Intrusiveness and Scalability of OMPT-Based Tracing Tools for Task-based OpenMP Applications

Rayan Raddatz de Matos[1], **Lucas Mello Schnorr**[1]

[1] Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{rayan.raddatz, schnorr}@inf.ufrgs.br

Abstract. *Task-based parallel programming has become popular for handling irregular parallelism in modern HPC applications. This paradigm requires tailored performance analysis tools, with the OpenMP Tools (OMPT) API being the state-of-the-art for tracing task-based execution events. However, since large-scale applications can generate enormous numbers of tasks, understanding the intrusion of OMPT callbacks and existing tracing tools is crucial. This article proposes a methodology to investigate and compare the intrusiveness and scalability of OMPT-based tracers, evaluating Score-P, Extrae, TiKKi, and custom tracers under various configurations that stress task numbers and registered events. We demonstrate that OMPT-based tracer intrusiveness varies significantly across tools, with some achieving low intrusion and good scalability while others exhibit substantial performance degradation as parallelism increases.*

1. Introduction

The constant increase in High-Performance Computing (HPC) capacity, seen in the world's most powerful machines [Dongarra and Keyes 2024], confirms the need for HPC application programmers to adopt higher levels of parallelism manifestation. Following this trend, the task-based paradigm to express such parallelism has become a popular and influential parallel programming model because of its flexibility and efficiency [Agullo et al. 2017]. It takes out of the programmer the responsibility for most load balancing and scheduling intricacies and gives such responsibility to a performance-oriented runtime. This recent trend has seen numerous runtimes emerge, such as StarPU [Augonnet et al. 2010], OpenMP tasks [Ayguadé et al. 2008], and PaRSEC [Hoque et al. 2017].

Task-based programs create dynamically scheduled tasks that execute out-of-order across threads while respecting dependencies, forming task dependency graphs. Since programmers lose direct scheduling control to the runtime system, understanding application behavior becomes challenging. To address this, programmers employ tracing tools that capture task lifecycle and thread information through manual or automatic instrumentation. Unfortunately, any type of tracing introduces computational overhead (extra CPU cycles, increased memory consumption, additional I/O operations), commonly identified as intrusion or tracing overhead [Hunold et al. 2022].

For OpenMP task-based applications, the OpenMP Tools (OMPT) API [Eichenberger et al. 2013] enables user-defined callback execution whenever a specific event occurs. Moreover, this interface appears in established tracing tools such as Score-P

[Mey et al. 2011], TiKKi [Daoudi et al. 2020], and Extrae [Llort et al. 2016] to trace and analyze OpenMP task-based programs. These tools typically implement an OMPT plugin capable of attaching to a supporting runtime like the one provided by the LLVM/Clang compiler. With an active OMPT-based plugin, the runtime registers typical events during the execution of a task-based application, such as task creation and execution. It also registers the graph of tasks created by the application. As the number of tasks grows larger for small-scale applications, it becomes important to correctly understand the intrusion caused by OMPT plugins.

This work presents a methodology and an investigation of the intrusiveness of modern OMPT-based tracing tools in stressful situations, such as when there are many tasks and worker threads. We evaluate Score-P, Extrae, TiKKi, and different in-house tracers under different configuration scenarios and compare them against a situation without tracing. Our contributions include (1) a comprehensive methodology for evaluating the intrusiveness and scalability of OMPT-based tracers in task-parallel applications, enabling systematic comparison of tracing tool overhead across different parallelism levels; (2) an empirical analysis revealing significant performance variations among established OMPT tracers (Extrae, Score-P, TiKKi), demonstrating that tool selection critically impacts application scalability in parallel environments; and (3) comparison against our in-house low-overhead tracing mechanism for OMPT capable to gathering the necessary information for Space-Time views and graph analysis.

Section 2 presents core concepts regarding task-based applications and tracing. Section 3 highlights related work and a discussion motivating our work. Section 4 discusses the experimental methodology used in this work, with details about the applications, tools, and systems used in the experiments. Lastly, Section 5 shows and discusses the experimental results, while Section 6 concludes this work.

Software and Data Availability. We endeavor to make our analysis reproducible for a better science. We made available a companion material hosted in a public GitHub repository at `https://github.com/rddtz/sscad2025-companion`. Our companion contains the source code of this article and the software necessary to handle the created datasets. We also include instructions to run the experiment and figures.

2. Background

2.1. Task-based paradigm and OpenMP

The task-based paradigm has become a popular parallel programming alternative to embrace parallelism because of its power and simplicity, making it easy to schedule complex workloads to multiple cores. The programmer defines the tasks and marks their parameters as input, output, or both (dependencies). At the same time, the runtime system is responsible for scheduling the tasks to different cores, handling data dependencies [Dongarra et al. 2017]. The general task graph is usually absent in a typical runtime system. However, we can generate an external representation of the Direct Acyclic Graph (DAG), as illustrated by Figure 1, to depict that each node represents a task (the color represent its type) and the edges represent the dependencies between tasks. This example graph is also known as a Task Dependency Graph (TDG). The runtime system can schedule tasks simultaneously, respecting the dependencies between them.

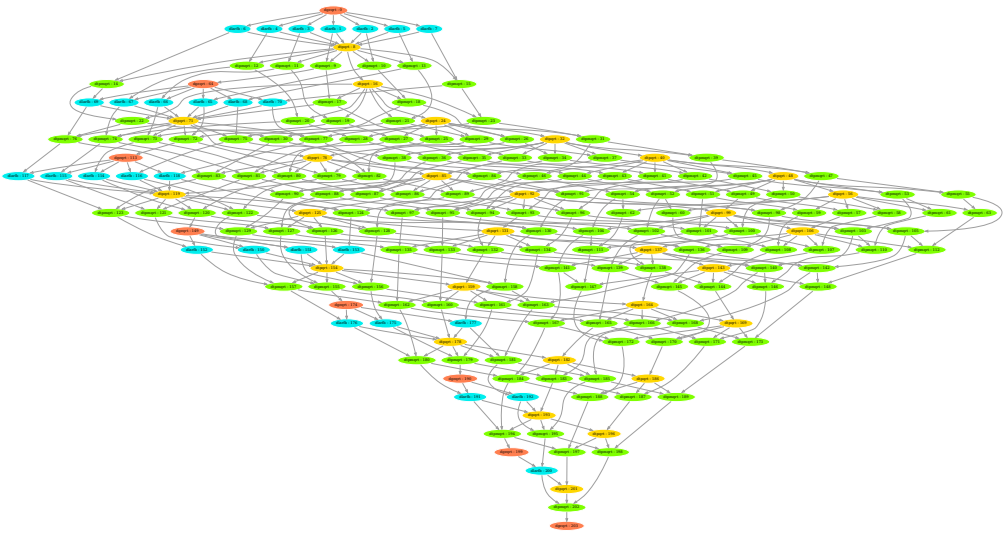

Figure 1. Task-based QR Factorization application DAG with 204 tasks.

Initially, the OpenMP supported only tasks with the `task` and `taskwait` constructs. This later changed with newer versions of OpenMP, introducing new and more complex features. For example, version 4.0 had as its main contribution the `depend` clause, which lets the programmer add additional per-task constraints in the task scheduling referring to its parameters. Using the `depend` clause, the task execution order depends solely on the satisfaction of its task dependencies, expressed by the programmer [Ayguadé et al. 2008]. This represents a fine-grain control of task dependencies, enforcing that the task only starts when all the input data it depends on is ready. Version 4.5 introduced the `priority` clause, which allows the programmer to hint to the runtime system about what tasks must be scheduled first by assigning a higher *priority* to them. In version 5.0, we see the arrival of the OpenMP Tools (OMPT) API and the `affinity` clause, which aims to let the programmer hint to the runtime system the desired location (typically a given core). Notice that the `affinity` and `priority` clauses are only hints, and currently not supported by all OpenMP runtime, even those widely available. As hints, the runtime system might not always schedule the higher priority tasks first or to their indicated locations.

2.2. Tracing task-based applications and the OpenMP Tools (OMPT) API

The understanding of parallel applications are paramount for its maintenance and improvement. Unfortunately, understanding a task-based application behavior is not something trivial. The runtime system can execute a task in any thread and can delay its executions instead of executing the task immediately, causing irregular parallelism. Fortunately, we can use tracing techniques in order to get detailed information about the application execution. *Post-mortem*, that is, after the end of the execution, this information is helpful to understand the general application behavior and identify performance issues. One can typically employ modern data science tools to derive decision-taking information.

With the need of tracing task-based programs, OpenMP 5 [OpenMP 2018] officially introduced into its specification the *OpenMP Tools Interface* (OMPT) [Eichenberger et al. 2013]. It takes form as a callback interface that enables plugins to

receive notification of OpenMP events from the runtime and then, for instance, trace OpenMP programs. The callback is just a function attached to a reference OpenMP event. Whenever the runtime reaches that specified event, it invokes the previously defined callback function. This function can then access internal task information, the start and end timestamp of tasks, the thread responsible for the task, and other valuable information. The OMPT interface not just facilitate to the programmer to write simple and straightforward tracing tools to better understand its applications, but also make possible to already established performance analysis tools to better support OpenMP programs, reducing the maintenance burden of previous methods [Feld et al. 2019]. Despite being a powerful tool to trace OpenMP programs, the OMPT support from current versions of OpenMP runtime systems GCC/libgomp and LLVM/libomp are relatively limited. The LLVM/libomp is the runtime that support most OpenMP events, such as the thread and task behavior necessary for our work. We therefore adopt LLVM/libomp for this work.

3. Related Work

With the popularization of the task-based model, different works studied task-based applications and its performance by analyzing factors that impacted its performance. [Miletto and Schnorr 2019] analyzes the influence of different runtime systems in the performance of the a task-based application, suggesting problems related to the task granularity in one of the OpenMP runtime systems. The topic of task granularity for task-based OpenMP applications is as well discussed in [Gautier et al. 2018], concentrated mainly in the LLVM/libomp implementation. Furthermore, various benchmarks were created to measure the performance of different aspects of OpenMP task-based applications. Examples include the Barcelona OpenMP Tasks Suite (BOTS) [Duran et al. 2009], created to evaluate tasking introduced in OpenMP 3.0, and KASTORS benchmark [Virouleau et al. 2014] to evaluate the `depend` clause introduced in OpenMP 4.0.

Other works focus on the understanding of parallel applications written in the task-based paradigm. For example, [Muddukrishna et al. 2015] focus on the characterization of OpenMP task-based programs using BOTS. They use as an example to demonstrate that task-based performance analysis is useful to diagnose performance problems and provides detailed behavioral explanations. [Pinto and Filho 2024] provides an workflow to integrate TiKKi [Daoudi et al. 2020], an OMPT-based tracing tool, into StarVZ [Leandro Nesi et al. 2020], an R programming language package for task-based applications performance analysis, enabling a better understand of OpenMP tasking paradigm.

This importance of understating and visualizing the performance of this parallel model arose the need for tracing and performance analysis tools that were capable of support task-based OpenMP applications and give valuable information about its execution. Before OMPT, [Schmidl et al. 2014] analyzed three different tracing tools, search for differences in the amount of data generated and suitability of these different tools. However, the creation of OMPT was fundamental to the appearance of more convoluted tracing tools that could support OpenMP tasks such as TiKKi [Daoudi et al. 2020] or ScalOMP [Daumen et al. 2019]. Traditional tools like Score-P [Feld et al. 2019] and Extrae [Llort et al. 2016] have also been ported to OMPT.

Previous work [Matos and Schnorr 2025] evaluated simple tracing tools using OMPT, concluding that OMPT intrusion is minimal when there is no tool attached and

that the task granularity is one of the major factors for intrusion. In this work, we deepen the investigation of intrusion of task-based OpenMP applications. We now evaluate in addition different and more complex tracing tools that rely on the OMPT interface, providing insights also into the quality of information observed by these different tracers.

4. Methods and Materials

The goal is to understand the intrusiveness and the scalability of representative tracing tools when varying factors. We briefly describe the representative applications, the six tracers, including our baseline, the three cluster partitions, and the Design of Experiments.

4.1. Applications: Cholesky Factorization, Gauss-Seidel and the QR Factorization

We selected three different OpenMP applications that follow the task-based paradigm. The first is a dense Cholesky factorization, with a variation to possibly consider the `priority` clause. The second is an implementation of the Gauss-Seidel method from [Nesi et al. 2021]. The third one is our implementation of the dense QR factorization following the Householder Transformation. All three applications are task-based tiled implementations employing the `depend` clause. The use of these alternative applications allow us to verify different behaviors of creating and tracing tasks. The choice for tiled implementations enable the control of the amount of tasks very easily. Both Cholesky and QR applications rely on LAPACK 3.11.0 for linear algebra operations, Cholesky also uses OpenBLAS 0.3.29. Section 4.4 presents our choices for the application's parameters used in the experiments.

4.2. Tracing Tools: Score-P, Extrae, TiKKi, Void, Printf, and OOT

We select six tracing tools with varying capabilities: Score-P, Extrae, TiKKi, Void, Printf, and OOT. We chose the first three tools (Score-P, Extrae, TiKKi) because we consider that they are popular and well established tools for tracing and analyzing the performance of OpenMP programs. We also implement three different tracing tools using the OMP Tools API (Void, Printf, and OOT) to evaluate the API performance. A short description of these tracers follows. **Score-P** is an unified performance-measurement system that serves as a common basis for other performance tools such as Periscope, Scalasca, Vampir, and TAU. Initially, Score-P used the OPARI2 [Gmbh et al. 2001] source-to-source instrumentation tool for OpenMP programs, but with the release of the OMPT, Score-P now rely on it for tracing OpenMP applications. **Extrae** is an instrumentation package from the Barcelona Supercomputing Center (BSC) who automatically instruments applications and creates Paraver trace files. OMPT can be enabled for the Extrae tool with the configuration file `extrae.xml`. **TiKKi** is a lightweight OpenMP tracing tool from Inria that captures all the necessary events to construct the application's task graph along with performance information related to these events. TiKKi trace files can be converted easily to different forms using the `ukilli` tool. **Void** is an OMPT plugin with all the necessary callbacks for tracing task-oriented events, but the content of these callback functions remain empty. We used this plugin to measure only the impact of the OMPT interface by itself. **Printf** is an OMPT plugin that traces tasks using one trace file per thread using the classical `printf` function of the C programming language to record activity, with all the buffering issues and expected associated intrusion. Finally, the **OOT** is our in-house OMPT plugin standing as *our-ompt-tracer* we develop based on the high-performance

and low-intrusion LibRastro [da Silva and de Oliveira Stein 2002] library. We expect the OOT to have the lowest overhead footprint in a functional tracer. Furthermore, we also execute the cases without any tracing tool attached – identified as an **Empty** run – so we can compare the previous mentioned tools to a normal execution of the application. As a final word, while the Score-P tool uses a compiler wrapper to instrument the application, our OMPT implementations needed manual instrumentation, and the Extrae and TiKKi tools are dynamically linked to the application so tracing can happen.

4.3. Hardware & Software configuration

We employ three partitions of the PCAD cluster at INF/UFRGS in the experiments. Table 1 specifies the hardware for each of them. We chose to test in these different hardware to see how applications and tracers behave in alternative setups.

Table 1. Hardware specification of the machines used for the experiments

Nome	CPU	RAM
Cei	2 x Intel(R) Xeon(R) Silver 4116, 2.10 GHz, 48 ths, 24 cores	96 GB DDR4
Hype	2 x Intel(R) Xeon(R) E5-2650 v3, 2.30 GHz, 40 ths, 20 cores	128 GB DDR4
Draco	2 x Intel(R) Xeon(R) E5-2640 v2, 2.00 GHz, 32 ths, 16 cores	64 GB DDR3

Environment variables define different aspects for the tracers. For Extrae and Score-P we attempt to only capture task-related OpenMP events aiming to a fair comparison with the other tracers, as these two tracers have a much wider set of capabilities. With Score-P 9.2, we appoint a total memory of 128MB through `SCOREP_TOTAL_MEMORY` and adopt a filter file to trace only OpenMP task-based constructs through `SCOREP_FILTERING_FILE`. With Extrae 4.3.0, we limit the buffer size of 10 million ($1 \cdot 10^7$) events using `EXTRAE_BUFFER_SIZE`. Like for Score-P, a filter file aims to trace only the wanted OpenMP constructs. This file is a modified version of the example filter file found in `extrae/share/example/OMP/extrae.xml` and informed through `EXTRAE_CONFIG_FILE`. As the TiKKi tracer has no official releases, we used commit `9721397c` from its main repository[1]. We also set a 10 million events buffer size for LibRastro used in the OOT tracer using `RST_BUFFER_SIZE`. These choices allow the entirety of an execution behavior be registered in-memory. At the end of an execution, tracers save the buffer in files. We ensure this to measure only the recording of the events, not the final IO cost. We use the LLVM/clang 14.0.6 compiler and the libomp runtime that comes along with it.

4.4. Design of Experiments (DoE)

In our Design of Experiments (DoE), we aim to observe the tracers behavior when varying two major main factors: the number of threads and the applications' task granularity. Table 2 shows the parameters used in the applications for the experiments and the amount of tasks created for each combination of problem and sub-problem sizes. When considering the number of threads, we execute from one thread to the maximum number of physical cores in each machine presented in Table 1. We avoid using Intel's Hyper-Threading mechanism with the corresponding logical cores as we observe a

[1] `https://gitlab.inria.fr/openmp/tikki`

higher experimental variability and small performance benefits, as our applications are mainly compute-bound. To correctly control the usage of physical cores, we set the `OMP_NUM_THREADS`, `OMP_PLACES` and `OMP_PROC_BIN` environment variables accordingly, bounding OpenMP threads to physical cores during the program execution. When considering the varying task granularity, we set different block sizes for a fixed problem size. We select the levels of this factor to established similar task and workload amount among the applications.

Table 2. The three applications, the problem size (matrix size), the block size and the corresponding number of tasks.

Application	Problem	Block	Tasks Created
QR factorization	2048	64, 128, 192, 258	11440, 1496, 385, 204
Cholesky factorization	2000	50, 80, 100, 200	11480, 2925, 1540, 220
Gauss-Seidel method	10000	250, 500, 1000, 2000	16000, 4000, 1000, 250

As consequence, our DoE comprises nine experimental sessions, representing the nine combinations of all three applications in the all three selected partitions of our platform. Each session has the following factors: tracer (seven levels), the problem size (one level, fixed, per-application), the block size (four levels, per-application), and the amount of threads (per-partition, according to node specification). These nine DoE are registered as Comma-Separated (`CSV`) files, where each line represent a configuration of factors. The order of configurations are purely random defined by `DoE.base` R package we use to create the sessions. The DoE also has 10 repetitions of each unique configuration, so we can measure performance variability in our results by calculating the mean of the 10 measurements for each configuration and a Confidence Interval of 99% assuming those measurements follow a Gaussian Distribution, something we have verified for the observations as we have kept a tight control of experimental parameters.

5. Results

Figure 2 depicts a Space-Time (ST) View for one execution of the Gauss-Seidel application in the Draco machine (top) and the QR application in the Cei machine (bottom), both using the maximum available threads for each machine and being traced with the OOT tracer. The ST shows the behavior of each thread (Y-axis) as a sequence of task execution (colored rectangles) along time (X-axis). The absence of colored rectangles typically represents idle time, as observed in the first thread, at the bottom of each facet. This idle time indicates that these threads submit tasks for execution, following the Sequential Task Flow (STF) paradigm, [Pei et al. 2022]. We can see that during the most significant part of the execution, the application fully uses all the available resources (no idle time, except for the first submitting thread), showing that the system is under stress for the whole execution. We also observe this behavior for the other sub-problem sizes for these applications, with some idle time appearing for the larger problem sizes.

For the amount of threads available, the performance of all adopted applications typically follows the strong scaling pattern for speedup and efficiency metrics. Such behavior then enable us to move forward with the analysis as all executions represent normal runs using OpenMP, where in the middle of the execution we are fully compute-bound.

We provide a results overview (Sec 5.1). We detail our results focusing on the Extrae lack of scalability (Section 5.2). Finally, we explain the differences among Void and OOT and Score-P (Section 5.3).

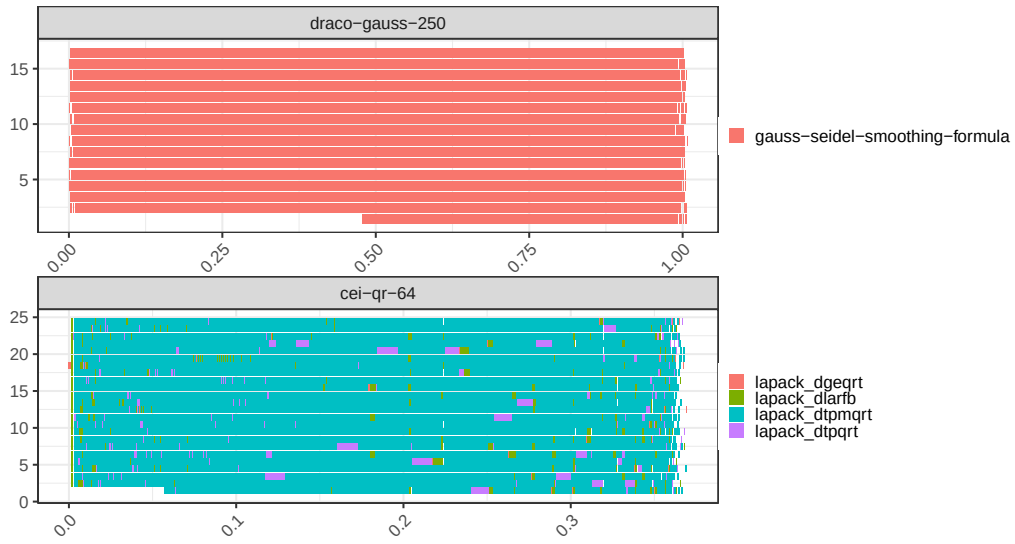

Figure 2. Space-Time View from OOT execution of Gauss-Seidel and QR applications, with sub-problems 250 and 64 respectively.

5.1. Overview of the tracing intrusion

We define a tracer intrusion as the difference between the time taken by executing an application with the enabled tracer tool and the empty execution of this same application and input. This metric ultimately is the mean intrusion time of using the tracing tool. Figure 3 shows, for each different tracer (X-axis), the mean intrusion time (Y-axis) for all executions (different block sizes) for the two applications (facets) and the respective error bars depicting the variability of the measurements. The Extrae and TiKKi tools have a higher intrusion when compared with the other tracing tools, while void and OOT have the least intrusion. Another observation is the enormous variability for the Gauss-Seidel application in some cases, especially in the Draco machine. In a few cases, the mean results for some of the tracers were lower than the mean for the Empty version execution. As one expects an actual tracing to cost more than no job done, we believe the observation happens due to very cheap tracing operations for some configurations suggestions an enormous experimental budget to clarify. A similar observation has also appeared in other independent work [Pinto and Filho 2024].

Figure 4 shows the intrusion for the Gauss-Seidel application when running with the maximum number of threads for each machine, separated per sub-problem sizes (facets). We can see that the Draco machine (green) has a higher variability, increasing when heading towards higher sub-problem sizes (lower parallelism scenarios). A similar behavior can be seen in the QR application with the 192 sub-problem size (not shown). We believe this is due to a combination of the single type of task of the Gauss implementation (see Figure 2) and Draco being the oldest hardware our testbed. Finally, the smallest sub-problem size demonstrates a higher intrusion for almost all the tracers, as we detail next.

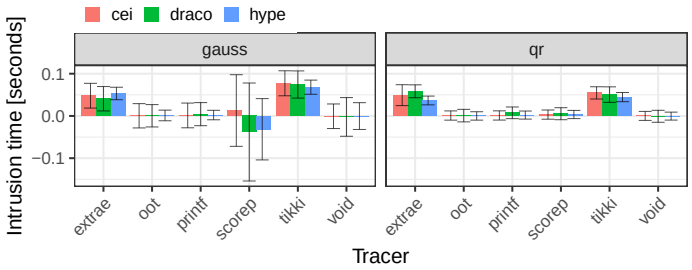

Figure 3. Mean intrusion for all configurations, by tracer and application.

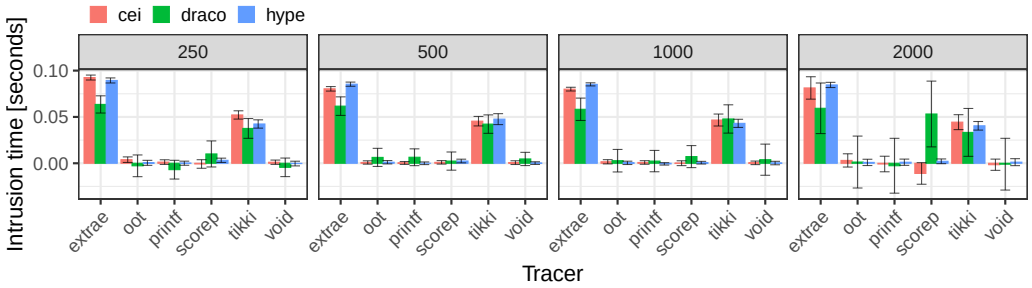

Figure 4. Mean intrusion for Gauss-Seidel application using the maximum available thread in each machine.

5.2. Detailed analysis and Extrae lack of scalability

As previously stated, the Extrae and TiKKi tools have a much higher intrusion when compared to other tools (see, for example, the right facet of Figure 3). Instead of depicting only the mean intrusion time across all block sizes for each tracer, we focus now on the measurements of the QR application. A similar behavior also appears for the other applications, especially the Gauss-Seidel. Figure 5 depicts an alternative and more detailed view. On each facet, we depict the intrusion time (Y-axis) as a function of the number of threads (X-axis) and machine (colors) to analyze the scalability of the tracing mechanism when a larger number of threads is present. We depict our results for all tracers (horizontal facetting) as a function of the block size (vertical facetting). Interestingly, we can now see that for Extrae and TiKKi, the intrusion is larger no matter the block size (more or less parallelism). We also observe that for TiKKi, Score-P, OOT, Printf and Void, as the number of threads increases, the intrusion time remains relatively stable across all machines. Even more interestingly, we can see that Extrae behaves very differently as the intrusion time increases as a function of the number of threads, demonstrating a lack of scalability probably related to the increasing amount of events and details. Extrae created the largest trace files out of every tracer, despite our efforts to control the tracing mechanism to gather similar data across all tracers.

5.3. Differences among Void and OOT and Score-P

As expected, the Void was the tracer with the least intrusion among the tested tools because it only quantifies the low cost of the OMPT interface and callbacks by itself. The OOT comes in second as the tracer has the lowest cost, as this tool only traces minimal information to create representative ST views and the task graph, such as task creation, task dependencies, and task completeness. By uniting this straightforward tracing of only

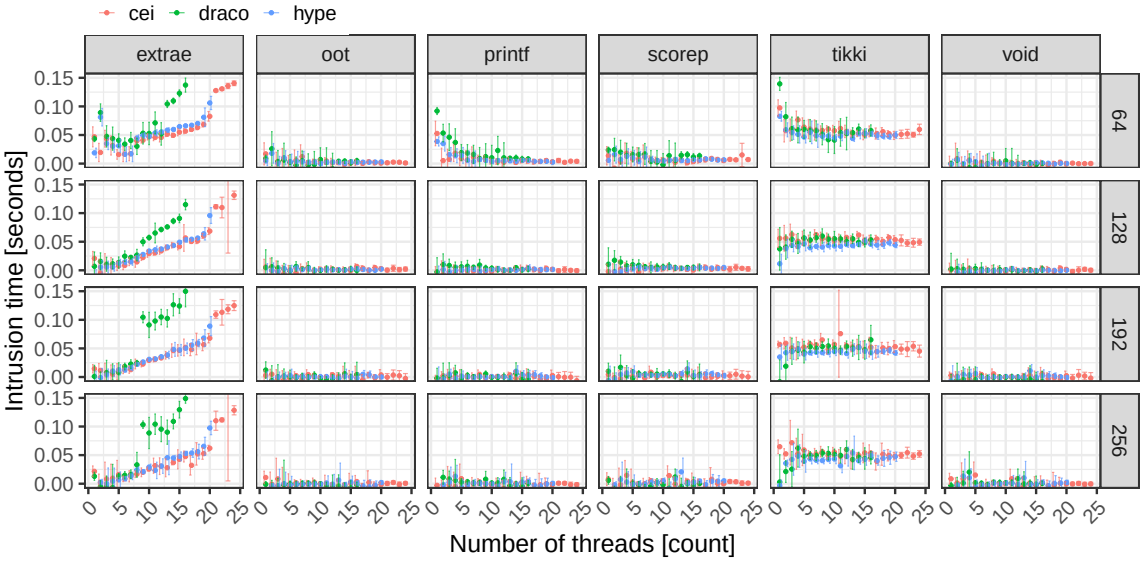

Figure 5. Mean intrusion for the QR application on the three machines separated by threads, sub-problem size and tracer.

desired events and the tracing using the LibRastro, we could achieve a simple, low-cost tracing for OpenMP programs. The results for the Score-P and Printf tracers are similar. While Printf works similarly to OOT by tracing minimal information, we abstain from using a dedicated library to trace events, relying on the Operating System's capability of I/O buffering for recording the data. This buffering difference explains the slight increase in the intrusion for the Printf tracer. Unlike our tools, Score-P is a more generic event-based tracing tool that supports different programming models, automatically instruments the application code, and has numerous options. The results for Score-P depict a low intrusion for the application, with an intrusion comparable to that of the Printf, while gathering much more performance data.

6. Conclusion

In this work, we propose a methodology to investigate and compare the intrusiveness and scalability of OMPT-based tracers for task-based programming. With this methodology, we compared established tools and also our own implemented OMPT tracers. For our tracers, we verified the low intrusiveness from the OOT tracer, showing promising results for all applications, and confirmed the low impact of the OMPT interface through the lack of intrusion for the Void tracer. For the established tools tested, we observed scalability problems for the Extrae tool for all applications, which rapidly increased the intrusion as the parallelism grew. We verified an inverse result for the Score-P tracer, which also showed promising results for intrusion and scalability in the tested applications. The TiKKi tool remains a mid-term solution, showing a considerably high intrusion in some cases, but a scalable solution as the number of threads increases. As future work, we intend to investigate further the extreme scalability effects on larger processors such as the NVIDIA Grace and Blackwell CPU Superchips.

Acknowledgments. We would like to thank the PCAD at INF/UFRGS for making infrastructure and hardware used in the experiments available. We also acknowledge the Brazilian National Council for Scientific Technological Development (CNPq) for their

financial support through the PIBIC-UFRGS scholarship. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, the FAPERGS (16/354-8, 16/348-8), Petrobras (2020/00182-5).

References

Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. P. (2017). Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Trans. on Paral. and Distrib. Syst.*

Augonnet, C., Thibault, S., and Namyst, R. (2010). *StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines*. PhD thesis, INRIA.

Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2008). The design of openmp tasks. *IEEE Transactions on Parallel and Distributed systems*, 20(3):404–418.

da Silva, G. J. and de Oliveira Stein, B. (2002). Uma biblioteca genérica de geração de rastros de execução para visualização de programas. In *Anais do I Simpósio de Informática da Região Centro*.

Daoudi, I., Virouleau, P., Gautier, T., Thibault, S., and Aumage, O. (2020). somp: Simulating openmp task-based applications with numa effects. In *The 16th Intl. Workshop on OpenMP*, page 197–211, Berlin, Heidelberg. Springer-Verlag.

Daumen, A., Carribault, P., Trahay, F., and Thomas, G. (2019). Scalomp: Analyzing the scalability of openmp applications. In *OpenMP: Conquering the Full Hardware Spectrum*, pages 36–49, Cham. Springer International Publishing.

Dongarra, J. and Keyes, D. (2024). The co-evolution of computational physics and high-performance computing. *Nature Reviews Physics*, 6(10):621–627.

Dongarra, J., Tomov, S., Luszczek, P., Kurzak, J., Gates, M., Yamazaki, I., Anzt, H., Haidar, A., and Abdelfattah, A. (2017). With extreme computing, the rules have changed. *Computing in Science & Engineering*, 19(3):52–62.

Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. (2009). Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 International Conference on Parallel Processing*, pages 124–131.

Eichenberger, A. E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., and Lorenz, D. (2013). Ompt: An openmp tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*, Berlin, Heidelberg. Springer Berlin Heidelberg.

Feld, C., Convent, S., Hermanns, M.-A., Protze, J., Geimer, M., and Mohr, B. (2019). Score-p and ompt: navigating the perils of callback-driven parallel runtime introspection. In *International Workshop on OpenMP*, pages 21–35. Springer.

Gautier, T., Pérez, C., and Richard, J. (2018). On the Impact of OpenMP Task Granularity. In *The 14th Intl. Workshop on OpenMP for Evolving Arch.*, pages 205–221. Springer.

Gmbh, F., Bericht, I., Malony, A., Shende, S., and Mohr, B. (2001). Design and prototype of a performance tool interface for openmp. *Journal of Supercomputing*, 23.

Hoque, R., Herault, T., Bosilca, G., and Dongarra, J. (2017). Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8.

Hunold, S., Ajanohoun, J. I., Vardas, I., and Träff, J. L. (2022). An overhead analysis of mpi profiling and tracing tools. In *Proceedings of the 2nd Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn Strategy*, pages 5–13.

Leandro Nesi, L., Garcia Pinto, V., Cogo Miletto, M., and Schnorr, L. M. (2020). StarVZ: Performance Analysis of Task-Based Parallel Applications. preprint at `https://inria.hal.science/hal-02960848`.

Llort, G., Filgueras, A., Jiménez-González, D., Servat, H., Teruel, X., Mercadal, E., Álvarez, C., Giménez, J., Martorell, X., Ayguadé, E., et al. (2016). The secrets of the accelerators unveiled: Tracing heterogeneous executions through ompt. In *International Workshop on OpenMP*, pages 217–236. Springer.

Matos, R. and Schnorr, L. (2025). Quantificando o impacto do rastreamento em aplicações paralelas openmp baseadas em tarefas. In *Anais da XXV Escola Regional de Alto Desempenho da Região Sul*, pages 109–112, Porto Alegre, RS, Brasil. SBC.

Mey, D. A., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A., Nagel, W. E., et al. (2011). Score-p: A unified performance measurement system for petascale applications. In *Proceedings of an Intl. Conf. on Competence in High Performance Comp.*, pages 85–97. Springer.

Miletto, M. and Schnorr, L. (2019). Openmp and starpu abreast: the impact of runtime in task-based block qr factorization performance. In *Anais do Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)*, pages 25–36.

Muddukrishna, A., Jonsson, P. A., and Brorsson, M. (2015). Characterizing task-based openmp programs. *PLOS ONE*, 10(4):1–29.

Nesi, L. L., Miletto, M., Pinto, V., and Schnorr, L. (2021). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*, chapter Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks, page 131–152. SBC.

OpenMP (2018). OpenMP application program interface version 5.0.

Pei, Y., Bosilca, G., and Dongarra, J. (2022). Sequential task flow runtime model improvements and limitations. In *IEEE/ACM Intl. Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–8. IEEE.

Pinto, V. and Filho, C. S. (2024). Improving performance visualization of openmp task-based applications. In *Anais do XXV Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 156–167, Porto Alegre, RS, Brasil. SBC.

Schmidl, D., Terboven, C., an Mey, D., and Müller, M. S. (2014). Suitability of performance tools for openmp task-parallel programs. In *Intl. Workshop on Par. Tools for HPC*, pages 25–37. Springer.

Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., and Gautier, T. (2014). Evaluation of openmp dependent tasks with the kastors benchmark suite. In *International Workshop on OpenMP*, pages 16–29. Springer.