CS 111 Project 4 answers

By Chunnan Yao (204568002) Jiaqi Huo (704589101)

Questions 1.1

1. Why does it take this many threads or iterations?

The more threads or iterations, the higher possibility that conflicts will happen during calling functions. Race condition will be easier to happen.

2. Why does a significantly smaller number of iterations so seldom fail?

Because when number of iterations is small, the computation will be so fast, thus threads run so fast that they have lower possibility to race with other threads.

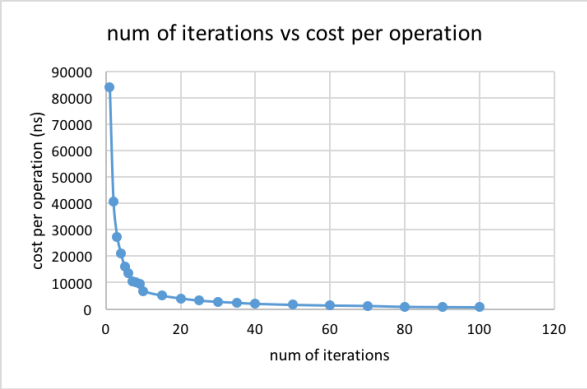


Figure 1-1 the average time per operation vs the number of iterations

Questions 1.2

1. Why does the average cost per operation drop with increasing iterations?

Because thread creation time is much larger compared with operation time. When number of iterations is small, large thread creation time is added to every operation time.

2. How do we know what the "correct" cost is?

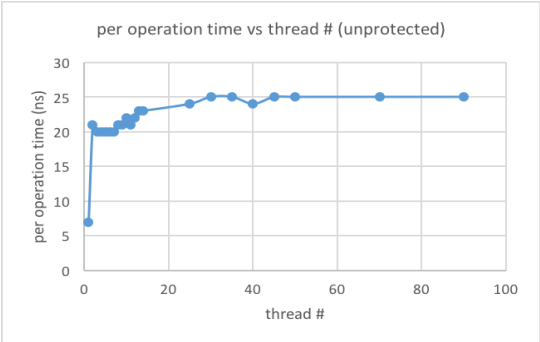
Set iteration time much larger than number of threads, so that we can eliminate the effect of thread creation time.

3. Why are the --yield runs so much slower? Where is the extra time going?

"yield" requires additional CPU time to do context switch.

4. Can we get valid timings if we are using --yield? How, or why not?

Yes we can. We can do pre-experiment to get average "yield" time and subtract "yield" time from the original per-operation time we got. But it is also an appropriation.



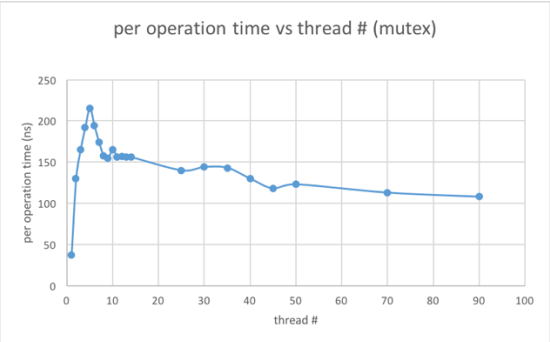
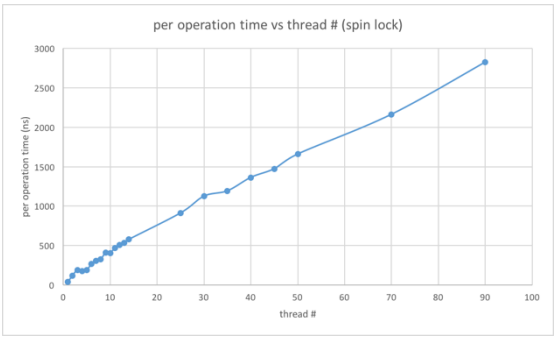


Figure 1-2

Figure 1-3



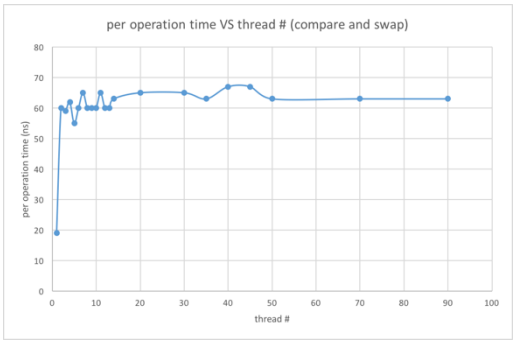


Figure 1-4

Figure 1-5

Putting them all together:

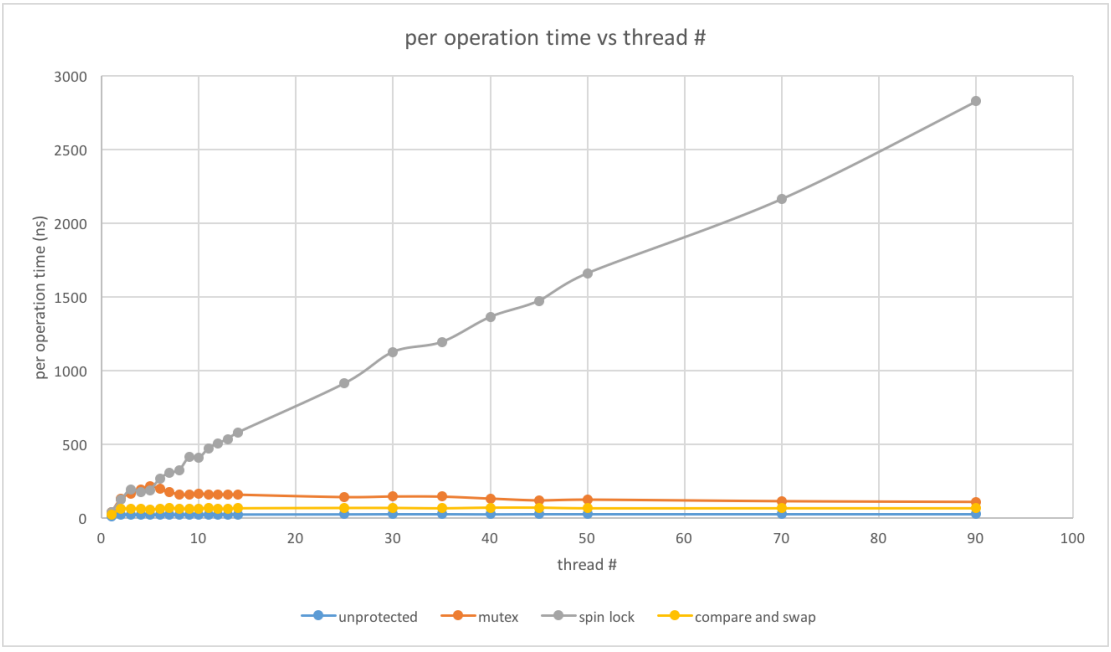


Figure 1-6 the average time per operation
vs the number of threads for all four versions
of the add function.

Questions 1.3

1. **Why do all of the options perform similarly for low numbers of threads?**

When the number of threads are small, the time spent for each thread in acquiring lock is small, because there will be lower possibility of conflict. So the differences between different options will not be obvious.

2. **Why do the three protected operations slow down as the number of threads rises?**

When there are large number of threads, the waiting time for the lock then counts a

lot for each thread.

3. Why are spin-locks so expensive for large numbers of threads?

Spin lock is implemented by busy waiting, while sync=m used mutex and sync=c used automatic remedy computation. Both sync=m and sync=c doesn't waste computation resource or wait just a little time when race occurs.

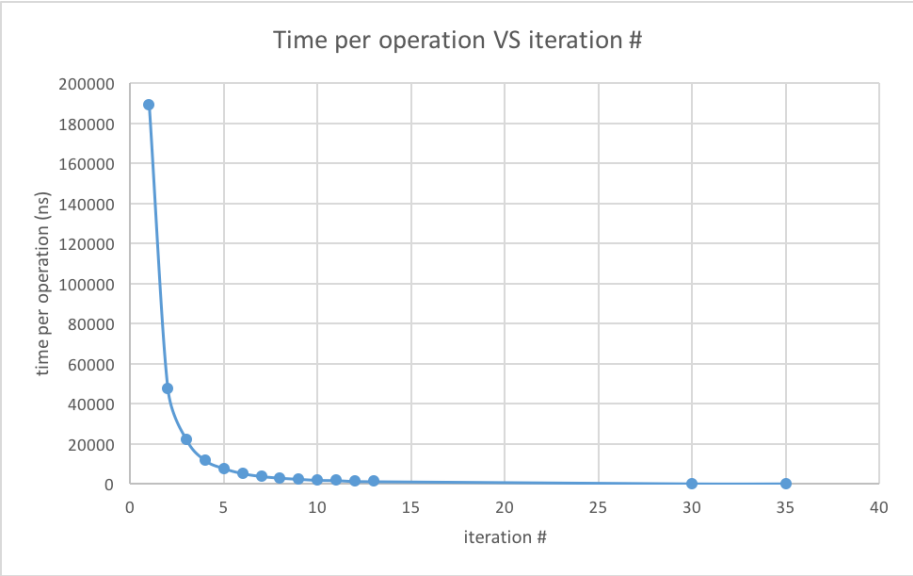


Figure 2-1 average time per unprotected operation
vs number of iteration (single thread)

Question 2.1

Explain the variation in time per operation vs the number of iterations? How would you propose to correct for this effect?

As the iteration increases, the time per operation descends greatly. This has the same reason as question 1.1. As when iteration is small, thread creation time makes most of the operation time. If we want to get data structure operation time more precisely, we should use large iteration to make up for the effect of thread creation time.

Through experiment, I found when iteration # >= 1000, per operation time will be stable. So in the following experiments, I will use iter=1000.

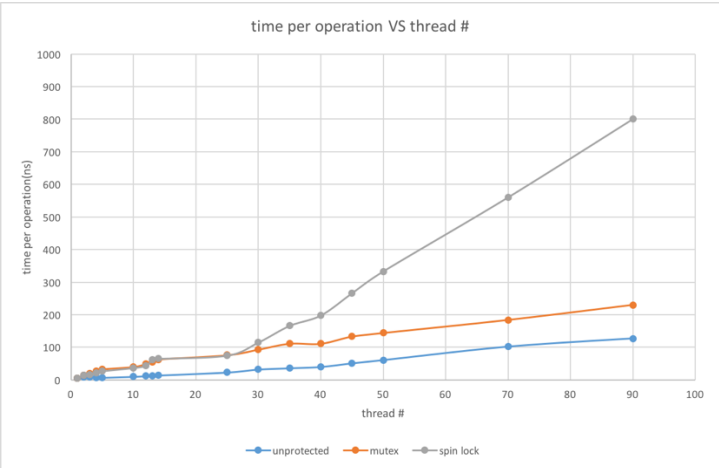


Figure 2-2 average time per operation
(for none, mutex, and spin-lock) vs number of threads.

Question 2.2

Compare the variation in time per protected operation vs the number of threads in Part 2 and in Part 1. Explain the difference.

In part 1, time per operation tends to be stable as the thread number increases. In part2, time per operation continues to grow linearly as the thread number increases. The reason is that for part 2, more threads will make linked list to be longer temporarily (longer than that is indicated by –iter). The actual length of the list will be linearly longer than that indicated by –iter. So more time will be spent to insert, lookup, delete, and calculate length, which makes per operation time longer.

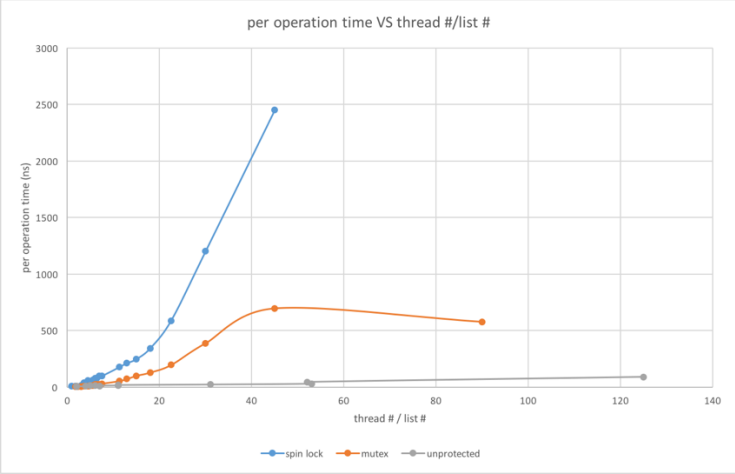


Figure 2-3 average time per operation
(for none, mutex, and spin-lock) vs the ratio of threads per list

Questions 2.3

1. **Explain the the change in performance of the synchronized methods as a function of the number of threads per list.**

For spin lock, per operation time grows quickly as threads per list increases. That's because the more threads per list, the more likely race condition will happen. Spin lock have to spend more time to "spin" until the thread acquires the lock.

For mutex, when threads per list is less than 43, per operation time grows. Then per operation time slightly decrease and tends to become stable. That's because the more threads per list, the more likely race condition will happen. More threads will try to acquire lock from mutex. This will make per operation time increase when there are still mutexes left idle. When mutexes are used up, there will be no more mutexes for thread to acquire, thus no more time will be spent.

2. **Explain why threads per list is a more interesting number than threads (for this particular measurement).**

Because lists have their own lock objects, even if number of threads are the same, per operation time will be totally different with different number of sub lists. More sub-

lists will allow better concurrency, namely, more threads, to operate effectively (potential race conditions will be less when sub-lists are more). That's why threads per list is more interesting than threads to gauge per operation time.

Question 3.1

1. **Why must the mutex be held when pthread_cond_wait is called?**

It's because we want to avoid race condition from other threads. Calling thread should have full control of its block condition to ensure deterministic program logic.

2. **Why must the mutex be released when the waiting thread is blocked?**

Because if later calling thread is put into sleep while the waiting thread is blocked, no thread will change the pthread_cond_wait's mutex reacquire condition and resume the thread, so threads will be either blocked or sleeping. The whole program will stick.

3. **Why must the mutex be reacquired when the calling thread resumes?**

It's because we want to avoid race condition from other threads. Calling thread should have full control of its block condition to ensure deterministic program logic.

4. **Why must this be done inside of pthread_cond_wait? Why can't the caller simply release the mutex before calling pthread_cond_wait?**

No. Other threads may have already changed calling thread's status between release mutex and calling thread_cond_wait.

5. **Can this be done in a user-mode implementation of pthread_cond_wait? If so, how? If it can only be implemented by a system call, explain why?**

No it can't. Because after sleep, no user-mode code can be still running in the process.