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 extientions to the OpenMP runtime library realted to the interoperability problem. Also, 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 when exiting the parallel region to prove that It would take longer time than awakening a sleeping thread.

1 Introduction

1.1 What is an OpenMP?

OpenMP is an implementation model to support the implementation of parallel algorithms. It is primarily designed for shared memory multiprocessors. The goal of OpenMP is to provide a standard and portable API for writing shared memory parallel programs [2].

OpenMP takes a directive-based approach for supporting parallelism. It consists of a set of directives that may be embedded within a program written in a base language such as Fortran, C, or C++. There are two compelling benefits of a directive-based approach that led to this choice: The first is that this approach allows the same code base to be used for development on both single-processor and multiprocessor platforms; on the former, the directives are simply treated as comments and ignored by the language translator, leading to correct serial execution. The second related benefit is that it allows an incremental approach to parallelismstarting from a sequential program, the programmer can embellish the same existing program with directives that express parallel execution. These directives may be offered within any base language (within the C/C++ languages, directives are referred to as pragmas). In addition to directives, OpenMP also includes a small set of runtime library routines and environment variables. These are typically used to examine and modify the execution parameters. The language extensions in OpenMP fall into one of three categories: control structures for expressing parallelism, data environment constructs for communicating between threads, and synchronization constructs for coordinating the execution of multiple threads [1].

1.2 How does OpenMP work?

OpenMP uses a fork/join execution model. OpenMP provides two kinds of constructs for controlling parallelism. First, it provides a directive to create multiple threads of execution that execute concurrently with each other. The only instance of this is the parallel directive. Second, OpenMP provides constructs to divide work among an existing set of parallel threads. An instance of this is the do directive.

An OpenMP program always begins with a single thread of control that has associated with it an execution context or data environment. This initial thread of control is referred to as the master thread. When the master thread encounters a parallel construct, new threads of execution are created along with an execution context for each thread. Each thread has its own stack within its execution context. The execution context for a thread is the data address space containing all the variables specified in the program. Multiple OpenMP threads communicate with each other through ordinary reads and writes to shared variables.

1.3 OpenMP Runtime Library

The OpenMP API runtime library routines are external procedures. The return values of these routines are of default kind, unless otherwise specified. Runtime library provides interface to the compiler. The runtime interface is based on the idea that the compiler "outlines" code that is to run in parallel into separate functions that can then be invoked in multiple threads. OpenMP provides several runtime library routines to assist you in managing your program in parallel mode. Many of these runtime library routines have corresponding environment variables that can be set as defaults. The runtime library routines enable you to dynamically change these factors to assist in controlling your program. In all cases, a call to a runtime library routine overrides any corresponding environment variable.

Generally, we can analyze the architecture into two perspectives: the parallelization regions and the data.

1. Region perspective. We use parallel to automatically create multi-threads. And each thread will be executed without order. However, we can use ordered clause to guarantee the code be executed in sequence. There are different types of parallel regions:
   - Section means the task is assigned to each thread.
   - Single means the task is assigned to a random thread.
   - Master means the task is executed in the master thread.

   For the default parallel regions, we can use schedule to design a way to assign tasks to different threads. Generally, we can implement this assignment in three ways:
   - Static: equally assign them to n threads.
   - Dynamic: assign them to the idle thread only.
   - Guided: implement the dynamic assignment reductively.

2. Data perspective. We have two kinds of variables. Variables that are defined before parallel region are shared among every thread, while those defined in parallel regions can be only accessed by certain threads. We use threadprivate to change those shared variables into a private one for each thread. This is done by generating a new private variable for every single thread. For those shared variables, we must pay attention to the data race problem, which defined as two different memory operations are trying to use a same variable, and different execution order may lead to different results. To solve this problem, we can use critical or atomic directive to guarantee that the data can be only accessed by one thread at a time. We can also set barriers to make sure all threads have been executed before starting any new threads. Sometimes the update of certain variables are stored only in registers, we can use flush to directly write the data back to memory to make sure that other threads will use the data that already been updated.

2 Challenges and Motivations

There are still some challenges in terms of OpenMP interoperability. OpenMP threads that are created by the parallel construct cannot interact with external systems. In other words, we are trying to enable the interoperability through flexible communication between OpenMp threads and user threads. However, the main goal of this work is to achieve a high level of resource utilization. So, it would be better if OpenMP threads can interact and communicate with user threads. To achieve this goal, we implement four new functions as follows:

1. int omp_set_wait_policy(ACTIVE |PASSIVE): set the waiting thread behavior. The function returns the current wait_policy, which could be different from intention of the call depending on the decision made by the runtime. If the value is PASSIVE, waiting threads should not consume CPU power while waiting; while the value is ACTIVE specifies that they should.
2. int omp_thread_create( ): 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.
3. int ompe_quiesce( ): to shutdown or unload the OpenMP runtime library.

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.

```
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

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.
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.

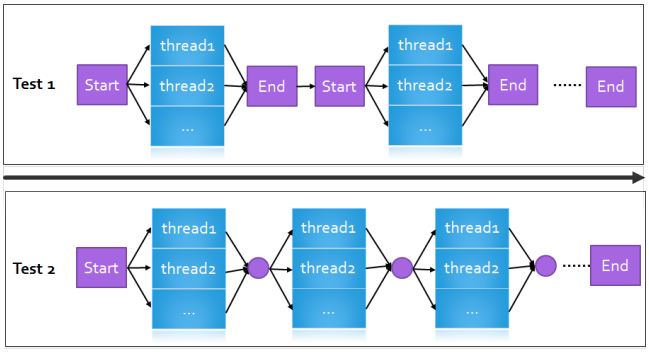
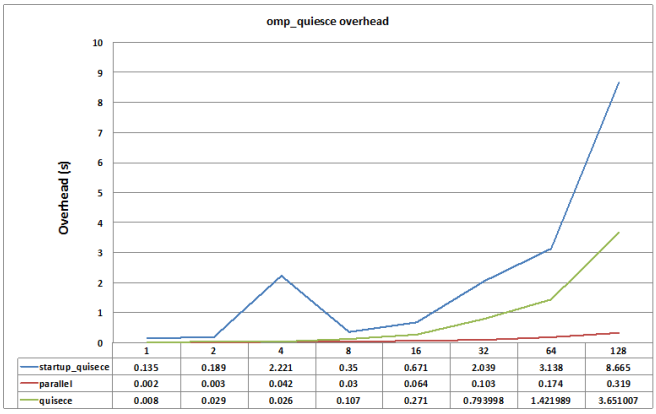


Fig. 4: omp_quiesce evaluation



	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

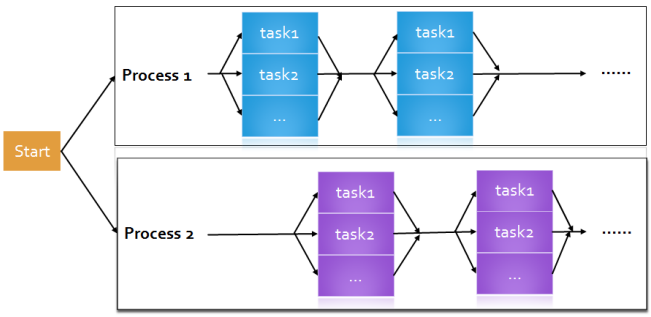


Fig. 6: waiting policy evaluation

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.

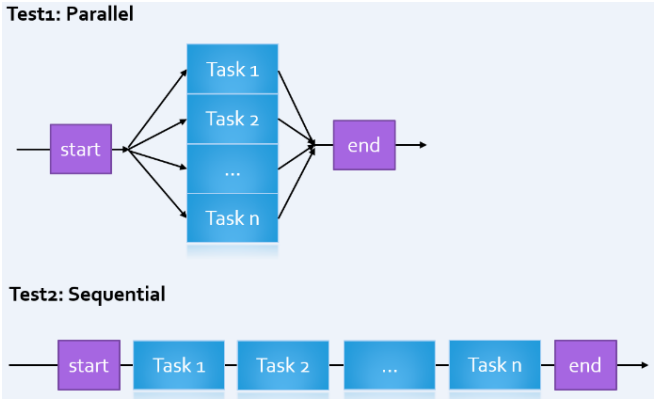


Fig. 8: creating thread evaluation

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

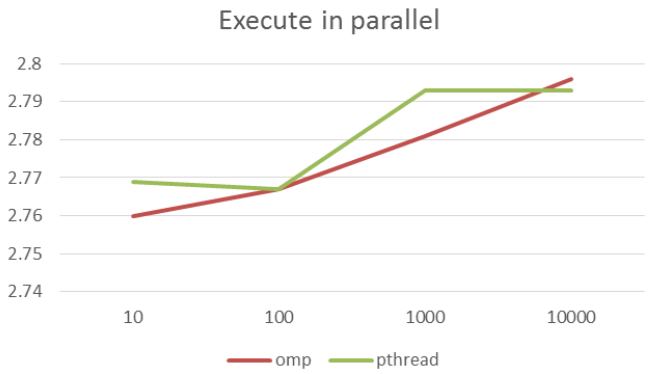


Fig. 10: Results of omp_thread_create in parallel

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

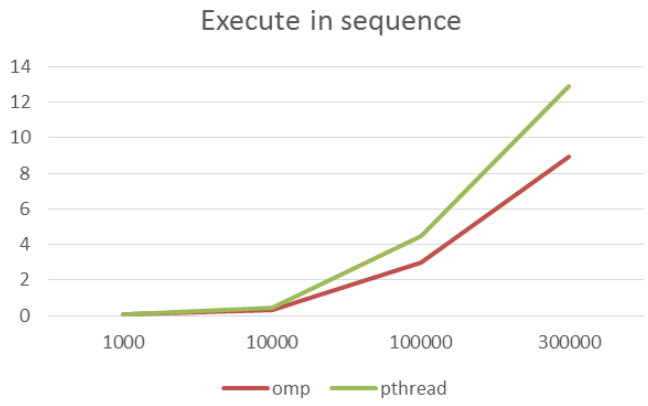


Fig. 12: Results of omp_thread_create in sequence

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

1. Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
2. Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.