Assignment 3: Spin Locks

Course No.	Name	Email
CSC458	Jiupeng Zhang	jzh149@ur.rochester.edu

Usage

Compile: `/path/to/jzh149/make`

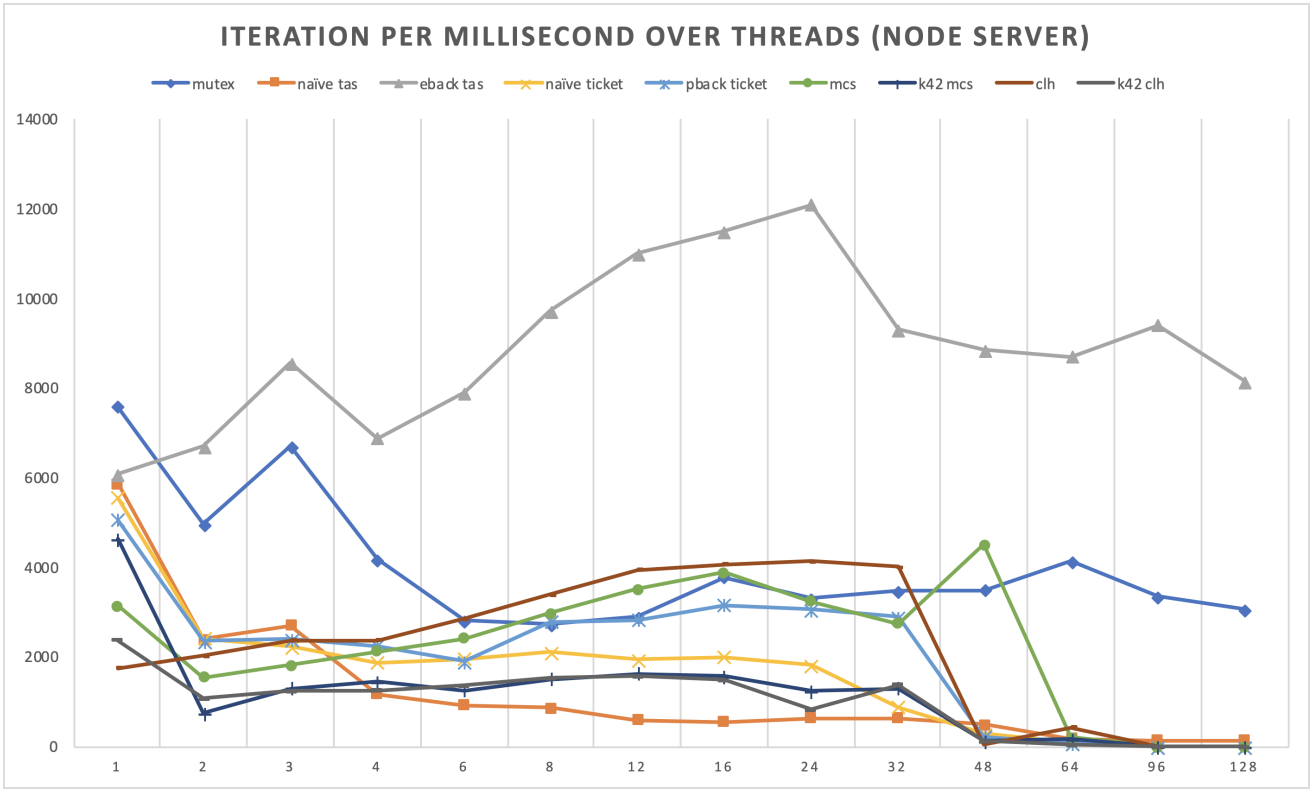
Run by default: `/path/to/jzh149/parcount`

Parameters:

- `-t` number of threads *(default: 4)*
- `-f` number of iterations *(default: 10000)*

Comparison of spin locks

I tested my program with 1000 iterations per thread. Suppose we set a parallel counter with *k* threads, the final counter values of *1000k* indicates a correct result. The following results are all under the correct counting, hence mainly show the differences in throughput within a consistent thread load.



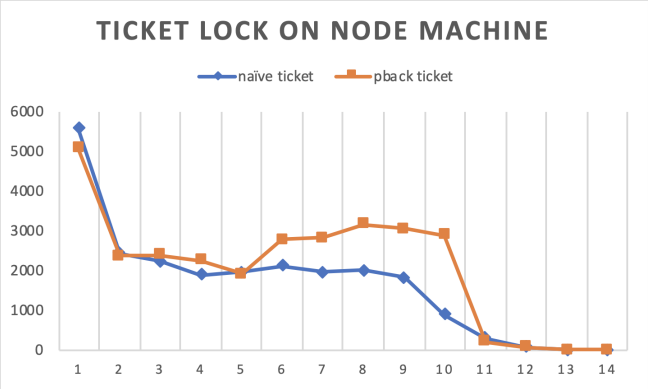
Following the order of given legends exclude the c++ mutex lock (introduced as a datum), we can say that the exponential backoff version of TAS lock has a surprisingly outstanding performance, given the reason that it properly ease the pressure of frequently requesting locks, especially in this trivial test with tiny and uniform (with same logic during the) critical sections. Without the backoff improvement of TAS, the naive TAS lock is hardly express its advantages because of taking too many CPU cycles which threaten the proportion of normal/valuable calculations.

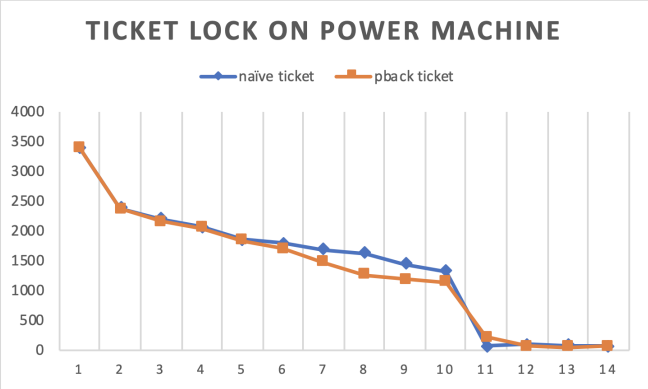
```
// Test-And-Set Lock (with exponential backoff)
void acquire() override {
    int delay = base;
    while (f.test_and_set()) {
    for (int i = 0; i < delay; i++);
        delay = std::min(delay * multiplier, limit);
    }
    atomic_thread_fence(std::memory_order_acquire);
}

void release() override {
    f.clear();
}
```

However, TAS locks may raise the *starvation problem*, to avoid this, we need more control over those who are acquiring the locks. To introduce *fairness*, we need to ticketing / queuing the upcoming requests, which will sacrifice some performance on these kinds of locks. For the ticket locks, all waiting threads are spinning on a global `now_serving` mark if its not their's turn for a move, such that as the total counting thread rises, this pure polling situation intensified, and occupied most of the CPU time which causes performance degradation.

A proportional backoff version of ticket lock comes to alleviate this issue. Unfortunately, the effect of backoff is not obvious, because the critical section is too small in our case, such that any tunning on `base` trying to match the time for a single critical section execution is trivial in performance (after several tests, I decided to set the base to roughly 20 loops waiting for a ticket pass).

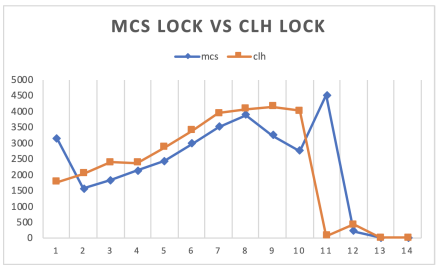


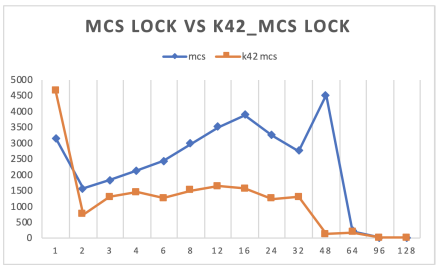


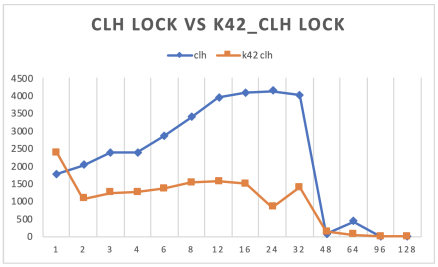

```
// Ticket Lock (with proportional backoff)
void acquire() override {
    int my_ticket = next_ticket.fetch_add(1), ns;
    while (true) {
        ns = now_serving.load();
        if (ns == my_ticket) break;
        for (int i = 0, pause = base * (my_ticket - ns); i < pause; i++);
    }
    atomic_thread_fence(std::memory_order_acquire);
}

void release() override {
    int t = now_serving.load() + 1;
    now_serving.store(t, std::memory_order_release);
}
```

Overall, these type of ticket locks offering fairness rely on remote/unified memory accessing to decide if it's a "right" turn for a move, such actions need expensive thread switching and scheduling costs (a fast drop at the end shows it reaches the bottleneck of the 64 hardware-threaded NODE machine). To solve this problem, we introduce the MCS and CLH lock to shift from the remote querying solution to a local spinning implementation (like the TAS locks), as well as offering the decent queueing.







An MCS lock is implemented based on the linked list, it put a marker on its own node for the spinning wait, and its predecessor with "wake" it up by flipping the marker so that it can pass the polling to enter the following critical section. However, the instantiation for `qnodes` for queueing (basic API version) paid for some of the performance.

```
// MCS Lock
void acquire(qnode &p) {
    p.next.store(nullptr);
    p.waiting.store(true);
    qnode* prev = tail.exchange(&p, std::memory_order_acquire);
    if (prev != nullptr) {
        prev->next.store(&p);
        while (p.waiting.load());
    }
    atomic_thread_fence(std::memory_order_acquire);
}

void release(qnode &p) {
    qnode* succ = p.next.load(std::memory_order_acquire);
    if (succ == nullptr) {
        qnode* t = &p;
        if (tail.compare_exchange_strong(t, nullptr)) {
            return;
        }
        while ((succ = p.next.load()) == nullptr);
    }
    succ->waiting.store(false);
}
```

The main difference between a CLH and MCS lock is about the location to place the `wait` boolean. Instead of putting it on its predecessor's node, an MCS lock put the marker on its own node and querying it to decide if it is its turn to go. One possible reason of doing that is it has a better cache coherence for the reason that the hot zone of memory accessing mainly focus on a running node instead of two.

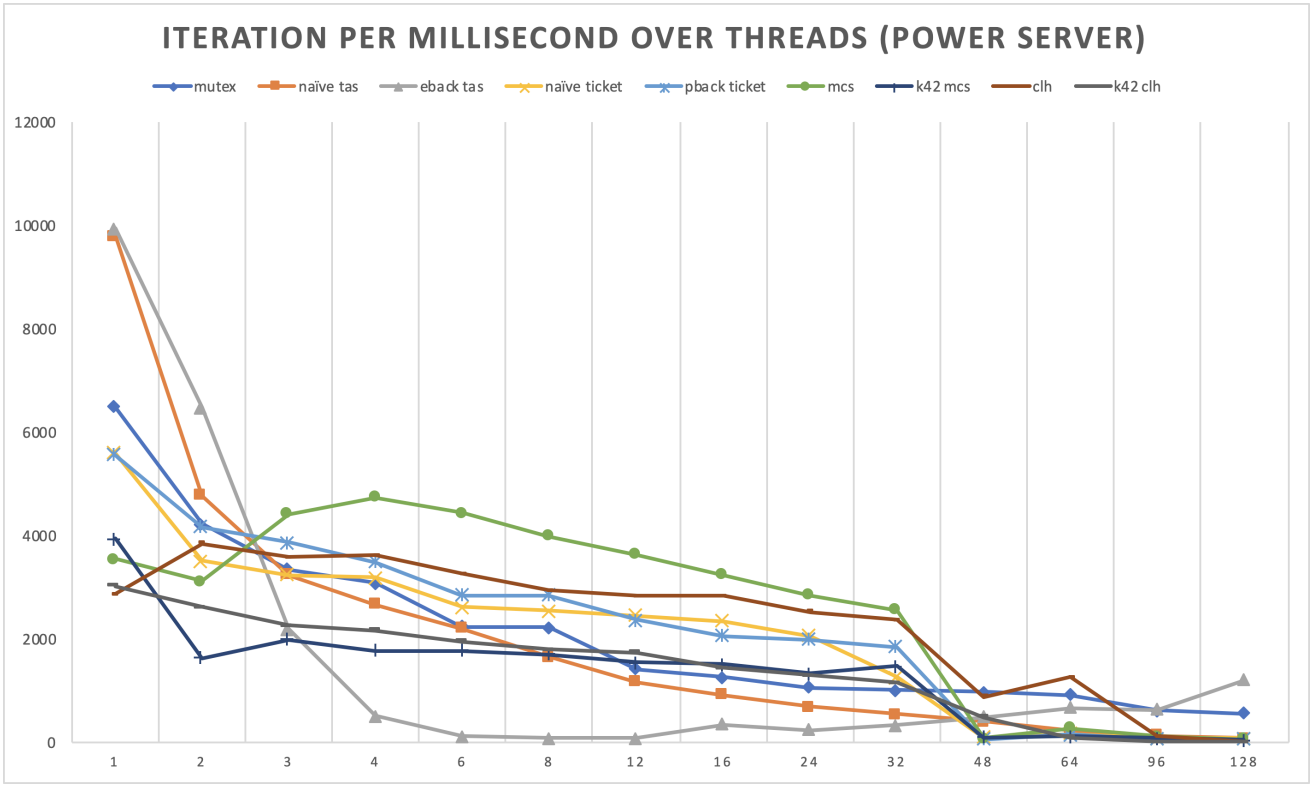
```
// CLH Lock
void acquire(qnode &p) {
    p.next.store(nullptr, std::memory_order_relaxed);
    p.waiting.store(true, std::memory_order_relaxed);
    qnode* prev = tail.exchange(&p, std::memory_order_acquire);
    if (prev != nullptr) {
        prev->next.store(&p);
        while (p.waiting.load());
    }
    atomic_thread_fence(std::memory_order_acquire);
}
```

```
void release(qnode &p) {
    qnode* succ = p.next.load(std::memory_order_acquire);
    if (succ == nullptr) {
        qnode* t = &p;
        if (tail.compare_exchange_strong(t, nullptr)) {
            return;
        }
        while ((succ = p.next.load()) == nullptr);
    }
    succ->waiting.store(false);
}
```

However, the CLH's scattered querying feature delays the downgrade, especially trans from a single thread turn to two. As you can see on the above picture, the brown line (CLH) indicates that for the segment from one thread to two, it avoids a downgrade on the throughput, where other queueing solutions obviously not. It is also worth mention that the descend at the end part has a lag effect. A possible reason for that is the CLH solution first reach the top line because more time is needed because of cross cache accessing.

Test on relaxed memory model (IBM Power machine)

> Because of its more relaxed memory model, the IBM machine may expose bugs that are hidden on the x86.



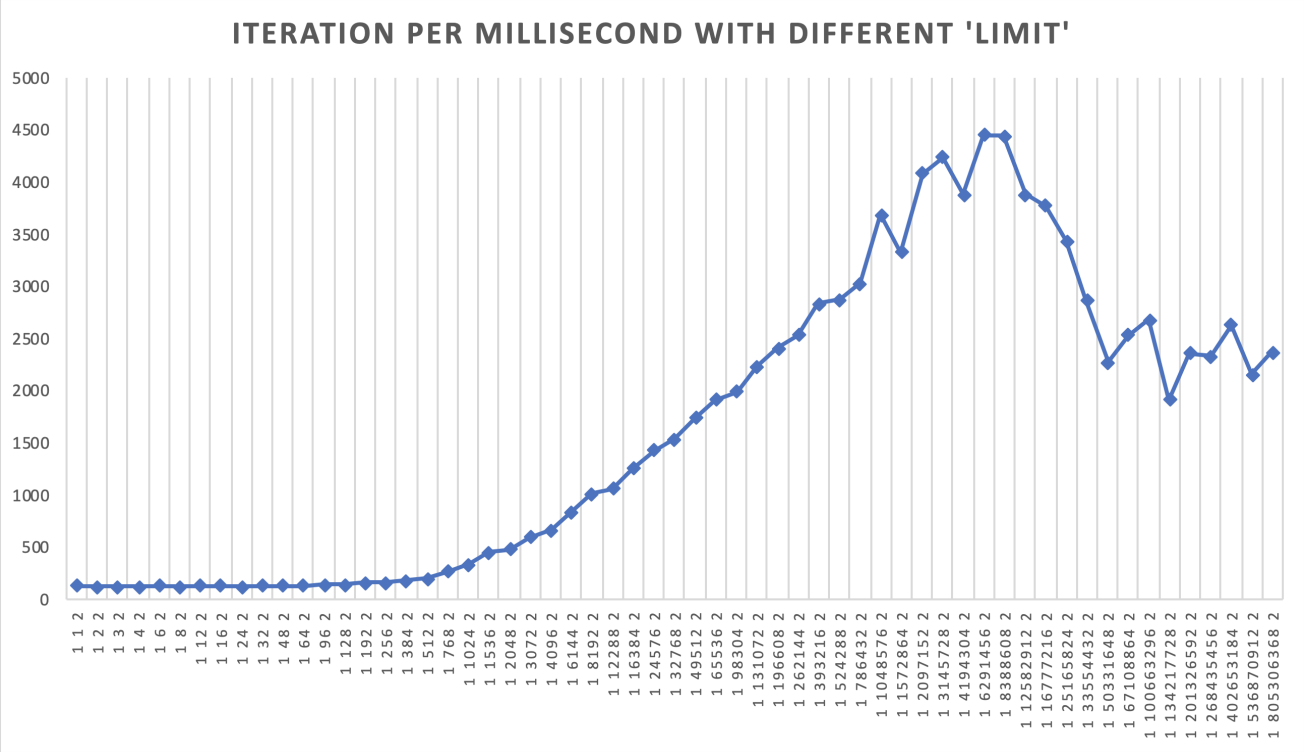
The PPC has more memory ordering relaxation than an x86 machine by supporting LL/SC instructions for synchronization (instead of the CAS instruction for x86 machines). By roughly compare the throughput trend, a PPC machine shows more sensitive during increasing threads than the NODE machine, even if it has a larger total throughput on this machine within a single thread.

Moreover, the effect of the exponential backoff TAS lock is weakened, and even much worse then I expected. A possible reason for that might be doing a "while loop" when executing a looseness loaded `test_and_set`, it might exist a delay when we read the updated value of `f`. And the backoff strategy Intensified this phenomenon. The second possible reason is that my tuned backoff parameters are not capable of this case, especially on specific machines, the solution for that is to take more experiments to figure out a well-tuned backoff configuration.

Test on relaxed memory model (IBM Power machine)

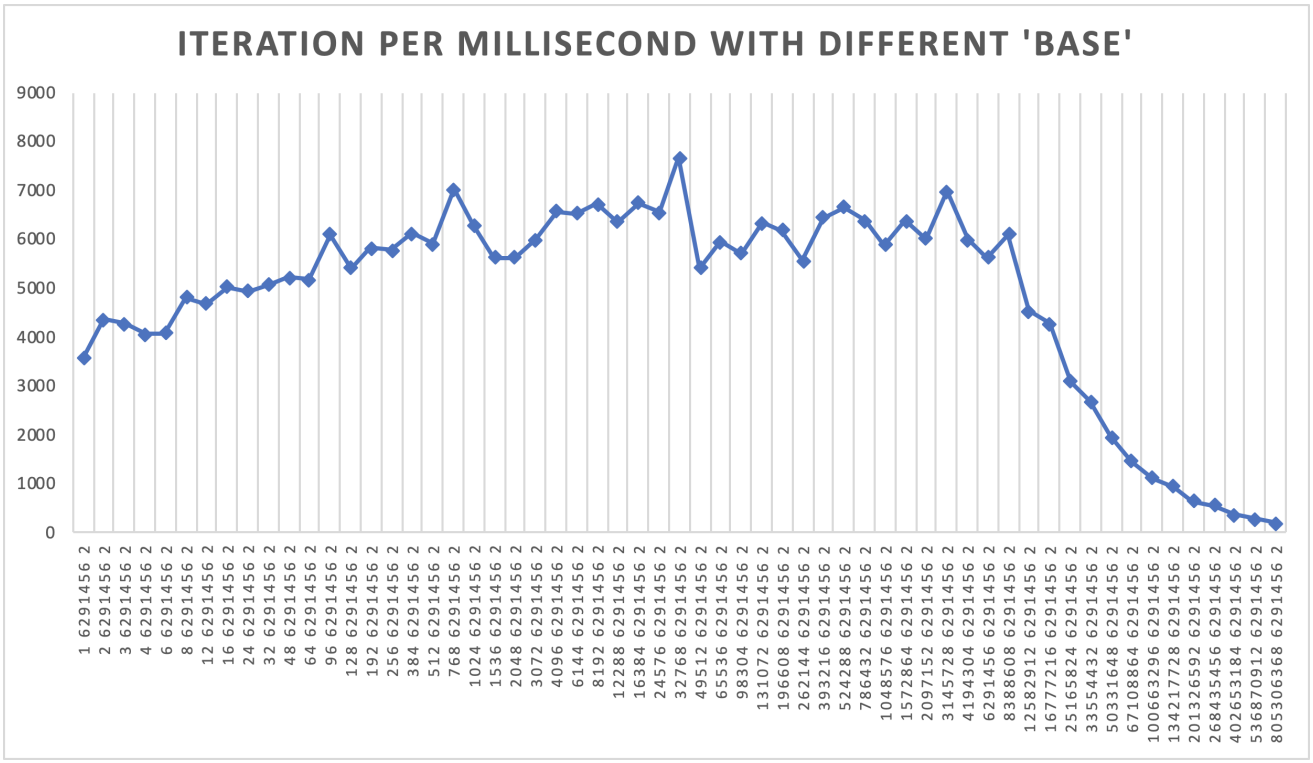
> Because of its more relaxed memory model, the IBM machine may expose bugs that are hidden on the x86.

The eback version tas lock uses an exponential increasing loop cycle to delay the lock from acquiring under a busy situation. To tune the parameters, a triad of (base, limit, and multiplier), I compared its performance under different configurations.

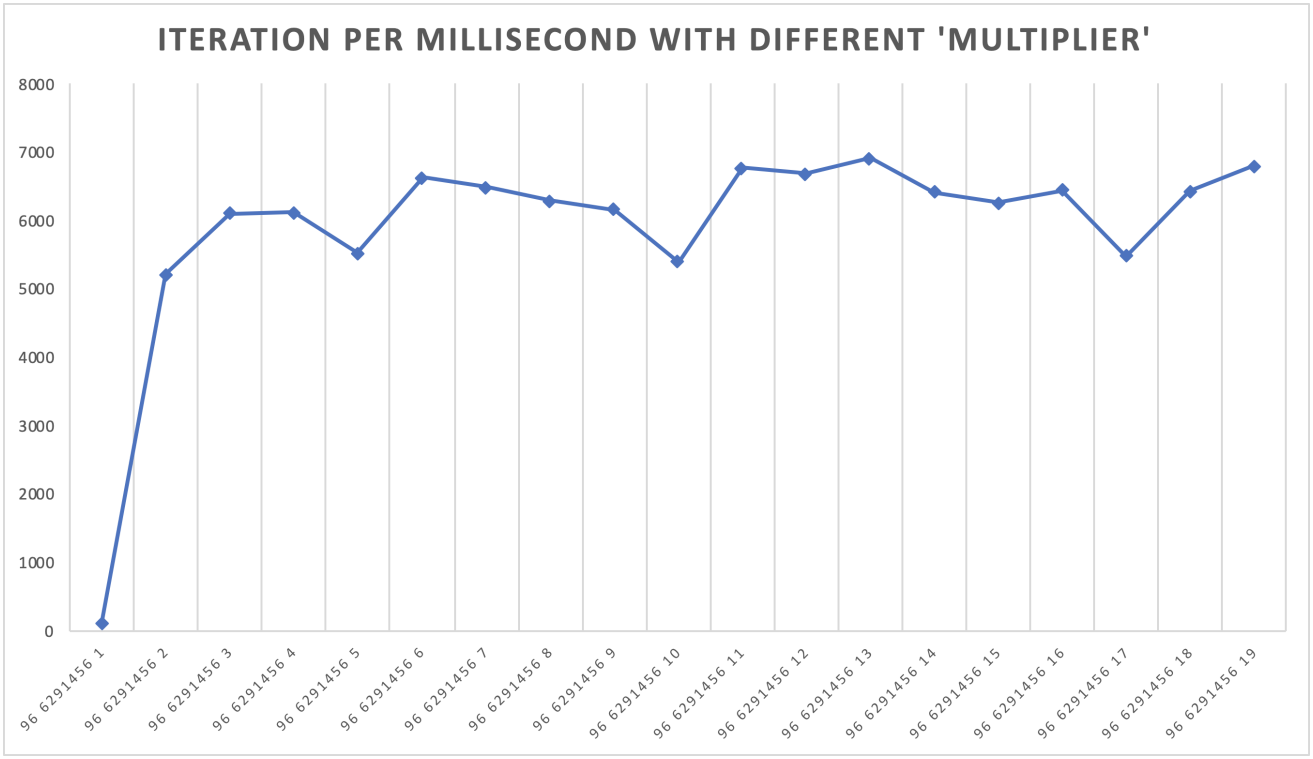


In experiment #1, I fixed the `base` and the `multiplier`, set them to (1, 2) as default, and then picked 50+ different `limits` to find a reasonable `limit` value for our case. After 10 tests, I picked their average and found that a `limit` for about 5,000,000 is a good choice, so I moved on to the next section to tune the `base`.

From the graph above, we can see that the performance was improved at the first, However, as we increase the threshold too much, it then dropped down drastically, that is because, at the very beginning, the `limit` is too strict so that it cannot offer enough backoff for each crash. Then, with an increasing threshold, it performs better. The tail part of the graph shows that choosing a *too large* `limit` raises bad consequences such that every thread becomes overly polite so that the program cannot get full use of the parallelism.



Setting a `base` solves the "slow-start" issue and make sure the back-off behaviors fast and efficient. Experiment #2 is to find a good `base` for our program solving the specific problem (trivial integer counting). We can find multiple peeks and falls on the above picture such that our shifting on `base` seems unstable. For safe, I chose the `base` following with an acceptable "drop pattern" as the `base` for the program (96). For the following part of the graph, the descending is because of the overly waiting.



In experiment #3, I decided how fast to grows our delay for the backoff. based on my subjective will that not to wait too much during the competition, I chose 6 as the multiplier, for the reason that it brings good enough performance with a relatively small multiplier.

After tuning my parameters (I randomly chose a triad of (256, 2048, 2) before), I got an about **1.6x** improvement on my current implementation (96, 6291456, 6). To compare with the naive TAS lock, I acquired for about **10.0x** enhance on average.

Sample test case

```
$ cd /path/to/cs458/03/
$ make
$ ./parcount -i 100000 -t 4

[-t] number of threads:    4
[-i] number of iterations: 100000

running test_cpp_mutex...
result of count: 400000
completed in 31.5551 ms

running test_naive_tas...
result of count: 400000
completed in 86.7362 ms

running test_eback_tas...
result of count: 400000
completed in 7.77462 ms

running test_naive_ticket...
result of count: 400000
completed in 48.5501 ms

running test_pback_ticket...
result of count: 400000
completed in 43.4282 ms

running test_mcs...
result of count: 400000
completed in 48.4597 ms

running test_k42_mcs...
result of count: 400000
completed in 67.1232 ms

running test_clh...
result of count: 400000
completed in 50.0262 ms

running test_k42_clh...
result of count: 400000
completed in 69.1083 ms
```