# Extending OMPT to support Grain Graphs

Peder Voldnes Langdal, Magnus Jahre, and Ananya Muddukrishna

Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
pedervl@stud.ntnu.no, {magnus.jahre,ananya.muddukishna}@ntnu.no

**Abstract.** The upcoming profiling API standard OMPT can describe almost all profiling events required to construct *grains graphs*, a recent visualization that simplifies OpenMP performance analysis. We propose OMPT extensions that provide the missing descriptions of task creation and parallel for-loop chunk scheduling events, making OMPT a sufficient, standard source for grain graphs. Our extensions adhere to OMPT design objectives and incur up to 1% overhead for BOTS and SPEC OMP2012 programs. Although motivated by grain graphs, the events described by the extensions are general and can enable cost-effective, precise measurements in other profiling tools as well.

**Keywords:** OMPT, performance analysis, performance visualization

## 1 Introduction

Programmers are required to write parallelized code to take advantage of the multiple cores and accelerators exposed by modern processors. The OpenMP standard API [3] is among the leading techniques for parallel programming used by programmers. All programmers have to do is incrementally insert OpenMP directives into otherwise serial code. The directives are translated by compilers into parallel programs that are scheduled by runtime systems.

Getting OpenMP programs to perform well is often difficult since programmers work with limited information. Program translation and execution happens in the background driven by compiler and runtime system decisions unknown to programmers. Performance visualizations depict these background actions faithfully and do a poor job of connecting problems to code semantics understood by programmers.

The grain graph is a OpenMP visualization method for OpenMP that shows performance problems on a fork-join graph of *grains* – task and parallel for-loop chunk instances [14]. Problem diagnosis becomes effective since programmers can easily match the fork-join structure to the code they wrote. Insightful metrics derived from the graph guide optimization decisions. The graph is constructed post-execution using profiling measurements from the MIR runtime system [13].

We have previously found [10] that except for task creation and parallel for-loop chunk scheduling, the upcoming OpenMP Tools API (OMPT) [7,18] standard can describe all profiling events required to obtain measurements for grain graphs. In the paper, we propose OMPT extensions that provide the missing descriptions. Our extensions adhere to the design objectives of OMPT and have a low overhead for standard benchmarks and programs from EPCC [1, 2] (up to 3% overhead for schedbench excluding statically scheduled loops with small chunks, 1.7% for taskbench), BOTS [6] (1%) and SPEC OMP2012 [16] (1%). Although our extensions are motivated by grain graphs, the events they describe are general and can enable cost-effective, precise measurements in other profiling tools as well.

## 2 Background

We explain required background information on OMPT and grain graphs in the section.

### 2.1 OMPT

The OpenMP Tools API (OMPT) [7,18] is an upcoming addition to the OpenMP specification to enable creation of portable performance analysis tools. OMPT supports asynchronous sampling and instrumentation-based monitoring of runtime events.

Tools based on OMPT, hereafter simply called tools, are a collection of functions that reside in the address space of the program being profiled. During startup, the runtime system looks for and calls a tool-provided function called `ompt_start_tool`, which returns pointers to the tool's initialization and finalization functions. It then calls the initialization function, which in turn registers callback functions with the runtime system to be called at specific events such as starting a thread, starting a worksharing region, task creation, and task scheduling.

The foremost design objectives of OMPT [7] are:

- Tools should be able to obtain adequate information to attribute costs to application source code and the runtime system.
- OMPT support incorporated in an OpenMP runtime system should add negligible overhead when no tool is in use.

### 2.2 Grain graphs

The grain graph is a recent visualization method for OpenMP that works at the level of task and parallel for-loop chunk instances, collectively called *grains* [14]. The graph captures the fork-join program progression familiar to programmers by placing parent and child grains in close proximity without timing as a placement constraint. Grains with performance problems such as work inflation,

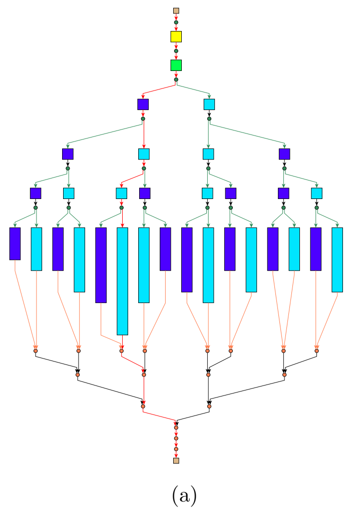
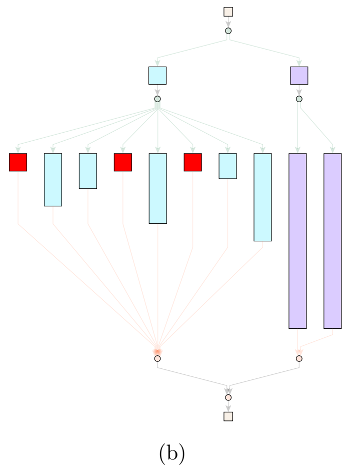

Fig. 1: Example grain graphs. (a) Graph of BOTS Fibonacci program for small input (n=32,cutoff=4). Grain colors encode definition site in source code. (b) Graph of a simple OpenMP parallel for-loop with 10 iterations executed on two threads with the `dynamic` schedule. Grain colors encode the worker thread. Chunks with low parallel benefit are highlighted in red.

inadequate parallelism, and low parallelization benefit are pin-pointed on the graph. Example grain graphs are shown in Figure 1.

The grain graph visualization is implemented in a reference prototype [15] that relies on detailed profiling measurements from the MIR runtime system [13]. Per-grain metrics from MIR such as execution time and parallelization cost are combined with the grain graph structure to derive metrics that guide optimizations.

*Parallel benefit* is a derived metric equal to a grain's execution time divided by its parallelization cost including creation time. Parallel benefit aids inlining and cutoff decisions by quantifying whether parallelization is beneficial. Grains with low parallel benefit should be executed sequentially to reduce overhead.

## 3 Extending OMPT

We propose two extensions to make OMPT a sufficient source for descriptions of profiling events required to construct grain graphs. The first extension enables measuring time spent in creating task instances. Task creation time is required to derive the parallel benefit metric of grain graphs. The second extension describes detailed parallel for-loop execution events including chunk assignment, enabling performance analysis at the chunk-level – a key feature of grain graphs. Both extensions adhere to OMPT design objectives (Section 2.1) and separate concerns similar to the rest of the interfaces. More details about the extensions follow.

## 3.1 Task Creation Duration

Creating a task instance typically involves pushing it into a task queue after allocating and initializing book-keeping data structures. This can take an uneven amount of time subject to memory allocation latencies and queue contention. An existing callback in OMPT called `ompt_callback_task_create` can notify tools that task creation is taking place. However, it does not allow measuring the duration of the process. Allowing tools to determine per-task creation time enables precise guidance about inlining and cutoffs. Also, situations where task creation duration estimates computed by tools are outdated or mismatched with the runtime system can be avoided.

To extend OMPT with the ability to inform tools about task creation duration, we considered three alternative approaches:

1. Add an endpoint parameter to the `ompt_callback_task_create` callback, and let the callback be invoked both at the start and end of task creation. This enables tools to measure the time between calls at the expense of changing the signature and the semantics of an existing callback.
2. Introduce a new callback that denotes the beginning of task creation and let the existing callback `ompt_callback_task_create` be called at the end of task creation. This approach differs from the first in that it avoids changing the signature of an existing callback but introduces a new one.
3. Measure the task creation duration inside the runtime system and report it to the tool as an extra parameter to `ompt_callback_task_create`. The advantage of this approach is that it avoids an additional callback invocation before each task creation event. However, it forces tools to agree on the notion of time. Some tools may require time measured in processor cycles, while others may only need microsecond precision. To complicate things further, the runtime system may decide to measure elapsed processor cycles using a hardware performance counter – a scarce resource for tools. In the case that multiple cycle counters exist, the tool would not necessarily know which counter is used by the runtime system.

We chose the third approach because it reduced callback overhead. The time agreement disadvantage was solved by allowing tools to register a function in tool-space that returns the current time. This function is called by the runtime system before and after task creation, and the difference between the two time values is returned as a parameter.

Our design for the task creation duration extension has the following new function signatures:

```
// The signature of the new ompt_tool_time callback to
// register a tool-space time function
```

```
typedef double (*ompt_tool_time_t) (void);
// The proposed new signature to ompt_callback_task_create
typedef void (*ompt_callback_task_create_t) (
  ompt_data_t *parent_task_data,
  const ompt_frame_t *parent_frame,
  ompt_data_t *new_task_data,
  ompt_task_type_t type,
  int has_dependences,
  double event_duration,          // A new addition to return duration
  const void *codeptr_ra
);
```

The event_duration parameter is typed as a double-precision floating point number to give tools increased precision and be consistent with omp_get_wtime. If the tool has not registered an ompt_tool_time function, the event_duration is reported as 0. We chose to return 0 instead of falling back to a low-precision timer consistent with omp_get_wtime so that no extra timing overhead is incurred if tools opt out of registering a time function.

## 3.2 Extended For-loop Events

Currently, OMPT lacks interfaces to understand chunks. Parallel for-loop support is also meager. The existing loop-focused callback ompt_callback_work carries little information about looping parameters. Tools can help programmers correctly diagnose parallel for-loop problems if enabled with per-chunk metrics such as creation duration, execution duration, and iteration range, as demonstrated by grain graphs [14].

We propose extending OMPT with two new callbacks, one for chunks and the other for loops, that improve the quality of information provided at loop events to tools, enabling them to measure the execution time of individual chunks and map chunks to iterations or worker threads.

The signatures for the new callbacks are shown below.

```
// The proposed ompt_callback_chunk signature
typedef void (*ompt_callback_chunk_t) (
  ompt_data_t *task_data, // The implicit task of the worker
  int64_t lower,          // Lower bound of chunk
  int64_t upper,          // Upper bound of chunk
  double create_duration, // Interval found from tool-supplied instants
  int is_last_chunk       // Is it the last chunk?
);
// The proposed ompt_callback_loop signature
typedef void (*ompt_callback_loop_t) (
  omp_sched_t loop_sched,         // Actual schedule type used
  ompt_scope_endpoint_t endpoint, // Begin or end?
  ompt_data_t *parallel_data,     // The parallel region
  ompt_data_t *task_data,         // The implicit task of the worker
  int is_iter_signed,             // Signed loop iteration variable?
  int64_t step,                   // Loop increment
  const void *codeptr_ra          // Runtime call return address
);
```

The proposed callback `ompt_callback_chunk` is called before a chunk starts execution. It describes the iteration range and creation time of the chunk. Chunk creation time is calculated using a tool-space `ompt_tool_time` function if provided, similar to approach for tasks (Section 3.1). The information can be used by tools to identify chunks that execute shorter than their creation time and guide chunk size selection, as demonstrated by grain graphs.

The new loop callback `ompt_callback_loop` is meant to be called instead of the existing `ompt_callback_work` whenever a parallel for-loop is encountered. This callback provides additional loop-level information such as loop increment and the schedule type at runtime. Schedule type is not always set in the source code and can be decided by the compiler, runtime system, and environment variables. The `is_iter_signed` parameter is used to inform tools about the signedness of the iteration variable, so that tools can cast the iteration bounds reported by `ompt_callback_chunk` to the correct type.

The extensions require minimal changes to existing OMPT implementations. The `ompt_callback_work` callback is simply replaced by `ompt_callback_loop` in code that processes the `for` construct. Calls to `ompt_callback_chunk` should be made in runtime system functions that handle assignment of chunks to worker threads executing dynamically scheduled for-loops.

A relatively larger change is required to handle statically scheduled for-loops where worker threads calculate their chunk iteration ranges directly through code inserted by the compiler. In this case, compilers should additionally generate calls to `ompt_callback_chunk`, preferably through a call to the runtime system. Calling the runtime system for every chunk is expensive if chunk sizes are small. We avoid this overhead when there is no tool attached, or when the attached tool has not registered for the `ompt_callback_chunk` callback, by conditionally calling the runtime system as shown in the pseudocode snippet below.

```
bool callbackPerChunk = __omp_runtime_should_callback_per_chunk();
while (UB = min(UB, GlobalUB), idx = LB, idx < UB) {
    if (callbackPerChunk) {
        __omp_runtime_for_static_chunk(...)
    }
    for (idx = LB; idx <= UB; ++idx) {
            BODY;
    }
    LB = LB + stride; UB = UB + stride;
}
```

## 4 Evaluation

Evaluation of the proposed extensions is discussed in this section.

### 4.1 Experimental Setup

Our test machine has two Intel Xeon E5-2630 2.2Ghz 10-core processors. Each core has private 32KB L1 instruction and data caches, and a 256KB L2 cache. Each processor has a shared 25MB L3 cache. The system has 64GB RAM and runs CentOS Linux with kernel version 3.10. OpenMP threads are pinned to physical cores.

We selected a wide range of benchmarks to test the extensions. Our benchmark set consisted of *schedbench* and *taskbench* micro-benchmarks from the EPCC OpenMP micro-benchmark suite [1, 2] and programs from BOTS [6] and SPEC OMP2012 [16].

Benchmarks from schedbench and taskbench capture overhead of supporting parallel for-loops and tasks respectively. Both sets have the following parameters: *Outer repetitions* specifies how many times to repeat the test, *test time* specifies the target time for each test, and *delay time* specifies the busy-wait duration inside loop iterations and tasks. We parameterized schedbench with 50 outer repetitions, test time 30 ms, delay time 0.1 µs, and 4096 iterations per thread to produce the same conditions on our modern test system as the original authors of schedbench [1]. We used default parameters for taskbench except for increasing the number of outer repetitions to 50 and the test time to 30 ms to significantly reduce variance. We report median measurements of 20 runs for schedbench and taskbench benchmarks.

We included all programs from BOTS and C/C++ programs from SPEC OMP2012 in our benchmark set. We used large inputs when available, medium otherwise for BOTS programs and reference inputs for SPEC OMP2012. We report the median measurements of 20 and 12 runs for BOTS and SPEC OMP2012 programs respectively. Nested parallelism in 352.nab causes high execution time variance, so we run it with nested parallelism disabled.

We ran benchmarks with and without tools attached. Tools attached had two variants: a *no-callback tool* that registered no callbacks and a *test tool* that registered relevant callbacks but did not execute any code within.

Benchmarks, tools, and different versions of the LLVM OpenMP runtime were compiled using LLVM Clang version 4.0 with $-O3$ optimization. The default OpenMP runtime system of Clang 4.0 supports an outdated OMPT specification. We refer to this runtime system as *TR2* since it supports a subset of the OMPT Technical Report 2. The group behind OMPT has augmented TR2 with support for the more recent OMPT Technical Report 4 [17]. We refer to this runtime system as *TR4*. We modified TR4 to include our extensions and called it *TR4E*. We also modified Clang to generated code that supports the chunk scheduling extension in statically scheduled for-loops. This modified compiler was used to compile benchmarks that linked with TR4E. Our modifications in

TR4E and Clang are consistent with the prevailing implementation style and are publicly available for review [11].

| *Runtime system* | **for** construct | **task** construct |
|---|---|---|
| *TR2* | `ompt_event_loop_begin` `ompt_event_loop_end` | `ompt_event_task_begin` |
| *TR4* | `ompt_callback_work` | `ompt_callback_task_create` |
| *TR4E* | `ompt_callback_loop` `ompt_callback_chunk` | `ompt_callback_task_create` |

Table 1: Callbacks registered by test tools are runtime system specific.

Callbacks registered by the tools are shown in Table 1. These differ because the runtime systems support different versions of OMPT. TR2 did not have a direct equivalent to the `ompt_callback_task_create` callback of TR4. We used the TR2 callback `ompt_event_task_begin`, a close match called once before task execution.

## 4.2 Experimental Results

We compare overhead of supporting the extensions and attaching the test variant of tool. Overhead of attaching the no-callback tool variant are not discussed for space reasons. We refer to callbacks that describe parallel for-loop and chunk events as loop and chunk callbacks respectively.

Results of schedbench experiments are shown in Figure 2. TR2 has the lowest overhead since it supports an outdated OMPT implementation with fewer features. TR4E incurs less than 1% overhead over TR4 when no tool is attached, except for the guided schedule with chunk size 1 where the difference is 2.8%. With a tool that registers only for the loop callback, the overheads of TR4 and TR4E are very similar. Again, TR4E adds the most overhead, 2%, for the guided schedule with chunk size 1. The proposed chunk callback incurs negligible overhead when not used by tools. When used, TR4E incurs up to 3% higher overhead than TR4, except for the case of statically scheduled loops with chunk sizes below 32 where enabling the conditional per-chunk runtime system call (Section 3.2) incurs upto 50% overhead given the fine-grained nature of iterations.

Results of taskbench experiments are shown in Figure 3. We note that performance flaws of TR2 have been rectified in the TR4. TR4E adds negligible task creation overhead over TR4 since it requires less than 1% extra instructions. The highest increase in overhead is 1.7%, seen when no tool is attached with the MASTER TASK micro-benchmark.

Results of experiments with SPEC OMP2012 and BOTS programs when no tool is attached are shown in Figure 4. Alignment and SparseLU are present in both benchmark suites. We show variants from SPEC OMP2012 since they use

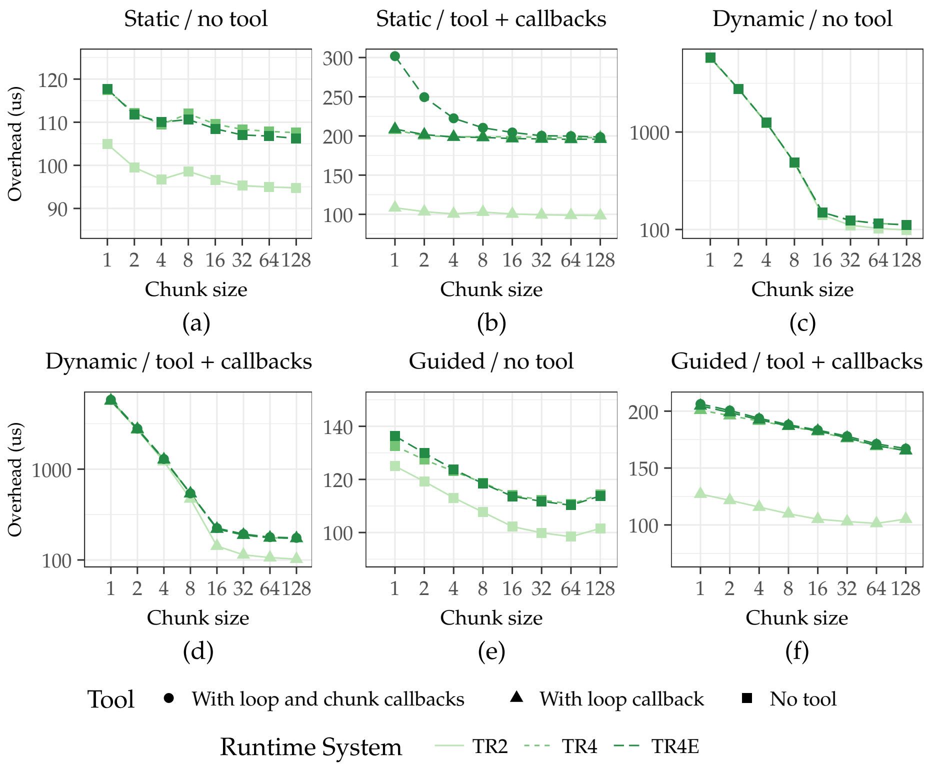

Fig. 2: Overhead of extensions measured with schedbench micro-benchmarks of EPCC are up to 3% except for statically scheduled small chunks that require a runtime system call per-chunk.

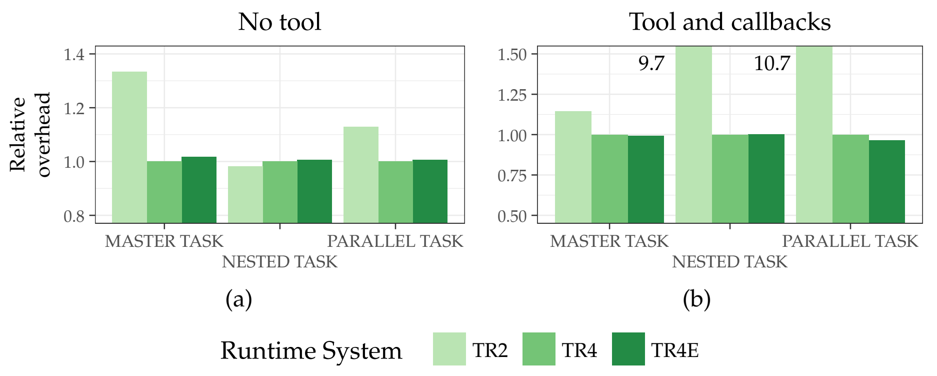

Fig. 3: Overheads of proposed task creation extension measured with taskbench micro-benchmarks from EPCC are negligible.

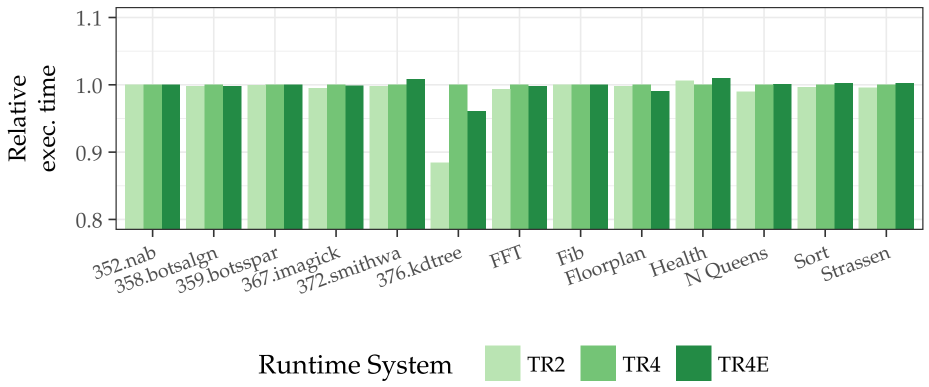

Fig. 4: Execution times with no tool attached for SPEC OMP2012 and BOTS programs are mostly identical. TR4 is the baseline. Programs with alpha-numeric names are from SPEC OMP2012.

larger inputs. 376.kdtree runs 4% faster due to incidental optimization opportunities used by the modified Clang 4 compiler. 372.smithwa and Sort runs 1% slower. Otherwise, execution times are mostly identical. Overhead is similarly low when attaching tools. We omit discussing this for space reasons.

## 5 Related Work

The main motivation for the proposed extensions is to construct grain graphs portably. However, the events described by the extensions have found use in other profiling APIs and tools.

The POMP API [12], a base for OMPT, included events to describe the start and completion of for-loop chunks.

Qawasmeh et al. [19] analyze timing and cache performance of runtime events including task creation to decide on optimal scheduling strategies in the OpenUH runtime system. They extend [20] the Sun/Oracle Collector API [9] to record the events. The task creation event in their design is described using separate start and stop events. The same two events are used by Servat et al. [21] for instrumenting the Nanos++ runtime system.

Drebes et al. [5] augment the LLVM OpenMP runtime system to collect parallel for-loop chunk traces. The traces are used to map chunks to worker threads in their Aftermath tool [4], enabling diagnosis of load imbalance problems. Unlike our extension, their implementation does not trace chunks of statically scheduled parallel for-loops – a dominant parallelization pattern. Excluding 367.imagick, 91/105 parallel for-loops in SPEC OMP2012 are statically scheduled.

Intel's VTune Amplifier [8] recently improved its OpenMP debugging feature set by characterizing loop schedules, chunk sizes, and time spent scheduling iterations. These are understood through source code inspection and sampling, provided profiled programs use Intel or GCC runtime systems. Our proposed OMPT extensions enable tools to portably compute similar metrics without need for source code inspection.

## 6 Conclusions

We presented extensions to OMPT that add a time duration parameter to the task creation callback, improve information provided by the loop callback, and introduce a new callback to describe chunk events, with the intention to construct grain graphs portably from any OMPT-compliant runtime system. Overhead incurred by the extensions is low – up to 3% for EPCC micro-benchmarks, excluding the use of the chunk callback for statically scheduled chunks of sizes below 32. Programs from BOTS and SPEC OMP2012 slowdown at most by 1%. The extensions adhere to OMPT design objectives, are implemented in a consistent, maintainable manner in a standard toolchain, and are publicly available [11]. Although motivated by grain graphs, the events described by the extensions are general and can enable cost-effective, precise measurements in other profiling tools as well.

## Acknowledgment

We thank Joachim Protze (RWTH Aachen), Jonas Hahnfeld (RWTH Aachen), and Harald Servat (Intel) for helpful comments and suggestions regarding the extensions. This paper is partially funded by the TULIPP project, grant number 688403 from the EU Horizon 2020 Research and Innovation programme.

## References

1. Bull, J.M.: Measuring synchronisation and scheduling overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP. vol. 8, p. 49 (1999)
2. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for OpenMP tasks. In: International Workshop on OpenMP. pp. 271–274. Springer (2012)
3. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5(1), 46–55 (1998)
4. 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 (Apr 2016)
5. 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)
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: Proceedings of the 2009 International Conference on Parallel Processing. pp. 124–131. ICPP '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/ICPP.2009.64

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: OpenMP in the Era of Low Power Devices and Accelerators. pp. 171–185. Springer, Berlin, Heidelberg (Sep 2013), dOI: 10.1007/978-3-642-40698-0_13
8. Intel: Intel VTune Amplifier Webpage (May 2017), `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
9. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y., Lin, Y.: An OpenMP runtime API for profiling. OpenMP ARB as an official ARB White Paper available online at http://www. compunity. org/futures/omp-api. html 314, 181–190 (2007), `http://www.compunity.org/futures/omp-api-old.html`
10. Langdal, P.V.: Generating Grain Graphs Using the OpenMP Tools API. Tech. rep., NTNU (2017), `https://brage.bibsys.no/xmlui/handle/11250/2434632`
11. Langdal, P.V.: LLVM OpenMP TR4E Alpha Release (May 2017), `https://doi.org/10.5281/zenodo.570288`, dOI: 10.5281/zenodo.570288
12. 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 Fourth Workshop on OpenMP (EWOMP 2002). pp. 1001–1025 (2002)
13. Muddukrishna, A., Jonsson, P.A., Langdal, P.: anamud/mir-dev: MIR v1.0.0 (Mar 2017), `https://doi.org/10.5281/zenodo.439351`
14. 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. 28:1–28:13. PPoPP '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2851141.2851156`
15. Muddukrishna, A., Langdal, P.: anamud/grain-graphs: Grain Graphs v1.0.0 (Mar 2017), `https://doi.org/10.5281/zenodo.439355`
16. Müller, M.S., Baron, J., Brantley, W.C., 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. IWOMP'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-30961-8_17`
17. OMPT Tools Interface Group: LLVM OpenMP Runtime with Changes Towards TR4. GitHub (2017), `https://github.com/OpenMPToolsInterface/LLVM-openmp`
18. OpenMP Language Working Group: OpenMP Technical Report 4: Version 5.0 Preview 1 (Nov 2016), `http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf`
19. Qawasmeh, A., Malik, A.M., Chapman, B.M.: Adaptive OpenMP Task Scheduling Using Runtime APIs and Machine Learning. In: 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA). pp. 889–895 (Dec 2015)
20. Qawasmeh, A., Malik, A., Chapman, B., Huck, K., Malony, A.: Open Source Task Profiling by Extending the OpenMP Runtime API. In: OpenMP in the Era of Low Power Devices and Accelerators. pp. 186–199. Springer, Berlin, Heidelberg (Sep 2013)
21. Servat, H., Teruel, X., Llort, G., Duran, A., Giménez, J., Martorell, X., Ayguadé, E., Labarta, J.: On the Instrumentation of OpenMP and OmpSs Tasking Constructs. In: Euro-Par 2012: Parallel Processing Workshops. pp. 414–428. Springer, Berlin, Heidelberg (Aug 2012)