A Quick Study on the Efficiency of Multithreading Technology with C++11

Wenjie ZHENG[*]

October 6, 2014

This article is aimed to give the readers some ideas about multithreading technology as well as the performance of multithreading in C++11 environment. Baptiste Wicht gave a primary study on this problem in his blog in 2012. However, he overlooked some issues. A more detailed study will be given here.

In this article, readers will find answers to these two questions:

1. Is multithreading always more efficient than a single thread?
2. *Mutex* or *Atomic type*, which one to use?

This article presumes that the readers have basic knowledge of multithreading, for example why we do multithreading, in which circumstances; what is mutex or atomic type; why we need it. If the readers don't, they are invited to read related documents first. For the general knowledge, they could consult the Wikipedia page. For the meaning and usage of mutex and atomic type, they could find enough infomation in Baptiste Wicht's blog.

The code used in my experiments is inspired by the one written by Baptiste Wicht. You can find my code in Github. The CPU on which I had my experiments is *Intel® Pentium® Processor 2020M*, a dual-core one without hyper-threading technology. Therefore, I will only be able to test up to 2 threads. The operating system is Windows 8.1 64 bit. I compiled and ran my code in Cygwin 64 bit, in using GCC 4.8.2 as compiler.

Now let us begin the topic.

Multi threads vs. Single thread

The answer to the first question depends whether the designed threads are parallel or sequential (supposing that the task is always equally distributed to all threads) .

If the threads are parallel, for instance in the code below, multithreading is more efficient:

[*]For questions or propositions, please email to wenjie.zheng@lip6.fr.

```
void iteration(){
    for(int i = 0; i < ITERATION; ++i);
}
```

Each thread runs this function, and they are parallel. In my experiment, the 2-thread program took only 3% more time than the 1-thread program. Since it completed twice the task, it is 94% more efficient than the 1-thread one.

If the threads interact among them, the program becomes partly sequential, which is the case in the following code:

```
void lock(){
    for(int i = 0; i < ITERATION; ++i){
        mutex.lock();
        mutex.unlock();
    }
}
```

No threads are allowed to get access to the mutex, until the thread currently owning it unlock it. Ideally, the 2-thread program will take exactly twice as much time as the 1-thread one, since it has twice the work to do. However, this was not the case in my experiment. In the best case, the 2-thread program took indeed a little more than twice the time. But in average, it took 3000% more time, which means its efficiency is only 6% of 1-thread program.

In conclusion, if the threads interfere each other seriously, multithreading is not recommended. Actually, since the example above shows the worst case (it does nothing between the lock and the unlock of the mutex), it gives an estimate of the lower bound of the efficiency of multithread program.

The discussion above teaches us two things. On the one hand, not all problems are suitable for multithreading technology; those which could be divided into parallel sub-problems are recommended to do so (e.g. traditional Monte Carlo method other than Metropolis-Hastings or Gibbs sampling). On the other hand, if we really want to use multithreading technology, always look for a better design to insure that the threads interact with each other as least as possible.

Mutex vs. Atomic Type

The figure in Baptiste Wicht's blog gives an illusion that atomic type always outperforms mutex. However, this is not ture. The answer to the second question is still it depends. Indeed, if one was better than the other in all circumstances, the latter one would have been replaced.

The code below is a simple but persuasive illustration, which shows the advantages of both simultaneously:

```
long s1 = 0;
void sum_mutex(){
    for(int i = 0; i < ITERATION; ++i){
        mutex.lock();
        for(int j = 0; j < NUM_TASKS; ++j){
            ++s1;
        }
        mutex.unlock();
    }
}

std::atomic<long> s2;
void sum_atomic(){
    for(int i = 0; i < ITERATION; ++i){
        for(int j = 0; j < NUM_TASKS; ++j){
            ++s2;
        }
    }
}
```

In both case, there are two loops. The difference is that the first uses mutex (NUM_TASKS additions are protected as a whole by a mutex), while the second uses atomic type. The result shows that the advantage of atomic type over mutex decreases with the increase of NUM_TASKS, and increases with the number of threads. For instance, in my experiment, I got the following result with a tuned value of NUM_TASKS :

Table 1: Comparison of both methods

used time (unit: μs)	1 thread	2 threads
using mutex	500	8325
using atomic type	1238	6218

We can see from Table 1 that in the 1-thread program, mutex outperformed atomic type, while in the 2-thread program, atomic type outperformed mutex.

The explanation is that the operation on atomic types are more expensive than primary types. In the 1-thread case, although there is extra cost in the lock and unlock of mutex, the addition of primary types finally earns more time than mutex itself costs. In the 2-thread case, although each single thread is more expensive for the one using atomic type, it provides more flexible cooperation between the two threads than the one using mutex, thus earns itself some time.

The discussion above teaches us that if the protected task is a simple one, use atomic type; otherwise (the protected task is heavy), use mutex.