Generating Grain Graphs Using the OpenMP Tools API

Peder Voldnes Langdal
Norwegian University of Science and Technology
pedervl@stud.ntnu.no

***Abstract*—Computers are becoming increasingly parallel. Many applications rely on OpenMP to divide units of work between a set of worker threads. Typically, this is done using parallel for-loops or tasking. Grain graphs is a recent method for visualizing program execution from a program perspective. It shows the control flow of a program in terms of fork and join points. Between such points, one can find grains, which are task instances or for-loop chunks. Attached to these grains are a set of metrics that inform the programmer of how they well they perform.**

However, generating grain graphs means using the MIR runtime system, as it is the only one designed to provide all the needed metrics. In this paper, I look at the OpenMP Tools API as an alternative. I show what data can be immediately obtained from the interface. Furthermore, I look at whether any metrics require extensions to OMPT. I find that most task-related metrics are available, but that information about for-loop chunks is missing from the API. I propose that an event for chunk scheduling is introduced, as well as an event denoting the end of task creation.

I. Introduction

Computers are rapidly moving towards architectures with multiple processing units. This necessitates creating programs that can utilize these units by having them perform work in parallel. Many techniques exist for doing this, one of which is OpenMP, an industry-standard for parallel shared-memory programming[1]. OpenMP allows programmers to insert directives into their code that lets the compiler generate programs that perform work in a parallel manner. Two common ways of doing this is by parallelizing for-loops and structuring a program as a set of tasks, which are units of work that can be distributed to worker threads.

In general, understanding the precise behaviour of parallel programs is hard[2][3], and OpenMP programs are no exception. While OpenMP makes it simple to parallelize programs, it can still be hard to identify and fix program bottlenecks. It has been pointed out that existing sampling based tools are insufficient for identifying problematic tasks, as they investigate task constructs defined in source code and not the task instances[4]. This makes certain optimizations harder, such as eliminating tasks with low parallel benefit. A similar problem exist for parallel for-loops, where uneven distribution of work can lead to load imbalance[5].

The grain graph is a recent visualization method for OpenMP programs[5]. Its main focus is visualizing program execution from a program perspective. This is done by creating a graph displaying the structure of the program, in terms of tasks, for-loop portions, and synchronization points. Furthermore, certain properties of tasks and for-loop portions, collectively called *grains*, are visually encoded in the graph. Using this visualization, the programmer can inspect the performance characteristics of each individual grain. Muddukrishna et al. use an in-house runtime system named MIR to obtain the grain-level metrics needed by the graph. More widely used runtime systems, such as those in LLVM, Intel ICC and GCC, have thus far not been able to provide such metrics.

There is currently an ongoing effort to extend the OpenMP standard with an API for performance analysis of OpenMP programs[6]. This API is called OpenMP Tools (OMPT). A key motivation for OMPT is the inherent tight coupling between performance analysis methods and runtime systems. Grains graphs and the MIR runtime system is an example of this. OMPT supports both sampling- and instrumentation-based measurement, as well as *blame shifting*[7], which is a technique that enables sampling-based tools to put the blame for waiting on the context that caused the waiting. Moreover, it is already implemented in recent versions of the Intel and LLVM compiler suites, and at the time of writing it is being considered for inclusion in the upcoming OpenMP 5.0 standard.

Until recently, OMPT has been in a continuous state of change, and as such I have found no other tools that visualize performance problems using measurements taken with OMPT. A tool that generates grain graphs using data collected through OMPT could fill this gap. However, it is apparent that OMPT does not provide all information required by grain graphs, one example being for-loop chunk information.

This paper builds upon this observation and presents an exhaustive examination of the raw information needed generate grain graphs and its derived metrics. Subsequently, the OMPT interface is studied in detail. I then describe what is missing and how OMPT can be extended to correct its shortcomings while respecting its expressed design objectives.

This work contributes a thorough answer on how to obtain from OMPT the data required to draw grain graphs. I show what data can be immediately obtained from OMPT, and what data needs extensions to OMPT. The proposed extensions would let the interface facilitate more detailed visualizations, a prime example being grain graphs. Hopefully, this will convince the OpenMP Architecture Review Board to extend the API to support this method.

II. Background

A. OpenMP

Creating programs that can fully utilize modern multiprocessors with shared memory can be a complex and frustrating process. OpenMP was designed to ease this process, specifically for high performance programs. It has become an industry-standard API for creating parallel programs on shared memory systems, using the C, C++ and Fortran programming languages[1]. It uses a compiler-assisted approach where the programmer inserts directives into the program to let the compiler know how a portion of the program is to be parallelised. The compiler then performs a code transformation that includes inserting calls to an OpenMP runtime. The runtime is a library that the program calls during execution, responsible for tasks such as thread management and work distribution.

Initially, OpenMP only supported worksharing constructs such as for-loops and sections. These let the runtime divide chunks of a for-loop, or non-iterative sections of code, among threads. The OpenMP 3.0 specification introduced tasks to better accommodate task-parallel programs. Tasks are independent units of work that themselves can create more tasks. They are more general and flexible compared to sections, as they allow dynamic generation of work and synchronization of child tasks. Task synchronization is different from barriers, which work on the thread-level to synchronize teams of worker threads. This makes implementing task-parallel algorithms simpler, as the programmer is distanced from runtime-centric abstractions like threads and barriers.

B. Performance Profiling

In order to pinpoint bottlenecks and improve program performance, programmers often need to perform profiling. Generally, this entails monitoring program execution in order to attribute various metrics such as execution time to program entities, of which functions are an example. There are two popular, distinct approaches for doing this.

The first is sampling, in which the profiler will perform measurements in regular intervals, and subsequently derive some performance metrics from these measurements. This approach can among other things reveal the time spent inside each function as well as the time spent executing application code and time spent on spin and overhead. Examples of tools using this approach are HPCToolkit[3] and Intel VTune Amplifier.

One thing to note is that current OpenMP runtimes do not come with functions for performance inquiries, as it is not a part of the OpenMP standard. Therefore profilers must use own methods to collect this data. In the case of HPCToolkit, programs are compiled with a signal handler that will record the current calling context. During sampling, signals are sent periodically to the program[3]. When the program is complete, the samples are analysed to present aggregated information, for instance in which functions or instructions most of the execution time is spent.

The second way to collect performance data is instrumentation. With this approach, the profiling system will receive calls at certain program events. The canonical manner in which this is done is to insert calls to the profiling system in the source code of the program that is to be profiled. This method is, among others, used in TAU[8] and Scalasca[9].

Using this approach, it is up to the programmer to find suitable instrumentation calls. If one is interested in the duration of a specific event, a typical approach is to invoke calls before and after the event. For another event, a single call might be sufficient. This enables much more fine-grained monitoring of the program. However, should the degree of instrumentation become too high, the behaviour of the program might change, yielding inaccurate performance analysis results[8]. This can be prevented by limiting the amount of instrumentation calls, as well as ensuring that the calls themselves do not incur unnecessary overhead.

Sampling can be combined with light-weight instrumentation to implement *blame shifting*[7]. Traditional sampling will often yield samples that show a thread waiting. This can be due to lack of work, synchronization, mutual exclusion, or something else. From the sample it is not trivial to identify the reason for waiting. Blame shifting is a technique for placing blame for waiting on the context that caused it.

This is done by using instrumentation calls that notify the profiling system that a thread is about to start waiting, along with information about the reason. For instance it can specify the lock that caused it to wait. The profiler can then record which thread is currently holding that lock. After this, whenever the profiler is doing sampling and observes that the thread is waiting, it can blame the thread holding the lock.

C. OpenMP Profiling APIs

In the past there have been multiple attempts to define a standard OpenMP profiling API. Two such efforts have been especially influential[10] in the design of OMPT, so they are introduced briefly here. OMPT is discussed separately later.

One of the efforts was the POMP API[11]. This is an instrumentation-based solution. It is proposed that the degree of instrumentation is controlled through compiler directives. It is argued that this is natural, as OpenMP programmers are already used to inserting directives into their programs. Moreover, monitoring of events is achieved by having the compiler insert extra calls to the OpenMP runtime.

The second effort was the Sun/Oracle Collector API[12]. While this proposal also contain instrumentation functionality, it was primarily designed to support statistical sampling of call stacks. Because the call stacks will be different across OpenMP runtime implementations, the authors also show how to construct a program-centric call stack that is independent of the runtime. Furthermore, this task is left up to the performance tool in order to keep the API lightweight.

III. Grain Graphs

The recently proposed OpenMP visualization grain graphs studies task instances and for-loop chunks, which the authors generalize as grains. It combines program structure with per-grain performance metrics to identify poor-performing grains.

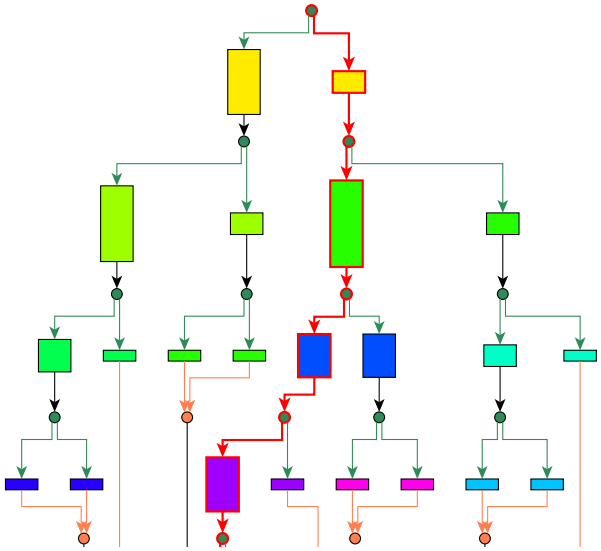

Fig. 1: A trimmed example of the grain graph of a recursive Fibonacci sequence program.

The program structure is presented as a directed acyclic graph. The graph's nodes are either grains, program fork points, or join points. Edges show the relationships between grains, fork points, and join points.

Moreover, the graph is supplemented with visual cues related to performance, some of which can be seen in Figure 1. The length of grain nodes denote their execution time, while the set of red edges shows the critical path of the grains. The graph can be further annotated with measurements of memory system behaviour such as cache miss ratios, or one of many derived metrics which will be discussed shortly. Should any of these metrics surpass reasonable values, the fill colour of the responsible grains can be used to separate poor-performing grains from well-performing ones. Favourable grains can also be dimmed, to further isolate the grains that should be optimized first.

Grain graphs necessitate collecting a set of properties per grain during program execution. Some properties are needed to uniquely identify grains. Others are used to generate the structure of the graph. Finally, some properties are necessary to derive performance metrics. The properties that must be collected during execution are italicized for clarity when first introduced. From these, other properties or metrics can be derived during the post-processing stage.

The following properties are enough to produce full, unreduced grain graphs. An unreduced graph shows grains further divided into fragments. It also shows bookkeeping nodes associated with parallel for-loops. As shown in [5], a grain graph is typically visualized in a reduced state to improve information density. A reduced graph is created by grouping nodes found in the unreduced graph.

A. Graph structure

The visual structure of a grain graph shows synchronization points, fork- and join-points. A fork is the result of a parent grain generating children grains. When the children complete their execution they synchronize with the parent, yielding a join point. For grains associated with task instances, this can be captured by storing its *parent*, *child number* and *synchronization point number*.

The child number indicates a grain's birth order. The synchronization point number is necessary in cases where a grain contains more than one implicit or explicit synchronization point. An example is when a task creates some children tasks, synchronizes with them, and then creates more. On a reduced grain graph, this will appear as a join followed by a fork, with no grain nodes in-between. Children that synchronize at the n-th synchronization point are shown to join at the n-th join point.

Parallel for-loops require some extra information. On the graph, n worker threads will give an n-way fork. On the graph, each branch will contain a series of bookkeeping nodes and chunk grains. Each grain is treated as the child of the grain that encountered the loop.

To generate the bookkeeping nodes, the bookkeeping time associated with each grain is collected as described later. To place grains on the correct fork branch, the *identifier of its thread* is used. To correctly order the grains on the same branch, the grains are placed in order of increasing child number. In other words, the successor of a chunk grain is the next sibling executed on the same thread.

B. Identification of grains

A simple way to identify grains is with an increasing *numeric identifier*. While uniquely identifying each grain present in a particular run of the program, this identifier is not guaranteed to correspond to the same grain across runs.

A run-independent unique identifier is necessary to compare grains in graphs generated from different program executions. Grains are either task- or for-loop chunk instances. Different identification schemes are used for these two cases.

Task instances are identified by the path one must travel to reach the task node from the starting node. This is enabled by the fact that the shape of the graph will not change for deterministic task-based programs, irrespective of the number of threads used. This is not true for grains corresponding to for-loop chunk instances, as the shape of the graph depends on the number of worker threads and the order in which chunks are assigned to workers. In this case, the grains are identified by the *thread that started the loop*, a *parallel for-loop sequence counter*, and the *iteration range*[5].

A grain might also be partially identified by its *source code location*. This identifier is important, because there is no point in finding poor-performing grains if the source origin of that grain is unknown. Alternatively it is possible to record the location of the grain code in the generated executable. Typically this will be an *outline function pointer*. Using this approach, the executable must contain the required information to link the function address to a source code location, for instance by examining the symbol table.

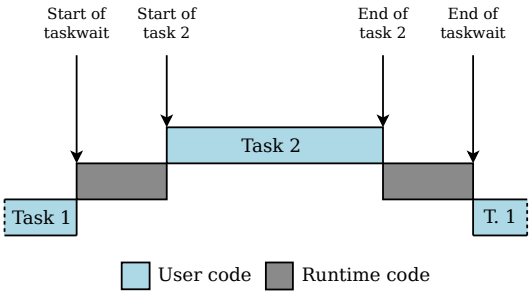

Fig. 2: Task 1 encounters a taskwait region, which causes task 2 to be scheduled on the same thread.

C. Grain performance

In order to indicate poor-performing grains, it is necessary with per-grain bookkeeping. Many raw properties are collected to calculate derived metrics. Here, I present these raw metrics in the context of the derived metrics that depend on them.

Firstly, the *execution time* is needed as it decides the length of grain nodes. The execution time of a grain is the accumulated time spent executing the grain. Grains associated with task instances may have their execution suspended at task creation and task synchronization points. As such, it is not sufficient to simply measure the time between the start and end of a task. The time a task spends in a suspended state should not count towards its execution time. In Figure 2, the execution of a `taskwait` region is shown. In this example, the time between the start and end of task 2 would not be counted in the execution time of task 1.

Execution time is also used to derive other metrics. One such metric is parallel benefit. It is a grain's execution time divided by the costs of parallelization as seen in the parent. This cost is defined as the grain's creation time plus average synchronization overhead. The latter is time used by the parent to synchronize with its children divided by the number of children. This metric introduces the need to collect *creation time* and *synchronization overhead time* per grain.

Another derived metric is instantaneous parallelism. It quantifies the degree of parallelism exposed at different times in the program at the fragment level. A fragment is a consecutive slice of grain execution. In Figure 2, the grain associated with Task 1 is shown as two fragments. This necessitates measuring a grain's *creation instant* relative to its parent's execution, its *synchronization instants*, and its *execution time*.

It is also necessary to record the *CPU identifier* for every grain. This is needed by the derived metric scatter. It is the median pair-wise NUMA distance of processors executing sibling grains.

Furthermore, Muddukrishna et al. suggest measuring grains' memory system behaviour[5]. This includes collecting *cache miss ratios* and *stalled cycles counts*. The latter can be used together with execution time, if measured in cycles, to calculate memory hierarchy utilization, which is the ratio of cycles used for computation to cycles spent on stalls.

Finally, each chunk grain is subject to some chunk assignment overhead. This is shown on the graph as bookkeeping nodes. The time spent on chunk assignment is encoded as the length of the bookkeeping nodes. It is also needed to calculate parallel benefit for chunk grains. This requires collecting *chunk overhead times*.

	T/F	Grain property	Purpose
Structure	☒☒	Parent identifier	Connect children with parents
	☒☒	Child number	Order children, place chunks in execution order
	☒☐	Sync. point number	Grain position relative to sync. points
	☒☒	Thread identifier	Place chunks in order, uniquely identify chunks
Identification	☒☒	Numeric identifier	Simple grain identification
	☒☒	Code location	Associate grains with source constructs
	☐☒	Loop sequence counter	Uniquely identify chunk grains
	☐☒	Chunk iteration range	Uniquely identify chunk grains
Performance	☒☒	Execution time	Node length, multiple derived metrics
	☐☒	Chunk overhead times	Bookkeeping nodes, derive parallel benefit
	☒☐	Creation time	Derive parallel benefit
	☒☐	Sync. overhead time	Derive parallel benefit
	☒☒	Create instant	Derive instantaneous parallelism
	☒☐	Sync. instants	Derive instantaneous parallelism
	☒☒	CPU identifier	Derive scatter
	☒☒	Cache miss ratio	Describe memory system behaviour
	☒☒	Stalled cycles	Derive memory hierarchy utilization

TABLE I: The per-grain properties collected during program execution. The T/F column shows whether a property applies to tasks, for-loop chunks, or both.

A summary of properties that must be collected during execution can be seen in Table I.

IV. The OpenMP Tools API

The OpenMP Tools API, called OMPT for short, is a recently proposed addition to the OpenMP standard. It is designed to enable the creation of portable, first-party performance analysis tools[10]. First-party means that the tool lives in the profiled program's address space. As there is currently no standard performance analysis API, performance tools are tightly integrated with specific OpenMP runtime implementations. By including performance measurement functionality directly in OpenMP, tools can be used with any standard-compliant OpenMP implementation. OMPT is likely to be included in the upcoming OpenMP 5.0 standard.

OMPT combines elements from the earlier POMP and Sun Collector APIs. It supports both instrumentation and sampling. OMPT also includes events intended to enable *blame shifting*. In this section, its design objectives, architecture and profiling capabilities are summarized.

A. Design objectives

The API is intended to allow high-quality tools with low overheads. Furthermore, its design objectives state that tools

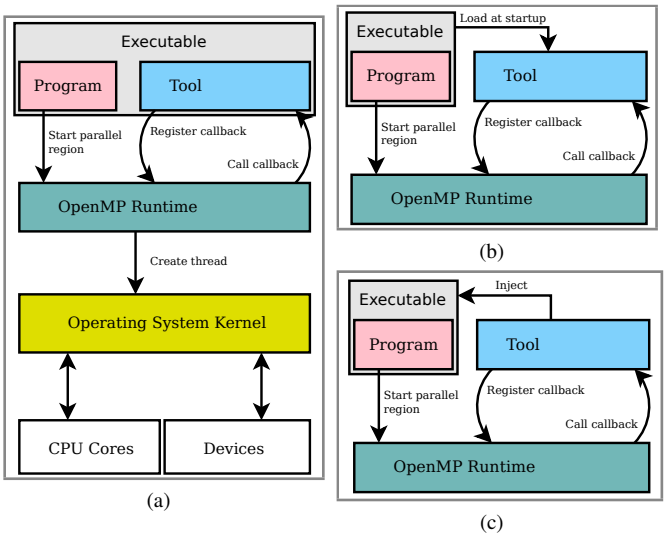

Fig. 3: Overview of an OMPT environment. The tool must exist in the address space of the program. Placing the tool in the address space can be done in three different ways: **(a)** static linking, **(b)** dynamic linking, or **(c)** library injection.

OMPT subset	Functions or events

Mandatory inquiry functions	`ompt_enumerate_states`
	`ompt_enumerate_mutex_kinds`
	`ompt_get_thread_data ompt_get_state`
	`ompt_get_parallel_info`
	`ompt_get_task_info`

Mandatory events	`ompt_event_thread_begin`
	`ompt_event_thread_begin`
	`ompt_event_parallel_begin`
	`ompt_event_parallel_end`
	`ompt_event_task_create`
	`ompt_event_task_schedule`
	`ompt_event_implicit_task`
	`ompt_event_runtime_shutdown`
	`ompt_event_control`

Selected optional events	`ompt_event_sync_region_wait`
	`ompt_event_sync_region`
	`ompt_event_worksharing`

TABLE II: Relevant excerpt from the OMPT interface.

should be able to associate costs with both the program and the OpenMP runtime and that incorporating OMPT support in an OpenMP runtime should add negligible overhead if no tool is using it. It also states that the API should not impose an unreasonable development burden on runtime developers or tool implementers, and that profilers that use call stack unwinding should be able to differentiate user- and runtime-imposed stack frames.

B. Environment

In order to make use of the OMPT API, the tool must reside in the application's address space. The OpenMP runtime will then call the tool's initialization function immediately after it has initialized itself. This allows the tool to set up its data structures, retrieve pointers to OMPT functions and register event callbacks. There are three distinct ways to place the tool in the address space. It may be statically linked into the executable, dynamically linked as a shared library, or injected into the process before the OpenMP runtime starts initialization.

Figure 3a shows a high-level overview of the application, the tool, the OpenMP runtime, and lower levels of a computer system. The OpenMP runtime is shown to interface with the operating system to create one or more threads, as this is one of its key responsibilities. The tool is shown to interface with a shared OpenMP runtime for OMPT functionality, but note that OMPT can also be fully compiler-implemented[10]. In Figure 3c, the tool is injected into the process. The advantage of this approach is that the application executable does not need to re-linked before doing performance analysis.

In order to let tools map instances of threads, parallel regions, tasks, and so forth to tool data structures, a data type called `ompt_data_t` is introduced. It is a union type that can represent an integer or a pointer, and it is passed by reference. This allows tools to either attach tool-specific data directly such instances, or alternatively maintain its own unique integer identifier.

C. Core functionality

A small set of mandatory features are required for an OpenMP environment to be minimally compliant. These mandatory features are roughly as follows: Unique identifiers must be maintained for instances of threads, parallel regions, tasks and so on. It must support classification of stack frames as user- or runtime-generated. After the OpenMP runtime has been initialized, it must call the tool's initialization routine unless the environment variable `OMP_TOOL` is set to `disabled`. It must also implement several async signal safe inquiry functions to retrieve information from the OpenMP runtime, maintain a state for each thread, and support a core set of event callbacks. These are what enable most performance measurements, so they are discussed in more detail below.

D. Mandatory functions

Every compliant implementation must support a set of inquiry functions, which among other things can be used for sampling. These functions are all listed in Table II. Their most important functionality is providing information about what a thread is doing.

A core set of event callbacks are also mandatory. During initialization, the tool must register callbacks for the events it is interested in. All callbacks are synchronous, meaning the program will not continue until the callbacks have completed execution. The event associated with the callbacks are listed in Table II. The `ompt_control` event allows the user program to send signals to the tool, as it is triggered when the program calls the function `ompt_control`.

Note that in addition to what is presented above, OMPT also contain inquiry functions and event callbacks related to features for programming heterogeneous systems. As these features are not needed here, they are omitted.

1) Optional functions: Optional features include a set of additional events for blame shifting and instrumentation. As these are quite numerous, they are not all listed here. However, there are three instrumentation events that are particularly relevant.

The first is `ompt_event_worksharing`. It is invoked when a worksharing construct, such as a parallel for-loop, is started, and when it is finished. The second is `ompt_event_sync_region`. This event is invoked before and after a task encounters a barrier, taskwait or taskgroup. The last event is `ompt_event_sync_region_wait`. It differs from the previous event in that the event starts when the a task is waiting inside a synchronization region, and ends when another task is scheduled on the thread, or when the region is exited. It can therefore generate multiple pairs of begin/end callbacks within the same region.

Note also that the specification is also lenient on the timing of observable state transitions and event callbacks. It states that for some states, the OpenMP runtimes have flexibility about whether to report the state early or late. It is up to the tool to account for this. For event callbacks, some might be called when an event occurs, when it is convenient, or never[10]. The return value for event callback registration must signal if and when a callback will be invoked.

V. Using OMPT to create grain graphs

I will now describe how to use the OpenMP Tools interface to acquire the raw properties needed to generate grain graphs. A summary of what and how the various properties can be obtained is shown in Table III. As I will demonstrate below, some properties are easy to collect while others rely on some nontrivial methods. Specifically, the `ompt_event_control` event can be used to extend the instrumentation capabilities of OMPT.

The `ompt_event_control` event allows the program to send signals to the tool via the `ompt_control` function. The function has two parameters named `command` and `modifier`, both unsigned 64-bit integers. These are passed to the tool when the corresponding event callback is invoked. This allows the programmer to implement some basic source code instrumentation. If this is combined with some creative methods in the tool, it is possible to obtain every property that the API cannot provide directly.

Experimental verification

The highest authority regarding the OpenMP Tools API is, at the time of writing, the technical report available in a public GitHub repository[10]. The API descriptions from this document are used to devise methods to collect the needed data. A tool has also been developed to verify that the methods work as expected. Like any tool using the OMPT interface, it registers relevant callbacks and maintains its own set of profiling data. It has been especially useful for more intricate methods that rely on accurately interpreting the terse API descriptions in the OMPT technical report.

Excluding empty lines, the tool is around 600 lines of C++ code. It is attached with the report. It was used together with the LLVM OpenMP runtime, as there is no OMPT reference implementation. The LLVM runtime's OMPT implementation follows the older OpenMP Technical Report 2[6] from 2014, with some changes from the more recent document[10]. It is therefore very similar to the updated OMPT proposal. The runtime supports 8/8 mandatory events, 10/14 blame shifting events, and 29/42 instrumentation events. One of the events needed to generate grain graphs was unsupported, namely `ompt_event_sync_region_wait`.

I shall now describe how to obtain each property step by ustep.

A. Graph structure

The structure is different in the case of task-based programs and programs with parallel for-loops. For the latter, the OMPT interface is lacking. How to acquire the properties needed for task-based programs will now be described.

1) Parent: For task-based programs, a pointer to the parent task's `ompt_data_t` is passed as a parameter in the callback for the event `ompt_event_task_create`. For programs with parallel for-loops, the parent is the initial task, found as a parameter to the `ompt_event_parallel_begin`.

2) Child number for task grains: The child number is equivalent to the order in which children are created. Therefore, the tool must keep track of the order in which the `ompt_event_task_create` callback is invoked. This can be done by maintaining a child sequence counter per grain during execution.

3) Child number for loop chunk grains: The OMPT interface does not provide any information about parallel for-loop chunks at all. To obtain data about chunks, I have designed a mechanism based on `ompt_control`. By insering a call to `ompt_control` inside the loop, it is possible to map out which iteration ranges ran on which threads. Each iteration range is then associated with a chunk.

A `command` value named `CHUNK_ITERATION` is defined. The current iteration counter is passed as the `modifier` for each iteration. By calling `ompt_get_thread_data` function inside the `ompt_event_control` callback, per-thread lists of iterations are created. One can then map iterations to ranges as a post-processing step. This can be done as shown in Algorithm 1. In simple terms, the algorithm finds the loop iteration increment from the two first iterations performed. After this point, whenever the difference between two iterations is not equal to the expected increment, one chunk is said to have ended while another one has started.

Figure 4 shows a for-loop where chunks are divided between two threads. The iteration increment is found to be two. While performing Algorithm 1 on the iterations of the first thread, one would find the iteration counter value jumping from 7 to 17. This would indicate the end of the first iteration range and the start of the second.

The algorithm works because OpenMP requires that parallel for-loops have a constant increment. A limitation of the

	T/F	Grain property	Obtainable	How or why not
Structure	☒☒	Parent identifier	Yes	Parameter of `ompt_event_task_create` and `ompt_event_parallel_begin` callbacks.
	☒☒	Child number	Yes, `ompt_control`	Maintain own sequence counter per grain. Need `ompt_control` mechanism for chunks.
	☒☐	Sync. point number	Yes	Maintain own sequence counter per task.
	☒☒	Thread identifier	Yes	By using `ompt_get_thread_data`.
Identification	☒☒	Numeric identifier	Yes, `ompt_control`	Maintain own sequence counter per grain. Need `ompt_control` mechanism for chunks.
	☒☒	Code location	Yes	Use return address parameter together with symbol table and debugging information.
	☐☒	Loop sequence counter	Yes	Maintain own sequence counter.
	☐☒	Chunk iteration range	Using `ompt_control`	Create list of iterations per thread, map to ranges late.
Performance	☒☒	Execution time	Yes, `ompt_control`	Note grain start, end, and suspensions times. Do bookkeeping to accumulate time.
	☐☒	Chunk overhead times	Using `ompt_control`	Calculate differences between loop start, chunks, and loop end.
	☒☐	Creation time	Using `ompt_control`	Insert begin/end calls before and after a task construct.
	☒☐	Sync. overhead time	Yes	Using the `ompt_event_sync_region_wait` callback.
	☒☒	Create instant	Yes, `ompt_control`	Calculate parent's elapsed execution time when grain created.
	☒☐	Sync. instants	Yes	Calculate elapsed exec. time inside `ompt_event_sync_region` callback
	☒☒	CPU identifier	Yes	Using system calls inside event callbacks
	☒☒	Cache miss ratio	Yes, `ompt_control`	Using hardware counters together with begin/end callbacks. Need `ompt_control` mechanism for chunks
	☒☒	Stalled cycles	Yes, `ompt_control`	Using hardware counters together with begin/end callbacks. Need `ompt_control` mechanism for chunks

TABLE III: Properties collected for each grain during execution, and how they can be obtained. The T/F column shows whether a property applies to tasks, for-loop chunks, or both.

Algorithm 1: Map iterations to chunk iteration ranges

Input: List L with n iterations
Output: List R of iteration ranges

$\Delta \leftarrow L[2] - L[1]$ `/* Iteration increment */`
$r_f \leftarrow L[1]$ `/* First iteration in range */`
$R \leftarrow \emptyset$

for *iteration* $L[i]$ *with* $i = 2 \dots n$ **do**
  if $L[i] - L[i-1] \neq \Delta$ **then**
    `/* Irregular increment means start of new chunk iteration range */`
    $R \leftarrow R + \{\ r_f\ ,\ L[i-1]\ \}$
    $r_f \leftarrow L[i]$
  end
  else if $L[i]$ *is last iteration* **then**
    $R \leftarrow R + \{\ r_f\ ,\ L[i]\ \}$
  end
  else
    continue
  end
end

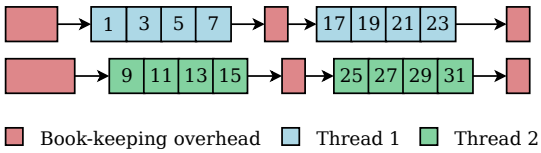

Fig. 4: A parallel for-loop iterating from 1 to 31, with an iteration increment of two and a chunk size of 4

algorithm is that it will treat chunks with successive iteration ranges as one large chunk if they are scheduled right after each other on the same thread. It will also give erroneous results if the chunk size is 1, as this makes the increment calculation impossible. To account for these two cases, one could set the chunk size manually for the loop using the `schedule` clause, and give this information to the tool through another call to `ompt_control`.

Algorithm 1 is described as a post-processing algorithm because it simplifies its definition. However, as shown in section V-D, it can be altered to create chunks during program execution instead. This is important, because it allows an OMPT tool to collect properties for chunks during program execution.

With this sorted out, I can finally answer how to obtain a chunk's child number: Whenever a new chunk is created, the per-grain sequence counter for the loop's parent task is used, and subsequently incremented.

4) Synchronization point number: To keep track of multiple synchronization points within a grain, the event `ompt_event_sync_region` can be used. It lets the tool know that a task is about to start synchronization. By maintaining a sequence counter per grain, a child grain's synchronization point number can be read from the parent grain's sequence counter at create-time.

5) Thread identifier: Getting the identifier of the thread that started executing a grain can be acquired by calling the inquiry function `ompt_get_thread_data` inside the callback for

either ompt_event_task_schedule or ompt_event_worksharing, for tasks and chunks respectively.

B. Identification of grains

1) Numeric identifier: For a particular run, each grain corresponding to task instances can be identified using the ompt_data_t type. This type can represent an increasing integer identifier or a pointer to tool data for the grain. For programs with parallel for-loops, whenever the tool detects that a new chunk grain has started, it can get its numeric identifier from a global grain sequence counter.

To uniquely identify grains across runs, the identification is a bit more involed. For a grain associated with a task instance, it is sufficient to simply enumerate the path from the root to the grain. This path is a consequence of the graph structure. This structure can be generated as described in section III-A. As shown above, the OMPT interface can be used to retrieve the necessary properties for task-based programs.

Grains associated with for-loop chunks have a different scheme for unique identification. Three properties are needed, one of which is the thread identifier which was also needed for the graph structure. The two other properties are:

2) For-loop sequence counter: Maintaining a parallel for-loop sequence counter is as simple as counting the number of times the ompt_event_worksharing callback is invoked.

3) Chunk iteration range: This property cannot be obtained directly using the OMPT API, but the ompt_control mechanism described earlier relies on using iteration ranges to find when one chunk instance has ended its execution, and another one has started. As such, it has already been found.

4) Code location: To correlate grains with their originating source code location, the callbacks associated with the task_create and worksharing events can be used. They both provide a pointer to the return address of the call to the OpenMP runtime routine that triggered the event. To see which compiled routine this address belongs to, the symbol table can be inspected. Compiling the program with a line number table is necessary to correlate a grain with a specific line number.

To inspect the symbol table of an executable on Unix-like systems, the nm tool can be used. The equivalent tool for Windows is DUMPBIN. Both tools can show line numbers if that information is presented in the executable.

C. Grain performance

1) Execution time for task grains: The execution time of a grain is the time spent executing the grain. For grains associated with task instances, the runtime will notify the tool that a task is getting scheduled for execution when it appears as the next_task_data argument of the ompt_event_task_schedule callback. When a task appears as the prev_task_data argument, it has been completed or suspended. We can measure the total execution time of a task as follows: When a task is created, its execution time is set to 0. Every time it is scheduled, the current time is noted. When it is suspended or completed, the time since it was last resumed is added to execution time.

2) Execution time for loop chunk grains: This property cannot be obtained directly using the OMPT API, but the ompt_control mechanism described earlier can be re-used. A for-loop chunk instance will run uninterrupted over its iteration range. As such, one can simply measure the time elapsed from the chunk's first iteration until the next chunk is encountered. The last chunk on every thread will have to instead use the end time of the loop to denote its end. This can be found from the ompt_event_worksharing callback.

3) Task creation time: There is no pair of begin/end callbacks for task creation. By inserting a call to ompt_control before and after each task construct in the source code, the tool can still measure creation time. Two distinct command values TASK_CREATE_BEGIN and TASK_CREATE_END are defined. During the function call associated with TASK_CREATE_END, the tool can use the inquiry function ompt_get_task_info to find the current (parent) task. As the creation time is a property of the child task that was just created, it must be assigned to the last created child of the current task.

4) Synchronization overhead time: This task-specific property measures the time spent inside a taskwait region. In Figure 2, it is the sum of the two grey regions spent inside the runtime during a taskwait. This can be done with the ompt_event_sync_region_wait callback. It is invoked when a task starts or stops waiting in a barrier, taskwait, or taskgroup region. One region may generate multiple pairs of begin/end callbacks if multiple tasks are scheduled on the waiting task's thread while it is waiting. When a task is created, its total synchronization overhead time is set to 0. If it starts waiting in a taskwait region, the current time is noted. When it stops waiting, we add the time since it started waiting to the synchronization overhead time.

5) Creation instant for task grains: The creation instant denotes when a grain is created relative to its parent's execution time. For grains associated with task instances, the ompt_event_task_create event can be used. When this callback is invoked, the duration of the interval since the parent was last resumed must be calculated. This is done by subtracting the parent's last resumed time property with the current time. This is then added to the parent's execution time to give the creation instant.

6) Creation instant for loop chunk grains: This property cannot be obtained directly using the OMPT API, but the ompt_control mechanism described earlier can be re-used. Every for-loop chunk instance is said to be created when the loop starts, because this is when they can start executing. As with the creation instants of task grains, the creation instant is relative to the parent task's execution time.

7) Synchronization instants: This task-specific property contains the all the times a taskwait region is entered relative to the task instance's execution time. Inside the ompt_event_sync_region callback, the duration since the task instance was last resumed is calculated. This is added to the execution time property, giving the synchronization instant. The instant is then added to the synchronization instants list.

8) CPU Identifier: The CPU Identifier can be collected when a grain is scheduled. The OMPT interface does not expose this information. However, for grains associated with tasks, the `ompt_event_task_schedule` callback is invoked on the thread that is about to execute the task. A system call can be used to get the identifier of the calling processor. For the Linux and Windows operating systems, these are available as `getcpu` and `GetCurrentProcessorNumber` respectively. In the case of untied tasks, it must be collected every time a task is scheduled, and stored in a list. The same system calls can be used whenever a new for-loop chunk is created.

9) Cache miss ratio: The cache miss ratio of a grain is the fraction of memory accesses within the grain that miss. It is equal to one minus the hit ratio. The hit ratio is equal to the number of hits divided by total accesses. To find the cache miss ratio for a specific cache level, one must therefore count the number of hits and misses to that cache. This can be done by using hardware counters. When a grain is scheduled, the counting starts. When it is suspended, the counts are added to temporary variables. When the grain is complete, the cache miss ratio is calculated. For task instances, the `ompt_event_task_schedule` callback is called when tasks are resumed, suspended, or completed. For for-loop chunks, the `ompt_control` mechanism described earlier is used to find when a chunk starts and ends.

10) Stalled cycle count: The stalled cycle count denotes the number of cycles spent waiting for data. The procedure for obtaining it is the same as for cache miss ratio - programmable hardware counters.

11) Chunk overhead times: This property cannot be obtained directly using the OMPT API, but the `ompt_control` mechanism described earlier can be re-used. I have already described how one can obtain chunk iteration ranges and their corresponding execution times. If one also persist each chunk's start and end instants, one can find the overheads before, after, and between chunks. Using Figure 4, the first book-keeping node on each thread will be the difference between the start of the loop and the start of the first chunk. The start instant of the loop can be noted during the `ompt_event_worksharing` callback. Subsequently, book-keeping nodes between chunks are found by taking the difference between their respective end and start instances. Finally, the last book-keeping node has a duration equal to the end of the last chunk and the time of the `ompt_event_worksharing` callback for the end of the loop.

D. Inferring chunks during execution

Earlier in this section, I showed a way to way to find chunks and their iteration ranges as a post-processing operation. However, it is possible to do it during program execution as well. The technique does not change much. Instead of looping over a list, one must now check if the increment since last iteration is equal to the expected iteration increment each time `ompt_control` is called. When an irregular increment is found, an iteration range has completed. This means that each thread working on a parallel for-loop must additionally store the iteration value of the last iteration.

It is now possible to have a chunk concept in the tool during execution. Whenever a team of worker threads start a parallel for-loop, as signalled by `ompt_event_worksharing`, each thread will create an instance of a tool-defined grain data structure. Whenever an iteration range has completed, the tool can calculate any remaining chunk properties and create a new grain data structure for the next chunk that is about to start.

VI. Discussion and Future Work

In this section, I discuss how well OMPT accommodate the generation of grain graphs, how it could be extended, and future work.

A. The good

The OpenMP Tools API is well thought out. In particular, it provides almost every piece of data that grain graphs need about task instances. The only exception is creation time, the time spent creating a task. The event `ompt_event_task_create` lets the tool know when a task is being created, but there is no corresponding callback when the creation of the task is complete.

However, by using the `ompt_control` function, it is possible to obtain more data than what OMPT can otherwise provide. In the case of the creation time property, one can insert a call to `ompt_control` before and after every task construct in the source code. The tool can then measure the duration between these two calls. As shown in the previous section, there are also other, more involved ways of using the function.

B. The less good

All worksharing constructs share the same OMPT events. When a parallel for-loop is encountered, the callback for the `ompt_event_worksharing` event is invoked, and the `wstype` argument indicates that the type of the worksharing construct is a loop. As such, the concept of parallel for-loop chunks does not appear anywhere in the OMPT interface.

I have devised workarounds for the lacking chunk information that rely on inserting calls to `ompt_control` inside each iteration. For tight loops, this could introduce significant overhead. Furthermore, the workarounds rely on collecting per-iteration data that is later used to find chunk iteration ranges, execution times and overheads. For large loops, the memory space needed could get quite large.

The workaround described in Algorithm 1 was initially conceived as a post-processing algorithm that relied on recording every iteration counter value executed on a thread and mapping them to chunks after the program had finished. This is obviously a sub-optimal solution, as it would require a lot of memory, especially when also collecting other chunk data that would also have to be assigned to chunks later. It was therefore optimized as discussed in section V-D, which is still not a perfect workaround. This illustrates that even with the

capabilities of `ompt_control`, extracting information about chunks is tedious.

However, it is not optimal having to insert a great amount of calls to `ompt_control` in your programs. It is also not particularly fun to create new algorithms to extract chunk properties, or to spend time optimizing those algorithms. One of the explicit design objectives of OMPT is that the API should not impose an unreasonable development burden on tool implementers[10]. I would argue that it is unreasonably clunky to extract information about for-loop chunks.

C. Proposing extensions to OMPT

I will now discuss how the OpenMP Tools API can extend its functionality to accommodate detailed profiling and visualization methods such as grain graphs. As shown above, it is particularly cumbersome to implement profiling methods that rely on obtaining information about parallel for-loop chunks. As such, this is considered to be the most important extension. A way of allowing the measurement of task creation duration is also proposed.

1) Identification of chunks: It should be possible to identify chunks. Like parallel and task regions, each individual chunk would be identified by a unique `ompt_data_t`.

2) An event for chunk scheduling: To allow measuring the duration of individual chunks, an event equivalent to `ompt_event_task_schedule` should be introduced for chunks. Whenever a chunk is about to start execution, the callback associated with the event would be invoked. Its parameters would at the very least include the task that encountered the construct, the prior chunk, if any, and the next chunk, if any. A natural extension would be to also pass the start and end of the iteration range, as well as the iteration increment.

3) A begin/end event pair for task creation: Many callbacks are associated with event pairs that denote the beginning and end of an interval. One example is the worksharing construct. Before the encountering task starts executing the first unit of work, the `ompt_event_worksharing` callback is invoked with the `endpoint` argument set to `ompt_scope_begin`. After the last unit of work, the same callback is called with `endpoint=ompt_scope_end`. I propose that the `ompt_event_task_create` callback is changed to work in the same way. I understand that the reason it is not currently so is likely that the OpenMP Architecture Review Board believe that it is not sufficiently useful, or that it would incur too much overhead.

However, I have found that grain graphs is not the only performance analysis method that relies on this information. Qawasmeh et al. have recently created a method for adaptive task scheduling that relies on characterizing, among other things, how well a program is scheduled with regard to load balancing[13]. To achieve this, they measure how long each thread spends performing a set of low-level runtime events. One such event is task creation.

In the case of grain graphs, the costs associated with task creation are added to the costs of child synchronization to form the total overhead of creating children. The parallel benefit of each child task is then calculated. This allows programmers to reason about whether creating certain tasks are worth the costs. I believe that these two cases show that there are legitimate reasons to consider extending OMPT to allow measuring task creation duration.

VII. Related Work

Here, I briefly summarize work related to interfaces similar to OMPT, and other tools that use OMPT.

A. Similar interfaces

In addition to the interfaces mentioned as background material, OPARI2 is a commonly used interface with similar capabilities.

OPARI2 is a source-to-source instrumentor. It is included in the Score-P instrumentation and measurement system[14]. It works by automically wrapping OpenMP constructs like parallel and task regions with calls to the performance monitoring interface POMP. Like OMPT, it distinguishes individual task instances and tracks their suspension and resumption points.

A comparison between OPARI2 and OMPT was performed by Lorentz et al. in 2014[15]. While OPARI2 excelled at passing along all relevant source code information, OMPT reflects the execution behaviour of the application more accurately. The information they provide is complimentary. The measurement overhead was found to be low for both, but the source code instrumentation of OPARI2 can sometimes interfere with compiler optimizations. Finally, Lorentz et al. proposed making the events associated with the start of an implicit task, and with a thread switching from one task to another, mandatory. This resulted in their inclusion as mandatory events in the current draft[10].

B. Program analysis based on OMPT

OMPT does not yet enjoy widespread use. The most important reason is likely that it has not yet been standardized. I have found no visualization methods that use data collected by OMPT.

Qawasmeh et al.[13] profiled a set of OpenMP programs running on the OpenUH OpenMP runtime in order to characterize how well they were scheduled. They collected, for each thread, the time spent inside low-level runtime events. Furthermore, they measured level 2 cache misses during the aforementioned events. The events were task creation, execution, suspension, completion, parallel region, implicit/explicit barrier, and loop/single/master region. The authors use an extended version of the Sun/Oracle Collector API[12]. It is not stated how time spent on task creation is measured.

Protze et al. attempt to use OMPT together with the LLVM ThreadSanitizer to implement a low-overhead data race detector[16]. However, it was found that this approach was not compatible with the ThreadSanitizer annotation interface. In addition, they rely on certain callbacks related to critical regions being called just before a synchronization point. However, the specification say that they should be called just after the synchronization point instead. Because of this, the authors

decided to integrate the required annotations directly into the Intel OpenMP runtime.

VIII. Conclusion

I have showed how the OpenMP Tools API can be used to obtain the data necessary to generate grain graphs. Some information is readily available, while other information require using the `ompt_control` function for source code instrumentation. When used to measure the time spent on task creation, this is an OK solution. However, using the function to extract information about parallel for-loop chunks is not straight-forward, and for tight loops it is very likely to introduce significant overhead.

A set of extensions to OMPT are proposed. In the case of for-loop chunk information, the extensions should result in considerably lower overhead when extracting per-chunk metrics compared to the alternative solution using `ompt_control`. The measurement of task creation duration is shown to be needed by others as well, so it should also be considered as an extension. In the future, I plan to examine this further, by extending the LLVM OpenMP runtime with the proposed extensions, and measuring how much overhead is introduced.

IX. Future Work

As mentioned, a prototype tool was used to verify that the methods described in section V worked as expected with the LLVM OpenMP implementation. The tool is able to create a slightly reduced grain graph for task-based programs. Because of the lacking support for information about loop chunks in OMPT, the tool is not yet able to create grain graphs for programs that implement parallel for-loops.

In the future, I plan to first extend the LLVM OpenMP runtime with the proposed functionality. I expect that the overheads associated with these extensions are low. Whether that assumption is correct will be revealed when I later measure the overheads and present the results. After extending OMPT, the aforementioned tool will be further developed, in order to generate full-fledged grain graphs on the extended LLVM OpenMP runtime.

References

[1] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[2] V. S. Adve, "Analyzing the behavior and performance of parallel programs," Ph.D. dissertation, Citeseer, 1993.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/cpe.1553/abstract

[4] D. Schmidl, C. Terboven, D. an Mey, and M. S. Mller, "Suitability of Performance Tools for OpenMP Task-Parallel Programs," in *Tools for High Performance Computing 2013: Proceedings of the 7th International Workshop on Parallel Tools for High Performance Computing, September 2013*. Springer, 2014, pp. 25–37.

[5] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain Graphs: OpenMP Performance Analysis Made Easy," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 28:1–28:13.

[6] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. Cownie, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OpenMP technical report 2 on the OMPT interface," *url: http://www.openmp.org/wp-content/uploads/ompt-tr2.pdf (visited on 07/07/2015)*, Jan. 2014.

[7] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing Lock Contention in Multithreaded Applications," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 269–280.

[8] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[9] M. Geimer, F. Wolf, B. J. Wylie, E. brahm, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[10] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. Cownie, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis," *OpenMP Tools Interface GitHub Repository*, Jan. 2016, work in progress. [Online]. Available: https://github.com/OpenMPToolsInterface/OMPT-Technical-Report/blob/master/ompt-tr.pdf

[11] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah, "A performance monitoring interface for OpenMP," in *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, 2002, pp. 1001–1025.

[12] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin, "An OpenMP Runtime API for Profiling," Sun Microsystems, Tech. Rep., 2007. [Online]. Available: http://www.compunity.org/futures/omp-api.html

[13] A. Qawasmeh, A. M. Malik, and B. M. Chapman, "Adaptive OpenMP Task Scheduling Using Runtime APIs and Machine Learning," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, Dec. 2015, pp. 889–895.

[14] A. Knpfer, C. Rssel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, and others, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[15] D. Lorenz, R. Dietrich, R. Tschter, and F. Wolf, "A Comparison between OPARI2 and the OpenMP Tools Interface in the Context of Score-P," in *Using and Improving OpenMP for Devices, Tasks, and More*, ser. Lecture Notes in Computer Science, L. DeRose, B. R. d. Supinski, S. L. Olivier, B. M. Chapman, and M. S. Mller, Eds. Springer International Publishing, Sep. 2014, no. 8766, pp. 161–172, dOI: 10.1007/978-3-319-11454-5_12.

[16] J. Protze, S. Atzeni, D. H. Ahn, M. Schulz, G. Gopalakrishnan, M. S. Mller, I. Laguna, Z. Rakamari, and G. L. Lee, "Towards Providing Low-overhead Data Race Detection for Large OpenMP Applications," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 40–47.

APPENDIX

A. Project Proposal Text

The project work involves extending OMPT, an API conceived by the OpenMP Tools group, with new interfaces for performance metrics that can be used to construct grain graphs.

The grain graph is a recent visualization method for diagnosing performance problems in OpenMP programs. Structurally, the grain graph is a directed acyclic graph whose nodes represent *grains* – task and parallel for-loop chunk instances, and edges represent parent-child relationships between grains. Performance crippling problems such as low parallelism, work inflation and poor parallelization benefit at highlighted at the grain level on the grain graph, enabling average programmers to make portable optimizations for poor performing OpenMP programs, reducing pressure on experts and removing the need for tedious trial-and-error tuning. The grain graph is constructed using per-grain metrics obtained from an in-house runtime system called MIR – a decision that rules out visualizing grain graphs of programs executed on popular runtime systems such as those behind ICC (Intel) and GCC.

OMPT (OpenMP Tools) is an API conceived by the OpenMP Tools group to standardize the development of first-party performance analysis tools for OpenMP programs. One of the main problems addressed by OMPT is the prevalent tight coupling of performance analysis methods and runtime systems as exemplified by the case of grain graphs and MIR. OMPT functions allow first-party tools to profile program execution through asynchronous sampling or instrumentation. Recent versions of ICC and a version of LLVM forked by the OpenMP tools group implement OMPT. At the time of writing, OMPT is mature enough to be considered by the OpenMP committee for addition to the upcoming revision 5.0 of the OpenMP standard.

To the best of our knowledge, there are no first-party tools that visualize performance problems using profiling metrics generated by OMPT. Grain graphs can readily fill this void, provided OMPT is used to generate the profiling metrics required to construct grain graphs. However, it is unclear whether OMPT can provide the required profiling metrics for constructing grain graphs without modification. At present we understand that OMPT lacks thread scheduling and for-loop chunk information required by grain graphs. It is likely that a deeper, grain graph-centric investigation of OMPT will reveal more missing interfaces and opportunities for improving existing interfaces.

The aim of the project work is to evaluate the hypothesis that OMPT can be extended to generate the profiling metrics required to construct grain graphs. The hypothesis will be evaluated using these high-level steps. First, the student conducts a review of the state-of-the-art in OpenMP performance profiling methods that are of relevance to OMPT. Next, the student identifies the profiling metrics required for constructing grain graphs that OMPT does not provide. Next, the student defines new interfaces that provide the missing data while respecting the API design philosophy of OMPT. If time permits, the student implements one or more of the newly defined interfaces, evaluates their overheads, and ensures that the overheads are within reasonable limits.