Design Document – Load Balancing Packet Distribution

1 Design Document

1.1 Modules

There will likely be four modules, to start with at the very least. One of the modules will serve as the entry-point for running the program, and will also contain the core of the program. Code for the serial and parallel packet distribution systems will be included here, as will code for worker thread routines per strategy. A second module will house all lock-related code, so that the main module can easily instantiate and interact with instances of each lock by referencing this module. Another module will be used for running unit tests as well as the set of four experiments. The fourth module will contain auxiliary code to be used by the other modules (e.g. bit-wise modulo for queues, calculating medians for experiment trials, etc.). The main module will consist of `executeSerial` and `executeParallel` functions, and one worker routine function per queue selection strategy. When the dispatcher is spawning worker threads in `executeParallel`, it will specify which routine it ought to carry out based on the input value S. Prior to the creation of worker threads, a global array of locks that maps to the array of Lamport queues will be initialized in cases where the strategy being used requires locks.

1.2 Locks

I will use L = { TASLock, ALock } in my experiments. I decided to use these two as my set of locks based on my observation that they exhibited the most stark dissimilarity in terms of performance (compared to other pairing of locks from 3a). Whereas the array-based Anderson lock scales relatively well via minimizing contention by having each thread spin locally, the TAS spinlock performs poorly in this regard, since it uses a single shared memory location for synchronization and can cause problematic bus traffic and cache invalidation under high contention. Only in cases of low contention did the TAS lock outperform the Anderson lock; past a certain number of threads, the Anderson lock maintained similar execution times for ever increasing amounts of contention, while the TAS lock's run-time steadily increased with contention. I am curious to see how different types of load distribution will impact this pattern, as well as the effects of the different queue acquisition strategies.

1.3 Awesome Strategy

I will improve performance by implementing a work-stealing scheduling strategy, so as to minimize worker idle time. Each worker will be assigned to its own Lamport queue and in theory we will want them to begin work on the packets they receive from their own queue. If a worker runs out of packets to process, it will randomly select another worker and "steal" a packet from their queue. If this fails, the worker will first check its own queue for newly added packets before proceeding to

repeat the stealing process. The conceivable performance benefits are incurred when there is enough variance between the workloads handled by worker threads such that a considerable portion of some threads' lifespan is spent waiting for more work to process; depending on the mean amount of work, idle time could potentially outweigh time spent computing checksums, which would be especially detrimental to running times of the program as a whole. Assuming that each worker can expect to receive the same number of packets, a simple termination condition would be for the worker to run so long as it has not yet processed the number of packets per source, T. However, this strategy can make further use of the work-stealing approach by allowing workers who have completed processing their own designated stream of T packets to steal from slower workers that are still active. If there are outlier worker threads (i.e. threads that take an exceptionally long time to process T packets relative to the other workers), this functionality would help to target the "weakest link" among the workers and thereby improve overall performance. In this case, the threads will only once all threads have exhausted their packet streams; all workers should terminate at roughly the same time, then. A simple implementation of this would be to keep a global counter that either keeps track of the number of remaining packets, or the number of packets already processed – both to the same effect. However, this plan may backfire in cases where there are more threads than available cores, since this would unnecessarily increase contention for locks guarding queues where there are still packets to be processed. To address this, we can have workers yield immediately prior to attempting to steal a packet, allowing descheduled workers to acquire a processor and make progress on their own queue. When there are fewer threads than cores, yielding should have no effect and hence not negatively impact performance. One aspect of my strategy I will likely need to tweak after running the other experiments/strategies is observing the "buffer" time between a thread being spawned and it receiving its first packet. This is an area of concern, since workers may start unnecessarily start attempting to steal packets from other queues in the midst of later workers still being spawned. In particular, I expect that measuring worker and dispatcher rates will help with addressing this potential issue.

I expect the benefits of my strategy to be more apparent when work distribution varies greatly from packet to packet – and hence from thread to thread – which would apply more so to exponentially distributed packets than uniformly distributed packets. This is because the greater the variance in workload between threads, the greater the difference is in their run-times, which consequently results in greater idle time to be capitalized by my strategy. This is not to say that an experiment with uniformly distributed packets will not benefit from this strategy, but my aim in implementing this strategy is **to address the disparities in thread run-time emblematic of exponentially distributed work.**

2 Test Plan

2.1 Correctness

I plan on re-purposing some of my unit tests from prior assignments, since 3b is an amalgamation of 2 and 3a. My unit test from 2 that involves comparing a series of enqueued and dequeued items for validating the correctness of my Lamport queue will be used, as will tests confirming that the parallel strategies all work by comparing their checksum results with that of the serial version of the code (the test of which should be trivial). Drawing from 3a, there will be unit tests to confirm that

the locks being used satisfy mutual exclusion: a counter test which involves increment a lock-protected counter using multiple threads and confirming that the counter reaches the desired amount, a test that checks whether a lock-protected counter is incremented by threads contiguously by observing the value of the counter after each increment and making sure that the values monotonically increase, etc. For the Anderson lock, there will be a test that determines whether the lock provides an ordering guarantee by comparing the number of increments across threads. Though there is no quantitative threshold at which we can say the test passes or fails, we can compare the increment variance among threads when using the Anderson lock against the increment variance exhibited when using a TAS lock, and make a reasoned conclusion based on the difference.

2.2 Performance/Hypotheses

For the **Idle Lock Overhead** experiment, I hypothesize that regardless of the values of W, the S = LockFree experiments will have the greatest performance (i.e. least overhead), followed by S = HomeQueue using L = TASLock, and then finally S = HomeQueue and L = ALock. It is obvious that the program without locks should not incur any overhead associated with lock usage, and therefore avoid the time penalty associated with using a lock. I would think that given the greater complexity of interfacing with an Anderson lock relative to a TAS lock, an uncontended path through the former would impose a greater time penalty, and thus provide worse performance than the latter. As W increases, checksum computation time will increase by roughly the same amount in both serial and parallel versions of the program, so the discrepancy in performance between the Anderson lock and TAS lock should not change considerably based on W, since there is no contention and hence will depend solely on the acquisition time of the locks (previously discussed).

For the **Speedup with Uniform Load** experiment, I expect performance for the lockFree strategy to discernibly taper off once the number of threads exceeds the number of available cores since some threads will necessarily have to wait for access to a processor and burn time, as I observed in assignment 2. For the homeQueue strategy, I expect the TAS lock to outperform the Anderson lock for all values of W, but especially for smaller W since then the checksum calculation work should not be considerably greater than the work spent retrieving a packet – this places more emphasis on lock acquisition, which the TAS lock is faster than the Anderson lock at under low contention. This is because each worker is strictly responsible for retrieving packets from its own queue, so no lock will ever experience contention, and thus the performance benefits of the Anderson lock in light of contention will not be taken advantage of. As W increases, I expect to see increasing speedup (if n > 1), as the program is able to take advantage of a multi-core system for a longer period of time (since a bigger W value implies longer computation times). However, I do expect performance to drop for the homeQueue strategy when *threads* > *cores* (as with the lockFree strategy mentioned before) since the benefits of this advantage can no longer be reaped.

For the **Speedup with Exponential Load** experiment, I anticipate that the different combinations of {S, L} will not scale as well compared to when using uniform loads, since there will be a greater imbalance in the amount of work across the worker threads. For instance, if the majority of worker threads have a workload lighter than the average workload, they will exhaust their packet streams well before the thread tasked with the heaviest workload. Despite finishing early, the program's run-

time hinges on *all* threads computing their respective checksums, so the single slowest worker can negatively skew performance. I think this effect will be exacerbated for larger values of W, since this leads to a greater spread of work between workers. I expect to see similar behavior with exponential loads as with uniform loads in all other respects (e.g. expected differences between the {S, L} combinations, expected behavior with varying n, etc.).

For the **Speedup with Awesome** experiment, my hypotheses are that there will be a positive correlation between speedup and the value of W, as well as between speedup and the number of workers. This is because a greater W value results in a greater imbalance of work between threads, which is capitalized by the work-stealing strategy. So long as *threads ≤ cores*, a greater number of threads should improve speedup too. Between the two locks, I expect the Anderson lock to outperform the TAS lock, especially for larger values of n, since the Anderson lock minimizes (the overhead) of contention, whereas the TAS does not, thus causing lag. If n is small enough, however, the TAS lock may fare better than the Anderson lock due to low contention and the relatively smaller overhead of operating the TAS lock itself.