OMPT: OpenMP Tools Application Programming Interfaces for Performance Analysis

Alexandre Eichenberger[*], John Mellor-Crummey[†], Martin Schulz[‡], Michael Wong[§],

Nawal Copty[¶], John DelSignore[‖], Robert Dietrich[**], Xu Liu[††], Eugene Loh[‡‡], Daniel Lorenz,
and other members of the OpenMP Tools Working Group

May 10, 2013

Abstract

To enable portable tools for performance analysis and debugging of OpenMP programs, we define an application programming interface (API) for tools that we propose for adoption as part of the OpenMP standard and supported by all OpenMP compliant implementation. The API is designed to allow performance tools to gather useful performance information from applications and to hide low-level OpenMP implementation idiosyncrasies from users. At the same time, the API is designed from the ground up to minimize the additional runtime overheads incurred while maintaining performance information. The API is designed to allow performance tools to gather useful performance information from applications and to hide low-level OpenMP implementation idiosyncrasies from users. At the same time, the API is designed from the ground up to minimize the additional runtime overheads incurred while maintaining performance information.

1 Introduction

There are two parts to the proposed interface: OMPT—a first-party API for performance tools, and OMPD—a shared-library plugin for debuggers that enables a debugger to inspect and control execution of an OpenMP program. This paper is about OMPT, the API for performance tools. A separate paper will describe OMPD.

1.1 OMPT

The design of OMPT is based on experience with two prior efforts to define a standard OpenMP tools API: the POMP API [4] and the Sun/Oracle Collector API [2, 3]. POMP is geared toward trace-based measurements, which has the shortcoming that its overhead can be significant because operations to be traced, e.g., an iteration of an OpenMP work-sharing loop, can take less time than recording an event in a trace. As an alternative to POMP's trace-based approach, the Sun/Oracle Collector API was designed primarily to support measurement and attribution of performance information using asynchronous sampling of call stacks. This sampling-based design enables construction of tools that attribute costs to full calling contexts without the drawbacks of tracing; namely, tools can record compact profiles with low runtime

[*]IBM T.J. Watson Research Center
[†]Rice University
[‡]LLNL
[§]IBM Canada
[¶]Oracle
[‖]Rogue Wave
[**]TU Dresden, ZIH
[††]Rice University
[‡‡]Oracle
Juelich Supercomputer Center

overhead. However, not all events can be traced prohibiting the implementation of subset of tools, such as trace analyzers or verification tools. OMPT builds on ideas of both POMP and the Sun/Oracle collector API to support asynchronous sampling and optional trace event generation. It extends them with support for *blame shifting* [5, 6] which shifts attribution of costs from symptoms to causes. The OMPT interface can be implemented either in entirely the compiler or entirely the OpenMP runtime system, as well as using a hybrid compiler/runtime option.

Most routines described in the OMPT API are intended only for use by tools rather than for direct use by applications. As a result, all OMPT API functions have a C binding only. A Fortran binding is provided only for a few application-facing inquiry and control functions, described in Section 6.

1.1.1 Design Objectives

OMPT tries to satisfy several design objectives for a performance tool interface for OpenMP. These objectives are listed in decreasing order of importance.

- The API should enable tools to gather sufficient information about an an OpenMP program executing under control of an OpenMP runtime system to associate costs with the program and the runtime system.
  - The API should provide an interface sufficient to construct low-overhead performance tools based on asynchronous sampling.
  - The API should enable a profiler that uses call stack unwinding to identify which frames in its call stack correspond to routines in the OpenMP runtime system.
  - An OpenMP runtime system should associate the activity of a thread at any point in time with a *state*, e.g., idle, which will enable a performance tool to interpret program execution behavior.
  - Certain API routines must be defined as thread-safe so that they can be invoked in a signal handler by a profiler as part of processing asynchronous events, e.g., handling sample events.
- Incorporating support for the API in an OpenMP runtime system should add negligible overhead to an OpenMP runtime system if the interface is not in use by a tool.
- The API should define support for trace based performance tools.
- Adding the API to an OpenMP implementation must not impose an unreasonable development burden on implementer.
- The API should not impose an unreasonable development burden on tool implementers.

1.1.2 Interface

To support the OMPT interface for tools, an OpenMP runtime system has two responsibilities: maintain information about the state of each OpenMP thread and provide a set of API calls that tools can use to interrogate the OpenMP runtime. Maintaining information about the state of each thread in the runtime system is not free and thus an OpenMP runtime system need not maintain state information unless a tool has registered itself, an environment variable directed the tool to track runtime state, or a debugger has demanded that runtime state information be maintained. Without any explicit request for tool support to be enabled, an OpenMP runtime need not maintain any information to support tools and may provide trivial (and thus, perhaps useless) answers to any API queries.

1.2 Document Roadmap

The document outlines aspects of the OMPT tools API. Section 2 describes state information maintained by the OpenMP runtime system for use by tools. Section 3 describes callback events for tools supported by the OpenMP runtime system. Section 4 describes tool data structures. Section 5 describes runtime system inquiry operations for tools. Section 6 describes runtime system inquiry and control operations available to applications.

2 Runtime State

To enable a tool to understand what an OpenMP thread is doing, when tools support has been turned on, an OpenMP runtime will maintain state information for each OpenMP (master, worker, or idle) thread that can be queried by a tool. The state maintained for each thread by the OpenMP runtime is an approximation of the thread's instantaneous state. When a thread not associated with the OpenMP runtime queries its state, the runtime returns `ompt_state_undefined`.

To enable low overhead implementations, an OpenMP runtime has some flexibility as to if and when it must report thread state transitions. For example, consider when a thread acquires a lock. One compliant runtime may transition the thread state to `ompt_state_wait_lock` early before attempting to acquire a lock. Another compliant runtime may transition a thread state to `ompt_state_wait_lock` late only if the thread begins to spin or block to wait for an unavailable lock. A third compliant runtime may transition the state to `ompt_state_wait_lock` even later - only after a thread waits for a significant amount of time.

Each thread maintains not only a state but also an `ompt_wait_id_t` identifier. When a thread is waiting for a lock, critical region, ordered, or atomic, and the thread is in the corresponding wait state, then the thread's `wait_id` field must point to the lock, critical region identifier, or atomic variable upon which the thread is waiting. A thread's `wait_id` is meaningless if the thread is not in a wait state.

State values 0 to 127 are reserved for current OMPT states and future extensions.

The supported states include Work States, Wait States for Non-Mutex, Wait States for Mutex, Overhead State and Miscellaneous State. Further details can be found in the draft OpenMP Tools Technical report [1].

3 Events

This section describes callback events that an OpenMP runtime may provide for use by a tool. Each callback has a particular type signatures defined for it.

There are two classes of events: mandatory events and optional events. Mandatory events must be implemented in any compliant runtimes implementations. Optional events are grouped in sets of related events. While each event can be individually included or omitted, we encourage tools to consider implementing all or none of the events in a given set.

A callback need not be registered for an event. An OpenMP runtime system will not make any callback unless a tool has registered to receive it.

3.1 Mandatory Events

The following events are mandatory and must be supported by a compliant OpenMP runtime system. These are create and exit events relating to Threads, Parallel Regions, and Tasks. The other are Application Tool Control events and Termination events. Further details can be found in the draft OpenMP Tools Technical report [1].

3.2 Optional Events

This section describes two sets of events. One set of events is used by sampling-based performance tools that employ a strategy known as blame shifting to attribute waiting to activity in contexts that cause other threads to wait rather than the contexts in which the waiting is observed.

Supporting any of the events in this section is optional for a compliant OpenMP runtime system.

3.2.1 Events for Blame Shifting

This section describes synchronous events used by sampling-based performance tools that employ 'blame shifting' to transfer blame for waiting from contexts where waiting is observed to code responsible for the waiting.[1] Using these callbacks, a tool employing blame shifting can attribute time a thread spends waiting for a lock to the context of the lock holder. Similarly, time threads spend waiting at a barrier can be

[1] Blame shifting has previously been demonstrated to be effective for attributing costs associated with threads idling while waiting to steal work in a work-stealing runtime [5], and spin waiting to acquire a lock [6]

attributed back to to code being executed by working threads while other threads wait. There are Idle State Entry/Exit, Barrier Idling, Taskwait Idling, Taskgroup Idling, Lock Release, Critical Release, Ordered Release, and Atomic Release. Further details can be found in the draft OpenMP Tools Technical report [1].

3.2.2 Events for Trace-based Measurement Tools

The following are synchronous events to support trace-based measurement of tasks. There are Task Creation and Destruction, Lock Creation and Destruction, Loops Begin and End, Sections Begin and End, Single Block Begin and End, Master Block Begin and End, Barrier Begin and End, Taskwait Begin and End, Taskgroup Begin and End, Lock Wait and Acquire, Nested Lock, Critical Section Wait and Acquire, Ordered Section Wait and Acquire, Atomic Block Wait and Acquire, and Flush. Further details can be found in the draft OpenMP Tools Technical report [1].

4 Tool Data Structures

4.1 Thread and Task Data

Each OpenMP thread and task instance provides an `ompt_data_t` data structure, which is a union of an integer and a pointer.

```
typedef union ompt_data_u {
    uint64_t value;    /* data under tool control    */
    void *ptr;          /* pointer under tool control */
} ompt_data_t;
```

The lifetime of the structure begins when a thread/task instance is created and ends when the instance is destroyed. While the value of a structure is preserved over the lifetime of the thread or task with which it is associated, tools should not assume that the address of an `ompt_data_t` structure remains constant over its lifetime.

When a thread/task instance is created, the callback associated with event creation must initialize the `ompt_data_t` structure. If there is no callback associated with this event, the OpenMP runtime initializes the structure value field to 0. The address of the `ompt_data_t` structure is passed to callbacks associated with the creation/destruction of threads/tasks. The address of the structure can also be retrieved on demand, e.g., by invoking an inquiry function in a signal handler.

If the `ompt_data_t` value field is 0 for a thread or task at the point that an exit callback would be made, the exit callback is not invoked. The tool is responsible for coordinating any concurrent accesses to `ompt_data_t` structures.

4.2 Parallel Region Identifier

Each OpenMP parallel region instance has an associated `ompt_parallel_id_t` that uniquely identifies the region instance.

```
typedef uint64_t ompt_parallel_id_t;
```

The `ompt_parallel_id_t` for a parallel region instance is unique across all instances of all parallel regions. The value of this structure is defined when a parallel region instance is created and passed to callbacks associated with creation/destruction of the parallel region instance. A parallel region's ID can be retrieved on demand, e.g., by invoking an inquiry function in a signal handler. Tools should not assume that `ompt_parallel_id_t` values for adjacent region instances are consecutive.

4.3 Wait Identifier

Each thread instance provides a `ompt_wait_id_t` data structure, which identifies what caused a thread to wait.

exit / reentry	reentry = null	reentry = defined
exit = null	case 1) initial task in user code case 2) explicit task that is created but not yet scheduled	initial task in runtime because of a parallel region or a task creation
exit = defined	non-initial task in user code	non-initial task in runtime because of a parallel region or a task creation

Table 1: Meaning of various values for `exit_runtime_frame` and `reenter_runtime_frame`.

```
typedef uint64_t ompt_wait_id_t;
```

For example, when a thread is waiting for a lock, this structure identifies the address of the lock. This structure is undefined when a thread is not in a wait state. The value of the `ompt_wait_id_t` structure is passed to callbacks associated with wait events, and also can be retrieved on demand, e.g., by invoking an inquiry function in a signal handler.

4.4 Pointers to Support Classification of Stack Frames

Each implicit or explicit task instance provides an `ompt_frame_t` data structure which contains pointers to OpenMP runtime procedure frames that appear above and below procedure frames associated with user task code.

```
typedef struct ompt_frame_s {
    void *exit_runtime_frame;    /* next frame is user code     */
    void *reenter_runtime_frame; /* previous frame is user code */
} ompt_frame_t;
```

The structure's lifetime begins when a task instance is created and ends when the task instance is destroyed. While the value of the structure is preserved over the lifetime of the task, tools should not assume that the address of a structure remains constant over its lifetime. Frame data is passed to some callbacks; it can also be retrieved for a task (e.g. by a signal handler). Frame data contains two components:

`exit_runtime_frame` This value is set once, the first time that a task exits the runtime to begin executing user code. This field points to the stack frame of the runtime procedure that called the user code. This value is NULL until just before the task exits the runtime.

`reenter_runtime_frame` This value is set each time that current task re-enters the runtime to create new (implicit or explicit) tasks. This field points to the stack frame of the runtime procedure called by a task to re-enter the runtime. This value is NULL until just after the task re-enters the runtime.

Advice to tool implementers: A monitoring tool using asynchronous sampling can observe values of `exit_runtime_frame` and `reenter_runtime_frame` before they are set to non-NULL values while in the runtime. Tools must be prepared to handle samples that occur in this brief window.

5 Inquiry Functions for Tools

Inquiry functions retrieve data from the execution environment for the tools. All inquiry functions are async signal safe.

5.1 Enumerate States Supported by an OpenMP Runtime

An OpenMP runtime system is allowed to support other states in addition to those described herein. For instance, a particular runtime system may want to provide finer-grain information about the nature of runtime overhead, e.g., to differentiate between the overhead associated with setting up a parallel region and the overhead associated with setting up a task. Further, a tool may not report all states defined herein, e.g., if state tracking for a particular state would be too expensive. To enable a tool to identify all states that an OpenMP runtime system implements, OMPT provides the following interface for enumerating all possibly reported runtime states.

```
_OMP_EXTERN int ompt_enumerate_state(
  int current_state,
  int *next_state,
  const char **next_state_name
);
```

When this interface is invoked for the first time, the value `ompt_state_first` should be supplied for `current_state`. The argument `next_state` is a pointer to an integer that will be set to the code for the next state in the enumeration. The argument `next_state_name` is a pointer to a location that will be filled in with a pointer to the name associated with `next_state`. Subsequent invocations of `ompt_enumerate_state` should pass the code returned in `next_state` by the prior call. The enumeration is complete when `ompt_state_last` is returned in `next_state`. The canonical way to enumerate the states supported by an OpenMP runtime system is shown below:

```
int state;
const char *state_name;
for (int ok = ompt_enumerate_state(ompt_state_first, &state, &state_name);
     ok && state != ompt_state_last;
     ompt_enumerate_state(state, &state, &state_name)) {
        // tool notes that the runtime supports ompt_state_t "state"
        // associated with "state_name"
}
```

5.2 Thread Data Inquiry

Function `ompt_get_thread_data` is an inquiry function to access data stored by the OpenMP runtime system for the current thread for use by a tool.

```
_OMP_EXTERN ompt_data_t *ompt_get_thread_data(void);
```

This inquiry function returns NULL prior to OpenMP initialization or when no tool is attached to the runtime. This function is async signal safe.

5.3 Thread State Inquiry

Function `ompt_get_state` is the inquiry function to determine the state of the current thread.

```
_OMP_EXTERN ompt_state_t ompt_get_state(
  ompt_wait_id_t *wait_id
);
```

The location specified by `wait_id` is updated point to the wait identifier associated with the current state, if any, or NULL otherwise. This function returns `ompt_state_undefined` prior to OpenMP initialization or when no tool is attached to the runtime. This function is async signal safe.

ancestor level value	meaning
0	current parallel region
1	parallel region directly enclosing region at ancestor level 0
2	parallel region directly enclosing region at ancestor level 1
...	

Table 2: Meaning of different values for the `ancestor_level` argument to `ompt_get_parallel_function`.

5.4 Parallel Region Inquiry

The OMPT interface defines two inquiry functions to access data stored by the OpenMP runtime for parallel regions. The first, `ompt_get_parallel_id`, returns the unique parallel id associated with this instance of the parallel region:

```
_OMP_EXTERN ompt_parallel_id_t ompt_get_parallel_id(
  int ancestor_level
);
```

Outside a parallel region, `ompt_get_parallel_id` should return 0. If a thread is in the idle state, then `ompt_get_parallel_id` should return 0. In all other cases, the thread should return the state of the enclosing parallel region, even if the thread is waiting at a barrier.

The second, `ompt_get_parallel_function`, returns a pointer to the compiler generated function used by the OpenMP runtime to encapsulate the code of the parallel region, if any, and NULL otherwise.

```
_OMP_EXTERN void *ompt_get_parallel_function(
  int ancestor_level
);
```

Both of the functions take an ancestor level as an argument. By specifying different values for ancestor level, one can access information about each parallel region, even if parallel regions are nested. The meaning of different values for the `ancestor_level` argument to `ompt_get_parallel_function` is given in Table 2.

These functions return the value 0 when requesting higher levels of ancestry than available, prior to OpenMP initialization, or when no tool is attached to the OpenMP runtime. These functions are async signal safe.

5.5 Task Region Inquiry

The OMPT interface defines three inquiry functions to access data stored by the OpenMP runtime for task regions. Function `ompt_get_task_data` returns the tool data associated with a given task. Function `ompt_get_task_frame` returns the tool frame associated with a given task.

```
_OMP_EXTERN ompt_data_t *ompt_get_task_data(
  int ancestor_level
);

_OMP_EXTERN ompt_frame_t *ompt_get_task_frame(
  int ancestor_level
);
```

The value returned by the `ompt_get_task_function` indicates the compiler-generated function used by the OpenMP runtime to encapsulate the code of the task construct, if any, and NULL otherwise.

```
_OMP_EXTERN void *ompt_get_task_function(
  int ancestor_level
);
```

ancestor level value	meaning
0	current task
1	direct parent of task at ancestor level 0
2	direct parent of task at ancestor level 1
...	

Table 3: Meaning of different values for the `ancestor_level` argument to `ompt_get_task_function`.

The meaning of different values for the `ancestor_level` argument to `ompt_get_task_function` is given in Table 3.

These functions return the value 0 when requesting higher levels of ancestry than available, prior to OpenMP initialization, or when no tool is attached to the OpenMP runtime. These functions are async signal safe.

5.6 Tool Support Version Inquiry

The function `ompt_get_ompt_version` returns the version of the OMPT interface supported by the runtime.

```
_OMP_EXTERN int ompt_get_ompt_version(void);
```

The version of OMPT described by this document is known as version 1.

6 Inquiry and Control Functions for Applications

The functions described in this section are the only ones with a Fortran interface in addition to a C/C++ interface.

6.1 Runtime Version Inquiry

The function `ompt_get_runtime_version`, with the type signature shown below

```
_OMP_EXTERN int ompt_get_runtime_version(char *buffer, int length);
```

fills `buffer` with a version-specific string of at most `length` characters. The suggested format is

```
<vendor>-<major version number>.<minor version number>[-<optional feature]*
```

Namely, a vendor name, major and minor version numbers, and, optionally, a list of zero or more features, separated by dashes. As an example, IBM's OpenMP runtime might return the following version string " IBM-1.1-core=1-blame=1-trace=0", indicating that IBM's OpenMP runtime supports the OMPT tools API core augmented with support for blame shifting, but not support for detailed tracing.

6.2 Tool Control

The function `ompt_control` can be called by an application to pass control information to a tool. The signature for this function is shown below:

```
_OMP_EXTERN void ompt_control(uint64_t command, uint64_t modifier);
```

A classic use case for the `ompt_control` routine might be for an application to start and stop data collection by a tool.

return code	meaning
0	event may occur; no callback is possible
1	event will never occur in runtime
2	event may occur; callback invoked when convenient
3	event may occur; callback always invoked when event occurs

Table 4: Meaning of return codes for `ompt_set_callback`.

7 Initializing OMPT Support for Tools

An OpenMP runtime need not maintain information to support tools and may provide trivial (and thus, perhaps useless) answers in response to invocations of any API inquiry functions. Section 7.1 describes normal initialization for a tool. A further section describing the environment variable control over tool initialization and the tool initialization API for debugger can be found in the draft OpenMP Tools Technical Report [1].

7.1 Initialization of a Tool

A tool must register itself with an OpenMP runtime system and then specify callbacks for events of interest. Section 7.1.1 describes the initializer for a tool. Section 7.1.2 describes registration of callbacks for OMPT events.

7.1.1 Initializer for a Full-featured Tool

A tool must register itself with an OpenMP runtime by overriding the following weak symbol:

```
_OMP_EXTERN int ompt_initialize(void);
```

The role of `ompt_initialize` is to register callbacks for specific events, e.g., creating a parallel region. A tool must register a callback for every event of interest using `ompt_set_callback`, as described in Section 7.1.2. The OpenMP runtime system defines a weak symbol version of `ompt_initialize` that returns 0; a tool-provided version must return 1.

Since only one tool-provided definition of `ompt_initialize` will be seen by an OpenMP runtime, only one tool can be registered. Ordinarily, `ompt_initialize` will be invoked by an OpenMP runtime immediately after the runtime initializes itself.

An OpenMP runtime system *may* allow registration of a tool after initialization of the OpenMP runtime at a *clean point*. An OpenMP runtime is said to be at a clean point when no pthread is inside a parallel region. An OpenMP runtime system will not necessarily attempt to register a tool at a clean point unless a debugger has previously called `ompd_enable(true)`as described in the draft OpenMP Tools Technical Report [1].

After a process fork, if OpenMP is re-initialized in the child process, the OpenMP runtime system in the child process will call `ompt_initialize` under the same conditions as it would for any process.

7.1.2 Callback Registration for a Full-featured Tool

Full-featured tools register callbacks to receive notification of various events that occur as an OpenMP program executes. A tool uses `ompt_set_callback` to register callback functions.

```
_OMP_EXTERN int ompt_set_callback(
  ompt_event_t event,
  ompt_callback_t callback
);
```

The function `ompt_set_callback` may only be called within the implementation of `ompt_initialize` provided by a tool, as described in Section 7.1.1 The possible return codes for `ompt_set_callback` and their

meaning is shown in Table 4. Registration of supported callbacks may fail if this function is called outside `ompt_initialize`. The `ompt_callback_t` type for a callback does not reflect the actual signature of the callback; OMPT uses this generic type to avoid the need to declare a separate registration function for each actual callback type.

The function `ompt_get_callback`, as shown below, may be called at any time to inspect whether a callback has been registered or not. If a callback has been registered, `ompt_set_callback` will return 1 and set `callback` to the address of the callback function; otherwise, `ompt_set_callback` will return 0.

```
_OMP_EXTERN int ompt_get_callback(
  ompt_event_t event,
  ompt_callback_t *callback
);
```

8 Conclusion

This paper defines an application programming interface (API) for tools that we propose for adoption as part of the OpenMP standard and supported by all OpenMP compliant implementation. Further details can be found in the draft OpenMP Tools Technical report [1].

References

[1] A.Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, D. Lorenz, and other members of the OpenMP Tools Working Group. Ompt and ompd: Openmp tools application programming interfaces for performance analysis and debugging, 2013. `http://openmp.org/mp-documents/ompt-tr.pdf`.

[2] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin. An OpenMP runtime API for profiling. Sun Microsystems, Inc.. OpenMP ARB White Paper. Available online at `http://www.compunity.org/futures/omp-api.html`.

[3] G. Jost, O. Mazurov, and D. An Mey. Adding new dimensions to performance analysis through user-defined objects. In *Proceedings of the 2005 and 2006 International Conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 255–266, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.

[5] N. R. Tallent and J. M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 229–240, New York, NY, USA, 2009. ACM.

[6] 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*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.