Improving performance visualization of OpenMP task-based applications

Vinícius Garcia Pinto, Christian Einhardt Sousa Filho

[1] Federal University of Rio Grande – FURG
Rio Grande, Brazil

vinicius.pinto@furg.br

Abstract. *OpenMP is becoming a more powerful environment for exploiting task-based parallelism. Recent specification versions add support for new task clauses, while the OMPT interface provides a standard API for performance monitoring. In this paper, we present a workflow to improve the performance visualization of OpenMP task-based applications. We rely on open-source solutions such as the Tikki OMPT tracing tool and the StarVZ performance analysis framework to create enriched space-time views. We demonstrate this workflow with three applications: Strassen matrix multiply, SparseLU factorization, and a dense Cholesky factorization. For two of them, our strategy enables a better understating of the performance impact of the OpenMP* `task depend`*,* `task wait`*, and* `priority` *constructions.*

1. Introduction

Task-based has emerged as a powerful programming model that efficiently exploits multi and many-core platforms [Dongarra et al. 2017]. Several parallel programming libraries [Duran et al. 2011, Augonnet et al. 2011, Gautier et al. 2013, Bosilca et al. 2013] support it with a wide scope of hardware configurations. While such libraries keep pace with the state of the art, their particular APIs are unfriendly or unknown to general users. On the other side, the OpenMP specification remains the first approach to parallelize applications on shared-memory platforms.

Best known for its widely used `parallel for` constructions, recent specifications of the OpenMP standard have progressively improved the support for task-based applications. Since such task features are less known, programmers can underestimate the influence of some of them on the overall performance.

Previous work [Garcia Pinto et al. 2018, Pinto et al. 2021] has demonstrated how task-oriented performance visualizations can be useful to better understand and improve the performance of task-based applications running on top of the StarPU runtime system. At the same time, the OpenMP specification introduced OMPT, a new native API for monitoring the execution of OpenMP programs. This new interface made the emergence of new tracing tools possible, such as Tikki [Daoudi et al. 2020].

With the popularization of the task-based model, classical parallel applications such as Computational Fluid Dynamics (CFD), dense and sparse linear algebra, sorting, and bioinformatics algorithms were ported into task-based implementations. Benchmark suites such as BOTS, Kastors, and `omp-tdb` provide implementations of some common

applications that illustrate the flexibility of the task-based model. BOTS [Duran et al. 2009] applications exploit the initial features of the OpenMP tasking model, such as task creation, tiedness, and cut-off mechanisms. Kastors [Virouleau et al. 2014] benchmarks focus on the data dependency support introduced at OpenMP 4. The `omp-tdb` suite [Schuchart et al. 2017] provides a sort of microbenchmark to assess the overhead associated with task creation and data dependency management.

In this paper, we present a performance analysis of three OpenMP task-based applications obtained from public sources. Our goal is to illustrate how performance visualization can be useful in understanding the performance gaps between similar codes. We selected two cases from the Kastors suite to compare the performance of codes using the `task depend` clause against the `taskwait` variation. We also employ a tiled dense Cholesky factorization to assess the performance influence of the `priority` clause.

Our main contributions are a visual performance analysis to help explain the performance gains when using task `depend` and `priority` clauses; an analysis of the overhead introduced by Tikki tracing; a workflow to enable the use of traces from the Tikki tracing tool into the StarVZ framework and Spack [Gamblin et al. 2015] packages to make the installation of Tikki and Kastors easier on user-space environments.

The paper is structured as follows. Section 2 presents the background, providing an overview of the OpenMP tasking model, the OMPT interface, and the performance analysis of Task-based applications. Section 3 presents our workflow for integrating the StarVZ visualization and Tikki tracing tools. Section 4 demonstrates how our workflow helps to explain the performance variations when employing different OpenMP tasking features. Section 5 discusses Related Work. Finally, Section 6 concludes with the main results and details our future work. A publicly available companion at `https://gitlab.com/viniciusvgp/sscad2024.git` includes the experiments data, Spack packages, and the source code to integrate Tikki and StarVZ.

2. Background

2.1. OpenMP tasking model

OpenMP [OpenMP 2021] is a standard for multithread parallel programming implemented by several compilers such as GCC and LLVM/CLANG. It is a widely used tool to enable the parallelization of C, C++, or Fortran programs. It started as a set of compiler directives to distribute loop iterations among threads. Aside from the loop directives, it includes additional features to make variables private or shared among threads and to define synchronization points and mutual exclusion regions.

In OpenMP programs, the `omp task` directive defines the next structured-block as a task. In addition to the code statements in the block, the task also includes a data environment with the (private) variables declared inside the block, shared variables inherited from a wider scope or task local copies of variables declared elsewhere.

The task execution can start immediately when an `omp task` directive is reached or can be delayed for later execution. In recursive codes, the additional clauses `if` and `final` are useful to limit the spawn of new tasks. All created tasks are independent and may be executed in any order by any thread. When some tasks should be completed before others can start, the programmer should define task barriers at proper places with the `omp`

`taskwait` directive in a fork-join design inspired by Cilk [Blumofe et al. 1996]. From OpenMP version 4.0, the `depend` clause enables the definition of dependent tasks by specifying the access mode of a task parameter. This way, the programmer can define that a given task cannot start before another has produced a piece of data. The idea is to replace most "global" synchronizations using `taskwait` with fine-grained ones inferred from data access mode.

Recent OpenMP versions include other task features, such as clauses `priority` (v4.5) and `affinity` (v5.0). The first one is a way to provide scheduling hints to the runtime library. Tasks tagged with a higher priority value may be executed first. While this cannot be used to force an execution order, it is useful to try to release critical application tasks earlier. The `affinity` clause is also a scheduling hint. It can be beneficial to consider data location when placing tasks (e.g., on NUMA architectures). Both GCC and LLVM/CLANG claim to fully support OpenMP version 4.5, while version 5.0 features are only partially supported.

2.2. OpenMP Tools Interface (OMPT)

OMPT is an interface introduced at OpenMP 5 to enable monitoring and performance analysis of OpenMP programs. It consists of a set of callbacks to track the beginning and the end of threads, parallel regions, tasks, and synchronization regions. OMPT runs within the runtime library and can be loaded when launching previously compiled binaries.

Tracking the behavior of an OpenMP program with OMPT boils down to providing some code that will be executed each time the corresponding callback is activated. In `C`, this code can be as straightforward as a set of `printf` calls to register a timestamp and some information about the event caught, e.g., event type and thread ID [Miletto and Schnorr 2019, Nesi et al. 2021]. More sophisticated approaches rely on integrating OMPT facilities into existing tracing tools, as Extrae [Llort et al. 2016], Score-P [Feld et al. 2019], libKOMP/Tikki [Daoudi et al. 2020], and others [Agrawal et al. 2018, Daumen et al. 2019, Pinho et al. 2020].

An important limitation is that the OMPT interface is not supported by the GNU Offloading and Multi-Processing library (`libgomp`), which is the default OpenMP runtime library for GCC-compiled programs. Currently, the use of OMPT is essentially restricted to CLANG-compiled programs using the LLVM runtime library `libomp`.

2.3. Performance analysis of Task-based applications

Current task-based frameworks support dataflow scheduling, providing fine-grained synchronizations that reduce resource idleness and improve performance. Classical global synchronization instructions (e.g., `barriers`), placed by programmers to ensure all previous operations completed at all resources are replaced by task' parameters access-mode annotations. At runtime, the library can dynamically analyze the parameters' access mode, considering the sequential creation of tasks. A task is delayed only if at least one of its input parameters is not ready.

Several parallel computing research tools proved the effectiveness and the flexibility of this model on multicore, heterogeneous CPU+GPU and multi-node platforms [Duran et al. 2011, Augonnet et al. 2011, Gautier et al. 2013, Bosilca et al. 2013]. Starting from version 4.0, the OpenMP standard [OpenMP 2021] also supports such depen-

dency structures by adding the `depend in|out|inout` clause to constrain the scheduling of the tasks specified with the `omp task` clause introduced in version 3.0.

The lack of global synchronizations favors the performance but also makes the application behavior more irregular. Most traditional performance analysis tools focus on the regular duration of the application phases delimited by two global synchronizations. Since task-based applications do not necessarily present such phases, performance analysis should focus on different aspects, such as outlier tasks, resource idleness, and analysis of the dependency chain of delayed tasks. Previous works [Pinto et al. 2016, Garcia Pinto et al. 2018, Pinto et al. 2021] presented a set of performance analysis strategies designed for task-based parallel programs that run on top of the StarPU runtime system [Augonnet et al. 2011]. Such strategies are now available at StarVZ tool that is distributed as an R package at CRAN[1].

3. Integrating Tikki and StarVZ

The Tikki tracing tool encompasses the library implementing a set of OMPT callbacks and a set of utility commands to process the generated traces. Following the OMPT design, the library is coupled to the previously built binary of an OpenMP program through dynamic library loading (e.g., with `LD_LIBRARY_PATH`). When executing the application, Tikki generates a series of event files to be analyzed *post-mortem*. Such files can be converted to different formats (e.g., `dot`, `csv`, `rec`, `paje`) to represent different execution details using the `ukilli` command.

The StarVZ performance analysis tool was designed to analyze traces produced by the StarPU runtime system. By itself, it does not include any trace collection facilities, relying on the StarPU tracing support. This way, StarVZ is not able to analyze pure OpenMP applications directly. To enable the analysis of OpenMP programs on StarVZ, we rewrote some preliminary steps originally used to convert StarPU traces in the FxT format into StarVZ internal tables. We work on replacing some of these steps to integrate Tikki's collected data into StarVZ's expected tables. Most collected data are equivalent, but some of them need conversion, renaming, and joining. We perform these steps in `bash` code using standard UNIX tools such as `grep`, `sed`, `gzip`, `recutils` and packages `readr`, `dplyr`, `tidyr`, and `tibble` from `R`. The remaining integration steps are done with commands from the StarVZ tool itself, which is also available as an `R` package. After that, the analyst can rerun the analysis many times, recycling previously generated tables.

Figure 1 depicts our workflow with eight steps, starting from the OpenMP application (label 1) until the final space-time visual representation (8). The application is launched loading the Tikki library (2) to instrument the binary and produce a set of `evt` files (3). Ukilli tool (4) converts them to a set of text files (5). We designed scripts (6) in shell and R to convert such files into StarVZ expected ones. The final plot (8) is generated using native StarVZ commands (7).

4. Case Studies

We select three applications using different features of the OpenMP tasking model. To assess the impact of such features on the application performance, we first report the

[1] `https://cran.r-project.org/package=starvz`

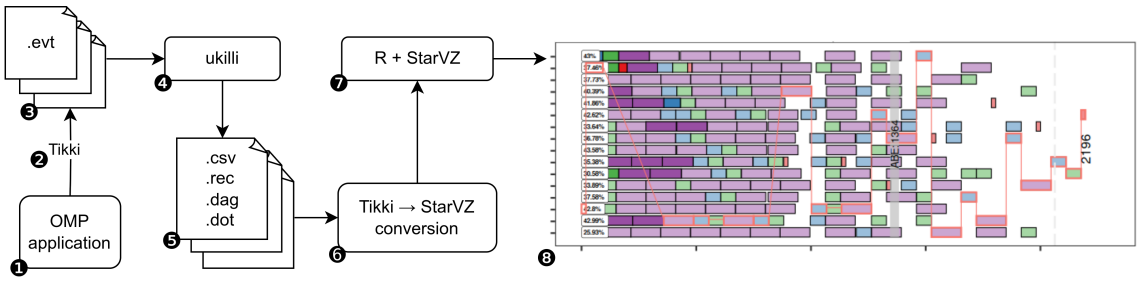

Figure 1. Workflow of Tikki and StarVZ integration.

makespans and then check if our performance visualization strategy helps explain the performance gaps.

4.1. Experimental Methodology

We selected three applications to demonstrate our strategy to provide support for performance analysis of OpenMP task-based applications on top of the StarVZ framework. Two applications (Strassen and SparseLU) came from the KaStORS benchmark suite [Virouleau et al. 2014], which is publicly available[2]. KaStORS provides two parallel versions for each application, one using independent tasks with simple `omp task` and `omp taskwait` directives and another with dependent tasks specified with `depend` clauses. The last application implements a dense Cholesky factorization using a blocked algorithm inspired by [Buttari et al. 2009]. This application is publicly available[3] and provides two variations, one with dependent tasks specified with `depend` clauses and another where such tasks include the `priority` clause. We repeat the execution of each application variation 15 times. As we intend to analyze the impact of different OpenMP tasking features on parallel executions, all our executions are multithreaded. For serial performance or scalability aspects, please refer to the original works [Virouleau et al. 2014, Nesi et al. 2021] from which we retrieve the benchmarks.

4.2. SW/HW Configuration

The experiments were executed on multi-core machines equipped with two Intel Xeon E5-2640 v3 processors, comprising 16 physical cores and 32 hardware threads. Each node has 64 GB of DDR4 RAM memory and runs Linux CentOS 7.3. Applications were built with clang 18.1.7, libomp 18.1.7 and Netlib Lapack 3.9.0. Tikki OMPT tool was compiled from `master` branch (`9721397c`), Kastors benchmark suite from `develop` branch (`aba7a004`) and Cholesky benchmark from `master` branch (`f256cf25`).

4.3. Tikki overhead

In this case, we compare the performance running with Tikki tracing enabled and without tracing. The last one is referred to as `original` and the other as `tikki`. To enable Tikki tracing, one must load a dynamic library when launching the application execution. However, here, we also changed the application code to include human-friendly task names by using the non-standard `ompt_set_task_name` extension provided by Tikki.

[2] `https://gitlab.inria.fr/openmp/kastors`
[3] `https://gitlab.com/lnesi/companion-minicurso-openmp-tasks/`

Table 1 summarizes the overhead introduced by the tracing for the three applications. For Cholesky and Strassen cases, the overhead is less than 1%. In some cases, the median of trace-enabled executions was even smaller than the pure one. We believe that the natural variation in the executions can explain this. A different situation was observed with the SparseLU application, where tracing the `task` version adds $\approx 3.9\%$ to the median makespan.

Table 1. Performance overhead of Tracing with Tikki. Median of 15 executions using 16 threads. For the Original case, the Interquartile Range (IQR) is shown in parentheses. For the IQR of Tikki cases, see Tables 2 and 3.

Application	`original` (s)	with `tikki` (s)	difference ($\%$)
Cholesky (prio `none`)	2.19676 (0.00596)	2.20876	0.546167
Cholesky (prio `correct`)	2.06697 (0.02906)	2.07414	0.347078
Cholesky (prio `wrong`)	2.2554 (0.03221)	2.24098	-0.639221
Strassen (`task`)	5.51985 (0.23744)	5.46932	-0.915406
Strassen (`taskdep`)	5.06152 (0.09609)	5.04346	-0.356691
SparseLU (`task`)	2.66188 (0.00343)	2.76444	3.852990
SparseLU (`taskdep`)	2.72660 (0.00413)	2.72992	0.121543

4.4. Influence of `taskdepend` clause

The `taskdepend` clause allows fine-grained synchronizations by replacing most of the `taskwait` barriers with strict dependencies inferred from the parameters access mode. We used the two applications from the Kastors Benchmark suite. Each one has two versions; the first uses simple `omp task` and `omp taskwait` directives in the OpenMP 3 style, while the second adds the `taskdepend` clause from OpenMP 4.

Table 2 summarizes results for the Strassen matrix multiplication using matrices of 8192×8192 with a Cutoff value of 64 and for the SparseLU factorization using matrices of 128×128 with sub-matrices of 64×64. For the Strassen application, the `taskdep` version reduces the median makespan by $\approx 7.8\%$. The interquartile range (IQR) shows that both cases present similar variability in the observed values.

To go deeper into the analysis, we used our workflow to convert Tikki traces into StarVZ-compatible format and then build StarVZ space-time visualizations. StarVZ space-time plots are similar to traditional Gantt charts commonly used in performance analysis. In such plots, the vertical dimension presents the list of computing resources (CPU cores or *threads*), while the application tasks appear horizontally over time in different colors for each type. StarVZ offers enriched space-time views with a series of improvements such as idleness quantification, highlight of important task dependencies, computation of theoretical bounds for the makespan, outlier highlighting, and task-aware temporal aggregation. The two space-time views of Figure 2 highlight the difference between both cases. During the most part of the execution time, the two versions perform similarly. However, in the beginning (see zoom area), the fine-grained synchronizations of the `taskdepend` version increase resource usage. The idleness quantification labels on the left show that the overall idleness ratio drops by half. However, the taskdep version does not bring significant performance gains for the SparseLU application. Moreover, as shown in Table 1, Tikki tracing adds a considerable overhead on the performance of the `task` version but not on the `taskdep` one, which prevents a fair analysis.

Table 2. Performance influence of `taskdepend` clause. Median of 15 executions using 16 threads with Tikki tracing enabled.

Application	task clause	Median (s)	IQR (s)
Strassen	`task`	5.46932	0.109190
Strassen	`taskdepend`	5.04346	0.111966
SparseLU	`task`	2.76444	0.124555
SparseLU	`taskdepend`	2.72992	0.033195

4.5. Influence of `priority` clause

The `priority` clause allows developers to provide scheduling hints on which tasks should be executed sooner. This feature is fully supported on the runtime libraries of both GNU/GCC and LLVM/CLANG. As a hint, the runtime libraries are not obligated to strictly respect such values and even totally ignore them, which is valid from the OpenMP specification.

In this case study, we select a dense Cholesky factorization following the PLASMA design of tiled algorithms. This implementation is available in the companion material of [Nesi et al. 2021] and includes two versions, one without priorities and a second one with priority values inspired by the Chameleon linear algebra library. The Chameleon library gives higher values to tasks releasing a higher number of dependencies, i.e., tasks with more direct or indirect children in the DAG, e.g., `potrf` ones. Additionally, we prepare a third version with inverted priorities. That is, the `potrf` tasks received a lower priority than any other task. Hereafter, we refer to these three versions as `none`, `correct`, `wrong`. The goal here is to check how much such a clause influences the application's performance. Table 3 summarizes the performance of these three versions running the factorization over matrices of size 5000×5000 elements with tiles of 500×500. The median of 15 executions shows that `wrong` version presents the worst performance.

We consider `none` version, i.e., without any priority clause, as the baseline since, at an initial approach, one would probably start parallelizing the application using just a basic `omp task` directive. Introducing the `priority` clause with the `correct` values reduces the median by $\approx 6.09\%$ over the `none` case. On the other hand, a hypothetical mistake by assigning inverted values (`wrong` version) to the `priority` clause would slow down the performance, increasing the median time by $\approx 1.45\%$. Moreover, providing the correct priority values not only reduces the median but also significantly reduces the variation in the observed makespan. This reduction in the variation can be confirmed by checking the interquartile range (IQR), which is $5\times$ smaller in the `correct` version.

Table 3. Performance influence of `priority` clause using the Cholesky factorization. Median of 15 executions using 16 threads with Tikki tracing enabled.

Priority clause	Median (s)	IQR (s)
`wrong`	2.24098	0.0583625
`none`	2.20876	0.0532135
`correct`	2.07414	0.0106255

Such summarized statistical values give us a notion of how the scheduling hints provided by the programmer can harm or improve the application's performance. To help

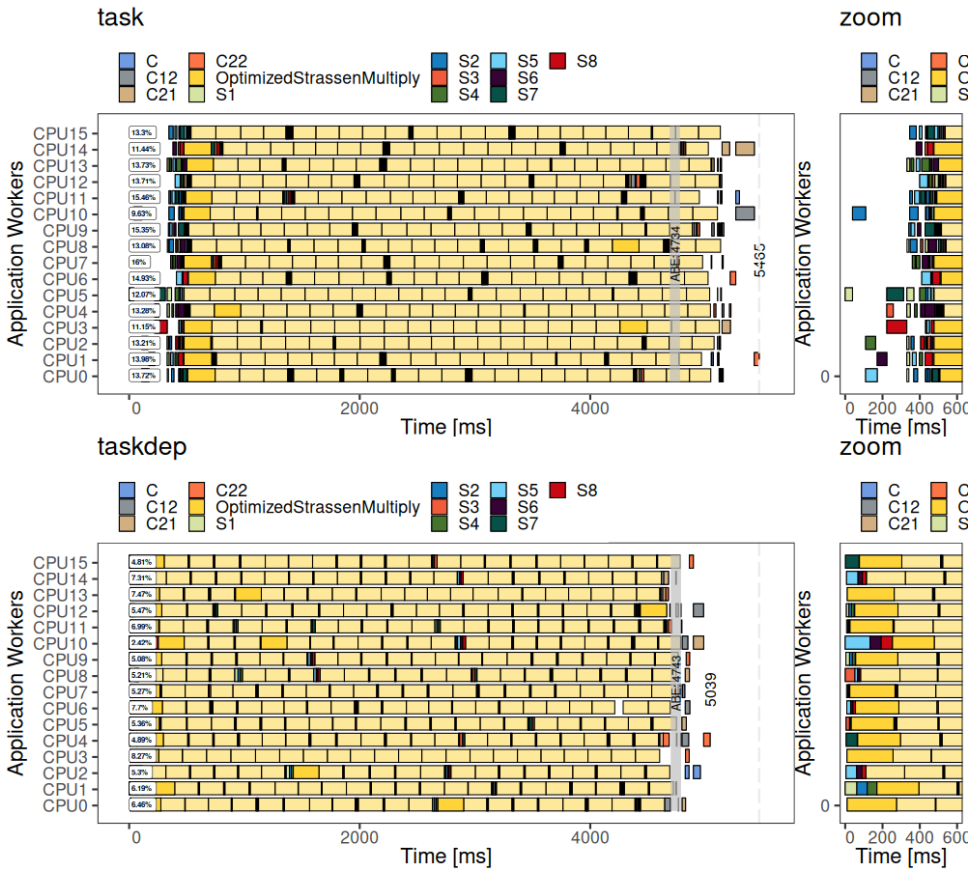

Figure 2. StarVZ space-time view for the Strassen application. The horizontal axes are synchronized. Zoom over the first 600ms on the right.

understand where such performance differences came from, we employed our visualization workflow to generate StarVZ space-time visualizations.

Figure 3 shows three space-time plots obtained from the execution where the makespan corresponds to the median of each case. The vertical gray bar around 1360ms represents the ABE value, a theoretical lower-bound estimation for the makespan without considering any dependency among tasks. The ABE value is quite equal for the three cases, which is expected since the three executions have the same load. The little variation on it (1357 vs 1364 vs 1359) can be explained by the minimal variation in the duration of each task. One can also observe that some tasks were drawn with darker colors (e.g., `gemm` ones). Such tasks were identified as outlier tasks by StarVZ. They appear mostly at the beginning of the three executions and do not seem to explain the performance differences. StarVZ is also able to recursively track the last dependency releasing a task. Here, we enabled the highlight of the critical path of the last task tagged with ID 219 in the three cases. We can observe that the dependency edges are mostly vertical, meaning such tasks were executed as soon as possible, at least at the end of the executions. Finally, we can focus on the resource idleness represented by white areas along the plot and resumed by the idleness quantification label on the left side. It is clear that the `wrong` execution presents much more idle periods and that they start earlier (≈400ms). This can be explained by the bad hints that favor the execution of numerous `gemm` tasks over the critical `potrf` ones. This decision delays the DAG unrolling, meaning that at some moments,

the number of ready tasks is insufficient to fill all the resources. Executions `none` and `correct` progress quite well until ≈900ms when the `none` has several resources in idle. The lack of scheduling hints caused a bad decision on the OpenMP runtime when starting the fourth `potrf` task. On the `correct` execution, this fourth `potrf` task started around 600ms on CPU11; on the `none` case, it started only around 900ms on CPU6. After this point, both executions progress in similar ways. The ratio between remaining tasks to execute and available resources made it possible to `none` case to complete only slightly after the `priority` one.

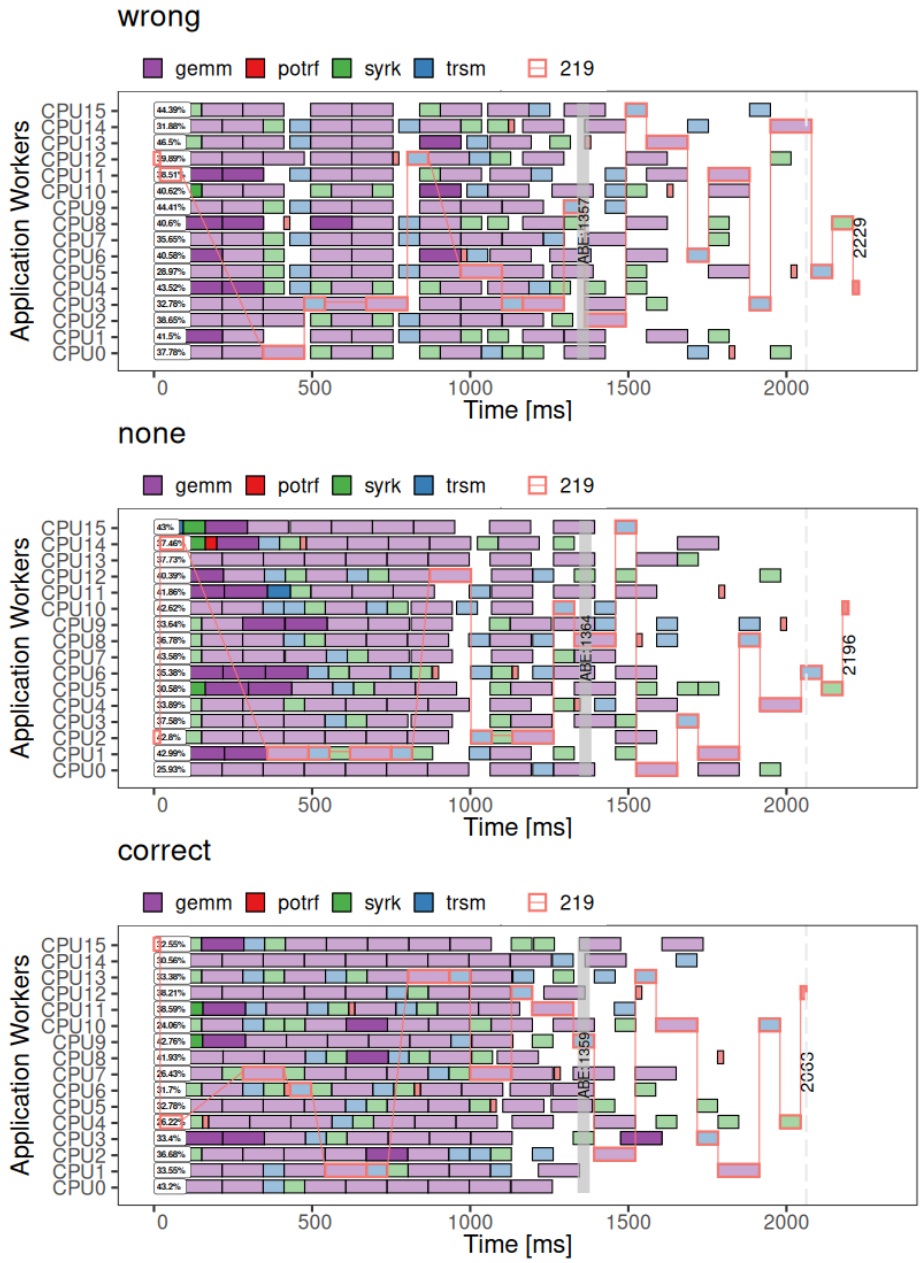

Figure 3. StarVZ space-time visualization for the three versions of the Cholesky factorization. The horizontal axis of the three plots is synchronized. The dashed gray line indicates the makespan of the best case.

5. Related Work

There are several works reporting performance analysis of task-based applications. Most of them rely on state-of-the-art runtime systems to explore heterogeneous multi-CPU and multi-GPU platforms. [Augonnet et al. 2010] and [Lima et al. 2015] propose and analyze new scheduling algorithms that consider data movements and heterogeneity using the StarPU and XKaapi, respectively. [Haugen et al. 2015] presents time-line visualizations with task dependencies for PaRSEC applications. Other works explore the OpenMP tasking model on multicore platforms. [Virouleau et al. 2014] explores the OpenMP `depend` clause reporting code changes and speed-up gains from several existing algorithms. [Miletto and Schnorr 2019] compares StarPU and OpenMP using a QR Factorization. [YarKhan et al. 2016] reports the PLASMA linear algebra library porting from the QUARK runtime system to OpenMP. This paper presents some visualization plots comparing executions using data dependencies with barrier ones. [Lima et al. 2019] reports the performance gains of the Lattice-Boltzmann Method when using task-based versions on top of OmpSs, StarPU, and XKaapi versus the traditional OpenMP parallel loop approach. [Agrawal et al. 2018] present improvements on the Intel Advisor to visualize DAG details of OpenMP task-based applications using the OMPT interface. [Daumen et al. 2019] presents an OMPT-based solution to analyze scalability problems on traditional loop-based OpenMP codes. [Pinho et al. 2020] relies on OMPT to profile OpenMP task-based codes, focusing on the OpenMP internal states as regions and barriers.

Most of these works rely on something other than trace visualization to reaffirm the observed results. Those offering such features rely on custom in-house or specific tool solutions. Others do not specify or use non-public tools. In this work, we present an alternative workflow entirely based on open-source tools such as Tikki and StarVZ.

6. Conclusion

In this paper, we presented a workflow for integrating the Tikki OMPT-based tracing tool into the StarVZ framework for performance analysis of task-based applications. We demonstrated our strategy by analyzing the performance of three task-based applications. Results show that our strategy successfully explained the performance gaps in two of them. For the third one, the overhead introduced by Tikki prevents a fair analysis.

In future work, we aim to extend the integration of Tikki and StarVZ on two axes. First, we plan to extend the data collected by Tikki to enable the existing StarVZ additional plots that show runtime library states, metrics on submitted and ready tasks, and data-handles management. The second point is to propose new StarVZ panels to depict information that is already available in Tikki traces as NUMA aspects and recursive tasks. Other future work includes broader analysis taking into account new OpenMP resources such as the task `affinity` clause and the support for GPUs.

Acknowledgements. This study was partially financed by the Brazilian funding agencies FAPERGS (ARD/ARC 23/2551-0000861-0) and CNPq (PIBIC). We are also thankful to the authors of [Virouleau et al. 2014, Daoudi et al. 2020, Nesi et al. 2021] for making the source code of the benchmark applications and the Tikki tool available.

References

Agrawal, V., Voss, M. J., Reble, P., Tovinkere, V., Hammond, J., and Klemm, M. (2018). Visualization of OpenMP* task dependencies using Intel® Advisor – flow graph analyzer. In *Lecture Notes in Computer Science, International Workshop on OpenMP, IWOMP 2018*, page 175–188. Springer International Publishing, Barcelona, Spain.

Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. (2010). Data-aware task scheduling on multi-accelerator based platforms. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1996). Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69.

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., and Dongarra, J. J. (2013). Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science and Engineering*, 15(6):36–45.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53.

Daoudi, I., Virouleau, P., Gautier, T., Thibault, S., and Aumage, O. (2020). *sOMP: Simulating OpenMP Task-Based Applications with NUMA Effects*, page 197–211. Springer.

Daumen, A., Carribault, P., Trahay, F., and Thomas, G. (2019). *ScalOMP: Analyzing the Scalability of OpenMP Applications*, page 36–49. Springer, Auckland, New Zealand.

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 and Engineering*, 19(3):52–62.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193.

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*. IEEE.

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 *Lecture Notes in Computer Science, International Workshop on OpenMP, IWOMP 2019*, page 21–35. Springer, Auckland, New Zealand.

Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., De Supinski, B. R., and Futral, S. (2015). The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12.

Garcia Pinto, V., Mello Schnorr, L., Stanisic, L., Legrand, A., Thibault, S., and Danjean, V. (2018). A visual performance analysis framework for task-based parallel ap-

plications running on hybrid clusters. *Concurrency and Computation: Practice and Experience*, 30(18).

Gautier, T., Lima, J. V., Maillard, N., and Raffin, B. (2013). Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE.

Haugen, B., Richmond, S., Kurzak, J., Steed, C. A., and Dongarra, J. (2015). Visualizing execution traces with task dependencies. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, SC15. ACM.

Lima, J. V., Gautier, T., Danjean, V., Raffin, B., and Maillard, N. (2015). Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures. *Parallel Computing*, 44:37–52.

Lima, J. V. F., Freytag, G., Pinto, V. G., Schepke, C., and Navaux, P. O. A. (2019). A dynamic task-based d3q19 lattice-boltzmann method for heterogeneous architectures. In *27th Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE.

Llort, G., Filgueras, A., Jiménez-González, D., Servat, H., Teruel, X., Mercadal, E., Álvarez, C., Giménez, J., Martorell, X., Ayguadé, E., and Labarta, J. (2016). The secrets of the accelerators unveiled: Tracing heterogeneous executions through OMPT. In *LNCS, Int. Workshop on OpenMP (IWOMP)*, page 217–236. Springer, Nara, Japan.

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

Nesi, L. L., Miletto, M., Pinto, V., and Schnorr, L. (2021). Desenvolvimento de aplicações baseadas em tarefas com openmp tasks. pages 131–152. SBC.

OpenMP (2021). OpenMP application program interface version 5.2.

Pinho, V., Yviquel, H., Machado Pereira, M., and Araujo, G. (2020). Omptracing: Easy profiling of openmp programs. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 249–256.

Pinto, V. G., Leandro Nesi, L., Miletto, M. C., and Mello Schnorr, L. (2021). Providing in-depth performance analysis for heterogeneous task-based applications with starvz. In *IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*. IEEE.

Pinto, V. G., Stanisic, L., Legrand, A., Schnorr, L. M., Thibault, S., and Danjean, V. (2016). Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach. In *Third Workshop on Visual Performance Analysis, VPA@SC 2016, Salt Lake, UT, USA, November 18, 2016*, pages 17–24. IEEE.

Schuchart, J., Nachtmann, M., and Gracia, J. (2017). *Patterns for OpenMP Task Data Dependency Overhead Measurements*, page 156–168. 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 *Int. Workshop on OpenMP (IWOMP)*, page 16–29. Springer, Salvador, Brazil.

YarKhan, A., Kurzak, J., Luszczek, P., and Dongarra, J. (2016). Porting the plasma numerical library to the openmp standard. *Int. J. of Parallel Programming*, 45(3):612–633.