The host thread/process interface in OMPi

VVD & the OMPi team

March 2016
Versions: 5/2010, 4/2008, 1/2008, 5/2007, 12/2006.

Abstract

We describe in detail the thread/process interface in ORT, the host runtime library of OMPi. This guide is targeted towards developers who want to develop their own thread/process libraries and use them through OMPi.

1 Introduction

Since version 2.0.0, the runtime system of OMPi has undergone a major reorganization so as to allow support of attached OpenMP 4 *devices*. The entire runtime system is implemented in `runtime/` and consists of two major runtime subsystems:

- the *host* subsystem which represents the traditional OpenMP runtime, operating on the main system (host) processors/cores.
- the *devices* subsystem which consists of specialized modules that:
  - provide runtime support for codes executing on attached devices
  - serve as interfaces between the host runtime and the device runtimes.

This document refers to the *host* subsystem. Documentation regarding devices and their corresponding interfaces is provided elsewhere.

OMPi's host runtime system (ORT) is implemented in `runtime/host/` (files `ort*.c`). ORT uses multiple *execution entities* (EEs) to control the execution of OpenMP programs compiled by OMPi. In earlier versions, the only type of execution entity supported was threads. However, since V1.0.0, an EE can be either a *thread* or a *process*. Apart from the EEs, no matter what their type is, ORT also employs locks and other related primitives, *but*:

> *it does not implement those primitives.*

ORT relies on an *execution entity library* (EELIB) to provide those primitives. All such libraries consist of at least two files: `othr.c` and `ee.h`, which lie in the appropriate subdirectory of `runtime/host/`. For example, the default library which is based on POSIX threads is in `runtime/host/ee_pthreads/` in OMPi's source distribution.

When implementing a new EELIB, one must choose a name for it (say `foo`), create `runtime/host/ee_foo/` and provide `ortconf.foo`, `othr.c` and `ee.h` in there. The library can be built into OMPi by using:

```
./configure --with-ortlib=foo [plus any other options]
cd lib
make clean
make
make install
```

as explained in the `README` and `doc/runtime.txt` files shipped with OMPi.

In the next sections we describe what ORT expects from EELIB and what facilities ORT provides to it.

2 What ORT expects from EELIB

ORT expects basically few things from EELIB:

1. A `#define` that states the type of EE provided.
2. Code that implements locks
3. Code that creates and joins EEs

Optionally, EELIB may provide barriers (see Section 3.2) and tasks (see Section 3.3). A couple more things required are discussed in Section 2.4.

2.1 The type of EE

In the top of `ee.h`, there must exist either

```
#define EE_TYPE_PROCESS
```

or

```
#define EE_TYPE_THREAD
```

If none is given, the EEs are assumed to be threads.

2.2 Locks

Regarding locks, the library must provide the following 5 self-explanatory functions:

```
int othr_init_lock(othr_lock_t *lock, int kind);
int othr_destroy_lock(othr_lock_t *lock);
int othr_set_lock(othr_lock_t *lock);
int othr_unset_lock(othr_lock_t *lock);
int othr_test_lock(othr_lock_t *lock);
```

```
#define EETYPE THREAD

/* The data types */
typedef struct {
                  pthread_mutex_t lock, ilock;
                  pthread_cond_t cond;
                  pthread_t owner;
                  int count;
                } nestlock_t;

typedef struct {
  int   type;                  /* normal/spin/nested */
  union {
    pthread_mutex_t    normal;   /* normal lock */
    nestlock_t         nest;     /* nest lock */
    pthread_spinlock_t spin;     /* posix spin locks */
  } data;
} othr_lock_t;

/* The functions */
extern int othr_init_lock(othr_lock_t *lock, int kind);
extern int othr_destroy_lock(othr_lock_t *lock);
extern int othr_set_lock(othr_lock_t *lock);
extern int othr_unset_lock(othr_lock_t *lock);
extern int othr_test_lock(othr_lock_t *lock);
```

Figure 1: Portion of an example `ee.h`. The 5 functions are implemented in `othr.c`

The library must support three different kinds of locks, namely normal, nested and spin locks. The first two are available to user OpenMP programs, while the spin locks are used internally by ORT. The actual kind of a lock is determined upon its initialization, where the second argument of `othr_init_lock()` can be one of `ORT_LOCK_NORMAL`, `ORT_LOCK_NEST`, `ORT_LOCK_SPIN`. However, the library must provide a single type for all locks, '`othr_lock_t`'; obviously, this must be a `union`. As a concrete example, for an EELIB based on POSIX threads, one could have the `ee.h` shown in Fig. 1.

2.3 Execution entities

For clarity, we will assume the library provides threads. For EELIBs that provide processes, the '`othr_`' prefix should be changed to '`oprc_`' in what follows. There are just three functions that ORT requires in order to manipulate EEs.

```
int  othr_request(int numees, int level);
void othr_create (int numees, int level, void *arg, void **info);
void othr_waitall(void **info);
```

First of all, it is important to note that the sole EE that exists when the application runs is called *initial EE* and it is *not* manipulated by the EELIB. Second, ORT asks always for *batches* of EEs, so as to create a team of threads using only one call to `othr_create()`.

When ORT needs to create a team of EEs it follows the steps below:

1. It calls `A = othr_request(N,L)` and asks for a specific number of EEs (`N`). The second parameter (`L`) refers to the nesting level. The initial EE is the only EE that is at level 0. The EEs created by the initial EE are all at level 1. In general, the EEs created by any EE in level x ('parent' EE) will all belong to level $x + 1$ ('children' EE).

   `othr_request(N,L)` should return an integer `A` indicating *the number of EEs that the threading library can provide*, which may be less than N; it may even be 0. In fact, the `pthreads1` library which used to ship with earlier versions of OMPi, works with only 1 nesting level. If any thread at level > 0 calls `othr_request()`, the library always returns 0.

2. It calls `othr_create(A,L,secret,&info)`. This is the actual function that creates the team of `A` EEs. The number `A` returned by the call to `othr_request()` is assumed to be *guaranteed*, i.e. `othr_create()` may not create fewer threads or ORT will break down. Similary, ORT may not ask for the creation of more/less than `A` threads.

   The parameters passed include the nesting level (`L` is the same as in the call to `othr_request()`), plus an argument to be known by the threads (`secret`—more on this in a while) and an `info` argument which will be explained below.

Here comes the first crucial point: the threads that will be created by EELIB *do not know the function they should execute*. Thus the created threads must execute an EELIB-provided "driver" function, which in turn will execute the unknown function through the following call[1]:

```
ort_ee_dowork(myid, secret);
```

where `secret` is what `othr_create()` provided and `myid` is the *id* of the thread:

the ids of the threads in a team of `A` threads, must be 1, 2, ..., A.

The id 0 is reserved for the parent of the team and should not be used for any child. *It is* EELIB*'s responsibility to provide correct ids for the threads it creates.*

This is almost all there is to it. Well, almost, since there is one more thing: `info`, and this is the second and final crucial point. If EELIB supports multiple levels of threads, then it is almost certain that for each team it creates it must do some kind of bookkeeping. Another thing that is also almost certain is that the created threads will need some kind of access to this bookkeeping (e.g. they might want to know who their parent is, they might need to have a shared lock to modify team-specific data, etc.). This is the reason behind the `info` parameter in the `othr_create(A,L,secret,&info)` call.

ORT maintains a specific variable for every thread that creates a team of threads (i.e. for every thread that becomes a parent). This variable is `info`, is declared as `void*` and is up to EELIB to use it. Thus, EELIB may for example

[1] Important: *the call must be made in the context of the thread*, i.e. nobody else may call this on behalf of the thread in question.

allocate memory, put its bookkeeping stuff there, and before returning from `othr_create()` make `*info` point to the allocated block.

Any child may access this `info` block in the parent of the team using:

```
stuff = ort_get_parent_othr_info(); /* Returns void* */
```

One thing that is good to know is that *it is guaranteed by* ORT *that the first time a thread becomes a parent, its* ***`info`*** *variable is initialized to* ***`NULL`****. After that, whether the same thread creates another team or not,* ORT *does not touch this* ***`info`*** *variable at all.* Although, it may not seem important, it might actually prove useful if EELIB needs to utilize some kind of memory recycler, in order to avoid repeated `malloc()`'s and `free()`'s.

The last function that ORT requires is `othr_waitall()`, for joining threads. It is only called by the parent of the created team; this function should wait until all children have finished their work. Upon return from this call, the team seizes to exist and the parent continues its way. The function is called with a single parameter, a pointer to the `info` variable that is associated with the parent.

If in `othr_create()` EELIB does indeed allocate a memory block and makes `*info` point to the allocated block, there should be a way to `free()` the allocated memory when the team finishes its work and the threads are done. The only place that this is possible is in `othr_waitall()`.

2.4 The rest of the required functions

EELIB should implement another 5 functions, as listed below:

```
int  othr_initialize(int *argc, char ***argv,
                     ort_icvs_t *icv, ort_caps_t *caps);
void othr_finalize(int exitvalue);
int  othr_yield(void);
int  othr_key_create(othr_key_t *key, void (*destructor)(void *));
void *othr_getspecific(othr_key_t key);
int  othr_setspecific(othr_key_t key, const void *value);
```

Starting from the bottom, the last 3 of them should behave exactly as the corresponding POSIX threads functions, so that ORT has a means of creating keys for thread-specific storage. One can instead use thread local storage (TLS) if available by the underlying system compiler (just `#define USE_TLS` in `ee.h`—see `runtime/ee_pthreads`).

The `othr_yield()` function should yield the processor so that other threads can run.

Finally, ORT calls `othr_initialize()` upon startup and `othr_finalize()` upon termination. This first function should initialize the threading library and prepare it for use by ORT. The first two parameters passed to the function are pointers to the program arguments, which the library may manipulate freely. The next parameter (`icv`) contains all the 'internal control variables' of ORT (see `ort.h`). Of particular importance are `icv->stacksize` and `icv->nthreads`; if they are ≤ 0, the EELIB is free to use its own values. If, however, `icv->stacksize > 0` then all created EEs must have that particular

stack size. If `icv->nthreads > 0`, then the EELIB must provide *at least* that many EEs.

The last parameter in `othr_initialize()` (`caps`) is the set of the library's 'capabilities' and must be filled by the EELIB. In particular, if the EELIB supports some kind of nested parallelism (i.e. levels > 1), then `caps->supports_nested` should be set to 1. If it supports dynamic adjustment of threads `supports_dynamic` should be set to 1. Finally, if it supports nested and non dynamic, it should set `supports_nested_nondynamic` to 1. This last one is used because some OpenMP systems support a limmited form of nesting which works only if dynamic adjustement is enabled. In particular, they can support nesting but they limmit the total number of threads that may be running at any point in time. `caps->max_levels_supported` should be set to 1 if the library does not support nested parallelism, or to a particular number if it can operate only up to a certain nesting level. If there is no limit to the nesting levels, this capability should be set to `-1`. Analogously, `caps->max_threads_supported` should be set to a maximum value or `-1` if there is no limit. Finally, `caps->default_numthreads` specifies the default number of threads that the library can provide if ORT does not ask for any particular number.

2.5 A couple more for processes

If the EE type is process, the EELIB must also provide:

```
int  oprc_pid();
void oprc_shmalloc(void **ptr, int size, int upd);
void oprc_shmfree(void *p);
```

which, correspondingly, return the process id of the EE, allocate space in shared memory and free it.

3 What facilities are provided by ORT to EELIB

Assuming one works in directory `runtime/host/othr_foo`, there are 3 basic files that one may include in `othr.c`:

```
config.h
../sysdeps.h
../ort.h
```

The first one resides in the `common/` directory of OMPi's source tree and provides some definitions created during configuration time, such as the name of the package ("ompi"), its version etc. One may need to perform some extra tests for the availability of certain system header files or library functions; the place to do those is file `configure.in`, so that the results are available as definitions in file `config.h`.

The second file contains some system-specific utilities and definitions. The definitions include the processor type of the system, the platform type, the operating system type, the size of the processors' cache lines, and a few others. This file provides two major facilities that can be useful for a threading library:

- A function to obtain the number of processors available. Use as in:

```
np = ort_get_num_procs();
```

- A macro to force serialization of memory operations, i.e. to provide a memory barrier / fence. Use as in:

```
...     /* Code above */
FENCE;
...     /* Code below */
```

  `FENCE` guarantees that all pending memory operations in the code above the fence will be finished *before* any operations below the fence start.

Finally, the third file (`../ort.h`) provides the rest of the facilities described below. In fact, this file includes the other two headers, so this is the only file one has to include, after all:

```
#include "../ort.h"
```

3.1 What is provided in `ort.h`

The first thing ORT provides is a set of memory allocation functions (`ort_alloc()`, `ort_calloc()`, `ort_alloc_aligned()`, `ort_calloc_aligned()`, `ort_realloc_aligned()`) which also check whether the requested memory block could not be allocated (causing an exit). The last three functions allocate memory aligned to the cache line size of the processor, which is a must for performance. The actual memory allocated contains usually more bytes than requested and starts at an unaligned position, so the functions return a pointer to the aligned portion of the memory block and set their 'actual' parameter to point to the actual (unaligned) block start; the latter is what one should `free()` if needed.

Finally, there is a function to display a warning and another one to diplay an error and `exit()` with a particular value.

3.2 Barriers

ORT implements a default barrier in `ort.c/ort.h` which makes use of the locks provied by EELIB. The barrier functions are also available for EELIB to use:

```
void ort_default_barrier_init(ort_defbar_t *bar, int numees)
void ort_default_barrier_wait(ort_defbar_t *bar, int thrid)
void ort_default_barrier_destroy(ort_defbar_t *bar)
```

You may want to experiment with your own barrier algorithms. In order to override the one in `ort.c`, all you have to do in your `ee.h` is:

```
#define AVOID_OMPI_DEFAULT_BARRIER
```

and implement the following 3 functions in your `othr.c`:

```
void othr_barrier_init(othr_barrier_t *bar, int n);
void othr_barrier_wait(othr_barrier_t *bar, int id);
void othr_barrier_destroy(othr_barrier_t *bar);
```

Of course, `ee.h` must provide the prototypes, plus the definition for `othr_barrier_t`. Finally, notice that in light of the tasking support in OpenMP, barriers are task scheduling points that require/make sure that all tasks of the binding thread team have finished before continuing. See `runtime/host/ee_pstrheads/` for an example.

3.3 Tasks

NOTE: This section is incomplete.

OMPi provides support for tasks in ORT, which is independent of the actual EE that is used. However, it also allows for an EELIB to provide its own task implementation, if needed. In order to do this, all you have to do is:

```
#define AVOID_OMPI_DEFAULT_TASKS
```

in your `ee.h`, and implement the following 3 functions in your `othr.c`:

```
void othr_new_task();
void othr_new_task_exec();
void othr_taskwait();
```

. . . to be completed.