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 Proposal

2.1 User threads

Definition A user thread is a thread that is not created by OpenMP implementation. A user thread could become an OpenMP initial thread.

The most common example of user threads are POSIX Threads, usually referred to as Pthreads with implementation available on most Unix-like POSIX-compliant systems. There are also implementations of other thread libraries, for example, Windows Native threads and language based threading support such as Java threads or others (e.g. qthreads).

Figure 1 shows an example of having three user threads (two PThreads and one thread of the main program) in an OpenMP program. The two three threads execute in parallel after the two PThreads are created. Each thread calls a function that will enter into OpenMP threading parallelism. So they all become OpenMP initial threads. How the user threads (PThreads in this example) interact with the OpenMP threading mechanisms in the runtime is up to the implementation. They may share the same OpenMP runtime instance or each has its own OpenMP runtime instance.

2.2 Impacts and Discussions

A user thread in a program adds additional level(s) in the overall "threading" hierarchy of a program. Those additional levels could be on top of OpenMP threading mechanism when a user thread becomes an OpenMP initial thread that creates OpenMP thread parallelism, or beneath the OpenMP threading mechanism when an OpenMP thread spawns a user thread, or the the mix of both. In the example from Figure 1, one can view this in a two-level threading parallelism: the top level user thread parallelism and the bottom level OpenMP threading parallelism.

These additional levels of threading increase the complexity of a program, both for users in the aspect of reasoning the parallel and synchronization behavior of a program, and also for the implementation and runtime system in terms of resource management and interactions. Adding to the complexity is the facts that a user thread may be created through a call to a library function whose paralelism (OpenMP) behavior is not known to the callee. Typical issues for example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <omp.h>

void *omp_parallel_foo(void *ptr);

int main(int argc, char * argv[])
{
     pthread_t thread1, thread2;
     const char *message1 = "pthread␣1";
     const char *message2 = "pthread␣2";

    /* Create independent threads each of which will execute function */
     pthread_create(&thread1, NULL, omp_parallel_foo, (void*) message1);
     pthread_create(&thread2, NULL, omp_parallel_foo, (void*) message2);

     omp_parallel_foo("master␣thread");

     /* Wait till threads are complete before main continues */
     pthread_join(thread1, NULL);
     pthread_join(thread2, NULL);
     return 0;
}

void *omp_parallel_foo(void *ptr )
{
    char *message = (char *) ptr;
#pragma omp parallel shared(message)
    {
        int id = omp_get_thread_num();
        printf("%s:␣OMP␣thread:␣%d,␣␣\n", message, id);
    }
    return NULL;
}
```

Fig. 1. Three user threads (two Pthreads and one main thread) with OpenMP

– Does each user thread use the same OpenMP runtime libraries or not? If not using the same library, how to handle symbol name conflicts of two more different OpenMP runtime libraries.
– For user threads that use the same OpenMP runtime library, does the user threads each create its own runtime instance or they share one?
– For user threads each of which has its own runtime instance (from the same or different runtime library), how to coordinate the resource management among those runtime instances to address such issues as oversubscriptions and the affinity between user threads?

It is important to note that approaches to address those issues are very implementation dependent, requiring protocol and agreement in the runtime behavior and/or interfaces of different OpenMP implementations. It may not be realistic to solve some of the issue from the language standard, and should be left to users to deal with them. In this aspect, we still hope this report could provide userful information and practices for users.

2.3 Oversubscription

Oversubscription happens when resources are claimed and held than what is needed. A program may request more OpenMP threads than the total amount of hardware threads available when entering a parallel region, which causes excessive competition among OpenMP threads for hardware threads and increases runtime overhead. When program execution enters into sequential stage after exiting a parallel region, those native threads that support the OpenMP threads in the parallel region may still alive in the background consuming CPU cycles but not doing actual work for users. This will make those hardware threads unavailable to others. Oversubscription impact the performance of an applications and the system, but should not introduce correctness issue to a program.

The two scenarios we mentioned above are the two kinds of oversubscriptions we should try to avoid[1]: **1) Active oversubscription**: Claiming or requesting more threads than what are available by the system. **2) Passive oversubscription**: Thread resources are not released after parallel execution. Holding hardware threads after parallel execution may not always hurt the performance overall, e.g. it will improve the startup performance of the upcoming parallel region. In this category, we are concerning those situations that actually impact the performance negatively.

Current OpenMP OpenMP currently (4.0) provides limited support for users to give hints to runtime for better managing OpenMP threads and native threads, which can be used to help reducing the impact of oversubscriptions.

OMP_DYNAMIC This also includes dyn-var ICV, omp_set_dynamic and omp_get_dynamic runtime routine. OMP_DYNAMIC could be either **true** or **false**. When setting the dyn-var ICV to be **true**, user will expect OpenMP implementation to adjust the number of threads to use for executing parallel regions in order to optimize the use of system resources.

OMP_WAIT_POLICY This also include wait-policy-var ICV, but no getter and setter routine. OMP_WAIT_POLICY could be set as either ACTIVE or PASSIVE. The ACTIVE value specifies that waiting threads should mostly be active, consuming processor cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin. The PASSIVE value specifies that waiting threads should mostly be passive, not consuming processor cycles, while waiting. For example, an OpenMP implementation may make waiting threads yield the processor to other threads or go to sleep. The details of the ACTIVE and PASSIVE behaviors are implementation defined.

OMP_THREAD_LIMIT This also include thread-limit-var ICV and omp_get_thread_limit getter runtime routine. The environment variable sets the maximum number of OpenMP threads to use in a contention group by setting the thread-limit-var

[1] Acknowledgements go to to Jeff Hammond for categorizing the twos and for introducing the two terms

ICV. The behavior of the program is implementation defined if the requested value of OMP_THREAD_LIMIT is greater than the number of threads an implementation can support.

OMP_DYNAMIC and OMP_THREAD_LIMIT are approaches to addressing active oversubscriptions, and OMP_WAIT_POLICY could be used to address passive oversubscriptions. Since there are no setters for ICVs for OMP_WAIT_POLICY and OMP_THREAD_LIMIT variable in the current standard, dynamically changing waiting policy and the maximum number of threads at runtime is not available.

The current support in OpenMP provides limited constrol on oversubscriptions, but are sufficient for lots of (if not most of) scenarios if the implemention is available. In the following of this report, we propose solutions that will provide more control for oversubscription.

Change wait policy dynamically to address passive oversubscription The idea is to provide a setter and getter for the wait-policy-var ICV. Compilers from IBM, Cray and Oracle have provide this feature [1, ?,?]. There are also different variants of this features depending how much details users can configure the wait policy.

1: omp_set_wait_policy(ACTIVE|PASSIVE) *setter* for the wait-policy-var ICV with ACTIVE or PASSIVE. This will allow programmer to explicitly change the policy at various points during a program's execution. An efficient implementation may use atomic write to the global ICV and all threads will react accordingly at some later point of the exectution after the ICV is set. So the effects may be delayed.

2: Finer-grained control with new environment variables and setter routine

2.4 Interaction between contention groups

2.5 Affinity with user threads

Coherence group (domains)

2.6 Common Interface

Application Binary Interfaces(ABI)

2.7 Interoperability with node-level programming model, e.g. MPI

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:

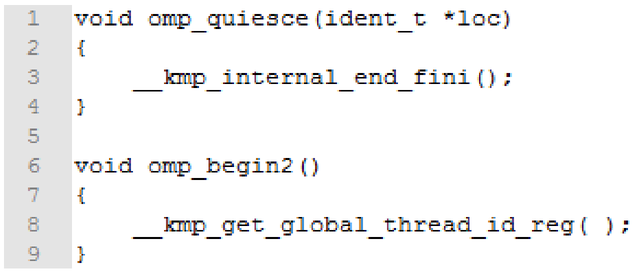
1. int omp_set_wait_policy(ACTIVE |PASSIVE): set the waiting thread behavior. The function returns the current wait_policy, which could be different from intention of the call depending on the decision made by the runtime. If the value is PASSIVE, waiting threads should not consume CPU power while waiting; 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 2, 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( ).



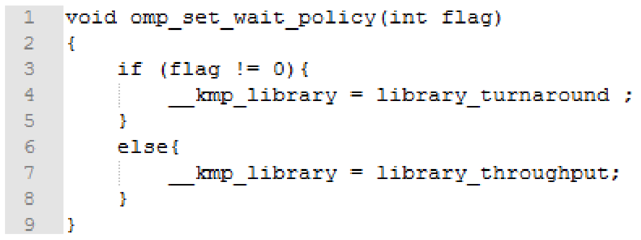
```
void omp_quiesce(ident_t *loc)
{
    __kmp_internal_end_fini();
}

void omp_begin2()
{
    __kmp_get_global_thread_id_reg( );
}
```

Fig. 2. 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 3. 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.

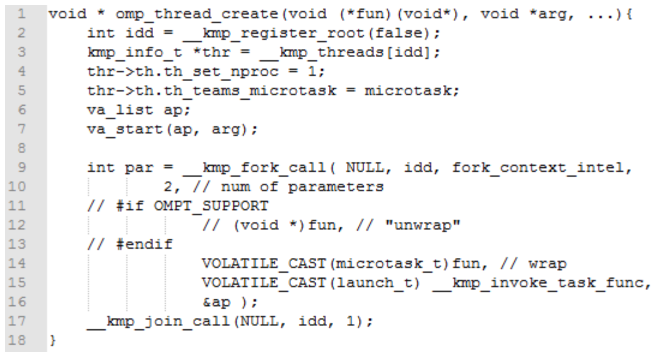


```
void omp_set_wait_policy(int flag)
{
    if (flag != 0){
        __kmp_library = library_turnaround ;
    }
    else{
        __kmp_library = library_throughput;
    }
}
```

Fig. 3. 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 4. 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. 4. omp_create_thread

4 Experimental Results

1. void omp_quiesce()
   Figure 5 shows the design of the quiesce evaluation. However, Figure 6 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.

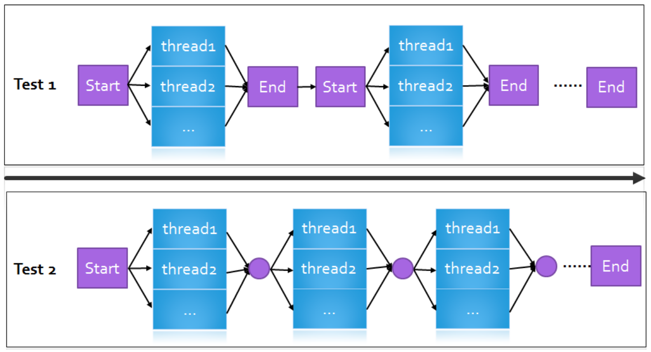
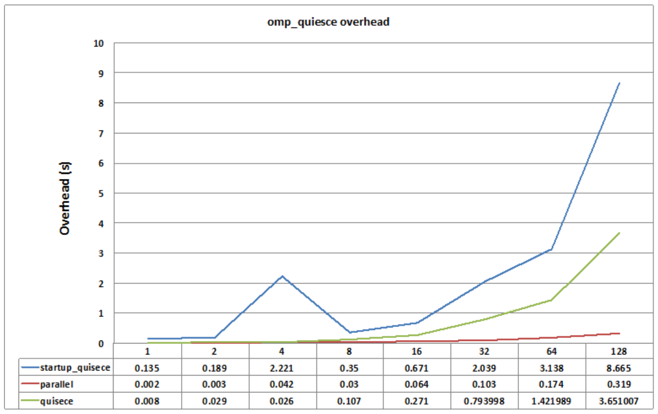


Fig. 5. omp_quiesce evaluation



	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. 6. 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 7 shows the design of the evaluation.

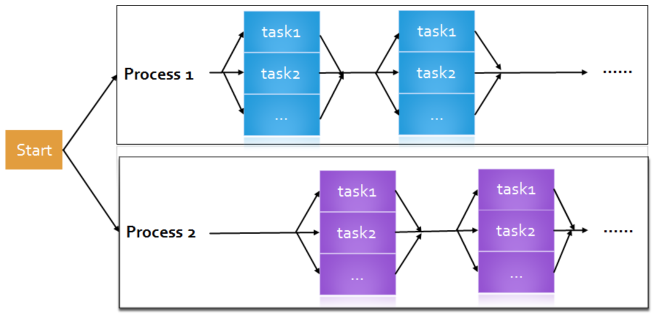
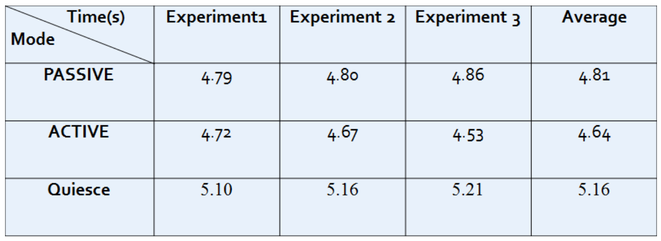


Fig. 7. waiting policy evaluation

As Figure 8 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. 8. 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 9 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.

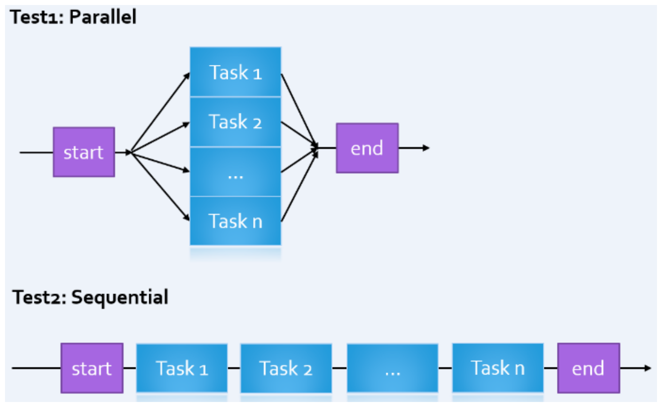
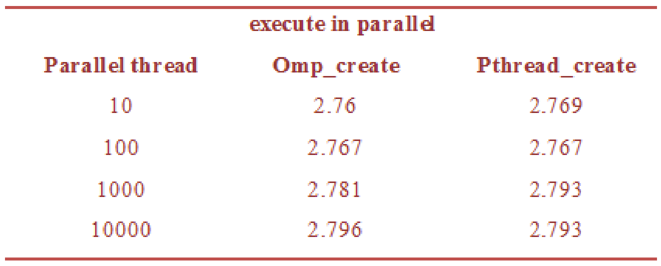


Fig. 9. creating thread evaluation

Figure 10 and Figure 11 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 12 and Figure 13 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. 10. Results of omp_thread_create in parallel

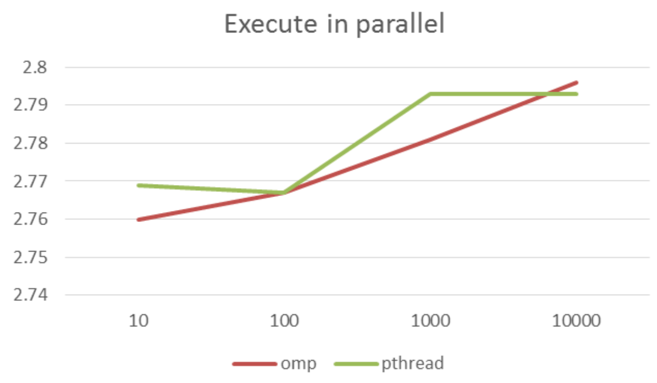
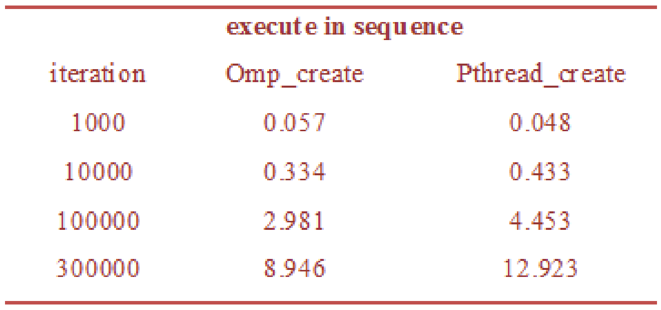


Fig. 11. Results of omp_thread_create in parallel



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. 12. Results of omp_thread_create in sequence

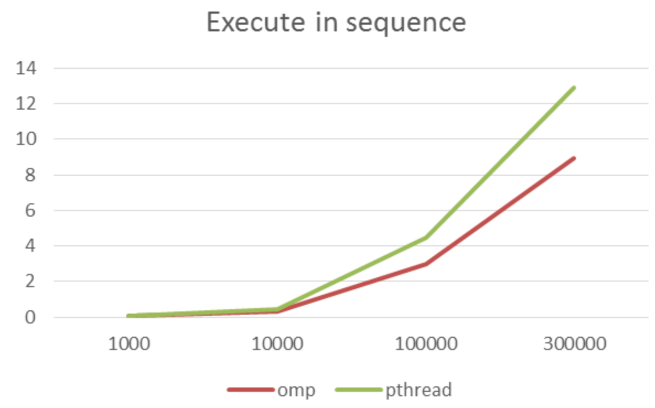


Fig. 13. Results of omp_thread_create in sequence

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. IBM Knowledge Center. XLSMPOPTS Runtime options:IBM XL C/C++ for Linux 12.1.0. `http://docs.oracle.com/cd/E24457_01/html/E21996/aewcb.html#gentextid-475`.