Test Suite for Minimally-Compliant OMPT Implementation*

Matt Zhenghuan Zhao John Mellor-Crummey

January 6, 2015

1 Introduction

OMPT defines an application programming interface through which one can write powerful monitoring or analysis software for OpenMP applications. However, traditionally various vendors usually have different OpenMP runtime, and as of now, there is not an easy way to automatically determine whether a particular OpenMP fully supports the OMPT standard. This motivates us to develop an automatic test suite for that purpose. Our test suite is fully automatic. After running it, you will be able to tell whether the runtime you are testing against is a minimally-compliant implementation of the OMPT specification.

1.1 Obtaining the Test Suite

The test suite can be obtained by cloning the master branch of its `git` repository, as shown below:

```
git clone https://code.google.com/p/ompt-test-suite/
```

1.2 Test Suite Organization

The regression directory contains three subdirectories: doc, mandatory, optional, and utils. The `doc` subdirectory contains the source for this document. The mandatory subdirectory contains subdirectories with regression tests for all of the mandatory APIs that a runtime needs to support. Inside the mandatory directory, there are subdirectories for OMPT initialization, events, and inquiry operations. At the top level, the utils subdirectory contains support code for the regression tests, including support for error reporting, sampling, timing, and regular expression matching.

1.3 Building the Test Suite

At present, the test suite contains Makefiles for two OpenMP runtimes: IBM's lightweight OpenMP runtime (LOMP), and Intel's open-source OpenMP runtime. The build system supports

- IBM's xlc compiler with LOMP,
- icc with Intel's OpenMP runtime, and
- gcc with Intel's OpenMP runtime.

1.3.1 Configuring an OpenMP Runtime System

Before building the OMPT test suite, one must first set an environment variable to specify the path to the runtime system.

- On an IBM POWER platform, set `IBM_COMPILER_ROOT` to the root of the IBM compiler and runtime tree, e.g., `/opt/apps/ibm`. The compiler and runtime tree must contain support for OMPT. The presence of OMPT support can be identified by the presence of subdirectories `ompt_light` and `opmt_full` in `IBM_COMPILER_ROOT/lib64`.

*This document is an incomplete draft that is meant to serve as a starting point for useful documentation

- On an `x86_64` platform, set `INTEL_OMP_ROOT` to the root of Intel open-source OpenMP runtime directory, e.g., the directory obtained by running `git clone https://code.google.com/p/ompt-intel-openmp`, and then running `make` in the `itt/libomp_oss` subdirectory.

1.3.2 Compiling the Test Suite

In the top level of the test suite, run the following commands, where xxx is replaced by the appropriate compiler: xlc, icc, or gcc.

```
cd regression
make compiler=xxx
```

The executables generated will reside under the same paths as their sources.

1.3.3 Running the Test Suite

After building the test cases, you can run them by executing the following command in the top-level regression directory:

```
make test
```

The Makefile will use the Python script `run_all_tests.py` located in the `utils` subdirectory to run all of the tests. Alternatively, you can execute a test directly by running it from the command line. For example, one may run the regression test for `ompt_get_parallel_id` by running the executable

```
./mandatory/inquiry_functions/test_ompt_get_parallel_id
```

1.3.4 How to interpret the results

After you execute a single test case, there is three possible return code (0, 255, 254). 0(CORRECT) means the runtime has cleanly passed this test case, 255(IMPLEMENTED_BUT_INCORRECT) means that the runtime has implemented the API the test case is testing but has failed, and finally 254(NOT_IMPLEMENTED) means that the test case hasn't implemented the API being tested at all.

Some test cases might cause segmentation fault when it fails on some runtime, we try to catch this by catching all SIGSEGV signals, and in this case, the return value is always 254 no matter what has actually happened, as it's impossible to recover the original program context at that point. The most robust way in this case is to run the tests with the python script, which then runs the tests in a different process, so all exceptions/seg fault are handled gracefully.

2 Test Case Details

This section enumerates test cases included in the test suite and describes what they are testing. API descriptions are copied directly from the OMPT specification [**?**].

2.1 Mandatory Events

A minimally compliant implementation of OMPT has to implement eight mandatory events: ompt_event_control, ompt_event_runtime_shutdown, ompt_event_parallel_begin, ompt_event_parallel_end, ompt_event_task_begin, ompt_event_task_end,
ompt_event_thread_begin, and ompt_event_thread_end.

2.1.1 ompt_event_control

- be able to register ompt_event_control event with a callback.
- the arguments passed to the callback are the values passed by the user to ompt_control.

2.1.2 ompt_event_runtime_shutdown

- be able to register ompt_event_runtime_shutdown event with a callback.
- exit with 0 only inside the callback in order to test whether the callback is executed before main returns.

2.1.3 ompt_event_parallel_begin

- be able to register ompt_event_parallel_begin event with a callback.
- running some inquiry functions to test whether the callback executes in the context of the task that encountered the parallel construct.
- validate arguments parent_task_id and parent_task_frame passed into the callback by comparing them with the results of the inquiry function inserted in the program.
- no duplicated parallel ids.
- nested parallel regions with depth 3 and with n threads at each level should generate a total of of (1+(1+n)*n) calls to the parallel begin callback.

2.1.4 ompt_event_parallel_end

- be able to register ompt_event_parallel_end event with a callback.
- running some inquiry functions to test whether the callback executes in the context of the task that encountered the parallel construct.
- parallel_id passed in parallel_end callback has to appear first in parallel_begin callback.
- equal number of calls to parallel_begin callback and parallel_end callback.

2.1.5 ompt_event_task_begin

- be able to register ompt_event_task_begin event with a callback.
- running some inquiry functions to test whether the callback executes in the context of the task that encountered the parallel construct(test by calling ompt_get_task_id, and ompt_get_task_frame in the callback).
- validate arguments parent_task_id and parent_task_frame passed into the callback by comparing them with the results of the inquiry function inserted in the program.
- no duplicated new task ids.

2.1.6 ompt_event_task_end

- be able to register ompt_event_task_end event with a callback.
- skip the context test because the callback can execute in the context of an arbitrary task on the thread that completed the explicit task
- task_id passed in task_end callback has to appear first in task_begin callback.
- equal number of calls to task_begin callback and task_end callback.

2.1.7 ompt_event_thread_begin

- be able to register ompt_event_thread_begin event with a callback.
- set up a nested parallel region(nested option is disabled) with depth 3 and n threads at each level, expect to see n calls to the thread_begin callback.
- thread with thread number 0 should have type ompt_thread_initial, while others should have ompt_thread_worker or ompt_thread_other(in our test case, we should expect to see ompt_thread_worker).

2.1.8 ompt_event_thread_end

- be able to register ompt_event_thread_end event with a callback.
- expect to see type ompt_thread_worker at the thread_end callback in our test case.
- thread_id passed into thread_end callback should appear first in thread_begin callback

2.2 Inquiry Functions

A minimally compliant implementation of OMPT also has to implement six inquiry functions: ompt_get_idle_frame, ompt_get_parallel_id, ompt_get_task_id, ompt_get_parallel_team_size, ompt_get_task_frame, ompt_get_state.

2.2.1 ompt_get_idle_frame

- be able to lookup ompt_get_idle_frame using the ompt_function_lookup_t function.
- a call to ompt_get_idle_frame on initial thread will always return NULL(test this behavior both outside and inside nested parallel regions).
- with omp_nested disabled, the new team that is created by a thread encountering a parallel construct inside a parallel region will consist only of the encountering thread. Thus, in the scenario, we should expect a consistent lowest idle frame for each thread.

2.2.2 ompt_get_parallel_id

- be able to lookup ompt_get_parallel_id using the ompt_function_lookup_t function.
- outside a parallel region, ompt_get_parallel_id should return 0.
- create a nested parallel region with depth 3, and call ompt_get_parallel_id with various depths to test whether it returns consistent results.

2.2.3 ompt_get_task_id

- be able to lookup ompt_get_task_id using the ompt_function_lookup_t function.
- if a tool requests a task ID at a depth deeper than the dynamic nesting of implicit and explicit tasks in the current execution context, ompt_get_task_id will return 0.
- similar to ompt_get_parallel_id, create a nested task with depth 3, and call ompt_get_task_id to see if it returns consistent results.

2.2.4 ompt_get_parallel_team_size

- be able to lookup ompt_get_parallel_team_size using the ompt_function_lookup_t function.
- when omp_nested is disabled, the current team size is expected to be the same as its parents team size.
- when omp_nested is enabled, the current team size should be different from the parents team size if we allow different degrees of parallelism at the two levels.
- function returns the value -1 when requesting higher levels of ancestry than exist.

2.2.5 ompt_get_task_frame

- be able to lookup ompt_get_task_frame using the ompt_function_lookup_t function.
- (please refer to the graph on page 32 in OpenMP Technical Report on the OMPT Interface) [**?**].
- the initial thread should see two frames r1, r2 on the stack such that r1 has reenter but not exit address and r2 has exit but not reenter address, and r1s reenter address should be less than the r2s exit address.
- test the behavior across threads; when thread 2 reach code C, it should see frames r4, r3, r1 on the stack but not r2(r2 is initiated after the runtime procedure created thread 2).
- function returns the value NULL when requesting higher levels of ancestry than exist.

2.2.6 ompt_get_state

- be able to lookup ompt_get_state using the ompt_function_lookup_t function.
- (Sampling-based testing, matching collect states on the master thread with a regular expression).
- outside a parallel region, expect sampled states to include ompt_state_work_serial.
- inside a parallel region with pure serial work, expect sampled states to include ompt_state_work_serial.
- in a parallel region doing a reduction, expect sampled states to include ompt_state_wait_barrier(reduction is fast and can be hard to catch, so use zero or more matching).
- in a parallel region with an explicit barrier, expect sampled states to include ompt_state_wait_barrier_explicit or ompt_state_wait_barrier.
- in a parallel region with an implicit barrier, expect sampled states to include ompt_state_wait_barrier_implicit or ompt_state_wait_barrier.
- in a task group with explicit taskwait construct, expect sampled states to include ompt_state_wait_taskwait followed by ompt_state_wait_taskgroup.
- in a parallel region with a single lock(ompt_lock_t object), expect sampled states to include ompt_state_wait_lock.
- in a parallel region with a nested lock, expect sampled states to include ompt_state_wait_nest_lock.
- in a parallel region with a critical construct, expect sampled states to include ompt_state_wait_critical.
- in a parallel region with atomic update, expect sampled states to include ompt_state_wait_atomic.
- in a parallel region with ordered construct, expect sampled states to include ompt_state_wait_barrier.

3 Future work

At present, the regression test suite is still in a fledgling state. Much work remains to make it complete (covering the whole API) and thorough (carefully validating as much of the semantics as possible.) At present, most of the optional events lack regression tests. Some of the tests, e.g., `ompt_test_thread_begin`, are relatively naive and don't test the full semantics associated with events. Result logging needs to be improved: some tests report the same error more than once because more than one thread encounters the code that triggers it. The tests should all include a timeout to protect against a test case that diverges.

Acknowledgments

The authors would like to acknowledge Laksono Adhianto at Rice University and Alexandre Eichenberger at IBM Research for laying the groundwork for this test suite. Some of test cases are directly based on their prior work.

References

[1] Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz. OMPT: An OpenMP® Tools Application Programming Interface for Performance Analysis. OpenMP Technical Report 2. April 2014.