CMSC 23010: Homework 3b Design Documentation

Will Thomas

November 17, 2022

Introduction

The goal of the following design documentation is to outline a cohesive plan for the design, correctness testing, and performance analysis of a program that will simulate packet checksums with numerous packet sources and numerous workers using explicit parallelism. Two different versions of the code will be implemented; a SERIALPACKET version which, much like HW2, will compute one checksum per packetsource serially, as well as a PARALLELPACKET implementation with three associated subversion: lockfree, homequeue, and awesome.

Design

Serial

The SERIALPACKET function will work exactly as it did in HW2. A single thread will request a packet from a certain source, compute a checksum, and then repeat with the next packet source until the appropriate number of packets have been checksummed. In this case, it will simply continue this process until the allotted time runs out.

Parallel

LOCKFREE:

The lockfree version of the parallel code will work exactly as the parallel version of HW2. Essentially, a dispatch thread will enter an infinite loop (really lasts as long as M) and, iterating through the queues, enqueue a packet in each (from the corresponding source). If the queue is full, the dispatcher will wait until there is room.

HOMEQUEUE:

The homequeue version is exactly the same as the lockfree version except that the associated worker for each queue must first obtain a lock before computing each checksum and relinquish after each computation.

AWESOME:

The awesome version will be different in that worker threads are no longer bound to specific queues; if a certain queue is full and needs help, free workers (workers with empty queues) can go and help out. My idea for the implementation of this is as follows:

Seeing as we have designed this program such that whenever the dispatch thread attempts to enqueue a packet to a full queue it is blocked and must wait for an opening in the queue, we can have a shared lamport queue pointer among all threads. Whenever the dispatch thread is forced to wait, it can set the value of this shared pointer to that of the queue that is full. Meanwhile, on the worker side, whenever a worker attempts to dequeue from its main queue and finds that it is empty, it should check to see if the shared pointer is NULL. If it is not NULL, it should attempt to obtain the lock to that queue and dequeue a single packet, after the computation of which the thread returns to its own main queue so as not to cause a packet buildup. NOTE: the lock to a certain queue should be released as soon as a packet has been dequeued. To ensure

that the dispatcher thread is not held up for too long, as soon as it is able to enqueue a packet into the once full queue it should set the pointer value to NULL (meaning no queue needs help) and continue on with its usual function.

This strategy should work for all packet types, but the most advantage should come from the uniform and especially from the exponential packet types.

Correctness Testing

Seeing as we were able to show that our locks work in the last assignment as well as that our queues work in HW2, we should only do minimal testing for SERIALPACKET as well as PARALLELPACKET with LOCKFREE version. As far as the other two versions of the PARALLELPACKET implementation, we can test HOMEQUEUE by checking that, in the case of one worker thread, every checksum is computed in the same order as it would have been in the LOCKFREE version. This test generalizes to multiple threads because they are all still single reader / single writer. We can test AWESOME by checking that every checksum is computed. That is, if we output every checksum (and each packet is generated with the same random seed), we should be able to check the output with the output to SERIALPACKET and they should contain the same checksums, although not necessarily in the same order.

As such, we can state our main invariants as follows:

1. No packets are dropped; all checksums are computed.
2. Queues work in a FIFO fashion (already established by HW2).
3. No more than one thread holds a certain lock at the same time (already established by HW3a).

Performance Hypothesis and Testing Plan

NOTE: All UNIFORM and CONSTANT experiments will be run 5 times, while EXPONENTIAL experiments will be run 11 times.

LOCKS USED: Although I do not think that it would be incorrect to say that Mutex and ALock performed the best out of the locks tested, I believe it would be a mistake to use ALock in this context. In the case of low contention, ALock had by far the worst initial performance plummet. Seeing as contention for locks should be relatively low given how AWESOME is designed (I do not expect there to be many threads that are free to help the full queue, implying low contention), I believe that I will implement TTAS and Mutex. I think TTAS will work sufficiently here given that, once again, contention should be quite low, and because the overhead is considerably smaller than that of ALock.

1. Idle Lock Overhead

   I expect that HOMEQUEUE will run about 17% as fast as LOCKFREE when using TTAS and about 14% as fast as LOCKFREE when using Mutex, as this was the slowdown seen for overhead in HW3b. I expect that these proportions will stay constant regardless of worker rate seeing as a lock must be acquired and released each time a checksum is computed.

2. Speedup with Uniform Load

   I hypothesize that as the value of W grows larger, the speedup of LOCKFREE relative to SERIALPACKET will start greater than 1 but quickly overtake it, seeing as the parallel overhead inherent in LOCKFREE will get smaller and smaller in comparison to the overall runtime and the benefits of parallelism in LOCKFREE will cause it to take over SERIALPACKET. Given that we have no real loadbalancing in effect, I suspect that HOMEQUEUE will continue being significantly slower than SERIALPACKET due to the cost of lock acquisitions and releases, although I also believe that as W grows large, the slowdown between HOMEQUEUE and SERIALPACKET will become smaller, although I am not sure it will ever overtake SERIALPACKET.

3. Speedup with Exponential Load

   I expect similar results as Uniform Load speedup except for a general smaller spread to the data. That is, because the work for a single packet has the capability of being so large, the parallel overhead in LOCKFREE and lock overhead in HOMEQUEUE will become increasingly irrelevant in comparison, although I still expect HOMEQUEUE to perform worse than LOCKFREE due to lock overhead.

4. Speedup with Awesome

   I expect that Awesome will be able to handily outperform the other strategies in cases of Exponential Packets with a large mean work time. This is because my strategy makes it so that the dispatcher does not get hung up waiting for a spot in a queue to open up, and not because it "skips" over that queue, but because available workers come to its aid. The design here will be between LOCKFREE and Awesome with both lock varieties, and values of $n \in \{1, 2, 3, 7, 13, 27\}$. W will be 8000 for all tests. I expect Awesome to overtake lockfree after $n = 7$.