Score-P and OMPT: Navigating the perils of callback-driven parallel runtime introspection

Christian Feld[1], Simon Convent[3], Marc-André Hermanns[1,2], Joachim Protze[3], Markus Geimer[1], and Bernd Mohr[1]

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich, Germany, {c.feld,m.a.hermanns,m.geimer,b.mohr}@fz-juelich.de
[2] JARA-HPC, Jülich, Germany
[3] IT Center, RWTH Aachen University, Aachen, Germany, simon.convent@rwth-aachen.de, protze@itc.rwth-aachen.de

Abstract. Event-based performance analysis aims at modeling the behavior of parallel applications through a series of state transitions during execution. Different approaches to obtain such transition points for OpenMP programs include source-level instrumentation (e.g., OPARI) and callback-driven runtime support (e.g., OMPT).
In this paper, we revisit a previous evaluation and comparison of OPARI and an LLVM OMPT implementation—now updated to the OpenMP 5.0 specification—in the context of Score-P. We describe the challenges faced while trying to use OMPT as a drop-in replacement for the existing instrumentation-based approach and the changes in event order that could not be avoided. Furthermore, we provide details on Score-P measurements using OPARI and OMPT as event sources with the EPCC and SPEC OpenMP benchmark suites.

Keywords: performance measurement · performance analysis · OpenMP

1 Introduction

The use of performance analysis tools that measure and analyze the runtime behavior of applications is a crucial part of successful performance engineering. Besides core-level optimizations such as proper vectorization and cache usage, particular attention needs to be paid to efficient code parallelization. In high-performance computing (HPC), OpenMP [26] is commonly used to parallelize computations on the node level to take advantage of the nowadays omnipresent multi-core CPUs. However, before the OpenMP 5.0 specification was released in November 2018, there has been no official interface for tools to capture OpenMP-related information. Nevertheless, performance monitoring tools have been able to obtain OpenMP-related measurement data for quite some time using different approaches.

For example, TAU [30], VampirTrace [14], Scalasca's EPIK [10], ompP [9], and Score-P [15] all leverage the *OpenMP Pragma And Region Instrumentor*

OPARI [20]. OPARI is a source-to-source preprocessor that rewrites OpenMP directives found in the source code, inserting POMP API calls [21] for instrumentation. These functions then have to be implemented by the respective tool to gather relevant performance data. Meanwhile, an extended version (OPARI2) is available using an enhanced API.

Another proposal for an OpenMP collector API was published in 2006 by Itzkowitz et al. [12]. However, with its restricted focus on sampling-based tools, this approach did not find widespread adoption. To the authors knowledge, it has only been implemented and used by the Sun/Oracle Developer Studio compiler's OpenMP runtime and the associated performance tools, as well as the OpenUH compiler [16] and TAU [30] as part of an evaluation by Huck et al. [11].

A first draft of the *OpenMP Tools Interface* (OMPT) was published by Eichenberger et al. [6] in 2013. Based on this interface, Lorenz et al. conducted an initial comparison between OPARI2 and OMPT in the context of Score-P [17]. However, early experiences in implementing OMPT support in both OpenMP runtimes and tools led to significant changes of the interface before it was integrated into the OpenMP specification with Technical Report 4 [24]. A slightly updated version is now part of the OpenMP 5.0 specification [26].

In this paper, we present our experiences with this OpenMP 5.0 version of the OMPT interface as implemented in the LLVM OpenMP runtime [3] with the Score-P instrumentation and measurement system. We describe the challenges encountered while trying to reconstruct the event sequences based on a logical execution view expected by Score-P's measurement core as well as the analysis tools building on top of Score-P from the OMPT events generated by the LLVM runtime. Moreover, we highlight major differences between the OMPT-based data collection and our previous OPARI2-based approach. Finally, we show a detailed overhead comparison between both approaches using the EPCC OpenMP benchmark suite [5] and the SPEC OMP2012 benchmarks [22].

2 The OpenMP Tools Interface

In this section, we will briefly introduce the OpenMP Tools Interface (OMPT) and highlight major changes compared to the initial draft [6] used in the study by Lorenz et al. [17]. OMPT is a portable interface enabling tools to gain deeper insight into the execution of an OpenMP program. The design of OMPT accommodates tools based on both sampling and instrumentation. For instrumentation, OMPT defines callbacks for relevant events to be dispatched during execution of a program. A tool can register callback handlers to record information about the execution which includes, for example, the types of threads, tasks, and mutexes, information on the stack frames, and more. Additionally, there are inquiry functions which can be used to extract additional information from within callback handlers, or signal handlers as typically used to implement sampling tools.

Changes to OMPT In the OpenMP 5.0 specification, tool initialization is now a three-way handshake protocol. This allows the OpenMP runtime to determine early during its initialization whether a tool is present or not. At the same time, a tool can decide against activation for a specific run.

Initially, a tool was able to identify a thread, a parallel team, or a task by an integer ID maintained by the runtime. Tracking OpenMP entities across multiple callback invocations therefore required potentially costly lookups. For most callbacks—a notable exception are the lock and mutex callbacks—the integer ID was replaced by storage for a 64-bit data word that a tool can use to maintain information on behalf of an OpenMP entity, thus enabling more efficient tool implementations.

Moreover, multiple events providing similar information have been folded into a single event callback. While reducing the number of callbacks simplifies the interface, it also reduces the possibilities for a tool to selectively choose a set of interesting events. The initial proposal also contained callbacks indicating that a thread is idling between participation in two consecutive parallel regions; these callbacks have been removed. We will see in Section 3, that the *implicit-task-end* event for worker threads can be dispatched late, so that the runtime might effectively report no idle time.

In contrast, callbacks for advanced OpenMP features such as *task cancellation* or *task dependences* have been added. While cancellation information can be relevant for maintaining the OMPT tool data objects, we do not yet see a use case in Score-P for logging these events. On the other hand, task-dependency information can be interesting to perform critical path analysis in tools like Scalasca. Another addition are callbacks for devices including callbacks for the initialization/finalization of devices as well as for data movements between host and devices. However, this part of OMPT is not yet implemented in the OpenMP runtime we are using for our experiments and therefore not considered in our implementation.

To allow a tool to relate events to source code, a pointer argument providing an instruction address was added to various callbacks. For ease of implementation, this pointer is defined as the return address: the next instruction executed after the runtime function implementing an OpenMP construct finished.

Since the order in which the OpenMP runtime and an attached OMPT tool are shut down is not necessarily well-defined, an `ompt_finalize_tool()` function has been introduced. This function can be called by the tool during its finalization and guarantees that any outstanding events that might have been buffered by the runtime get dispatched. If the OpenMP runtime was already finalized, however, all events have been dispatched and this function call results in a no-op.

3 Implementing OMPT support in Score-P

Score-P—the tool we focus on in this paper—is a highly scalable and easy-to-use instrumentation and measurement infrastructure for profiling, event tracing, and online analysis of HPC applications. It currently supports the analysis tools Scalasca [10,31], Vampir [14], Periscope [4], and TAU [30], and is open for other tools that are based on the Open Trace Format Version 2 (OTF2) [7] or the CUBE4 [28] profiling format as well as tools that implement a Score-P substrate plugin [29] for event consumption. As outlined before, up until now Score-P uses the source-level instrumentor OPARI2 to rewrite and annotate OpenMP directives to gather OpenMP-specific performance data. To limit the number of required changes in the analysis tools, we aim for generating the same (or at least very similar) event sequences based on a logical execution view from the OMPT events generated by the LLVM runtime. In the following, we describe the various challenges encountered and how we addressed them.

During development and for the experiments in this paper we used an OpenMP runtime implementation based on LLVM/7.0 including a patch which implements `ompt_finalize_tool()` [2]. This implementation roughly represents the interface as defined in Technical Report 6 [25], without callbacks for device-related events. Semantically there is no big difference to the OMPT specification in OpenMP 5.0. The resulting Score-P development version implementing the new OMPT functionality can be downloaded from [8]. Compilation was consistently done using the Intel compiler, version 19.0.3.199 20190206.

Event sequence requirements Score-P stores event data independently per logical execution unit in buffers called *locations*. Events in these locations are required to have monotonically increasing timestamps (*monotonicity requirement*). In addition, as the Score-P event model is based on *regions* that correspond to regions in the source code, most events are paired, either as `ENTER/LEAVE` or `BEGIN/END` pairs. These pairs must be properly nested within a location, otherwise the profile measurement or trace analysis fails (*nesting requirement*). Here, special care is taken for events generated from within explicit OpenMP tasks as the nesting requirement might be violated in task scheduling points [18]. For parallel constructs that affect several locations, the happened-before semantics must be reflected in the ordering of timestamps (*HB requirement*). For example, all timestamps belonging to events from within a `parallel` construct must not be larger than the corresponding *parallel-end* timestamp. With OPARI2's instrumentation being entirely inside the parallel region, this requirement is always fulfilled, and thus analysis tools rely on it to calculate performance metrics. To minimize synchronization overhead, the OMPT specification is less strict regarding cross-location happened-before relationships, as detailed below.

`parallel` construct: overdue events OPARI2 as well as OMPT use the event sequence depicted in Figure 1 for a `parallel` construct with T_0 as the encountering thread. The events for each thread T_0-T_2 are written to individual Score-P locations, where the encountering thread and the master child thread share one location. With OPARI2 instrumentation, all events of all locations are dispatched before the closing *parallel-end* on the encountering thread. Timestamps taken at dispatch time are guaranteed to meet the fork-join happened-before semantics. The OMPT specification, however, does not impose the requirement on non-master child threads to dispatch the *implicit-barrier-end* and *implicit-task-end*

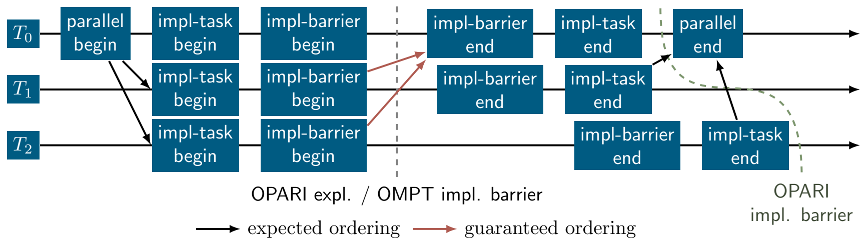

Fig. 1: Event sequence and ordering for a `parallel` construct.

callbacks earlier than the corresponding *parallel-end* on the encountering thread, only all *implicit-barrier-begin* events are guaranteed to be dispatched before the *implicit-barrier-end*. That is, there might be two *overdue* events per non-master child thread waiting for being dispatched even if the parallel region was already *joined*, as highlighted for thread T_2 in the diagram above. The only guarantee for these *overdue* events is that they are dispatched before any further events on this thread.

As a first consequence, timestamps taken when the overdue events are being dispatched likely violate the HB requirement. The only *implicit-barrier-end* and *implicit-task-end* timestamps guaranteed to conform to the assumed ordering are those on the master thread. To retain the happened-before timestamp order in Score-P, we chose to use these timestamps for all remaining *implicit-barrier-end* and *implicit-task-end* events, thus having identical timestamps per event type for all threads in the team.

`parallel` construct: non-deterministic scheduling The next consequence arises from the *combination* of (1) the freedom of the runtime to postpone events, (2) the mapping of OpenMP threads to Score-P locations, and (3) potential non-determinism in mapping of logical OpenMP threads to system threads. Whereas the first item has been described above, the other items need some additional explanation.

Score-P establishes a fixed mapping of OpenMP threads to Score-P locations based on OpenMP nesting characteristics, where the nesting characteristic is determined by the sequence of OpenMP thread numbers from the initial thread to the current one. This mapping is established in *implicit-task-begin* events by assigning a location to thread-local storage. The reasons for a fixed mapping are (1) to provide the user with the *logical* execution view, that is, present events per OpenMP thread number instead of per system thread, and (2) to maximize scalability regarding memory and the number of generated output files. As each distinct nesting characteristic is assigned a single Score-P location, locations are reused in subsequent parallel regions if a nesting characteristic has come to light previously[4]. In contrast, the system thread executing an OpenMP nesting characteristic might change in subsequent parallel regions.

[4] In addition, the master thread reuses the encountering thread's location.

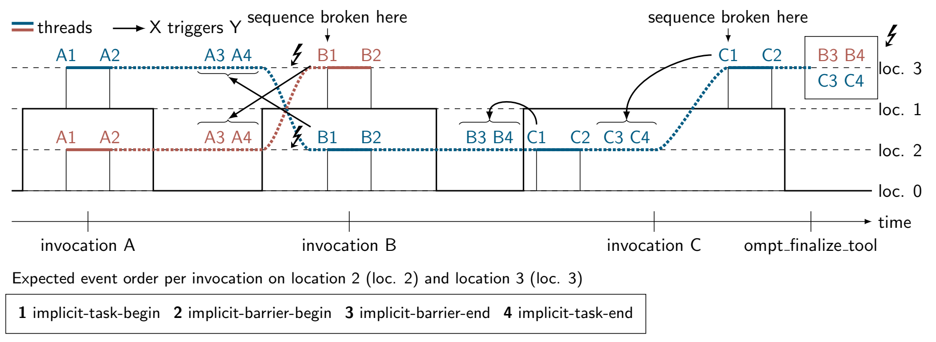

Fig. 2: Three invocations of identical nested parallel regions with two threads in each team. For invocation B the non-master OpenMP threads of the inner regions are invertedly mapped to system threads, for invocation C the non-master threads of both inner regions are mapped to the same blue system thread. A location corresponds to a unique OpenMP ancestry sequence.

In the advent of *overdue* events combined with a non-deterministic OpenMP thread to system thread mapping we observe two anomalous schedules which tend to break the monotonicity requirement and may lead to data corruption. Figure 2 illustrates these schedule decisions. Assume a parallel region with a team size of two that executes a nested parallel region, also with team size of two, for three subsequent invocations A, B, and C. The two inner parallel regions expose work for four OpenMP threads with different nesting characteristics, thus Score-P will create four locations. The Score-P locations created in invocation A are reused in invocation B and C because of identical nesting characteristics. The first form of the anomaly manifests in a *OpenMP thread to system thread assignment* switch between invocations; while the inner region's non-master implicit task n was served by system thread i in the first invocation, it is served by system thread j in the second one and vice versa for the other inner region, see transition from invocation A to B in the figure. Each two system threads *blue* and *red* carry two *overdue* events A3 and A4 from invocation A to be written to location 3 (blue) and location 2 (red), respectively. Each thread triggers its overdue events before its B1 event of invocation B. In B1 the switch manifests as a location change. As no ordering is enforced by OMPT, thread *red* might write B1 concurrently with thread *blue* writing the overdue events A3 and A4 to location 3 and vice versa for location 2. This race condition potentially violates the monotonicity requirement on either location—the overdue events A3 and A4 need to be written before B1—or worse, leads to corrupted data. Note that there is no race condition in the absence of overdue events.

The second form manifests in invocation C in Figure 2 being executed by just three of the four threads; the two inner region's non-master implicit tasks get both executed by the same system thread (blue). This time there is no issue on location 2 as all events are delivered in the expected order. The problem arises for location 3 during runtime shutdown. The undelivered events B3 and B4 (red) are dispatched and will violate the monotonicity requirement. If the undelivered C3 and C4 (blue) are dispatched concurrently, data might get corrupted in addition. The runtime implementation we used showed this anomalies only with nested `parallel` constructs.

To address these two anomalies, we need to ensure that any overdue events for a given location are written *before* processing an *implicit-task-begin* event from a subsequent invocation on the same location. Translated to Figure 2, invocation B and location 3, this means to write A3 and A4 from thread *blue* before B1 from thread *red*. Thus, the first thing to do in B1 is to detect whether there are overdue A3 and A4 events for location 3. To do so, we use location-specific data transferred from invocation A to invocation B, saving a Score-P representation of the latest implicit task data together with synchronization handles. This data is cleared from the location once the overdue events have been processed completely. If the overdue event data is still available when thread *red* dispatches B1, thread *red* takes ownership and processes A3 and A4 first—using the location-specific data provided by thread *blue* in invocation A—while preventing thread *blue* to do the same. If thread *blue* is first, it takes ownership and processes A3 and A4 while blocking thread *red* working on B1 during this time. Applying this synchronization for every *implicit-task-begin* will processes all overdue events except the ones waiting for being dispatched when the program finishes, here C3 and C4 from thread *blue*. These are explicitly triggered by calling `ompt_finalize_tool` during Score-P's shutdown and handled without additional effort. The fine-grained synchronization necessary to orchestrate this mechanism uses atomic updates and two spin-mutexes per location.

Developing this overdue-handling mechanism to maintain the established event sequence for the `parallel` construct was the biggest challenge in implementing support for OMPT in Score-P. Once this was achieved, implementing other OMPT callbacks was straightforward.

4 Differences in event sequence and source information

To investigate differences emerging from using OMPT callbacks compared to the traditional OPARI2 instrumentation, we ran experiments from the OpenMP 4.5 Examples [8, 23].

Worksharing constructs Implicit barriers synchronize worksharing constructs, unless a `nowait` clause was given. For OPARI2, these implicit barriers conceptually belong to the construct, that is, the events are nested inside the enclosing construct's `ENTER` and `LEAVE` events. In contrast, OMPT dispatches the implicit barrier events after the worksharing's *end* event. The different event order is exemplified with a minimal example using the `worksharing-loop` construct, see Listing 1. This event-sequence change is seen for all worksharing constructs.

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i<20; i++)
      work();
}
```

```
 ENTER Region: "!$omp parallel"
   ENTER Region: "!$omp for"
-    ENTER Region: "!$omp barrier"
-    LEAVE Region: "!$omp barrier"
   LEAVE Region: "!$omp for"
+  ENTER Region: "!$omp barrier"
+  LEAVE Region: "!$omp barrier"
   ENTER Region: "!$omp barrier"
   LEAVE Region: "!$omp barrier"
 LEAVE Region: "!$omp parallel"
```

Listing 1: For the `worksharing-loop` construct, `ENTER` and `LEAVE` events for the implicit barrier are created inside the construct (OPARI2 in red) or outside the construct (OMPT in green).

Barriers An OMPT implementation might distinguish between *implicit* and *explicit* barriers, but the LLVM runtime we used currently does not. OPARI2, on the other hand, distinguishes between barrier types. Whereas explicit barriers are easily instrumented by OPARI2, implicit ones need special attention. An implicit barrier is transformed to an instrumented explicit barrier, and for worksharing constructs a `nowait` clause is added to the corresponding construct. This way timing information can be obtained and the semantics stay unchanged. However, there are cases where the compiler can safely merge consecutive implicit barriers[5]. By transforming the implicit barrier, OPARI2 prevents the compiler from performing this optimization.

Tasking OPARI2 takes care that undeferred tasks will not create any events by evaluating the `if` clause. Similar behavior was implemented with OMPT by evaluating the task type. For the remaining tasks, there are some changes regarding the sequence of events written by Score-P. In general, the OMPT specification allows to signal the switch from one task to the next task. However, the current implementation in the LLVM runtime first signals a switch back to the scheduling task before switching to the next task. This additional switch is not observed with OPARI2, which leads to a reduced number of recorded scheduling events. Task switches in OPARI2 are triggered when a task starts running and potentially after scheduling points have been processed [18]. As OPARI2 does not instrument all scheduling point types yet, untied tasks will break the nesting requirement when scheduled in an unsupported type. OMPT provides a robust and complete picture in this regard.

With OPARI2 it is possible to measure the duration of task creation, as the instrumentation provides distinct task-create-begin/end events. OMPT's *task-create* does not provide timing information, nevertheless we mapped it to the task-create-begin/end pair to preserve the existing event sequence.

[5] See, for example, `Example_barrier_regions.1.c` from the OpenMP 4.5 Examples [23] where the implicit barrier of the inner parallel region is omitted.

	■ llvm-ompt-off	■ llvm-ompt-on	■ scorep-opari2	■ scorep-ompt
OMPT Runtime	no	yes	no	yes
Score-P Adapter	—	—	OPARI2	OMPT

Table 1: Matrix of measurement setups used in the evaluation.

Relation to source code To optimize a program after performance analysis, a user needs to relate analysis hotspots to source code. OPARI2, as a source-level translator, has comprehensive knowledge of source locations. Line number and filename of instrumented OpenMP constructs are hard-coded into OPARI2's output files. OMPT's means to relate OpenMP events to their source is to provide a return address (`codeptr_ra`) as a callback argument which is mapped to a Score-P handle dynamically. This address does not point to the corresponding OpenMP construct, but to the application code being executed once the OpenMP region related to the event is completed. Usually the instruction before this address resolves to the corresponding *filename:lineno* source location[6].

Other differences between OMPT and OPARI In addition, we want to mention differences regarding the following constructs just briefly:

Named criticals While OPARI2 provides the optional name of a `critcal` construct, OMPT distinguishes the underlying locks by a numeric `wait_id`.

Atomic construct The LLVM runtime only dispatches callbacks for *atomic*-events if the compiler is not able to emit a native atomic instruction. OPARI2 is able to instrument all `atomic` constructs, but due to the large relative overhead involved, it allows for deactivating this feature.

Section construct The LLVM runtime currently does not provide events regarding the `section` construct (within the `sections` construct) although the specification defines the corresponding `ompt_callback_dispatch`.

`omp_test_lock` and `omp_test_nest_lock` The LLVM runtime does not yet distinguish between *locks* and *test locks* and their nested counterparts.

5 Evaluation

We used the EPCC OpenMP micro-benchmark suite [5] and the SPEC OMP2012 benchmarks version 1.0 [22] to evaluate the measurement dilation introduced by the Score-P measurement adapters using OPARI2 and OMPT. The platform for our evaluation is the cluster partition of the JURECA supercomputer [13] operated by the Jülich Supercomputing Centre of Forschungszentrum Jülich in Germany. All measurements were taken on the same JURECA node, which consists of two Intel Xeon E5-2680 Haswell CPUs (2.5GHz, 12 cores each) and 128GB

[6] To convert addresses into file names and line numbers, we rely on the *Binary File Descriptor library* (BFD) [1] and debug symbols in the binary.

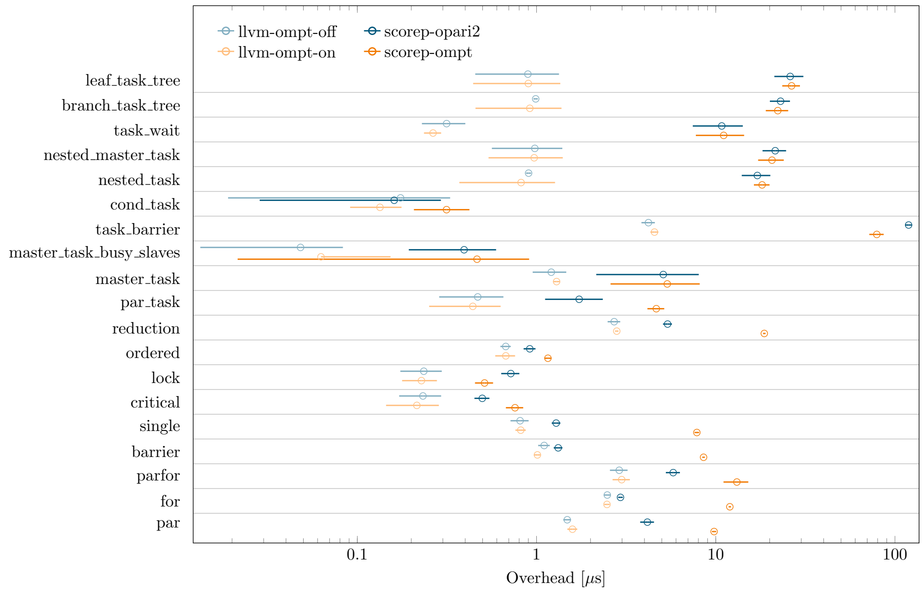

Fig. 3: Overhead reported by the EPCC OpenMP Benchmark Suite for individual OpenMP constructs in the four different measurement setups with 12 threads on a single socket of a JURECA Cluster Module [13] node.

RAM. For easier evaluation and reproducibility, we used the Jülich Benchmarking Environment (JUBE) [19] in version 2.2.2 to configure and run the measurements [8]. The Score-P measurements were done in profiling-only mode.

In our evaluation, we explore four different measurement setups as shown in Table 1. As OPARI2 (scorep-opari2) does not need OMPT runtime support, we disabled it in the LLVM runtime and provide a baseline measurement for this setup (llvm-ompt-off). For the OMPT adapter (scorep-ompt), we used a separate installation of the same LLVM runtime version with OMPT support enabled and also provide a separate baseline measurement (llvm-ompt-on). Data for baseline measurements are indicated by desaturated colors, whereas vivid colors indicate measurements with Score-P attached. Blue indicates OMPT to be disabled in the measurement, whereas orange indicates OMPT to be enabled.

EPCCbench The EPCC OpenMP micro-benchmark suite was developed to identify overheads created by individual OpenMP constructs. We use it here to compare the overhead that Score-P adds to the OpenMP measurement of individual constructs for each adapter—OPARI2 and OMPT—by comparing the overhead reported by the benchmark with and without Score-P attached.

Figure 3 shows the measurements on a single node of the JURECA cluster with 12 threads bound to a single socket[7]. For these measurements, we inten-

[7] We used `OMP_PROC_BIND=close` and `OMP_PLACES={0}:12` for all measurements.

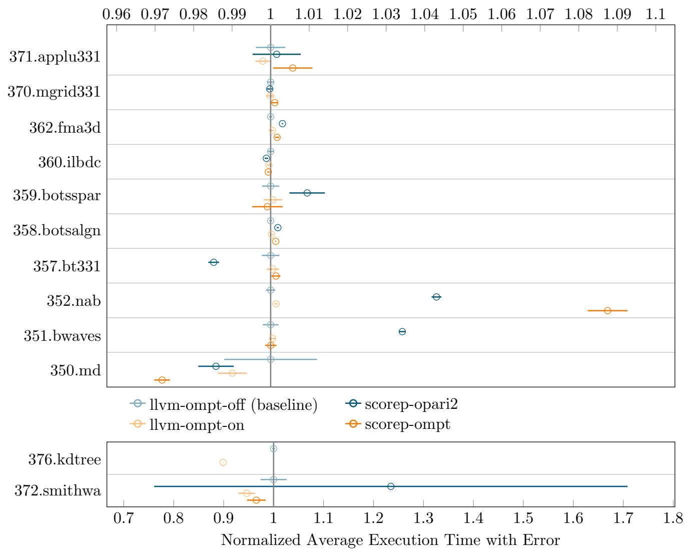

Fig. 4: Normalized execution time of the configured SPEC OMP2012 benchmark applications for the four different measurement setups using the *ref* input size with 12 threads on a single socket of a JURECA Cluster Module [13] node.

tionally did not occupy both sockets of the JURECA node to eliminate potential NUMA effects in the measurements caused by inter-socket memory accesses. We ran the benchmark with 150 outer repetitions, a test time of $5000\mu s$, and the delay time set to $15\mu s$. The EPCC benchmark uses the configured outer repetitions to provide an average overhead and uncertainty bounds for it as shown in the figure.

We notice that measurement setups llvm-ompt-off and llvm-ompt-on show very similar performance, i.e., OMPT overhead is minimal if no tool is attached.

While most of the task constructs are equally costly with OPARI2 and OMPT, we see a higher overhead with the Score-P OMPT adapter for kernels involving worksharing and barrier constructs. Analysis revealed that the additional overhead is caused inside Score-P by mapping `codeptr_ra` callback arguments to Score-P-handles concurrently[8]. We are confident to be able to improve this mapping in a future implementation. However, there will always be more overhead involved compared to OPARI2, as in this case all required information is statically available after source-to-source translation.

SPEC OMP2012 To evaluate the influence that users may expect of the two different Score-P adapters on measurements of real-world applications, we measured the runtime of 12 benchmarks of the SPEC OMP 2012 benchmark suite. We used the `runspec` command to build the respective benchmark applications,

[8] The `addr2line` lookup is done only once per address and is negligible.

but used JUBE to run the experiments. To enable time measurements even without the presence of a performance tool, we introduced coarse-grained time measurement and output around the outer iteration, excluding initialization and I/O where possible, to minimize external influences on the measurement.

Figure 4 shows measurements using the *ref* input size. As absolute execution time with this input size spreads significantly across the different benchmarks, we normalized the data. The average time of each application in measurement setup llvm-ompt-off acts as the baseline for the other measurement setups reported for that application. Therefore all of these measurements are displayed as 1, crossed by the vertical baseline indicator. For each data point, the average of 5 runs is reported, error bars indicating the standard deviation. The measurements show that for most of the SPEC applications, the runtime dilation due to the Score-P measurements is within an acceptable range independent of the adapter used. 352.nab generates a large number of worksharing and barrier events which are—due to the contended `codeptr_ra` lookup—likely to cause the additional overhead seen with the OMPT adapter [27]. More than 99% of 357.applu331's events are flush events. For these, we also do a `codeptr_ra` lookup, but apparently with less contention. 351.bwaves with its numerous, subsequent `parallel do` constructs revealed a smaller number of parallel and barrier events in the OMPT case, which might be due to the compiler's ability to fuse subsequent loops. A more in-depth investigation is needed, though. The measurements for 376.kdtree aborted for both the OPARI2 and the OMPT adapter, as memory requirements for the excessive number of explicit tasks could not be fulfilled by Score-P. The reason for the large standard deviation of the OPARI2 measurement of 372.smithwa could not yet be determined and is still under investigation.

6 Conclusion

With the availability of an official OpenMP Tools Interface, instrumentation-based performance tools need to consider to replace the common source-level OPARI2 approach, mainly to reduce the maintenance burden in the long run. In this paper, we presented the challenges implementing an OMPT tool based on the LLVM runtime as a drop-in replacement for OPARI2 in the context of Score-P and described the unavoidable changes in the order of OpenMP events. OMPT provides a *runtime* execution view, but as Score-P-based analysis tools historically rely upon a *logical* execution view, our first implementation tried to retain the latter. This choice presented a challenge handling the `parallel` construct, whereas implementing other OMPT callbacks was straightforward and provided sufficient measurement data to serve as a replacement.

From the EPCC micro-benchmarks, we saw that OMPT overhead is minimal if no tool is attached, recording task events is costly with both OPARI2 and OMPT, and our OMPT tool consistently generates higher overhead for worksharing and barrier constructs. The latter is caused by contended mapping of `codeptr_ra` callback arguments to Score-P-handles within Score-P and will be addressed in the future. However, this overhead does not propagate in great severity to real-world applications from SPEC OMP2012 but manifests in programs with a high number of `codeptr_ra` lookups.

Once additional OpenMP runtimes with OMPT support are available from compiler vendors, we are eager to verify whether they also provide sufficient data to our tool to replace the source-level OPARI2 approach. In addition, we will investigate how analysis tools consuming the Score-P measurement data have to be adapted to deal with the remaining differences in event order.

Acknowledgements

The authors gratefully acknowledge the computing time granted through JARA-HPC on the supercomputer JURECA [13] at Forschungszentrum Jülich. Part of this work was performed under the POP2 project and has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 824080.

References

1. GNU Binutils. `https://sourceware.org/binutils/`
2. LLVM runtime with experimental changes for OMPT. `https://github.com/OpenMPToolsInterface/LLVM-openmp/commits/tool_finalization_tr7`, branch tool_finalization_tr7, commit dcf2962eb6d92d82e74bd374f27e6ef836a5e2b3
3. Support for the OpenMP language in LLVM. `http://openmp.llvm.org`
4. Benedict, S., Petkov, V., Gerndt, M.: Periscope: An online-based distributed performance analysis tool. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009. pp. 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
5. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for OpenMP tasks. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7312, pp. 271–274. Springer (2012). https://doi.org/10.1007/978-3-642-30961-8_24, `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite`
6. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In: OpenMP in the Era of Low Power Devices and Accelerators. LNCS, vol. 8122, pp. 171 – 185. 9th International Workshop on OpenMP, Canberra (Australia), 16 Sep 2013 - 18 Sep 2013, Springer, Berlin/Heidelberg (Sep 2013). https://doi.org/10.1007/978-3-642-40698-0_13, `http://juser.fz-juelich.de/record/138577`
7. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In: Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011. Advances in Parallel Computing, vol. 22, pp. 481–490. IOS Press (2012). https://doi.org/10.3233/978-1-61499-041-3-481

8. Feld, C., Convent, S., Hermanns, M.A., Protze, J., Geimer, M.: [Reproducibility] Score-P and OMPT: Navigating the perils of callback-driven parallel runtime introspection (Jun 2019). https://doi.org/10.5281/zenodo.3251871
9. Fürlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. In: Proceedings of the first International Workshop on OpenMP (IWOMP) (2005)
10. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience **22**(6), 702–719 (April 2010). https://doi.org/10.1002/cpe.1556
11. Huck, K.A., Malony, A.D., Shende, S., Jacobsen, D.W.: Integrated Measurement for Cross-Platform OpenMP Performance Analysis. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) Using and Improving OpenMP for Devices, Tasks, and More. pp. 146–160. Springer International Publishing, Cham (2014)
12. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP Runtime API for Profiling. White paper (2002), `http://www.compunity.org/futures/omp-api.html`
13. Jülich Supercomputing Centre: JURECA: Modular supercomputer at Jülich Supercomputing Centre. Journal of large-scale research facilities **4**(A132) (2018). https://doi.org/10.17815/jlsrf-4-121-1, `http://dx.doi.org/10.17815/jlsrf-4-121-1`
14. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set, pp. 139–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68564-7_9, `https://doi.org/10.1007/978-3-540-68564-7_9`
15. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany. pp. 79–91. Springer Berlin Heidelberg (Sep 2012). https://doi.org/10.1007/978-3-642-31476-6_7, `http://dx.doi.org/10.1007/978-3-642-31476-6_7`
16. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An optimizing, portable OpenMP compiler. Concurrency and Computation: Practice and Experience **19**(18), 2317–2332 (2007)
17. Lorenz, D., Dietrich, R., Tschüter, R., Wolf, F.: A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P. In: Proc. of the 10th International Workshop on OpenMP (IWOMP), Salvador, Brazil, September 2014. LNCS, vol. 8766, pp. 161–172. Springer International Publishing (Sep 2014). https://doi.org/10.1007/978-3-319-11454-5_12, `http://dx.doi.org/10.1007/978-3-319-11454-5_12`
18. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to reconcile event-based performance analysis with tasking in OpenMP. In: Proc. of 6th Int. Workshop of OpenMP (IWOMP), Tsukuba, Japan. Lecture Notes in Computer Science, vol. 6132, pp. 109–121. Springer (Jun 2010). https://doi.org/10.1007/978-3-642-13217-9_9
19. Lührs, S., Rohe, D., Schnurpfeil, A., Thust, K., Frings, W.: Flexible and Generic Workflow Management. In: Parallel Computing: On the Road to Exascale. Advances in parallel computing, vol. 27, pp. 431 – 438. International Conference on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep

2015, IOS Press, Amsterdam (Sep 2016). https://doi.org/10.3233/978-1-61499-621-7-431, https://www.fz-juelich.de/jsc/jube/

20. Mohr, B., Malony, A., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. The journal of supercomputing **23**, 105 – 128 (2002). https://doi.org/10.1023/A:1015741304337, http://juser.fz-juelich.de/record/25115, record converted from VDB: 12.11.2012
21. Mohr, B., Malony, A.D., Hoppe, H.C., Schlimbach, F., Haab, G., Hoeflinger, J., Shah, S.: A performance monitoring interface for OpenMP. In: Proceedings of the 4th European Workshop on OpenMP (EWOMP'02). Rome, Italy (Sep 2002)
22. Müller, M., Baron, J., Brantley, W., Feng, H., Hackenberg, D., Henschel, R., Jost, G., Molka, D., Parrott, C., Robichaux, J., Shelepugin, P., van Waveren, M., Whitney, B., Kumaran, K.: SPEC OMP2012 an application benchmark suite for parallel systems using OpenMP. In: Proceedings of the 8th international conference on OpenMP in a Heterogeneous World. pp. 223–236 (06 2012). https://doi.org/10.1007/978-3-642-30961-8_17
23. OpenMP Architecture Review Board: OpenMP Application Programming Interface – Examples – Version 4.5.0. http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
24. OpenMP Architecture Review Board: TR4: OpenMP Version 5.0 Preview 1. Specification (November 2016), http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf
25. OpenMP Architecture Review Board: TR6: OpenMP Version 5.0 Preview 2. Specification (November 2017), http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf
26. OpenMP Architecture Review Board: OpenMP application program interface version 5.0. Specification (November 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
27. Protze, J., Hahnfeld, J., Ahn, D.H., Schulz, M., Müller, M.S.: OpenMP Tools Interface: Synchronization information for data race detection. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) Scaling OpenMP for Exascale Performance and Portability. pp. 249–265. Springer International Publishing, Cham (2017)
28. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4: From performance report explorer to performance analysis tool. Procedia Computer Science **51**, 1343–1352 (Jun 2015). https://doi.org/10.1016/j.procs.2015.05.320
29. Schöne, R., Tschüter, R., Ilsche, T., Schuchart, J., Hackenberg, D., Nagel, W.E.: Extending the functionality of Score-P through plugins: Interfaces and use cases. In: Niethammer, C., Gracia, J., Hilbrich, T., Knüpfer, A., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2016. pp. 59–82. Springer International Publishing, Cham (2017)
30. Shende, S.S., Malony, A.D.: The Tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (May 2006). https://doi.org/10.1177/1094342006064482, http://dx.doi.org/10.1177/1094342006064482
31. Zhukov, I., Feld, C., Geimer, M., Knobloch, M., Mohr, B., Saviankou, P.: Scalasca v2: Back to the future. In: Proc. of Tools for High Performance Computing 2014. pp. 1–24. Springer (2015). https://doi.org/10.1007/978-3-319-16012-2_1