AfterOMPT: An OMPT-based tool for fine-grained tracing of tasks and loops

Igor Wodiany[1] (✉), Andi Drebes[2], Richard Neill[1], and Antoniu Pop[1]

[1] Department of Computer Science, The University of Manchester, United Kingodom
{igor.wodiany,richard.neill,antoniu.pop}@manchester.ac.uk
[2] Inria and École Normale Supérieure, Paris, France
andi.drebes@inria.fr

Abstract. We present AfterOMPT, a new trace-based tool for analyzing the execution of OpenMP applications using the OMPT interface to capture accurate information on loop partitioning, distribution of iteration spaces across workers, task scheduling, and synchronization events. In contrast to previous works that rely on specific, instrumented runtime libraries, our tool is able to collect information from any runtime implementing the OMPT interface. In order to visualize the information from the collected traces, we have extended the Aftermath performance analysis tool with appropriate renderers for OMPT events. We also propose an extension of the OMPT interface for the collection of more detailed information on scheduled OpenMP loops. Experimental results show a tracing overhead of under 5% for the majority of studied benchmarks, increasing more significantly for those with highly fine-grained workloads.

Keywords: OpenMP · OMPT · Performance analysis · Tracing

1 Introduction

There are many factors that impact the performance of OpenMP [15] programs which may result in an inefficient utilization of the executing system, such as a limited amount of parallelism exposed by the application itself, interactions with the runtime system, locality of memory accesses, and sub-optimal use of explicit parallel constructs. Such performance bottlenecks are difficult or even impossible to detect using static analysis and thus require tracing of dynamic events and post-mortem analysis. In order to precisely identify the source of performance issues, it is further necessary to be able to attribute such events to specific instances of parallel constructs and to the OpenMP workers.

Aftermath [5] is a trace-based tool for performance analysis of parallel programs and has been extended for OpenMP programs in prior work [4]. The tool provides developers with accurate traces for loop and task execution and

This work was supported by the grant EuroEXA H2020-754337. Antoniu Pop is funded by the RAEng University Research Fellowship. Igor Wodiany is supported by the Department of Computer Science Kilburn Scholarship and the University of Manchester Presidents Award.

is able to capture synchronization events. Its ability to trace and visualize the distribution of loop iteration spaces across workers allows programmers to track the origin of work imbalance caused by inappropriate chunk sizes, unsuited loop scheduling strategies or a mismatch between data placement and work distribution on machines with non-uniform memory access. However, Aftermath relies on an instrumented version of the OpenMP runtime to generate traces. This comes with a cost for setting up the execution environment and bears the risk of an outdated runtime library, as the instrumentation likely requires updating with every new version of the runtime.

With the inclusion of the OpenMP Tools (OMPT) interface [7] in the OpenMP standard, it became possible to develop portable profiling tools that can be attached to any compatible runtime. The OMPT interface defines a set of callbacks that are invoked by the runtime for specific events throughout the execution. Tools can use this interface to capture information associated to these events and to write this information to a trace file. In order to eliminate the need for a specific, instrumented runtime for Aftermath tracing and thus provide a portable tool for OpenMP performance analysis, we have developed *AfterOMPT*, a library that implements the OMPT callbacks to collect dynamic events and write them to a trace file using the Aftermath tracing API. We have further extended Aftermath with OMPT-specific rendering functions that enables the visualization of such traces.

While the set of callback functions specified by the OMPT interface covers a basic set of OpenMP events, it is unsuited to capture dynamic information about the distribution of loop iteration spaces across workers: the OMPT work callback (`ompt_callback_work`) can only capture an aggregated execution of all iterations assigned to a specific worker, without any loop-specific details. The more recent dispatch callback (`ompt_callback_dispatch`) is an attempt to mitigate this issue, but potentially incurs a high overhead as is is called at the beginning of each loop iteration.

Langdal et al. [10] identified similar issues with OMPT in the context of Grain Graphs, a chronogram-based tool for visualizing OpenMP applications in the form of hierarchical graphs. Although their work provides specific implementation details and detailed overhead analysis, it lacks concrete examples on how information obtained from those callbacks can be used to optimize the performance of applications.

In this work, we provide concrete case studies to make a case for extending the tracing interface. Specifically, this paper makes the following contributions:

– We present *AfterOMPT*, a new Aftermath-based tool implementing the OMPT interface for portable and detailed performance analysis.
– We present two case studies to support an extension of the tracing interface with loop-related OMPT callbacks.
– We provide experimental results showing that the instrumentation overhead is below 5% for the majority of our studied benchmarks.

The rest of the paper is organized as follows. Section 2 introduces Aftermath and sets the terminology that AfterOMPT borrows from Aftermath. In

```
for(int k = 0; k < 2; k++) {
  #pragma omp parallel
  {
    #pragma omp for schedule(static, 2) // First loop
    for(int i = 0; i < 32; i++) { foo(); }
    foo();
    #pragma omp for schedule(dynamic, 2) // Second loop
    for(int i = 0; i < 32; i++) { foo(); }
    foo();
  }
}
```

Listing 1.1: Example with two loop constructs

Section 3, we discuss the implementation of our profiling tool and use of existing and proposed OMPT callbacks. We then present two case studies in Section 4 illustrating how AfterOMPT can be used to collect and inspect OpenMP traces. Tracing overhead is analyzed in Section 5. Related work, concluding remarks and directions for future work are presented in Section 6 and Section 7.

2 Aftermath

Aftermath[3] is a free and open source tracing and visualization tool for performance analysis. The project has transitioned from a tool supporting a specific set of parallel frameworks [5,4] to a framework-independent toolbox for building specialized tools for performance analysis. Multiple models can co-exist at the same time and are supported by an extensible, template-based type system for the definition of the trace format, trace processing and the in-memory data model. The four main components of the tool are:

– *A type system*, offering a declarative description of on-disk and in-memory tracing data structures and their relationships, from which functions for creation, management, storage and processing of trace data are generated.
– *A tracing library* that defines set of functions to create, write and read Aftermath trace files used for the instrumentation of runtimes and for building data capturing tools.
– *A rendering library* providing a set of functions to visualize trace data in graphical user interfaces or tools for bulk rendering.
– *A configurable graphical user interface (GUI)*, used for trace inspection and performance analysis by the end user. The GUI is defined by a customizable interface file that assembles different graphical widgets and a data-flow graph for trace processing, both of which can be modified on-the-fly during execution. Multiple GUI definitions and data-flow graphs can co-exist, providing specialized tools for specific frameworks or specific types of analyses.

It is worth noting that the timeline in the Aftermath GUI is hierarchical and nodes (e.g., worker threads, cores) can be collapsed, so that statistics shown on a

[3] https://www.aftermath-tracing.com/

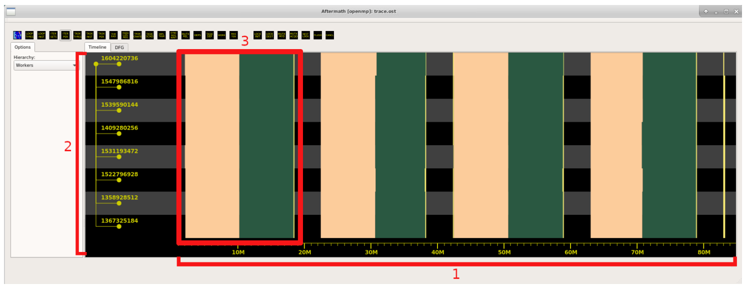


Fig. 1: Visualization of iteration periods from Listing 1.1 in the Aftermath GUI: (1) Timeline; (2) Worker Threads/Cores; (3) Execution of a single loop instance

lane are the accumulated values. As an example, consider three CPUs, with the first one spending 40% of the time in function f, the second one spending 30% of the time also in f, and the third one spending 60% of the time in function g for the same time interval associated to a pixel. The non-collapsed view would show three lanes: two with the pixel in the colour associated to f and another one with the pixel in the colour associated to g. When collapsed, the dominant function becomes f and the pixel would be rendered with the colour of f. Using this approach a large number of cores can be divided into smaller groups and the user can expand/collapse nodes to adjust granularity of the displayed data.

Previous support for OpenMP in Aftermath relied on a specific, instrumented OpenMP runtime to generate the trace files. In this paper, we present an OMPT-based tool that can be used to trace any runtime supporting OMPT. To this end, we extended the Aftermath type system to represent OMPT events and implemented callbacks with calls to Aftermath's tracing library. Since our OMPT-based tool is intended to entirely replace the legacy OpenMP support in Aftermath based on the instrumented runtime library, we refer to both our tool and Aftermath simply as Aftermath for the remainder of the paper.

Throughout the paper, we use the terms *loop instance*, *iteration set*, *iteration period*, *task instance* and *task period* introduced in [4] and defined as follows. We say that a parallel loop is *instantiated* when control flow reaches the instructions associated with a static definition of a parallel loop in the source code and its iteration space is distributed across workers according to its scheduling strategy. Each such encounter is defined as a *loop instance*. Similarly, we speak of a *task instance* when referring to the dynamic instructions executed by a task statically defined in the source code. The iteration space of a loop instance is split into *iteration sets*, each of which is assigned to exactly one worker. The iteration set corresponds directly to the loop chunk. The execution of an iteration set consists of one or more *iteration periods*, defined as contiguous intervals of execution of dynamic instructions of the loop associated to the loop instance. For flat loops each *iteration set* consists of exactly one *iteration period*, that corresponds to the execution of a specific loop chunk. For loops containing loops' nests the *iteration set* is split between multiple *iteration periods*, representing execution on the given nest level. Similarly, execution of a *task instance* is split into one or more *task periods*.

To illustrate these concepts, consider the Listing 1.1 where two loops are each executed twice. This results in four *loop instances* in total—two for the first loop and two for the second loop. Each instance is split into 16 distinct *iteration sets* with each set containing two iterations invoking the function `foo()`. Since the loop body does not contain nested parallel regions and does not spawn tasks, each *iteration set* will be associated with one continuous *iteration period*.

A visualization of the execution is given in Figure 1. The four distinct loop instances are presented as beige and green regions executing on a total of 8 worker threads, each identified by their thread ID on the left side of the figure. Those coloured regions represent alternating *iteration periods*, in this case corresponding directly to loop chunks (*iteration sets*). The thin yellow line after each loop instance represents the loop's implicit barrier, while the rightmost yellow line represents the barrier at the end of the parallel section.

3 Tracing Using OMPT Callbacks

In this section, we present the implementation of AfterOMPT based on the OMPT interface. We discuss which OMPT callbacks are used and what information is captured through the interface. We also present workarounds for cases in which additional information is required, but is not provided by OMPT.

3.1 Labeling instances

The ability to associate dynamic events with specific instances of OpenMP constructs and to combine the data captured from multiple callbacks requires a mechanism to reliably identify particular instances. AfterOMPT implements a labeling mechanism for this, associating each instance of a supported OpenMP construct with a unique label that identifies it. Each label is composed of two components: the thread ID of the worker instantiating the construct and the value of the worker's monotonically increasing sequence counter. Since the thread ID and the counter value are unique and private to a worker, workers can generate labels independently and concurrently without any need for synchronization. Once an instance has been created, its label is stored within an associated task data structure. Any related event that is captured through an OMPT callback function afterwards has access to the task's data and can thus store the serialized event data along with a reference to the instance in the trace file.

3.2 Tracing loops

The current OMPT interface provides very limited information on loops via the `ompt_callback_work` callback function. Neither the loop bounds nor the partitioning of the iteration space into chunks are exposed, which prevents tracing the distribution of the iteration space using OMPT alone. While this issue was identified in [10], and we base our work on that proposal, we propose further changes to the callback signatures necessary to generate more complete traces.

```
typedef void (*ompt_callback_loop_begin_t) (
  ompt_data_t* parallel_data, ompt_data_t* task_data,
  int flags,
  int64_t lower_bound, int64_t upper_bound,
  int64_t increment,
  int num_workers,
  void* codeptr_ra);

typedef void (*ompt_callback_loop_end_t) (
  ompt_data_t* parallel_data, ompt_data_t* task_data);
```

Listing 1.2: Callback signatures for loop tracing

For loop tracing, AfterOMPT uses two callbacks with signatures as defined in Listing 1.2, one invoked at the beginning of the loop and one invoked at the end of the loop. In line with the work of Langdal et al. those callbacks replace the current work callback whenever a loop is executed. However, rather than using the `endpoint` argument as the authors proposed, we use two distinct callbacks with names ending with `*_begin` and `*_end`. This simplifies the implementation by reducing the data required to be traced at the end of the loop, as work-sharing information is instead provided implicitly through the bounds, the increment, the number of workers and the flags indicating the schedule. This defines a compact representation from which the distribution across workers can be recovered by the callback function, compared to an explicit set of chunks and distribution.

The `codeptr_ra` argument refers to an address of an instruction of the loop body and can be used as a unique identifier for the source code location of the instantiated loop construct.

In order to trace loop chunks, we propose an additional callback function with the signature shown on Listing 1.3. In contrast to the signature proposed by Langdal et al., we do not include a parameter marking the final chunk, since this chunk is always followed by the loop end event and can thus be recovered post-mortem. We also omit the loop chunk creation time parameter, as we currently do not use it, however we aim to investigate potential use cases in the future.

Using this information, iteration sets can be recovered by mapping each occurrence of the loop chunk event into the new iteration set. To recover iteration periods we consider four cases: (1) The new period starts when the chunk gets dispatched and finishes when the next chunk gets dispatched; (2) The new period starts when the chunk gets dispatched and finishes when the loop ends; (3) The new period starts when the loop at the nest level n ends and finishes when the loop at the nest level $n-1$ ends; (4) The new period spans the execution time between the end of one loop and the start of the another loop at the same nest level, e.g., `for{ for{} /* Period (4) */ for{} }`

3.3 Tracing tasks

While the detailed tracing of loops requires an extension of the OMPT interface, tasks can be traced using the existing callbacks `ompt_callback_task_create` and `ompt_callback_task_schedule`. These events are captured as discrete events

```
typedef void (*ompt_callback_loop_chunk_t) (
  ompt_data_t* parallel_data, ompt_data_t* task_data, int64_t
  lower_bound, int64_t upper_bound);
```

Listing 1.3: Callback signature for the tracing of loop chunks

in the trace, with the task instance beginning and end events associated post-mortem, to reduce the run-time overhead. All instances of a given task construct can also be retrieved post-mortem by iterating over all task-creation events for the task's address, as provided by the `codeptr_ra` parameter within the task creation callback function. Tasks periods can be easily determined from scheduling points captured through the task schedule callback.

3.4 Tracing synchronization events and regions

Barriers, taskwait states, critical sections, master, single and parallel regions can be accurately traced by recording the information provided by the associated OMPT callbacks and by matching the invocations of the callbacks indicating the beginning of an event with the invocation of the callback indicating its end.

4 Case Studies

We now present two case studies using the tracing interface and show that AfterOMPT provides performance insights which are unavailable to developers without the proposed extensions of the OMPT interface for loop-related callbacks. In the first study, we show how an uneven distribution of work across the iteration space of a parallel loop can be inspected with our tool. The second study shows how AfterOMPT can be used to assess the effect of pipeline parallelism on the performance of an application.

4.1 Experimental Setup

We implemented new callbacks for dynamic loops and loop chunks, and static loops in the LLVM 9.0 OpenMP runtime[4]. This work is based on the implementation of the Aftermath instrumented OpenMP runtime [4].

For static loops, each worker can determine its part of the iteration space independently from the others, solely based on its thread ID, the chunk size, the loop bounds and the loop increment. Since this does not require invocation of the OpenMP runtime, static loops cannot be traced from within the runtime and require static instrumentation by the compiler. We have therefore used the modified version of Clang proposed in [10], where the compiler inserts the required callback directly into static loops within the application.

An alternative approach, not used in this paper, that does not require the modified compiler involves setting the compile-time schedule to *runtime* (with `schedule(runtime)`) and then runtime schedule to *static* with a specific chunk

[4] Artifacts and sources available at: https://github.com/IgWod/ompt-loops-tracing

size. This forces the application to distribute the work using the runtime functions—the same ones that are used by the dynamic scheduling.

For trace recording, processing and visualization, we have extended the latest branch of Aftermath with new types representing OMPT events. To leverage the existing OpenMP support in Aftermath and to avoid code duplication with our new OMPT-based implementation, we have also added an extra processing step in the Aftermath GUI that converts OMPT events into native Aftermath OpenMP types. Finally, AfterOMPT comes as a standalone library that implements required callbacks with Aftermath tracing API to capture required data.

All experiments have been carried out on a platform with two Intel Xeon Silver 4116 processors, each of which has 12 cores (24 threads) running at 2.10 GHz. The 112 GiB memory is split across 2 NUMA nodes. The system was running Ubuntu 18.04.4 LTS with kernel version 4.15 and Hyper-Threading enabled.

For the case studies, we have limited the execution to 12 threads (6 physical cores) on a single socket in order to improve readability of the visualized traces and to exclude any NUMA-specific effects. The subsequent overhead analysis has been carried out using all 48 threads (24 physical cores).

4.2 Identifying slow iterations in unbalanced loops

The first case study demonstrates how AfterOMPT's loop tracing capabilities can be used to inspect a non-uniform distribution of work across the iteration space of a parallel loop. We illustrate this on the integer bucket sort (*IS*) from the NAS Parallel Benchmark suite [3], version 3.4 with a custom data set.

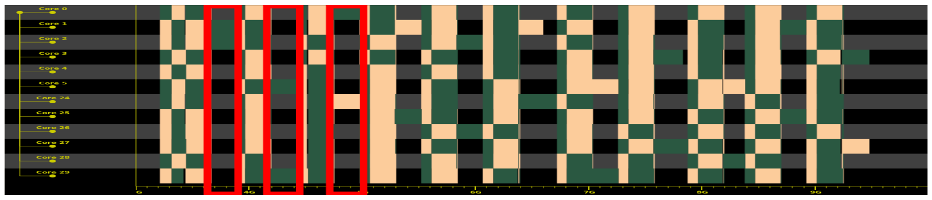
The bucket sort algorithm sorts a sequence of N integer values by distributing these values into K buckets, sorting each bucket individually and concatenating the sorted buckets into a final, sorted sequence. The distribution into the buckets is based on the maximum value $V_{\max}$ of the input sequence: a value v is put into the bucket with the index $\left\lfloor (K-1) \cdot \frac{v}{V_{\max}} \right\rfloor$. The amount of work required to sort a bucket depends on the number of values in the bucket, which in turn depends on the distribution of values of the input sequence.

The *IS* benchmark consists of three parallel loops in the main processing function. Two loops distribute keys into buckets and one loop sorts one bucket per iteration. In the following analysis, we show how AfterOMPT can be used to determine an uneven data distribution in the *IS* benchmark and to determine for which iterations the amount of work differs substantially.

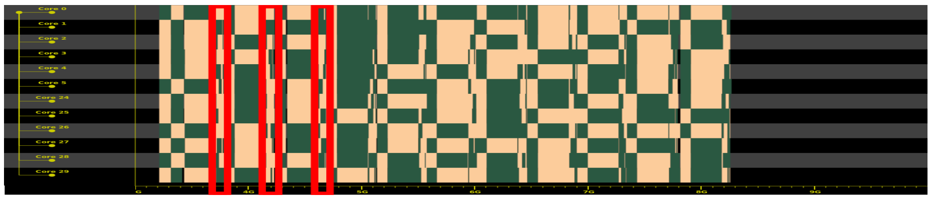
To this end, we have first changed the range of generated integer values to $(1048576, 1064960)$[5]. Since this range does not start at zero and does not end at the hard-coded $V_{\max}$ of the implementation, the buckets for low and high values remain empty, while the buckets for "medium" values each receive a significant part of the keys.

With the default parameters, the execution takes 2.58 s for the input class C and the input range adjusted to the interval above. A visualization of the

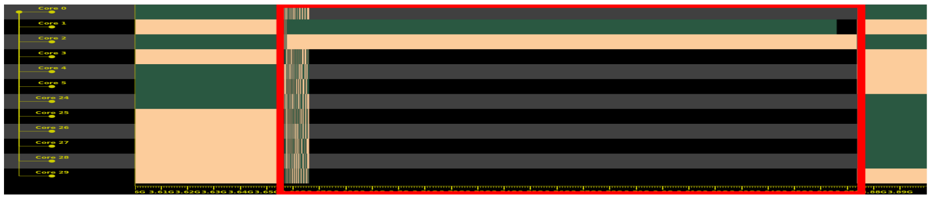
[5] Partial verification of this changed dataset fails as it relies on pre-defined ranks for keys at specific locations, but full verification passes, so that we can assume that the algorithm executes correctly.



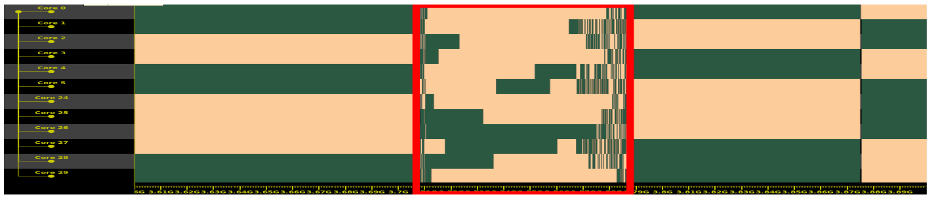
(a) Before (full program)



(b) After (full program)



(c) Before (sub-optimal *loop instance*)



(d) After (optimized *loop instance*)

Fig. 2: *Iteration periods* before and after changing the number of buckets in *IS* with bucket sorting loop instances marked in red

loop iteration intervals from the execution trace is given in Figure 2a. We have outlined in red the first three loop instances which are sorting the buckets. For each such instance, only two workers have significant iteration periods, indicated by the green and beige intervals within the red rectangles. For the remaining workers the visualization shows the alternating black and gray of the background, which means that these workers are mostly idle. Outside of the red rectangles, the loops process keys with a constant amount of work per iteration.

The zoomed visualization on the first imbalanced loop instance given in Figure 2c confirms the imbalance and clearly shows that two iterations are significantly slower than most of the remaining iterations. Further inspection with our tool shows that these are the iterations for buckets 128 and 129. The remaining

1022 iterations processing the buckets 0 to 127 and 130 to 1023 are very short, since these sort empty, or almost empty buckets.

The performance can be easily improved, simply by increasing number of buckets from 1024 to 4096 as this effectively distributes the work for a single bucket from the initial settings to more buckets and thus more workers. With a higher number of buckets, the execution time can be reduced to 2.11 s, which corresponds to a speedup of 1.22×. The absence of large gaps in the visualization of the trace in Figure 2b confirms the improved work balance. The long iteration periods from the original settings could be reduced by a factor of 3 with the increased number of buckets.

In conclusion, the visualization of *iteration periods* helped identifying the loop imbalance and allowed for attribution of the intervals to specific iterations of a specific parallel loop in the code. Repeated tracing with changed settings further allowed for rapid qualitative and quantitative evaluation of the changes.

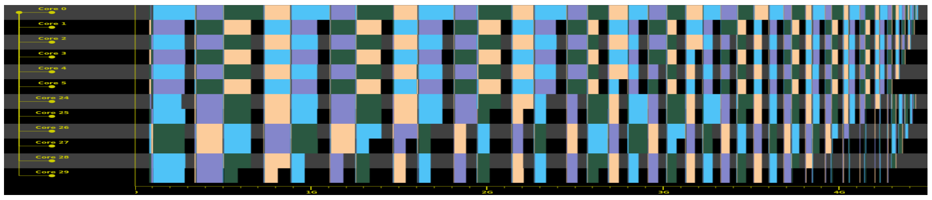
4.3 Comparison of loop-based and task-based implementations

In the second case study, we illustrate how AfterOMPT can be used to evaluate and compare different implementations of the same benchmark. We use the *SparseLU* benchmark of the Barcelona OpenMP Task Suite (BOTS) [6] and compare two loop-based versions, using different schedules, with the unmodified, task-based implementation. We first investigate the effect of the loop schedule in our modified versions on the performance, before assessing the effects of pipeline parallelism of the task-based version.

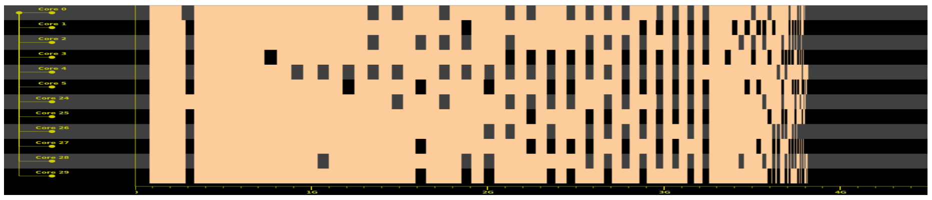
We produce a first loop-based implementation from the benchmark by commenting all task pragmas in the *for-omp-tasks* version of the application. This results in a benchmark whose parallelism is exposed solely through parallel loops with the default schedule, synchronized with barriers.

The execution time on our test system for this version is 2.08 s for default input size ($S1 = 50 \times 50$, $S2 = 100 \times 100$). The visualization of the execution trace provided in Figure 3a reveals significant work imbalance. Inspection of the iteration periods shows that the bulk of the execution time is spent in the code region executing the `bmod` function.

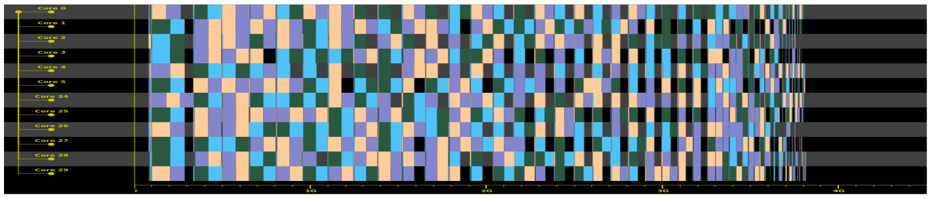
To mitigate the work imbalance, we have changed the default static schedule (no schedule specified) to a dynamic schedule with a chunk size of a single iteration (`schedule(dynamic, 1)`). This decreases the execution time to 1.81 s, corresponding to a speedup of 1.15×. Although this represents a significant improvement, the gaps in the visualization of the execution trace after modification shown in Figure 3b indicate that there is still potential for improvement. To identify the cause of the remaining imbalance, we investigate the *iteration periods* shown in Figure 3c. The duration of the periods is relatively uniform, indicating that the distribution of work is even across iterations. However, the barriers between loop instances have a significant impact as they cause a significant fraction of the workers to idle if the number of available iterations is not a multiple of the number of workers. Furthermore, the available parallelism decreases over time, leaving more and more workers idle towards the end of the execution. Since the



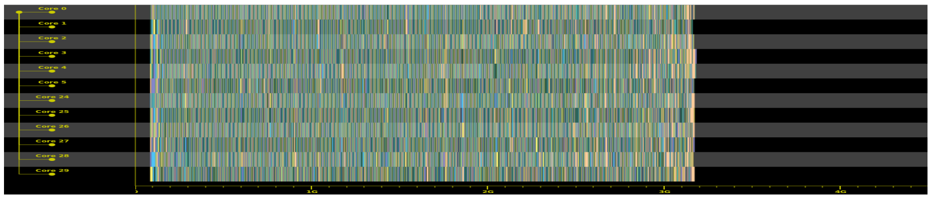
(a) *Iteration periods* (static schedule)



(b) *Loop instances* (dynamic schedule)



(c) *Iteration periods* (dynamic schedule)



(d) Tasks of task-based implementation

Fig. 3: Traces for loop-based and task-based implementations of *SparseLU*

number of iterations is data-dependent, any statically configured chunk size or loop schedule will lead to imbalance for certain problem instances.

The original, unmodified version the benchmark uses the parallel loops only to spawn parallel tasks. The barrier only synchronizes task creation, but not completion, thus exposing pipeline parallelism which allows all the workers to be kept busy for most of the time (Figure 3d) and reduces the execution time to 1.47 s (1.41× speedup). This shows that pipelining parallelism in the original implementation has a significant impact on performance.

5 Overhead Analysis

To obtain meaningful traces, it is crucial that the tracing mechanism does not perturb the execution of the application. In this section, we evaluate the tracing overhead using selected applications from BOTS [6] and the C implementation[6] of the NPB 2.3 [3] benchmarks. In our experiments, we trace threads, task creation, task execution, the beginning and end of loops, and the beginning and end of the execution of loop chunks via the callback functions `thread_{begin,end}`, `task_create`, `task_schedule`, `loop_{begin,end}` and `loop_chunk`.

To stress the tracing mechanism for loops, we selected *CG*, *EP*, *LU*, *MG* and *SP*, excluding *BT* and *FT* as they failed to build[7], as well as *IS* as it does not report its execution time.

For task-based benchmarks, we selected *alignment*, *fft*, *fib*, *floorplan*, *health*, *nqueens*, *sort*, *sparselu* and *strassen* from BOTS. The *uts* benchmark was excluded as we encountered frequent application segmentation faults when running it on the experimental machine. We used the *omp-tasks-tied* version for the BOTS benchmarks, except *alignment* and *sparselu* for which this version was unavailable, and the *for-omp-tasks-tied* version was used instead.

Each benchmark was executed with default values, except for *fib* where N was increased to 35 to avoid the high variation of the very short execution for the default value. The largest available input files were used for the BOTS benchmarks that require input files, except for *uts*, where *small.input* was used. The NPB benchmarks operated on the C input class, with the exception of *SP* which was given the A input class in order to avoid excessive experiment duration. Each benchmark was executed 50 times, where for the same reason *SP* was instead executed 20 times.

Figure 4 shows the relative mean increase of the execution time when tracing is enabled, compared to the execution without tracing. The reported values were obtained by dividing the execution time of each run of the benchmark with the tool attached, by the mean execution time of 50 runs of the baseline (no tool attached). The value above each bar indicates the mean relative change and error bars indicate the standard deviation.

The relative overhead for three of the loop-based NPB benchmarks, *CG*, *EP* and *MG*, was very low, with all three recording an increased execution time of under 3%. A higher relative overhead was recorded for the remaining two NPB benchmarks *LU* and *SP*: averaging $6.0\% \pm 4.0\%$ for *LU*, and $35.2\% \pm 5.7\%$ for *SP*. The increased relative overhead for these benchmarks is due to their large number of very fine-grained loop-chunks (especially for *SP*), resulting in a large number of invoked callbacks relative to the overall work done.

The overhead results varied across the task-based BOTS benchmarks, with values under 3% for *alignment*, *fft*, *sort*, *sparselu* and *strassen*, and values up to 24% for the remaining benchmarks. As with the NPB results, the more significant relative overheads resulting for these benchmarks is due to their large number of short-lived task instances, thereby invoking significantly more tracing-callbacks relative to their workload. Analysis of the *floorplan* benchmark shows that the average duration of an AfterOMPT tracing-callback was around 200

[6] https://github.com/benchmark-subsetting/NPB3.0-omp-C

[7] The compilation error is caused by the potential bug in the unofficial C port of the benchmarks and does not appear in the official Fortran implementation.

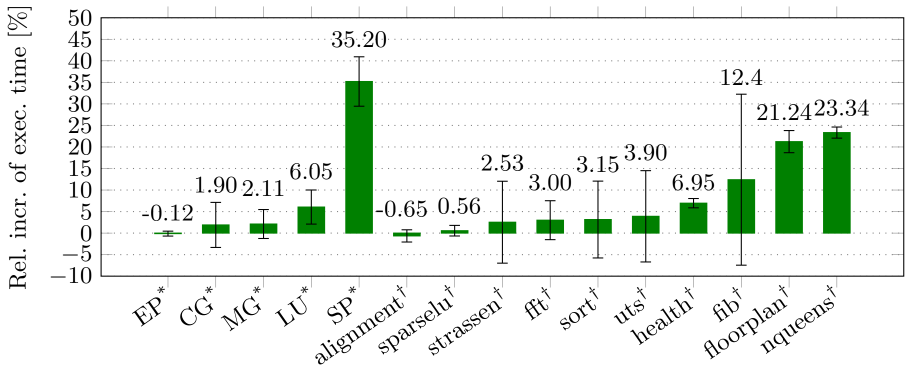


Fig. 4: Profiling overhead for the selected benchmarks from NPB 2.3 and BOTS (*loop-based, †task-based)

cycles, compared to the average total duration of a task instance (including the tracing overhead) of around 2400 cycles. All of these measures include the overhead of the OMPT interface itself. For more details, including cases with empty callbacks and OMPT disabled, we refer to [10].

In conclusion, the average overhead of AfterOMPT was found to be under 5% for the majority of the benchmarks (9 out of 15), increasing to under 7% for two further benchmarks, and greater than 10% for only four of the most fine-grained benchmarks. As the AfterOMPT tracing infrastructure incurs an overhead of only around 200 cycles per callback invocation, its generally low impact on the overall program execution time—while dependent on the workload granularity—means that it is highly suitable for tracing and analysing many target OpenMP applications. Moreover the overhead does not depend on number of threads, as synchronization within the tool is kept to minimum—one critical section per thread initialization —meaning each thread is traced independently.

6 Related Work

Langdal et al. [10] were first to investigate extending the OMPT interface with loop related callbacks. While they discuss implementation details and overhead analysis for the potential new loop callbacks, they do not provide detailed use cases to support proposed changes. We complement their work by presenting detailed scenarios, showing how information associated with those callbacks are useful in practice.

Their work was done in the context of Grain Graphs [12]. Compared to Aftermath, Grain Graphs is a chronogram-based application that represents OpenMP programs in a hierarchical graph form. It allows detection of limited parallelism, load imbalance and synchronization issues, however the visual representation does not attribute profiled constructs to specific cores. Aftermath presents traces on per worker timelines allowing to detect additional anomalies, such as problems related to NUMA.

Score-P [11] [8] is a profiling and event tracing infrastructure for HPC applications. It allows tracing of OpenMP applications with either by POMP2 [9] instrumentation using source-to-source compiler or with OMPT interface. It generates traces in the formats (OTF2 or CUBE4) compatible with several analysis tools such as Vampir [13] or TAU [13] toolkit. However Score-P does not support tracing of loops using OMPT with granularity offered by AfterOMPT.

Extrae [1], a tool for capturing execution trace with interfaces for MPI, OpenMP, pthreads, OmpSS and CUDA. Captured data can be later viewed with Paraver [16] visualization tool. Although data collection using OMPT interface is supported, it has the same limitations as Score-P.

Finally Intel VTune [2] does not support OMPT and uses VTune instrumentation API in the OpenMP runtime in addition to the sampling based profiling.

7 Conclusion and Future Work

We presented AfterOMPT, an OMPT-based tool for tracing, visualization and performance analysis of OpenMP applications that is portable across OpenMP runtimes. We motivated an extension of the OMPT interface that allows for fine-grained analysis of parallel loops. We showed that our tool allows for a detailed analysis of both loop-based and task-based applications. With a tracing overhead as little as 200 cycles per OMPT callback function, the resulting increase in the execution time is less than 5% for many benchmarks and only leads to a significant increase for very fine-grained work sharing. In the future we plan to extend our tool further with the visualization of tasks trees and also integrate OpenMP hardware event profiling proposed before in [14].

References

1. Extrae. https://tools.bsc.es/extrae, Accessed: 2020-05-25
2. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html, Accessed: 2020-05-25
3. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., et al.: The NAS parallel benchmarks. The International Journal of Supercomputing Applications **5**(3), 63–73 (1991)
4. Drebes, A., Bréjon, J.B., Pop, A., Heydemann, K., Cohen, A.: Language-centric performance analysis of OpenMP programs with Aftermath. In: International Workshop on OpenMP. pp. 237–250. Springer (2016)
5. Drebes, A., Pop, A., Heydemann, K., Cohen, A.: Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 274–283. IEEE (2016)
6. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: 2009 international conference on parallel processing. pp. 124–131. IEEE (2009)

7. 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: International Workshop on OpenMP. pp. 171–185. Springer (2013)
8. Feld, C., Convent, S., Hermanns, M.A., Protze, J., Geimer, M., Mohr, B.: Score-P and OMPT: Navigating the perils of callback-driven parallel runtime introspection. In: International Workshop on OpenMP. pp. 21–35. Springer (2019)
9. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y., Lin, Y.: An OpenMP runtime API for profiling. OpenMP ARB White Paper (2007), Available online at http://www.compunity.org/futures/omp-api.html
10. Langdal, P.V., Jahre, M., Muddukrishna, A.: Extending OMPT to support grain graphs. In: International Workshop on OpenMP. pp. 141–155. Springer (2017)
11. Lorenz, D., Dietrich, R., Tschüter, R., Wolf, F.: A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P. In: International Workshop on OpenMP. pp. 161–172. Springer (2014)
12. Muddukrishna, A., Jonsson, P.A., Podobas, A., Brorsson, M.: Grain graphs: OpenMP performance analysis made easy. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–13. ACM (2016)
13. Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing scalable applications with Vampir, VampirServer and VampirTrace. In: PARCO. vol. 15, pp. 637–644 (2007)
14. Neill, R., Drebes, A., Pop, A.: Accurate and complete hardware profiling for OpenMP. In: International Workshop on OpenMP. pp. 266–280. Springer (2017)
15. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Version 5.0) (2018)
16. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31 (1995)