=1.0
ReproMPI	MPI_Bcast, 1024 B, nrep=1000000	MPI_Bcast, 1024 B, nrep=300000

4 EXPERIMENTAL RESULTS

We present the results of our experiments on a single node and on 32 nodes. The dataset with the raw experimental results can be found at https://zenodo.org/record/6535636.

Figure 3 summarizes the experimental results obtained on one compute node, where each MPI application is run with 32 processes, each of which is mapped to a dedicated core. The figure shows the composition of the different time slots that were defined in Section 3.1. For all ECP proxy application, we can clearly observe that most of the running time is spent in the MPI main program (the net time, yellow). We can also see that Extrae spends more time writing the final performance results compared with the other tools, which can be seen for AMG and miniAMR. For ReproMPI, the time difference between Extrae and the other tools becomes significant.

In Figure 4, we examine the net overhead caused by each tool with respect to an application run without the utilization of a performance analysis tool. Therefore, the figure does not contain a bar for the baseline, as we normalize the net time measured for each tool to this baseline. Our initial hypothesis was that applying performance tools would add a noticeable overhead of roughly 5–10%. But the results shown in Figure 4 showed a completely different picture. All performance tools worked extremely well for all ECP proxy applications tested. Only, for the stress test with ReproMPI, visible overheads could be observed. We have to emphasize that, in the experiment with ReproMPI, we call `MPI_Bcast` 10^6 times and each broadcast is synchronized with an `MPI_Barrier`, leading to a total of 2×10^6 collective calls with 32 processes. It is therefore not surprising that tracing tools such as Score-P and Extrae entail an overhead of 13% and 20%, respectively, which is still very low for such a scenario.

Now, we turn to the results for 32 compute nodes, which are shown in Figure 5. We start again by looking at the contribution of each individual time. We need to point out that we were unable to obtain results for mpiP with AMG and for Extrae with ReproMPI. In the case of mpiP, it simply never finished creating the profile for AMG. In the other case with Extrae and ReproMPI, we could observe that Extrae was still trying to merge the trace files to create a single Paraver trace, but we stopped it after 45 minutes when the trace file had reached a size of 130 GB. When inspecting the results shown in Figure 5, we were not surprised to observe that tracing tools like Extrae and Score-P had spent a significant fraction of the overall runtime in writing the final trace file. In these cases, the hook and wrap times are significantly increased. A little surprising was the fact that mpiP required a comparatively long time for creating the final profiles. Notice that the hook and wrap time can heavily be impacted by the speed of the file system. Therefore, a faster file system may mitigate the relatively large writing overhead.

In Figure 6, we compare the net overheads introduced by each performance tool to the MPI applications. Notice the different scales on the y-axes for each MPI application. Similarly to the results for one compute node, all performance analysis tools performed extremely well, leading to a net overhead of less than 5% overhead for virtually all ECP proxy applications. Extrae on miniAMR and Pilgrim on AMG were the only cases that significantly exceeded the 5% overhead threshold. Overall, we had not expected to measure such small net overheads across the different performance analysis tools and proxy applications.

5 CONCLUSIONS

In this work, we have examined the composition of the running time of MPI applications when being monitored with a performance analysis tool. Our main goal was to quantify the overhead introduced by the various profiling and tracing tools inside the main application, which is the part of the application after `MPI_Init` and before `MPI_Finalize`. Performance tools should not perturb MPI applications in this time period, as the recorded performance result may otherwise just represent artifacts.

We defined and investigated different overheads that may occur in a performance analysis run of an MPI application. Our results showed that all current MPI performance analysis tools, although often having completely different objectives, entail a very small overhead in the main part of MPI applications, i.e., they do not significantly perturb these applications.

Our study of the overheads introduced by performance analysis tools is far from complete. There are many more experimental factors that can be explored, e.g., the number of processes, the number of compute nodes, or the utilization of threads at node level with OpenMP or Kokkos [22]. Finally, we need to point out a lesson learned during this study. It is extremely hard (or perhaps impossible) to get a fair comparison of the various performance analysis tools. Many tools are themselves built on a large software stack. Even small modifications of the stack may change the results and the outcome. There are many pitfalls, since most tools have a rather steep learning curve. However, in sum, all performance tools significantly exceeded our expectations in terms of overhead.

ACKNOWLEDGMENTS

We would like to thank Markus Geimer for answering our countless questions concerning Score-P.

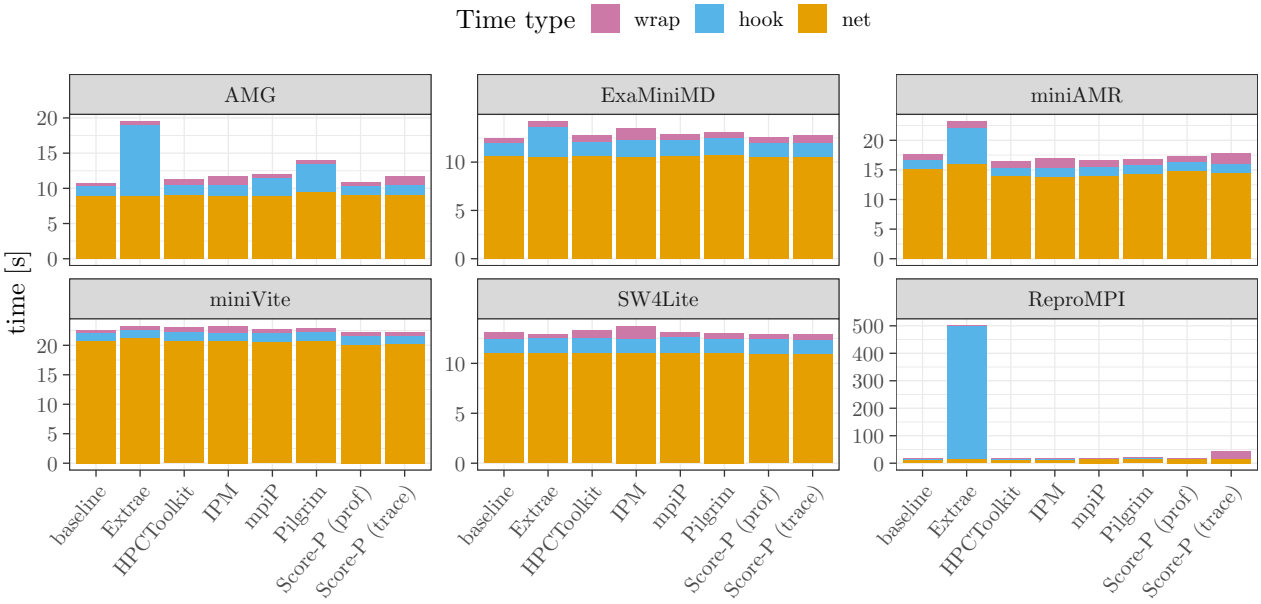

Figure 3: Composition of performance analysis times; 1×32 processes; machine: *Hydra*.

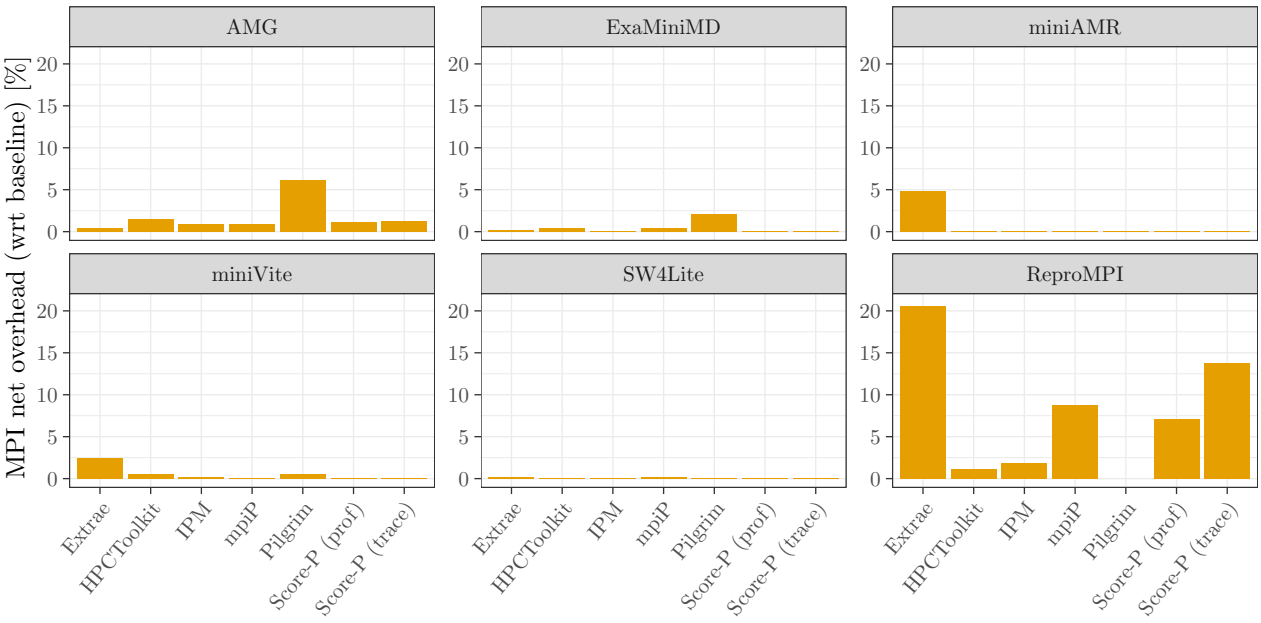

Figure 4: MPI net time overhead of performance analysis tools w.r.t. baseline execution; 1×32 processes; machine: *Hydra*.

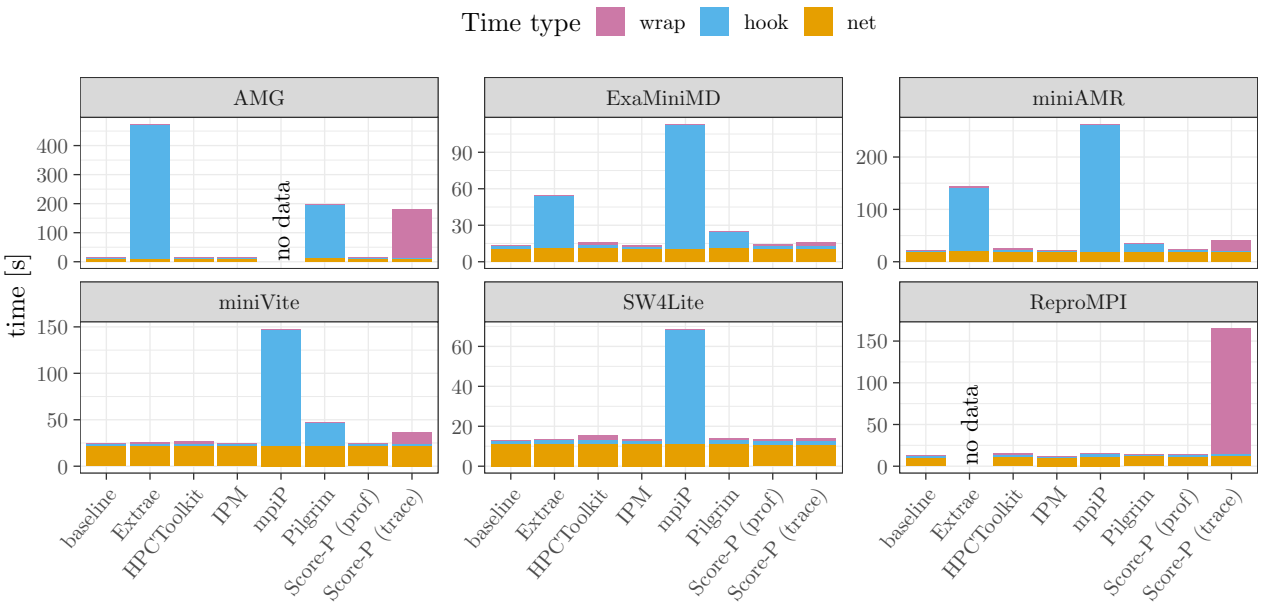

Figure 5: Composition of performance analysis times; 32×32 processes; machine: *Hydra*.

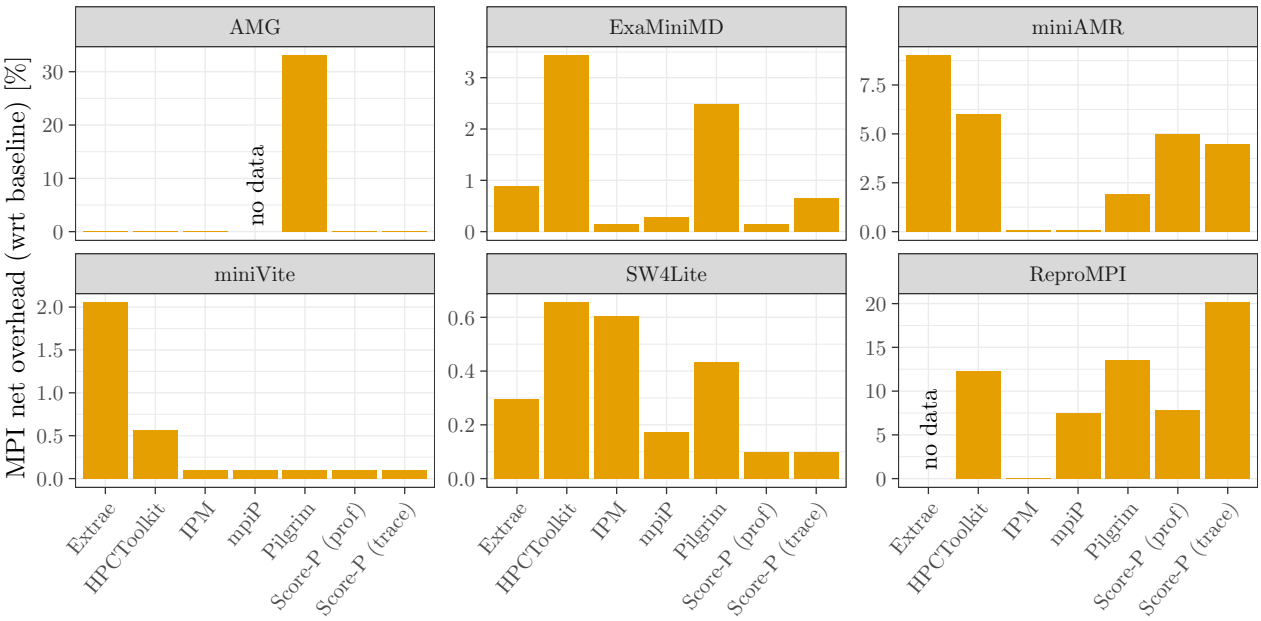

Figure 6: MPI net time overhead of performance analysis tools w.r.t. baseline execution; 32×32 processes; machine: *Hydra*.

REFERENCES

[1] 2020. Exascale Proxy Applications. https://proxyapps.exascaleproject.org/

[2] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.* 22, 6 (2010), 685–701. https://doi.org/10.1002/cpe.1553

[3] 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 Supercomputing (SC)*. IEEE Computer Society, 550–560. https://doi.org/10.1109/SC.2016.46

[4] BSC Performance Tools. 2022. Extrae. https://tools.bsc.es/extrae

[5] BSC Performance Tools. 2022. Paraver. https://tools.bsc.es/paraver

[6] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI usage on a production supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE / ACM, 30:1–30:15.

[7] I-Hsin Chung, Robert Walkup, Hui-Fang Wen, and Hao Yu. 2006. MPI tools and performance studies - MPI performance analysis tools on Blue Gene/L. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. ACM Press, 123. https://doi.org/10.1145/1188455.1188583

[8] Bengisu Elis, Dai Yang, Olga Pearce, Kathryn Mohror, and Martin Schulz. 2020. QMPI: A next generation MPI profiling interface for modern HPC platforms. *Parallel Comput.* 96 (2020), 102635. https://doi.org/10.1016/j.parco.2020.102635

[9] 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

[10] Sascha Hunold, Jordy I. Ajanohoun, and Alexandra Carpen-Amarie. 2021. MicroBench Maker: Reproduce, Reuse, Improve. In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS 2021)*. IEEE, 69–74. https://doi.org/10.1109/PMBS54543.2021.00013

[11] Sascha Hunold and Alexandra Carpen-Amarie. 2016. Reproducible MPI Benchmarking is Still Not as Easy as You Think. *IEEE Trans. Parallel Distributed Syst.* 27, 12 (2016), 3617–3630. https://doi.org/10.1109/TPDS.2016.2539167

[12] Sascha Hunold and Alexandra Carpen-Amarie. 2018. Hierarchical Clock Synchronization in MPI. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*. IEEE Computer Society, 325–336. https://doi.org/10.1109/CLUSTER.2018.00050

[13] Benjamin Klenk and Holger Fröning. 2017. An Overview of MPI Characteristics of Exascale Proxy Applications. In *Proceedings of the 32nd ISC High Performance (LNCS, Vol. 10266)*, Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David E. Keyes (Eds.). Springer, 217–236. https://doi.org/10.1007/978-3-319-58667-0_12

[14] 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 *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

[15] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. 2007. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Prceedings of the ParCo 2007 (Advances in Parallel Computing, Vol. 15)*. IOS Press, 637–644.

[16] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. 2009. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distributed Comput.* 69, 8 (2009), 696–710. https://doi.org/10.1016/j.jpdc.2008.09.001

[17] 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

[18] David Skinner. 2005. *Performance monitoring of parallel scientific applications*. Technical Report. https://doi.org/10.2172/881368

[19] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham V. Bangalore, Ignacio Laguna, and Kathryn Mohror. 2021. Understanding the use of message passing interface in exascale proxy applications. *Concurr. Comput. Pract. Exp.* 33, 14 (2021). https://doi.org/10.1002/cpe.5901

[20] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. 2009. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer, 157–173. https://doi.org/10.1007/978-3-642-11261-4_11

[21] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. LIKWID: Lightweight Performance Tools. In *Competence in High Performance Computing 2010 - Proceedings of an International Conference on Competence in High Performance Computing, Schloss Schwetzingen, Germany, June 2010*, Christian H. Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum (Eds.). Springer, 165–175. https://doi.org/10.1007/978-3-642-24025-6_14

[22] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Q. Dang, Nathan D. Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan R. Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[23] J. Vetter and C. Chambreau. 2006. mpiP: Lightweight, Scalable MPI Profiling. https://github.com/LLNL/mpiP

[24] Chen Wang, Pavan Balaji, and Marc Snir. 2021. Pilgrim: scalable and (near) lossless MPI tracing. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 52:1–52:14. https://doi.org/10.1145/3458817.3476151