OpenMP Runtime Interoperability

Ali Alqazzaz, Zijun Han, and Yonghong Yan

Department of Computer Science, Oakland University
{aalqazzaz,zhan,yan}@oakland.edu

Abstract. OpenMP has become a very successful user-model for developing parallel applications. However, there are still some challenges in terms of OpenMP interoperability. In this paper, we introduce some extensions to the OpenMP runtime library related to the interoperability problem. We evaluate and compare the performance of the different waiting thread behaviours (PASSIVE |ACTIVE). In addition, we introduce a new function to shutdown or unload the whole runtime library, which enables the greatest degree of interoperability with other threading models.

1 Introduction

Parallel and large-scale applications are typically developed using multiple parallel programming interfaces in a hybrid model, e.g. MPI+OpenMP, and using one or mulitple prebuilt scientific and/or platform-specific libraries such as MKL. Each of these programming models and libraries has their own runtime to handle scheduling of work units and management of computational and data movement tasks. There have been challening issues for using these models in one application, including compatibility issues for compiling and linking, oversubscription of resources at runtime, and the naming conflicts that programmers have to create workaround wrappers to deal with.

This report propose solutions to the interoperability and composability challenges faced by OpenMP programming interface, includling those between multiple OpenMP implementations and/or multiple OpenMP runtime instances of the same implementation, OpenMP with native threads (pthreads and Windows Native threads), OpenMP with other threading languages and library such as C++11, TBB and Cilkplus, and OpenMP with inter-node programming models such as MPI, PGAS implemenation, etc. We think the similar challenges exist in other threading based libraries and language implementations, and believe the solutions we provided in this technical report will work for them too.

Interoperability and composability are closely related, while the interoperability sounds to improve the interactions between multiple models while composability is meant to improve the modular use of OpenMP with itself and other models. One is from the aspect of system while the other is more concerned with software engineering. Both should be considered when developing solutions.

For parallel programming languages and libraries, most implementations rely on system native threading (pthread or Windows Native threads) mechanisms to

acquires system resources. Each implementation of the same or different programming models has their own mechanism for scheduling user-level tasks and operations, which is the core part of a runtime system. The interoperability challenges are then concerned with how much we want two or more runtime instances (for the same or different high-level programming interfaces) to interact with other other for computational resource sharing and data movement. Thus solutions to these challenges are more in the scope of runtime and implementation, than in the level of programming interfaces and compiler transformations.

2 Challenges and Proposals

3 Implementation

In general, to implement those four functions, we follow the three steps:

- Define this function in file kmp_csupport.c, write down the implementation.
- Declare this function in file kmp.h, using KMP_EXPORT in front the declaration.
- Export this function in file dllexports, assign a unique ID for this function.

1. void omp_quiesce()
   The purpose of this function is to shutdown or destroy all OpenMP threads in the thread pool. We have implemented it, as shown in Figure 1, by using the Intel internal call to _kmp_internal_end_fini, which unloads the runtime library. Then, we have to register the master thread again so it can generate team of threads later when needed. This can be done by calling the _kmp_get_global_thread_id_reg( ).

```
1  void omp_quiesce(ident_t *loc)
2  {
3      __kmp_internal_end_fini();
4  }
5
6  void omp_begin2()
7  {
8      __kmp_get_global_thread_id_reg( );
9  }
```

Fig. 1: omp_quiesce

2. void omp_set_wait_policy(PASSIVE |ACTIVE)
   The idea of this function is to set the waiting thread behavior. PASSIVE value means that waiting threads should not consume CPU power while waiting. In other words, the OpenMP runtime system will put them into a sleep mode. On the other hand, ACTIVE value means that waiting threads

should keep asking the CPU for work to do. The intention of doing this function is to measure the differences in performance between these different modes. The implementation of this function is done by using the internal _kmp_stg_parse_wait_policy as shown in Figure 2. The current OpenMP runtime system uses the library_turnaround to indicate the ACTIVE mode and library_throughput to indicate the PASSIVE mode. We pass an integer as its parameter. If it equals to 0, we set the wait policy to be passive, otherwise, active. We found a variable named _kmp_library in the environment setting file which has four different status for the waiting policy. So, we change this value accordingly, then we call a function _kmp_aux_set_library to set the changed value to the OpenMP environment.

```
1  void omp_set_wait_policy(int flag)
2  {
3      if (flag != 0){
4          __kmp_library = library_turnaround ;
5      }
6      else{
7          __kmp_library = library_throughput;
8      }
9  }
```

Fig. 2: omp_set_wait_policy

3. int omp_thread_create()
   The purpose of this function is to give the user the ability to create an OpenMP thread without using #pragma omp parallel directive, and lets it be a user thread similar to pthread. The implementation of this function is shown in Figure 3. So, we are creating one thread to execute the passed function. If there are enough available threads in the thread pool, we will get one thread from the thread pool and assign the task to it. If no thread is available in the thread pool, we create a new thread to execute this task, and then put the new thread back into the thread pool after completing its job.

4 Experimental Results

1. void omp_quiesce()
   Figure 4 shows the design of the quiesce evaluation. However, Figure 5 below shows that the running time of all variables (startup_quisece, parallel, and quiesce) increase as we increase the number of threads used. The running time of creating the parallel region is very small because the OpemMP just creates that once. Then, it puts them in a global thread pool to be used next time needed. However, the time cost represented by the quiesce term refers to the time required to shutdown the whole runtime library. In other words, after each parallel region we remove all threads in the global thread pool. Finally, the startup_quiesce term implies the time required to initialize the parallel region and the time taken to shutdown the runtime library.

```
void * omp_thread_create(void (*fun)(void*), void *arg, ...){
    int idd = __kmp_register_root(false);
    kmp_info_t *thr = __kmp_threads[idd];
    thr->th.th_set_nproc = 1;
    thr->th.th_teams_microtask = microtask;
    va_list ap;
    va_start(ap, arg);

    int par = __kmp_fork_call( NULL, idd, fork_context_intel,
            2, // num of parameters
    // #if OMPT_SUPPORT
                // (void *)fun, // "unwrap"
    // #endif
                VOLATILE_CAST(microtask_t)fun, // wrap
                VOLATILE_CAST(launch_t) __kmp_invoke_task_func,
                &ap );
    __kmp_join_call(NULL, idd, 1);
}
```

Fig. 3: omp_create_thread

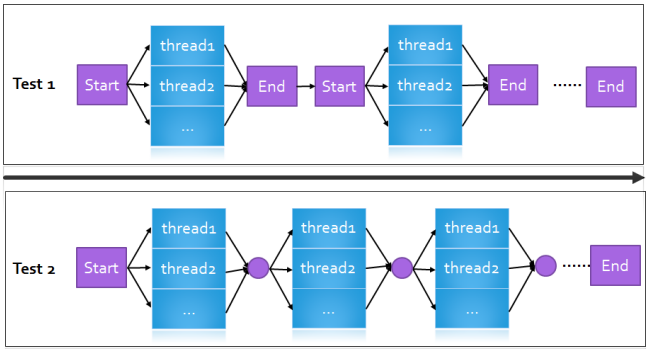
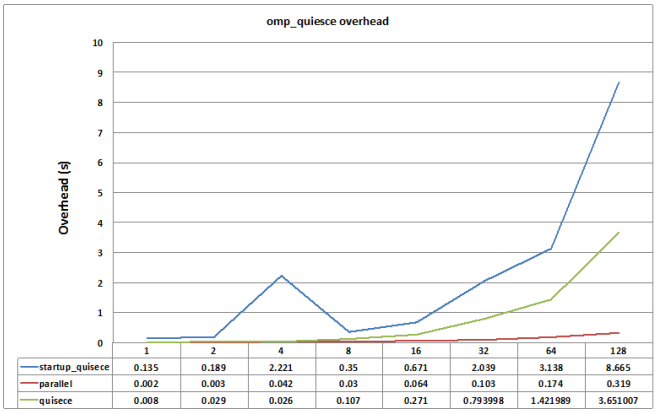


Fig. 4: omp_quiesce evaluation



	1	2	4	8	16	32	64	128
startup_quisece	0.135	0.189	2.221	0.35	0.671	2.039	3.138	8.665
parallel	0.002	0.003	0.042	0.03	0.064	0.103	0.174	0.319
quisece	0.008	0.029	0.026	0.107	0.271	0.793998	1.421989	3.651007

Fig. 5: omp_quiesce results

2. void omp_set_wait_policy(PASSIVE |ACTIVE)
   We need to create two processes since each process will only maintain and share one thread pool. For those two process, each task is execute using 1s, and we need to create enough threads to make full use of the calculation power of one CPU. We tested it in three cases: passive, active, and quiesce/restart the runtime environment. Figure 6 shows the design of the evaluation.

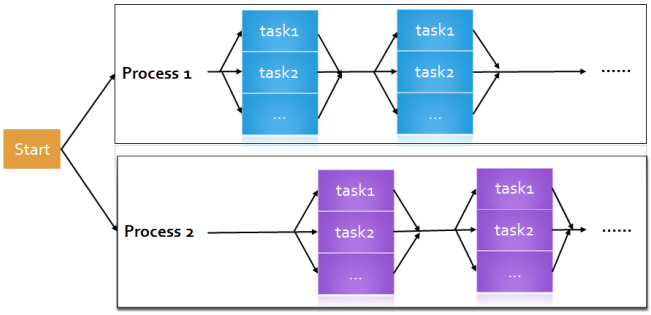


Fig. 6: waiting policy evaluation

As Figure 7 below shows, there is no a big difference between the two behaviors. The reason is that the OpenMP uses only one global thread pool for all OpenMP threads created by multiple pthreads. So, the small difference comes from the time required to awake a sleeping thread. By doing this experiment, we have understand more about the way that OpenMP deals with the thread pool.

Time(s) / Mode	Experiment1	Experiment 2	Experiment 3	Average
PASSIVE	4.79	4.80	4.86	4.81
ACTIVE	4.72	4.67	4.53	4.64
Quiesce	5.10	5.16	5.21	5.16

Fig. 7: Waiting Policy Results

3. int omp_thread_create()
   We compared this function with creating pthread to execute a list of tasks. So, for this function we have tested it in two different ways. Figure 8 shows the design of the evaluation. For the first way, we put different number of tasks in one parallel region, so that every omp_thread_create() or pthread_create() function will be run in parallel. On the other hand, we use different iterations

to execute the omp_thread_create() or pthread_create() functions in sequence, and compare the running time.

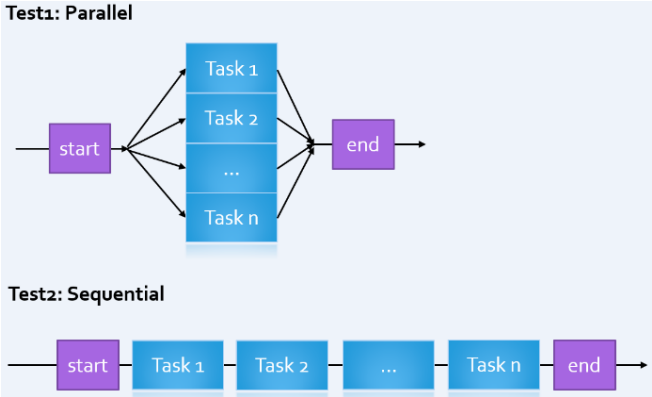


Fig. 8: creating thread evaluation

Figure 9 and Figure 10 show the result of the first approach (execute in parallel). It clearly shows that there is almost no differences between them. This is might be because that we are doing it inside the parallel region.However, Figure 11 and Figure 12 show the result of the second approach (execute in sequence). They show that omp_thread_create() gives a better performance that pthread_create( ). So, it would be a good feature if the user can do this instead of creating another pthread.

execute in parallel		
Parallel thread	**Omp_create**	**Pthread_create**
10	2.76	2.769
100	2.767	2.767
1000	2.781	2.793
10000	2.796	2.793

Fig. 9: Results of omp_thread_create in parallel

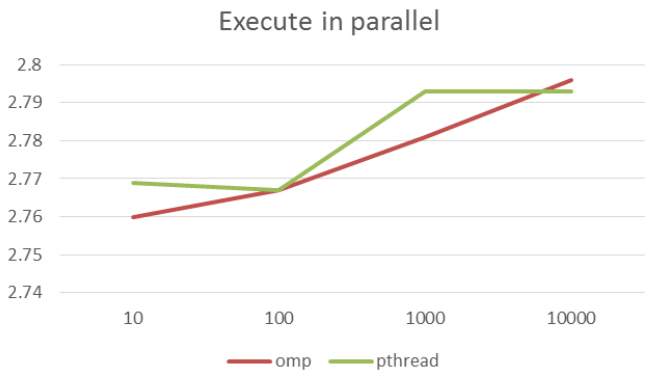


Fig. 10: Results of omp_thread_create in parallel

execute in sequence		
iteration	Omp_create	Pthread_create
1000	0.057	0.048
10000	0.334	0.433
100000	2.981	4.453
300000	8.946	12.923

Fig. 11: Results of omp_thread_create in sequence

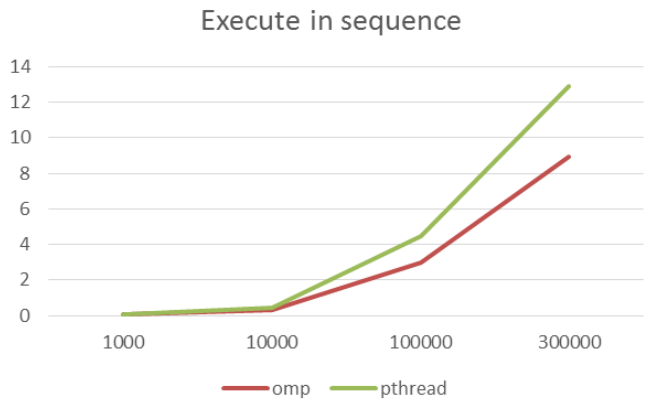


Fig. 12: Results of omp_thread_create in sequence

5 Conclusions and Future Work

In conclusion, we have seen that there are many features can be added to the current OpenMP Runtime Library in order to improve the OpenMP interoperability. One feature is that allowing the user to create a new OpenMP thread and assign a task to it instead of creating new user thread. We have implement a function to allow users to get one thread from the existing thread pool is any threads are available, and assign one task to this thread, this helps to take advantage of the OpenMP thread pool and wont need to create a new thread to work on it, which helps to save the memory usage and speed up the runtime.

We have studied the waiting policy of the OpenMP and how the current OpenMP Runtime System deals with the thread pool. Considering there are two waiting policies, one called throughput (passive), which is designed to make the program aware of its environment (that is, the system load) and to adjust its resource usage to produce efficient execution in a dynamic environment. While the other one called turnaround (active), which is designed to keep active all of the processors involved in the parallel computation in order to minimize the execution time of a single job. We cannot simply say which one is better than the other, it depends one the executing environment. When setting the wait policy to be passive, after a certain period of time has elapsed, the useless thread will stop waiting and sleep. Thus active mode may be better for high-density of OpenMP tasks. While, a passive mode with a small blocktime value may offer better overall performance if your application contains non-OpenMP threaded code that executes between parallel regions.

In addition, we have implemented a new function to shutdown the whole runtime library when exiting the parallel region. Since all threads are maintained in the same thread pool, quiesce will reap every threads to free the memory, which sometimes help to clear the runtime environment when the task density is lower and we dont need to wake up most of the thread in the thread pool. However, when entering new parallel regions, we need to make sure that we register the current working thread as our root thread, so that new runtime environment can be built on it. It cost time to restart another parallel region, thus works slower when lots of tasks in the task queue.

As a future work, we should continue adding more functions to the existing runtime system to improve the OpenMP interoperability, such as omp_attach/omp_detach, omp_exit/omp_join. By doing this, we could have a better OpenMP runtime library that optimizes the resources utilization.

References