2019 IEEE The 3rd International Conference on Circuits, Systems and Devices

Optimization Methods for Deterministic Thread Library Dthreads Scheduling

Jinyu Li
School of Computer Science and Technology
Harbin Institute of Technology Weihai
Weihai, China
e-mail: rikinyu@163.com

Zhenzhou Ji
School of Computer Science and Technology
Harbin Institute of Technology Weihai
Weihai, China
e-mail: jizhenzhou@hit.edu.cn

Yihao Zhou
Command Information Support Center
Zhejiang Civil Air Defense
Hangzhou, China
e-mail: zyhinchina@gmail.com

***Abstract*—Dthreads is a multi-core deterministic thread library with good performance in recent years. It has great application value for programs with deterministic execution requirements. This paper focus on Dthreads scheduling process. In view of program's potential parallelism characteristics and the requirement of determinism, two optimization methods are introduced to improve the parallelism. The validity and performance of these two methods are verified by experiments. The results show that the two optimization methods can improve the running efficiency of program and guarantee determinism.**

Keywords-dthreads; deterministic execution requirements; scheduling process; parallelism

I. INTRODUCTION

Since the beginning of the era of multi-core processors, multi-thread programming has become the programming method to maximize the use of resources. Unlike single-core processor, multiple tasks may be performed simultaneously on more than one processor core. These tasks share storage resources and come with a lot of competition and interference. Nondeterminism could occur at this time. For a program, executing multiple times under the same input but with different results is fatal. The technology of the deterministic parallel execution is widely thought that is the key to deal with this problem at present [1]. By controlling the synchronization, competition and interference between parallel tasks, the program can deal with various tasks in accordance with the predetermined order and rules, so that the program can still obtain consistent and correct results under the complex scheduling environment of multi-core and multi-thread.

To implement deterministic parallel execution technology is to perfect the program with nondeterminism into deterministic parallel execution. Analysis of the sources of nondeterminism shows that there are two forms of competition in multithreaded program: data competition and synchronous competition. Therefore, deterministic parallel execution technologies are generally divided into two categories: strong deterministic technology that solves data competition and synchronous competition, and weak deterministic technology that only focuses on synchronous competition [1]. Obviously, weak deterministic technology has relatively small overhead and can get acceptable results, so it is common in practice.

Currently, multithreaded programming mostly follows POSIX's standard thread library rules. Pthread is a widely used thread library in Unix-like systems [2]. The standard library defines types, constants, and functions based on C/C++. It provides basic operations for threads: thread management, operations on mutexes, operations on condition variables and synchronization management between threads using read and write locks. Although Pthread has some ways to implement determinism, it does not guarantee that writing multithreaded programs is deterministic.

Dthreads is a deterministic multithreaded runtime library proposed by Tongping Liu and others from the University of Massachusetts. It adds some deterministic mechanisms to change the nondeterminism of traditional Pthread. Meanwhile it can be directly compatible with traditional Pthread without changing user programs and only needs to modify the linked dynamic link library when generating executable files. It brings great convenience to parallel programmers based on C/C++. In the article [3], authors introduce the mechanism of Dthreads, including memory isolation, deterministic commit protocol, deterministic synchronization mechanism, deterministic memory allocation and so on.

This paper is based on deterministic thread library Dthreads. We point out two improvement methods of Dthreads scheduling by analyzing the execution of running tasks. Finally, the effectiveness of methods are evaluated through experiments.

II. ABOUT DTHREADS

Dthreads is a new thread library that is fully compliant with POSIX thread library (Pthread) standard. It is implemented by rewriting the libc and Pthread library files related to memory allocation and thread operation behavior. The modified functions can be more consistent with the mechanism of deterministic system, so as to build a deterministic execution runtime and ensure the deterministic output of the program [4].

Here we abstract Dthreads into three modules: control information module, memory management module and scheduling control module. Fig.1 shows the abstract structure of Dthreads. Control information module maintains memory page information and scheduling control information needed for deterministic execution. This information is shared by all threads in real time. Memory management module is responsible for memory isolation (a kind of deterministic execution mechanism). In Dthreads, it is proposed to convert threads into lightweight processes to isolate communication between other threads. All behaviors for shared memory modification must wait for authorization. The sign of authorization is to hold a token. With lightweight processes having private memory resources, creating copies of shared resources ensures that threads have the same data as private pages before the beginning of each parallel phase. In parallel phase, threads modify their own copies. And in serial phase, threads submit their copies to shared memory in deterministic order and modify shared pages through byte-to-byte comparison. In this way, Dthreads implements the effect of no data competition in parallel phase and one-time modification of shared memory in serial phase. These behaviors depend on the coordination of memory management module.

Scheduling control module is responsible for switching between serial and parallel phases in the process of program running. It contains thread queue submodule and token submodule. Thread queue submodule maintains a lot of queue-like data structures, such as thread queue, condition variable waiting queue, etc. These queues are important references to ensure scheduling determinism. Token submodule serves the serial phase, and determines the order of token acquisition according to the deterministic strategy, so as to perform memory submission and synchronization operations deterministically.

When using Dthreads, user program only needs to add the header file "<pthread.h>" like using Pthread, and then adds the dynamic link library option at compile time. When running, the program initializes the environment and creates an object of class "determ" to manage the global control information, and then enters the parallel phase. In parallel phase, because threads are created according to lightweight processes, they have their own private space. These threads copy pages from global memory to private space and modify data in private space. When a thread arrives at the synchronization statement, scheduling control module calls "waitFence" to start pre-Fence and joins the thread into the pre-Fence waiting queue. When the number of threads in the queue satisfies the condition, each thread leaves pre-Fence and requests token from the token submodule. The system grants the token in the order of "threadindex". The first thread that gets token submits its private copies to shared memory and executes its own task, then enters post-Fence and waits for the serial process to end. The subsequent threads that get token compare their modified pages with the shared pages to complete the modification, execute the serial process and enter post-Fence. When all threads reach the post-Fence, the program enters a new parallel phase.

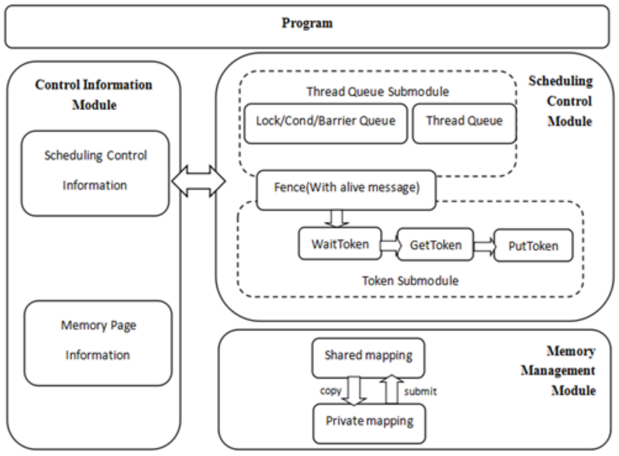

Figure 1. Dthreads modules abstract structure diagram.

III. DTHREADS SCHEDULING ANALYSIS AND OPTIMIZATION

As we know, Dthreads will go through several serial and parallel phases in running process. For a program with synchronous operation, serial phases are inevitable. Therefore, reducing serial time as much as possible is a means to improve efficiency.

According to the running process of Dthreads, we know that Dthreads changes from parallel to serial by setting up a previous fence(pre-Fence). When all parallel threads run into synchronization statements, they enter the queue of pre-Fence and wait for serial operation to begin. The timing to start the serial phase is when all threads reach pre-Fence. The last arriving thread start serial phase and execute it serially according to the strategy.

It is worth noting that Dthreads have used "synchronization point submission copy" to avoid data competition and reduce the frequency of modification of shared memory, but there are two potential factors that can reduce the degree of program parallelism under the above mechanisms. First of all, during the serial phase, only one thread occupies the processor, other threads are waiting, and other cores do not execute the program. Secondly, in parallel phase, Fence is set to ensure determinism. The rules require all threads enter into serial phase when a synchronization statement is encountered. Some threads reach the synchronization soon, and other threads have a great synchronization distance. It will cause threads with short synchronization distance to wait for a thread with long synchronization distance.

A. Partial Parallelization

The output is unique and deterministic when using Dthreads because serial phases follow a given path. In reality, it is possible to synchronize without conflicting access to resources (i.e., no mutually exclusive operation between some threads). According to the regulations, all threads must execute sequentially in a serial phase, which reduces the parallelism of the program. So we want to compress serial phases and add parallel factor in serial phase. From programming point of view, when multiple threads have the need to access shared memory, setting mutex to avoid some problems (e.g., dirty data) is an effective method in parallel programming. Therefore, in a standard multi-threaded program, mutex must be set to limit access to the same resource. This programming feature provides an idea for serial phase parallelization.

We can divide threads into mutex threads group and other synchronization threads group according to the type of synchronization operation applied to the system during synchronization phase and data structure "LockEntry" in the system. We group all threads that competing for the same mutex into one mutex thread group, then in a serial phase we can get several mutex thread groups and one other synchronization operation thread group, each mutex group can be stored in computer as a queue. When Fence starts the serial phase, different thread groups execute in parallel, and threads in the same thread group execute in serial. When each thread group completes its task, there is also a serial operation between thread groups to synchronize local results to global shared memory. We set up a group-Fence here. When the last thread in each thread group calls "putToken", system will call "waitFence" to make each thread group wait for "group synchronization". All thread groups are detected to have called "waitFence" then group synchronization phase will be started and all of thread group modification will be submitted in order according to smallest thread index in the thread group. Finally, all threads enter the post-Fence to complete the whole serial phase. Fig. 2 shows the process of partial parallelization.

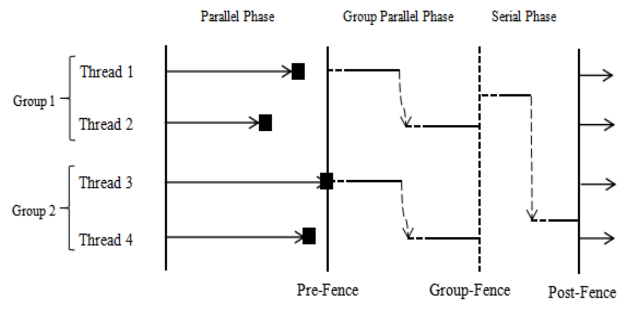

Figure 2. An overview of partial parallelization. The dashed line in serial phase represents the atomic commit page operation and the solid line represents synchronous operation diagram.

Everyone familiar with Linux programming knows that conditional variable depends on the protection of mutex. In addition, multiple mutexes are sometimes called in the case of data competition. For these situations that synchronization process is nested with other synchronization processes, the above strategy cannot be taken into account when dividing thread groups. In order to ensure deterministic execution, we have to abandon parallelism and wait for the inter-group serial phase to execute sequentially.

This strategy increases the parallelism of program and ensures that the program is still deterministic. This is because the strategy limits all threads with mutually exclusive operations to one thread group, and the operations between groups are mutually non-interference operations. For each thread, the timing of quitting synchronization operation and ending the serial phase is clear, and the strategy of partial parallelism does not change fundamental characteristic of serial phase. Such parallelism has not effect on determinism. It should be pointed out that the conditions for increasing parallelism are very harsh. In addition, when too little work is done in critical area or too many pages are submitted, the promotion may not be particularly obvious. The following experiments also show that not all programs using deterministic thread libraries can benefit from it.

B. SKIP Strategy

Different threads perform different tasks, and the distance from parallel phase to synchronization statement varies. This may lead to a thread with short synchronization distance waiting for long synchronization distance thread. And the effect has no great advantage over serial execution. This is not ideal for parallel programs. We expect long synchronization distance thread to abandon the nearest serial phase voluntarily and other threads can enter synchronization as soon as possible. According to our findings, running time is not a measure of whether to skip the serial phase because different cpus may have different frequency. The method of prediction based on the number of instructions executed needs to use program instrumentation like DMP [5], which is not very practical. So a mechanism called SKIP has been proposed and implemented.

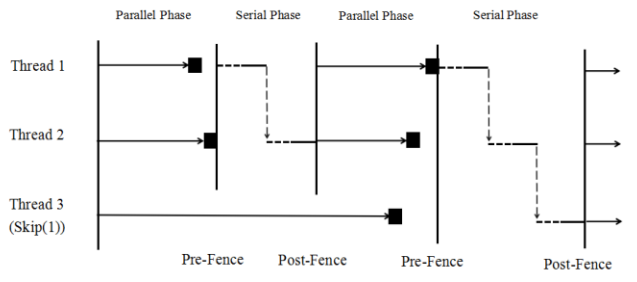

Figure 3. An overview of SKIP strategy.

This mechanism adds SKIP state and SKIP function to Dthreads. The function SKIP has a parameter "times" that indicates the number of serial phases to skip. When function SKIP is encountered in parallel phase, the thread is set to SKIP state and joins SKIP queue. Thread in this state is the same as ordinary threads except for skipping serial phase. When normal threads call "waitFence" and are about to enter a new serial phase, function "waitFence" subtracts the SKIP counter by one and removes threads with a value of 0 from SKIP queue. These threads will enter the pre-Fence in this round and participate in the serial phase. Finally, this function maintains global thread information based on the number of remaining threads in the SKIP queue.

Synchronized operations in Dthreads will start serial operations. Under the same input, a program has a fixed number of serial phases, and so does each thread. Since threads specify the number of serial phases to skip, the operation is also clear about which serial phases are affected. From the concept of determinism, this strategy is consistent with determinism.

```
Init(Mutex1,Cond,Mutex2)
Function CONSUMER(threadindex)
    Pthread_mutex_lock(Mutex1)
    Pthread_cond_wait(Cond,Mutex1)
    Print()
    Pthread_mutex_unlock(Mutex1)
    If(threadindex<1)
        Pthread_mutex_lock(Mutex2)
        Print(Count())
        Pthread_mutex_unlock(Mutex2)
End Function
```

```
Function PRODUCER(threadindex)
    While(--num>=0)
        Pthread_mutex_lock(Mutex1)
        Pthread_cond_signal(Cond)
        Pthread_mutex_unlock(Mutex1)
End Function
Function Main()
    For i = 1 to 4
      Pthread_create(threadinc[i],CONSUMER)
    Pthread_create(threaddec,PRODUCER)
    Pthread_join(threaddec)
End Function
```

Figure 4. Program to verify the validity of the serial partial parallelization.

When implementing this strategy, we have following situations to consider. The first case is when a thread calls function SKIP after Fence starts. Because the thread must not have run to synchronization statement at this time, we only need to ensure that function SKIP can modify the global thread information correctly, then the serial phase can proceed as expected. We use the global lock provided by Dthreads to achieve global thread information consistency. Another case is when a parallel thread encounters a synchronous statement earlier than expected and the value of the SKIP counter is not 0. Although the number of skipped rounds is known, the number is uncertain due to the different execution environments. So the idea of ending SKIP state early is not feasible. In other words, it should be set that the thread must wait for the counter to return to 0 before performing synchronization.

IV. Experimental Analysis

The experiment is conducted on a PC with CPU Intel Core i5-4690 3.50GHz has 4 cores and 3.7GB memory. The operating system is Ubuntu 14.04.4 LTS, running with Linux kernel version 4.2.0-27-generic.

TABLE I. Comparison between Pthread and Optimized Dthreads

Times	Pthread Sequence	Optimized Dthreads Sequence
1	3 0 1 2	0 1 2 3
2	2 1 0 3	0 1 2 3
3	0 1 2 3	0 1 2 3
4	1 3 0 2	0 1 2 3
5	3 2 1 0	0 1 2 3

Firstly, we verify the validity of the serial partial parallelization. We design an experimental program based on producer-consumer model. The program also involves competing for different locks. The pseudo-code of the program is shown in Fig. 4. We use Pthread and optimized Dthreads to do experiment. Four consumer threads are set up in experiment. As shown in Table I, the results of Pthread are different, but the results of Dthreads are the same. This is because deterministic thread library is guaranteed by scheduling mechanisms such as Fence and Token, and the scheduling is carried out according to the prescribed principles. We believe that the mechanism is deterministic.

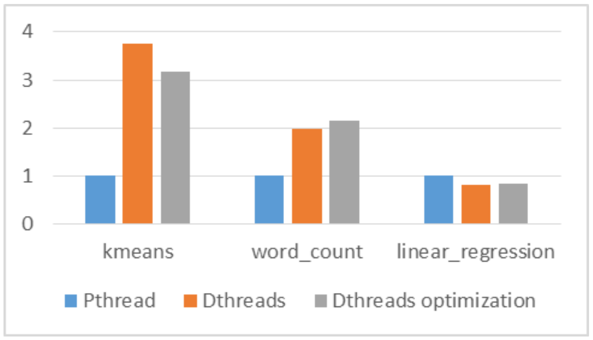

Figure 5. Program run time histogram (normalized).

Next, we use Phoenix-2.0 (i.e., a kind of test set) to complete the performance test. This test set is a data-intensive parallel program set suitable for multi-core processor platform [6]. We select three test programs with Pthread mode: kmeans clustering algorithm, word_count and linear_regression. The number of threads is 8. Each program runs 20 times, and eliminates the highest and lowest results. The average result is normalized according to the Pthread results. Fig. 5 is a histogram after statistics. It can be found that the efficiency of the two versions of Dthreads is not much different. There are several phenomena that we can notice:

1. Using Pthread in linear_regression is on average longer than using two versions of Dthreads. In fact, because the test program itself runs in a short time, it is more sensitive to the computer environment. Through tracing, we find that linear_regression program only performs one round of synchronization operations with multiple threads. Under short and few synchronization operations, the efficiency of the program is not much different from that of Pthread. This

shows that under short synchronization, the impact of these mechanisms of Dthreads is very limited, but the results are deterministic.

2. Kmeans have more synchronization operations, and the overhead of deterministic mechanism is obvious. The performance of the optimized version is slightly better than that of native Dthreads, which shows that serial partial parallelization has played a predicted role to some extent.

3. In word_count, the optimized version of Dthreads takes more time. We found that the program spends a lot of working time in parallel process, and the load of serial synchronization operation is not larger than that of submitting pages, so partial parallelization strategy does not play a very important role. On the contrary, the additional overhead of inter-group serial scheduling affects efficiency of the program. This is consistent with previous analysis.

Similarly, we do experiments to evaluate SKIP strategy. Since the strategy is optimized at the programmer level and targeted threads have obvious differences in synchronization distance, we wrote a test program shown in Fig. 6.

```
Init(Mutex)
Function LONGPARALLEL()
   SKIP(1)
   Do_some_work(Heavy_load)
   Pthread_mutex_lock(Mutex)
   Print(k=1)
   Pthread_mutex_unlock(Mutex)
End Function
Function SHORTPARALLEL()
    Pthread_mutex_lock(Mutex)
    Print(k=2)
    Pthread_mutex_unlock(Mutex)
    Do_some_work(Light_load)
    Pthread_mutex_lock(Mutex)
    Print(k=3)
    Pthread_mutex_unlock(Mutex)
End Function
Function Main()
    Pthread_create(thread,LONGPARALLEL)
    For i=1 to n
      Pthread_create(t[i],SHORTPARALLEL)
    Pthread_join()
End Function
```

Figure 6. Program to verify the validity of SKIP strategy.

TABLE II. TEST RESULTS OF SKIP STRATEGY

Times	Results
1	2(2) 2(3) 2(4) 3(2) 1(1) 3(3) 3(4)
2	2(2) 2(3) 2(4) 3(2) 1(1) 3(3) 3(4)
3	2(2) 2(3) 2(4) 3(2) 1(1) 3(3) 3(4)
4	2(2) 2(3) 2(4) 3(2) 1(1) 3(3) 3(4)
5	2(2) 2(3) 2(4) 3(2) 1(1) 3(3) 3(4)

Table II shows the results of SKIP strategy. Once the position of function "SKIP" in the program and the number of rounds to be skipped are determined, the output of the program will be carried out according to the deterministic principle. The results of program execution will satisfy the requirements of determinism.

From the result, the numbers in parentheses represent "threadindex" given by Dthreads. Thread 1(i.e.,long parallel) joins serial phase in the second round. At this time, the first element of global queue "tokenpos" defaults to thread 2 (the first thread in the previous round), so thread 2 runs first in this round. However, this does not affect deterministic output. The sequential logic of Dthreads serial phase can be changed by user if they have special requirements for output sequence.

Under the condition of using Dthreads library with SKIP strategy, we let the user program run with SKIP and without SKIP respectively. As shown in Fig. 7, we have done 10 experiments at random, and the results show that the efficiency of program without SKIP is not as good as that with SKIP strategy. It shows that SKIP improves the effect to a certain extent.

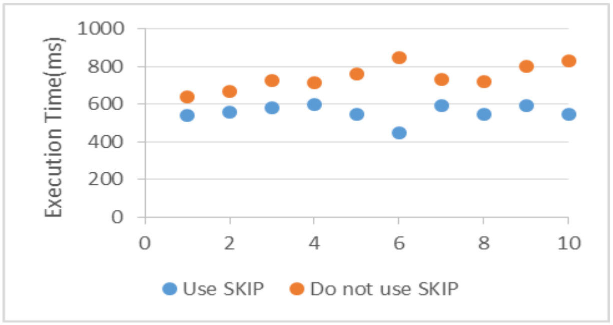

Figure 7. Execution time scatter diagram of SKIP.

V. CONCLUSION

Based on the Dthreads scheduling process, this paper makes analysis and experiment on the two improved ideas. Partial parallelization improves parallelism of program operation moderately, but its scope of action is limited because the parallelism condition is not easy. The experimental results show that this method is available. SKIP strategy provides an optimization method from the programmer's point of view. The experiment proves its effectiveness and good performance. In conclusion, Using these two methods correctly can make the scheduling process more efficient.

REFERENCES

[1] Z. Xu, L. Kai, C. Chen, "Deterministic Parallel Technique", Chinese Journal of Computers, vol. 38, May. 2015, pp. 973-986.

[2] Butenhof D R. "Programming with POSIX threads". Addison-Wesley Professional, 1997.

[3] L. Tongping, C. Curtsinger , and E. D. Berger . "Dthreads: efficient deterministic multithreading." Acm Symposium on Operating Systems Principles DBLP, 2011, pp. 327-336.

[4] Fei Y, et al. "Comparative modelling and verification of Pthreads and Dthreads". Journal of Software: Evolution and Process 30.5(2017):e1919.

[5] Devietti J, Lucia B, Ceze L, Oskin M. "DMP: deterministic shared memory multiprocessing." International Conference on Architectural Support for Programming Languages & Operating Systems ACM, 2009, pp. 85-96.

[6] Yoo R M, Romano A, Kozyrakis C. "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system." IEEE International Symposium on Workload Characterization, 2009, pp. 198-207.