Question 1.1.2

How much more costly is a fetch&add than a write?

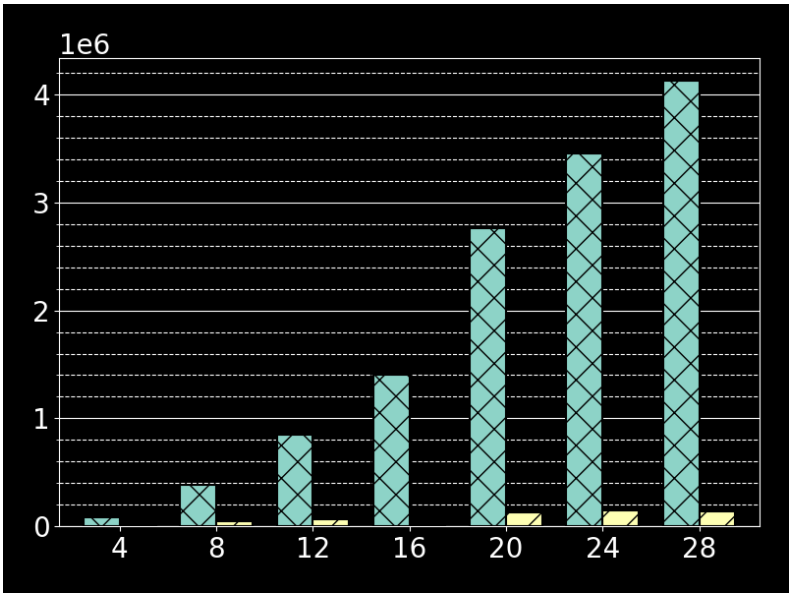
In my experiment I did 10 million iterations with fetch and add with atomic variable as well as write with -O0 optimization. For the write I decided to only write same thing to a variable in every iteration. Average time for 10 million iterations was as follows:
Fetch and Add: ~105 ms
Write: ~22.9 ms
This clearly shows that fetch and add is several about 4.6x more expensive as compared to simple write. This can be attributed due to the micro architectural differences between simple store and atomic store. In simple store the value can be cached in private cache while in an atomic store the value is always atleast written to the shared cache.

How much more costly is it for many threads to fetch&add on a single address vs fetch&add on disjoint addresses?

For this I conducted an experiment where multiple threads (upto 28) were writing to single address vs multiple addresses with atomic fetch and add. I did not use padding for disjoint addresses. I computed the average latency for both the single address FAA as well as disjoint address FAA.

Overall the observations were recorded in the graph below where the blue bars are the disjoint FAA vs the yellow bars are the single address FAA. On average the disjoint FAA showed several times better performance.

Performance gain increased from 3.6 to 30 when going from 4 to 28 threads.



Question 1.2.2
how does this improve scalability relative to the single fetch&add counter? What parameters affect the performance of this counter (and how impactful are they)?

For the approximate counters their performance is better even for single thread as compared to FAA counter. But difference is exponential in terms of scalability. While the performance of FAA counter decreases when the number of threads increase the performance of approximate counter increases.

For 4 threads the approximate counter counted 5.6 billion while the FAA counter only counted 60 million for a fixed test. For single thread the approximate counter counted 1.5 billion vs 148 million for FAA counter.

For the approximate counter, the gain is not linear especially when the number of threads is increased. This can be explained due some variables still being shared (Global count). While the FAA counter does not show any error, the approximate counter has error in its final value which can increase as square of number of threads: cn^2 where n is the number of threads.

Question 1.3.2: Sharded Locked Counter

For the sharded locked counter with the implementation as given in the image on the right, we can see following:

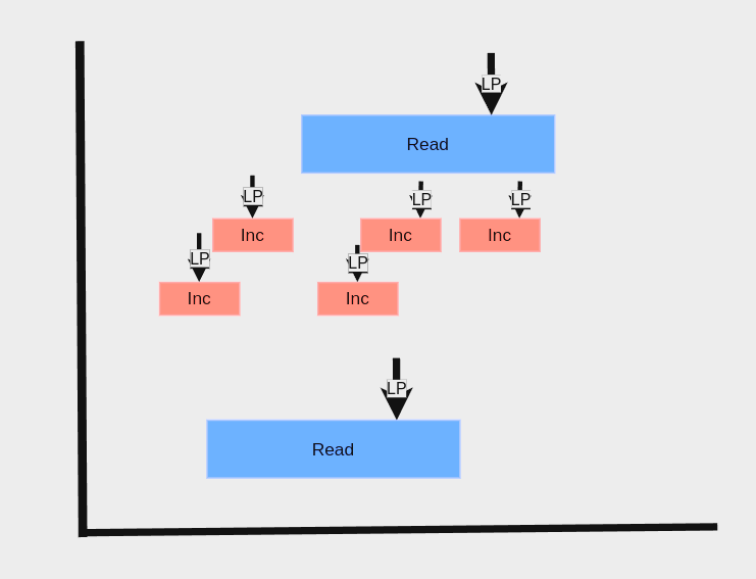
```
132     int64_t inc(int tid) {
133         pthread_spin_lock(&sh[tid].plock);
134         sh[tid].data++;
135         pthread_spin_unlock(&sh[tid].plock);
136         return 0;
137     }
138     int64_t read() {
139         int64_t thread_local sum=0;
140         for(int i=0; i<__numThreads; i++){
141             pthread_spin_lock(&sh[i].plock);
142             sum+= sh[i].data;
143         }
144         for(int i=0; i<__numThreads; i++){
145             pthread_spin_unlock(&sh[i].plock);
146         }
147         return sum;
148     }
```

For Increment:

1. Every thread has its own lock
2. Every thread has its own data variable
3. Data variable is only being written once
4. Lock and data is being written only by one thread, but can be read by multiple threads

For Read:

1. It does not have any writes to shared data
2. Read is performed only after the locks from every thread are acquired.



Following combination of events can occur:

1. **Multiple threads are incrementing simultaneously:** since there is no shared data, we can choose any point to be linearization point, as every choice of linearization point will lead to valid sequential execution.
2. **Multiple threads are reading simultaneously:** since there is no shared data, we can choose any linearization point, as every choice of linearization point will lead to valid sequential execution.
3. **Increments are happening simultaneously with Read:** For this case we cannot simply choose any operation as linearization point, since increment changes data variable which is shared with read(). And the read of data variable in read() function, is not atmoic for multiple threads i.e: it spans over a loop where multiple threads are being read.

Linearization point for Inc: since there is only one operation that is changing state of syste: we can call the increment operation as the Linearization point.

Linearization Point for Read(): since read is spanned in a for loop, it is a non atomic, multi-instruction read of shared variables from the system. Before unlocking the spinlock, we have a for loop that accesses the shared data variable to add it into the sum. This shared variable is accessed only after lock is acquired for that thread. T**hat means, after the lock is acquired for a specific thread, that thread will not be able to issue its increment until the unlock happens.**

Hence, no thread can update its counter once it has been read by the read function. Once all the locks have been acquired, no thread will be able to issue a increment. **Hence the acquisition of final lock is the point which will record all the increments happened before that point**. Therefore, it will always lead to a valid sequential execution, as long as all the increments are scheduled before the final acquisition of lock. Thus, this is a valid linearization point and the argument holds for all possible executions.

Question 1.3.3 Sharded Wait Free Counter

```
int64_t inc(int tid) {
    sh[tid].data.fetch_add(1, std::memory_order_relaxed);
    return 0;
}
int64_t read() {
    int64_t sum=0;
    for (int i=0; i<__numThreads; ++i){
        sum+=sh[i].data.load(std::memory_order_relaxed);
    }
    return sum;
}
```

For this proof, I am using some pretext from previous question. For the sharded wait free counter, as previously, we can see the following:
1. The increment function only writes to its own data variable in atomic manner. Hence this is the only instruction and this can be the linearization point.

2. For the read function, we have a loop going over atomic reads from the data variables of each thread and addition with sum variable. This is a non-atomic operation hence multiple increments can be scheduled while the for loop is going on.

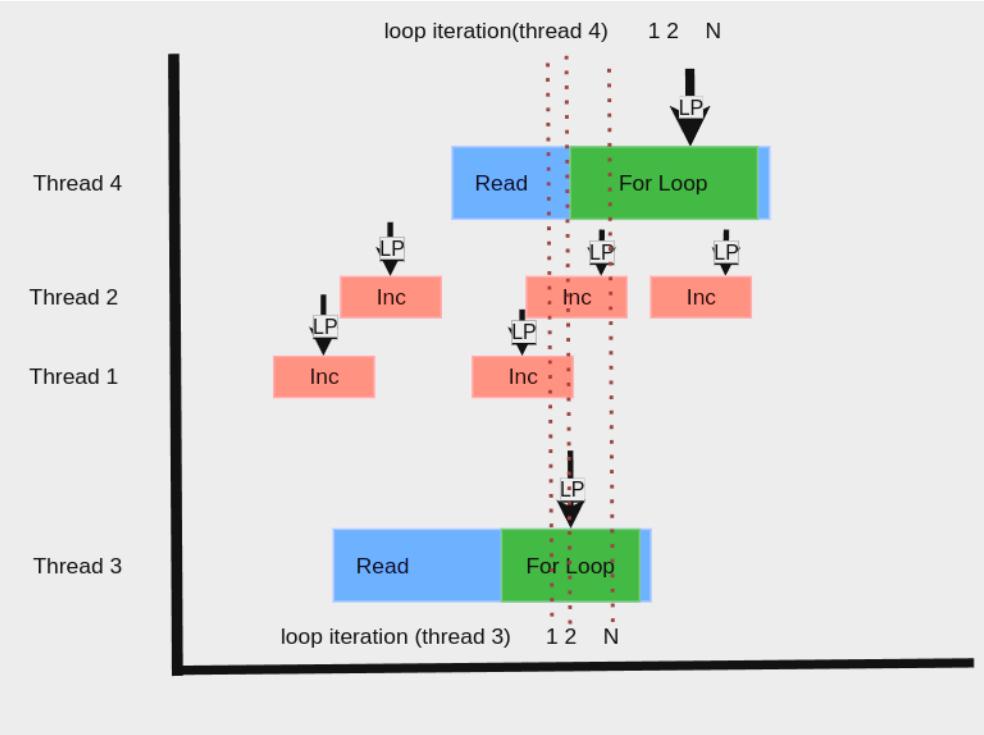
Let's analyze this problem in the same way as before. We can have the following possible parallel scenarios:

1. Increments is happening simultaneously with other increments: This would not pose a problem, since there is no data sharing among threads. Every thread has its own data variable that it increments atomically. All possible parallel executions would lead to valid sequential executions.
Hence in this case we can choose any linearization point.

2. Read is happening simultaneously with other reads: Reads do not modify any system data, hence the reads cannot cause any contention with each other. Hence all possible executions of parallel reads will lead to valid sequential execution.

3. Increment is happening simultaneously with Read: For this case we can easily see, that choosing a wrong linearization point can lead to invalid sequential executions. For example, let's say for loop is going from 1 to 10 for 10 threads, let's say that we choose the fifth iteration of for loop as the linearization point, some executions can lead to invalid sequential behaviour. If for example, thread 3 increments its counter when the for loop is at 1, the final sum will include this increment. On the contrary, if thread 2 increments when the loop is at 3, this increment will not be included in the final sum. Hence two these two parallel executions will not lead to valid sequential execution. **Hence we can see, for certain parallel executions, certain choices of linearization points will yield the wrong results, while certain choices can yield valid sequential execution.**

The following diagram can better help us to see which executions can yield valid sequential consistency, and if certain choices of linearization points can actually lead to sequential consistency for all possible executions.



Based on the above diagram and example we can see that for our code, **loop going 1 to N number of threads, if a linearization point is chosen at k iteration of loop going from 1 to N such that there is no increment of thread from k to N after this iteration, this would be a valid choice of linearization point leading to sequential consistency for all possible executions.**
Thus we can see, as long as the linearization point is chosen at k iteration, no numerically marked thread from k to N total threads calls its increment, it will be a valid point. Since there are only a maximum of N threads, it can be seen that the linearization point must exist between 0 and Nth iteration of the loop.