How to get most of OMPT (OpenMP Tools Interface)

Joachim Protze (protze@itc.rwth-aachen.de), Tim Cramer, Jonas Hahnfeld, Simon Convent, Matthias S. Müller

Hands-on

Clone instructions:
bit.ly/OMPT-Handson
What is OMPT?
Tools interface in the OpenMP spec

- Makes your tool compatible with any standard compliant OpenMP runtime
- Invokes callbacks for defined OpenMP events (e.g. “parallel-begin”)
- Maintains tool data for OpenMP scopes (e.g. “parallel-blob”)
- Provides signal-safe inquiry functions to learn about OpenMP runtime information
Available runtime implementations, roadmap

- **IBM lightweight OpenMP runtime:**
  - OMPT implementation is available on early-access systems (e.g. LLNL)
  - Successfully used with OMPT-based ARCHER tool

- **LLVM/OpenMP runtime:**
  - [https://github.com/OpenMPToolsInterface/LLVM-openmp](https://github.com/OpenMPToolsInterface/LLVM-openmp)
  - towards_tr4 branch is up-to-date with currently voted internal OpenMP spec
    - TR4 + fixes
  - Just recently fixed some performance issues
  - Starting review process after LLVM release is finished
  - Reference OMPT-tool: runtime/test/ompt/callback.h

- **From the spec point of view:**
  - The internal spec evolved (LLVM runtime is up-to-date with internal spec)
  - Interface almost stable, but we expect some tiny improvements for SC‘17 release
Agenda

• Basic OMPT usage
• OMPT-based tracing/profiling tool
• Use multiple OMPT tools at the same time
• OMPT-based sampling tool
• OMPT for accelerators
Using OMPT


git clone --recursive [https://git.rwth-aachen.de/OpenMPTools/OMPT-Examples.git](https://git.rwth-aachen.de/OpenMPTools/OMPT-Examples.git)

Hands-on

`cmake required!`  
`OMPT-Examples $ bash bootstrap.sh`
OMPT tool initialization (example1/hello.c)

```c
#include <stdio.h>
#include <omp.h>
#include "initialization.h"

int main()
{
    #pragma omp parallel num_threads(2)
    {
        printf("Hello from thread %i of %i!\n", omp_get_thread_num(), omp_get_num_threads());
    }
    return 0;
}
```

- $CC -fopenmp -L../INSTALL/lib/ hello.c
- ./a.out

libomp init time: 0.001135
Hello from thread 0 of 2!
Hello from thread 1 of 2!
application runtime: 0.001877

**Important for static tool, if OpenMP runtime of the compiler has no OMPT support**

**Hands-on**

Execute example1 with static tool:
```
example1 $ make run-static
```
OMPT tool initialization (example1-initialization.h)

```c
typedef struct my_ompt_fns_t { ompt_initialize_t i; ompt_finalize_t f;
    double time; double init; } my_ompt_fns_t;

int ompt_initialize (ompt_function_lookup_t lookup, ompt_fns_t* fns)
{
    my_ompt_fns_t* data = (my_ompt_fns_t*) fns;
    data->init = omp_get_wtime();
    printf("libomp init time: %f\n", data->init - data->start);
    return 1; //success: activates tool
}

void ompt_finalize (ompt_fns_t* fns)
{
    my_ompt_fns_t* data = (my_ompt_fns_t*) fns;
    printf("application runtime: %f\n", omp_get_wtime() - data->init);
}

ompt_fns_t* ompt_start_tool (unsigned int omp_version, const char *
*runtime_version)
{
    static my_ompt_fns_t data = {&ompt_initialize, &ompt_finalize, 0, 0};
    data.start = omp_get_wtime();
    return (ompt_fns_t*) &data; //success: registers tool
}
```

§4.6: Tool callbacks may not use OpenMP directives or call any runtime library routines described in Section 3.
Bringing the tool into the game

- Link tool statically into the application
- Link tool dynamically into the application
  - Make sure tool is linked before the OpenMP runtime (check ldd)
  - If OpenMP runtime is linked statically, tool must be loaded before OpenMP runtime is initialized
  - For some compilers take care of „as-needed“!
- Ld-preload the shared tool library
- Use OMP_TOOL_LIBRARIES environmental variable to let the runtime load the shared tool library

- If you load multiple tools with the different mechanisms, it is not specified which tool is found.
- If a detected tool returns NULL on ompt_start_tool, the runtime continues to detect another tool.

Hands-on

Execute example1 with all mechanisms:
example1 $ make run
OMPT runtime entry points (example2/callback.c)

```c
static ompt_get_thread_data_t ompt_get_thread_data;
static ompt_get_unique_id_t ompt_get_unique_id;

//...
int ompt_initialize(
    ompt_function_lookup_t lookup,
    ompt_fns_t* fns)
{
    ompt_set_callback_t ompt_set_callback = (ompt_set_callback_t)
        lookup("ompt_set_callback");
    ompt_get_thread_data_t ompt_get_thread_data = (ompt_get_thread_data_t)
        lookup("ompt_get_thread_data");
    ompt_get_unique_id_t ompt_get_unique_id = (ompt_get_unique_id_t) lookup("ompt_get_unique_id");

    //...
}
```

- Function for registering callbacks (next slide)
- Unique integer identifier across OpenMP threads
- Thread-local storage for OpenMP threads
Registering callback functions (example2/callback.c)

```c
#define register_callback_t(name, type) 
do{
    type f_##name = &on_##name;
    if (ompt_set_callback(name, (ompt_callback_t)f_##name) == ompt_set_never) {
        printf("0: Could not register callback '" name "'\n"); 
    }
}while(0)

#define register_callback(name) register_callback_t(name, name##_t)

int ompt_initialize(ompt_function_lookup_t lookup, ompt_fns_t* fns) {
    //...
    register_callback(ompt_callback_implicit_task);
    register_callback(ompt_callback_parallel_begin);
    register_callback(ompt_callback_parallel_end);
    // register_callback_t(ompt_callback_sync_region_wait,
    //    ompt_callback_sync_region_t);
    //...
    return 1; //success
}
```

Ensure matching function signature, before casting to void*

Some function signatures are reused for multiple callbacks
Implementing callback functions (example2/callback.c)

```c
static void on_ompt_callback_implicit_task( ompt_scope_endpoint_t endpoint,
                                      ompt_data_t *parallel_data, ompt_data_t *task_data, unsigned int team_size,
                                      unsigned int thread_num )
{
    uint64_t tid = ompt_get_thread_data()->value;
    switch(endpoint)
    {
        case ompt_scope_begin:
            counter[tid].cc.implicit_task_scope_begin += 1;
            task_data->value = ompt_get_unique_id();
            break;
        case ompt_scope_end:
            counter[tid].cc.implicit_task_scope_end += 1;
            break;
    }
}
```

Some callbacks are used for begin- and end-event

Hands-on

Execute example2:
```
example2 $ make run
```
OMPT Multiplex

All sources available at (and already included in the Hands-on):

git clone https://git.rwth-aachen.de/OpenMPTools/OMPT-Multiplex.git
Sometimes a single tool is not enough

• Similar as in P^nMPI, the initial idea was to create an OMPT tool, that can load multiple client tools
• Configuration file to specify the details about client tools
• Data-blob is a vector of data-blobs
• OMPT^n coordinates the access to tool specific data-blobs
• Needs to provide specialized inquiry functions for all clients
Cascading OMPT tools

• OMPT-Multiplex
  – A tool can load another tool like the OpenMP runtime would do
  – Implemented as header-only
  – Looks for `<Tool-name>_TOOL_LIBRARIES`, to load another tool and execute `ompt_start_tool`
  – Unlimited cascading of tools possible

• Limitation:
  – Loading the same tool twice results in infinite recursion and SEGFAULT
    ➢ Build two versions of the same tool with different TOOL_LIBRARIES-var

• License: MIT license to allow broad usage
Details on multiplexing OMPT

- Multiplexing of callback invocation
  - `ompt_callback_parallel_begin(data->ptr->first)`
  - `ompt_callback_parallel_begin(data->ptr->second)`

- Multiplexing of runtime entry points
  - The lookup-function provides specialized runtime entry points for own and client tool
    - `client_get_parallel_info()` returns `(data->ptr->second)`
    - `ompt_get_parallel_info()` returns data

Tool (Multiplex)

Client Tool

Multiplexing diagrams:

1. OpenMP parallel
   - Application
   - `multiplex.h`
   - `ompt_callback_parallel_begin(data->ptr->first)`
   - `ompt_callback_parallel_begin(data->ptr->second)`

2. Client Tool
   - `ompt_callback_parallel_begin(data->ptr->first)`
   - `ompt_callback_parallel_begin(data->ptr->second)`
   - `client_get_parallel_info()` returns `(data->ptr->second)`
   - `ompt_get_parallel_info()` returns data
OMPT-Multiplex basic usage (example3/Makefile)

• Just define the intended name for TOOL_LIBRARIES-var and include the header in the tool source file, that implements ompt_start_tool:

```c
#define CLIENT_TOOL_LIBRARIES_VAR "COUNT_TOOL_LIBRARIES"
#include <ompt_multiplex.h>
```

• Even easier, define and include the header at compile time (see Makefile):

```bash
$(CC) -DCLIENT_TOOL_LIBRARIES_VAR="COUNT_TOOL_LIBRARIES" \
    -include ompt_multiplex.h callback.c -fPIC -c -o callback.o
```

• OMPT-Multiplex manages an individual data-pointer for each tool
• Callbacks are delivered first to the own tool, then to the client tool
• OMPT-Multiplex registers callbacks only for those that are registered by any of the tools

Hands-on

Execute example3:
example3 $ make run
OMPT-Multiplex advanced usage (example4 initialization.h)

- The basic mode needs to allocate ompt_data_t[2] for any new OpenMP scope
  - Entry for own data, entry for client data
- The advanced mode allows to add a field for the client data into the own data-blob
- Define an accessor for the client data field:

```c
#define OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_THREAD_DATA
#define OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_PARALLEL_DATA
#define OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_TASK_DATA
#define CLIENT_TOOL_LIBRARIES_VAR "INIT_TOOL_LIBRARIES"
#include <ompt_multiplex.h>
```

- The Init-tool doesn’t use the data-blob at all, so the client should access the data stored in the OpenMP runtime

**Hands-on**

Execute example4:
```
example4 $ make run
```
OMPT-Multiplex advanced usage 2 (example4/callback.c)

- We expect the tool to free the data-blob in the scope-end callback, therefore the client is called first for these events.
- In general, the advanced mode only makes sense, if a tool stores objects in the data-pointer.
- Provide a delete function, to avoid the inverted callback invocation order:

```c
static ompt_data_t* get_client_data(ompt_data_t*);
static void delete_data(ompt_data_t*);
#define OMPT_MULTIPLEX_CUSTOM_DELETE_THREAD_DATA delete_data
#define OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_THREAD_DATA get_client_data

//...
typedef struct my_ompt_data{
    uint64_t own;
    ompt_data_t client;
} my_ompt_data_t;

ompt_data_t* get_client_data(ompt_data_t* data){
    return &(((my_ompt_data_t*)data->ptr)->client);
}
void delete_data(ompt_data_t* data){free(data->ptr);}
```
Asynchronous tool activity
OMPT States (example5/sample.cc)

- OMPT enumerate states:

```cpp
std::map<int, std::string> ompt_state_map;
int ompt_initialize(...){ ...
    int state = omp_state_undefined;
    const char *state_name;
    ompt_enumerate_states_t ompt_enumerate_states = (ompt_enumerate_states_t) lookup("ompt_enumerate_states");
    while (ompt_enumerate_states(state, &state, &state_name)) {
        ompt_state_map[state] = std::string(state_name);
    }
}

- Store a copy of the state-map in the thread-blob (see thread-begin)

Destructors for global variables might already be executed before the last thread receives the last signal handler
OMPT sampling tool (example5/sample.c)

• The signal handler:

```c
static void handler(int sig, siginfo_t *si, void *uc)
{
    if (!ompt_get_thread_data || !ompt_get_state) return;
    ompt_data_t *data = ompt_get_thread_data();
    if (!data) return;
    threadData *thread_data = (threadData*)(data->ptr);
    ompt_wait_id_t waitId;
    omp_state_t state = ompt_get_state(&waitId);
    thread_data->ompt_thread_state_map[state]++;
}
```

- Gives a rough idea about the state of the OpenMP thread.
- Might be NULL at shutdown!
- Always access ptr! Hard to debug if you access data.

Hands-on

Execute example5:
```
eexample5 $ make run-sample
```
OMPT stack trace support

- Stack traces of OpenMP tasking applications are confusing
- OMPT can help cleaning up the stack trace
- OMPT can give additional stack information

```c
int fib(int i){
    if (i<=1) return 1;
    int a,b;
    #pragma omp task shared(a)
        a = fib(i-1);
    #pragma omp task shared(b)
        b = fib(i-2);
    #pragma omp taskwait
    return a+b;
}
int main(){
    int result;
    #pragma omp parallel sections
        result = fib(5);
    return result;
}
```

Execute example5:
example5 $ make run-sample-dt
OMPT for OpenMP Devices
OMPT target support

Tracing on Device with cross-compiled Tool

- All OMPT events can also occur on a target device
- Alternative 1: Additional library / tool on device to collect data on device
OMPT target support

Tracing on Device with Tracing API

- Alternative 2: Asynchronous buffer handling with OMPT
- With buffering API
  - No additional (vendor/hardware-dependent) library required anymore
  - Device-sided events are collected within the runtime
- Prototype support in Score-P
OMPT target support

Tracing on Device with Tracing API

- Execution call sequence

- Initial support for
  - FPGAs in Nano++ / Extrae (Llort et al., IWOMP 2016)
  - Intel Xeon Phi in LLVM / liboffload (https://github.com/OpenMPToolsInterface)
OMPT native target support

- Wrapper for a native accelerator tracing API (like CUPTI)
- Correlate HW counter to OpenMP scopes (e.g., target regions)
- Native record includes a describing string, a start and an end time
  - Allows a time line representation, which is meaningful to a user
  - Useful even if the tool does not fully understand the native record
- Prototype support in HPC Toolkit
Conclusions

• OMPT ready to build your host-focused tool on it
  – Implementation for devices is coming

• Event driven callback interface supports tracing/profiling tools

• Interface allows to stack multiple tools, even though not specified
  – Header-only implementation of OMPT-Multiplex is available

• Asynchronous inquiry functions support introspection with sampling tool

• OMPT for accelerators provides multiple workflows
  – Integrating native event information and OpenMP specific information
Thank you for your attention.

Find slides at:
bit.ly/OMPT-Handson