Design Document

Alex Miller

May 29, 2021

Modules, data structures, interfaces, invariants:

Our code base will consist of the following modules:

- The highest ranked modules containing C code will be two files – one called serial.c, the other parallel.c. These modules will, respectively, implement serial and parallel queue performance tests.

  serial.c

  - This file will implement the functionality of serial packet and checksum generator performance test, configurable to the following variables:
    - M - an integer representing the time in milliseconds that the experiment should run.
    - n - an integer representing the total number of sources
    - W - an integer representing the expected amount of work per packet
    - U - an char representing a flag specifying what sort of packet distribution to utilize in our tests. t indicates the use of Uniformly Distributed Packets , while f Exponentially Distributed Packets.
    - s - a non-negative integer, for seeding the packet generator. In practice this will correspond to trial numbers when testing is run.
  - Using user input, this file will create the specified packet generator. Call this generator *packet_gen*. It has n sources, a median work value of W, and utilizes the seed s.
  - It will also bind the specified method of extracting packets from *packet_gen* to a pointer desginated *packet_method* – the options for packet generators are Uniform and Exponential.
  - It will then pass *packet_gen*, *packet_method*, as well as any necessary variables to chksum_serial(); this method generates checksums from *packet_gen* from n sources using a average packet size of W and *packet_method* for M milliseconds. This method returns an integer, T – it first describes the number of packets that get fully processed in the allotted time. I will discuss the specifics of this method's implementation below.
  - serial.c will finally output T to the terminal, before cleaning up any leftover resources and exiting.

  parallel.c

  - This file will implement the functionality of parallel packet and checksum generator performance test, configurable to the following variables:
    - Like serial.c, this file is configurable to M, n, W, U, and s.
    - D - an integer representing the depths of the queues to be used in this test
    - L - a char representing the lock type to be used in our load balancing implementations. Setting this character to t, p, a, and m will direct worker threads to use either our Test and Set, the pthread Mutex, our Anderson's Array, or our MCS lock, respectively, in order to dequeue. The specifics of these lock implementations are discussed below.
    - S - a char representing a load balancing strategy. Setting this character to L, H, and A will direct our performance test to utilize either a LockFree, HomeQueue, or Awesome load balancing strategy, respectively. The specifics of these different strategies are discussed below.

- Using user input, this file will create the specified packet generator. Call this generator *packet_gen*. It has n sources, a median work value of W, and utilizes the seed s.
- It will also bind the specified method of extracting packets from *packet_gen* to a pointer desginated *packet_method*.
- parallel.c will then pass *packet_gen*, *packet_method*, *lock*, D, L, S, as well as any necessary variables to chksum_parallel(); this method generates checksums from *packet_gen* from n sources using a average packet size of W and *packet_method* for M milliseconds. This method differs from chksum_serial() in that it utilizes n additional worker threads in order to generate checksums, utilizing the load balancing technique specified by S. This method returns an integer, T – it first describes the number of packets that get fully processed in the allotted time. I will discuss the specifics of this method's implementation below.
- parallel.c will finally output T to the terminal, before cleaning up any leftover resources and exiting.

- The next level of modules will implement the functionalities of chksum_serial() and chksum_parallel(). As such it will be comprised of three files – chksum.c, queue.c, and lock.c

  chksum.c

  - This module will describe chksum_serial() and chksum_parallel, the methods that will be used by serial.c and parallel.c.

    long chksum_serial()
    Input:
    * PackeSource_t packet_gen : a pointer to a packet source
    * volatile Packet_t * (*packet_method)(PacketSource_t *, int) : a pointer to a method for extracting packets from said source
    * int n : the number of sources that packets are being generated from
    * int M : the number of milliseconds this test should run for

    This repeatedly simply loops through n sources within a non-terminating while loop, utilizing *packet_method* to extract new packets from *packet_gen*. Before this loop begins, a timer is started. At the top of the loop, if this timer exceeds M milliseconds, the main thread returns. Otherwise, the loop continues to generate checksums for source $i \in [n]$. Each time the method creates and processes a packet, a counter is incremented, who's final value is returned by this method once it returns from the loop.

    long chksum_parallel()
    Input:
    * Like chksum_serial, this method is configurable by *packet_gen*, *packet_method*, n, and M
    * int D : an integer representing the depths of the queues to be used in this test
    * L : a char representing the lock type to be used in our load balancing implementations.
    * S : a char representing a load balancing strategy.

    This method is responsible for initializing the data structures needed for the functioning of n worker threads calculating checksums of packets generated by *packet_gen*, extracted by *packet_method*, and placed in FIFO Lamport queues.
    It will call create_queue_pool(n, D, L) in order to allocate space for *packet_queue_t* array of size n, each with depth D, each associated with a lock of type L. Call this array Q. Q represents a pool of Lamport queues.
    Following the initialization of these data structures, this method binds the pointer:
    * void * (*worker_method)(void *)

    to one of three methods describing the following functionalities depending on the specification of S. If S is:
    * L, then threads should utilize a Lock-Free strategy. This strategy dictates 1 - 1 correspondence between queues and threads, as specified in previous assignment:

- Within an infinite while loop, a thread attempts to dequeue from its associated queue within another while loop.
- If it succeeds, the thread generates a checksum for the dequeued packet and frees the packet. Then, the thread checks if the queue has signaled the thread to close; if not, it increments the queue's through_count variable and returns to the top of the outer loop. Otherwise the thread just returns.
- Otherwise, it checks if the queue has signaled the closing of the thread. If such a close has been signaled, the thread returns. Otherwise it yields and returns to the top of the inner while loop.

The design implies the following invariants:

- Threads dequeue and process packets in the order they are generated – each i-th packet placed in a queue has to be dequeued and processed before the i+1-th packet placed in said queue
- It is not an invariant that threads completely dequeue every packet in a completed queue – the queue may signal a close even if it still contains pointers to packets.
- No packet gets processed after its queue has signaled its closing.

* H, then threads should utilizes the Home-Queue strategy. This is like the Lock-Free strategy, with the main difference being that, while the Lock-Free strategy does not utilize locking, the Home-Queue threads utilize a set n locks, each of which are associated in a 1 - 1 correspondence with the n queues in Q. Each thread using this strategy locks their queue's lock before calling its dequeue() method, and unlocks it after completing said call. In our case, that would mean encapsulating the while loop in which each worker attempts to dequeue from its queueu.
This design implies the same invariants as the LockFree strategy.
* A, then the threads should utilize my Awesome strategy. Unlike the previous two strategies, the Awesome strategy does not maintain a 1 - 1 correspondence between worker threads and queues. Rather, threads cycle through queues, attempting to dequeue packets. It does maintain a 1 - 1 correspondence between queues and locks. Once a worker thread comes to a queue it:
  - Within an infinite while loop, the thread attempts to acquire a queue's associated lock with it's *trylock*() method.
  - If it succeeds, the thread waits to enter the critical section:
    CS START
  - The thread attempts to dequeue from the queue.
  - If it succeeds, it first unlocks the queue's lock. The thread then generates a checksum for the dequeued packet and frees the packet. Then, the thread checks if the queue has signaled the thread to close; if not, it atomically increments the queue's through_count variable and returns to the top of the outer loop. Otherwise the thread just returns.
  - Otherwise, it checks if the queue has signaled the closing of the thread. If such a close has been signaled, the thread unlocks the queue's lock and returns. Otherwise it moves on to the next queue.
    CS END
  - Otherwise, if the lock is busy, the thread moves onto the next queue.
  - Queues are indexed by a thread starting with the one it is passed as an argument (discussed later), and using a circular index counter.

  This design is favorable because it allows worker threads to search for work among the allotted queues. This should increase throughput by mitigating or otherwise smoothing over divergences in packet size between queues as they continue to take in packets from different sources – this should improve performance over our non load balancing implementation from HW2 when using an Exponential packet generator. Furthermore, the use of *trylock*() ensures that no thread gets stuck waiting on a queue that otherwise might be busy being dequeued by another thread, keeping lock contention minimal and keeping threads from piling up on a queue.
  The design implies the following invariants:
  - All threads eventually exit after every queue has signaled its closing

- It is not an invariant that threads completely dequeue every packet in a completed queue – the queue may signal a close even if it still contains pointers to packets.
- No packet gets processed after its queue has signaled its closing.
- For a given queue, worker threads dequeue in the order they acquire the queue's associated lock.
- It is not an invariant that packets get processed in the order they are placed into the queue.

These functionalities will be implemented in $void *L\_worker(void *args)$, $void *H\_worker(void *args)$, $void *A\_worker(void *args)$, respectively. These are the methods that can be bound to *worker_method*.
Each of these functions will take in a pointer to a *packet_queue_t* struct as an argument. This type will be described soon.

The thread this method is called in is the dispatcher thread – this thread starts n worker threads, which will be responsible for calculating checksums of packets generated by *packet_gen*, extracted by *packet_method*, and placed in Q. Each thread will generate checksums in the manner proscribed by *worker_method*. In order to do this, the dispatcher thread, $\forall q \in [Q]$, spawns a thread to call *worker_method* with $\&Q[q]$ as an argument.

After chksum_parallel() has initialized and spawned its worker threads and data structures, it carries out a performance test:

* After starting a timer, it repeatedly cycles through the n queues in Q, each associated with one of n packet sources.
* One queue at a time, the dispatcher thread will generate the queue's next packet, wait until the queue contains space for the packet, write to the queue, and then move on to the next queue.
* It will repeat this process so long as the timer has been running for less than M milliseconds.
* Once the timer is up, the dispatcher threads signals all threads to stop by setting the queue pool's shared *done* boolean to $True$, joins threads, and adds up the counters stored in the *through_count* attributes of the queues in Q. Call this value T. This value represents the combined total throughput of all threads over the course of the test.

This method terminates by cleaning up the resources initialized above, freeing packets remaining in the queue pool, and returning the value stored in C.
As such, this method describes a neutral interface with which to test the throughput of a dispatcher distributing work among n worker threads, which can be configured to utilize varying and programmable locking algorithms and load balancing strategies.
The design implies the following invariants:

* The dispatcher enqueues the i-th packet of source k into queue k before it enqueues the $i+1$-th packet
* The dispatcher does not signal one thread to stop before it signals another thread to stop
* No packet gets processed after its queue has signaled its closing.

queue.c

– This module will describe the construction and functionality of *packet_queue_t* struct. This struct implements a FIFO Lamport queue. It's declared with the following struct members and methods:

* int i : each queue contains an integer describing its index within a queue pool
* int D : an integer describing the depth of the queue
* volatile int head, tail : Volatile integers specifying the locations of the head and tail of the queue
* Packet_t packets : a Lamport queue with depth D
* volatile bool *done : a pointer to a volatile boolean designating whether or threads should keep reading from queues. This is allocated and initially set to $false$ in *create_queue_pool*().

    - lock_t *L : a pointer to a instance of a locking algorithm.
    - int through_count : a long integer that records how many packets get processed that get passed through a queue.
  - Methods:
    - int enq(packet_queue_t *Q, volatile Packet_t x) : queue a *Packet_t* instance in the Queue Q. Return 1 if Q is full, 0 if otherwise and x was enqueued successfully.
    - volatile Packet_t *deq(packet_queue_t *Q) : dequeue the next *Packet_t* from Q. Return a NULL value if the queue is empty
    - packet_queue_t *create_queue_pool(int num_q, int D, char L) : allocate an array of *num_q* packet_queue_t structs, each of depth D. Each queue should have a lock of type L associated with it. If L does not specify a char code for a valid lock (if L is not t, p, m, or a), then no locks are allocated.
    - void destroy_queue_pool(packet_queue_t *Q) : Destroy an array of packet_queue_t structs allocated as a block queues

lock.c

- The locks mentioned above, will be implemented in a file called lock.c. The file will describe the implementation of a Test and Set lock, an Anderson's Array Lock, and an MCS lock, as specified in the textbook and translated into C. The Mutex lock will be called from the pthread library, but wrapped so as to work with our *lock_t* interface. The interface has the following attributes associated with it:
  - char type : a reference to a lock type, as specified by input
  - void *l : a pointer to a lock object; this pointer is passed to the initialization, lock, and unlock functions.
  - void (*init_thread)(void *) : This function initializes any thread specific structures required by a lock. It should be called before a thread acquires the lock the first time.
  - void (*try_lock)(void *) : a pointer to the lock's try_lock function; it throws an exception to the calling thread if the lock object is busy. Otherwise, the calling thread acquires the lock.
  - void (*lock)(void *) : a pointer to the lock's lock function; it should be called on the void *l (which points to the initialized lock) in order to acquire the lock.
  - void (*unlock)(void *) : a pointer to the lock's unlock function; it should be called on the void *l (which points to the initialized lock) in order to release the lock.
- Each locking algorithm type will have the following methods associated with it:
  - void init(int n): initialize a lock to synchronize n threads; n is only used in initializing locks that allocate space for locks prior to use, such as Anderson's lock. Otherwise, n is discarded.
  - destroy(void *lock): destroy the initialized *lock* and free any memory allocated to support it
  - void init_thread(void *lock): in the case of the MCS lock, this method needs to be called by each thread utilizing the lock before it acquires the lock for the first time.
  - int try_lock(void *lock) : try_lock *lock*
  - void lock(void *lock): lock *lock*
  - void unlock(void *lock): unlock *lock*
- This interface should be able to be initialized and freed either one at a time or as a block. As such, this module will also contain the following methods:
  - lock_t *new_lock(char type, int n) : initialize a single lock of the kind specified by *type* which will manage n threads.
  - lock_t *new_lock_pool(int size, char type, int n) : initialize a lock pool with *size* locks of the kind specified by *type*, all of which will each manage n threads. This pool is allocated as a block of contiguous memory
  - int destroy_lock(lock_t *L) : destroy a single lock_t instance
  - int destroy_lock_pool(int size, lock_t *L) : destroy a pool of contiguously allocated locks of size *size*.

- Our last module consists of testing and analysis protocols, test_script.sh and analysis.py
  - test_script.sh will consist of a shell script that utilizes serial.c, and parallel.c. This test script will execute these files and record their output, generating experimental data in an automated fashion and storing that data in csv files. This file will be used to measure the throughput of my serial and parallel implementations under a variety of conditions and through a number of trials.
  - This script cannot be used to verify the consistency across implementations as serial.c and parallel.c are not deterministic by virtue of input.
  - I will use 5 trials per test, in order to get a fair spectrum of inputs while not hogging class resources.
  - I will use $M = 2000$ ms for all tests, though it remains to be seen whether this value is sufficient. Testing is needed.
  - As to the two locking algorithms that this script will specify in order to run the performance test described in *checksum_parallel*(); I plan on utilizing the Mutex and Anderson locks when testing my implementation. As shown in experiment 2 of my hw3a handin, these were consistently the two fastest locks (under contention) of the four I implemented. This quality will be desired in cases where queues pile up on a single queue.
  - As to my testing of my Awesome strategy: As demonstrated in HW2, load balancing becomes pertinent when dealing with exponential packets. Therefore in order to test my Awesome load balancing strategy, I will redo Experiment 3: Speedup with Exponential Load, but instead of using the HomeQueue strategy I will use my new Awesome strategy.
  - analysis.py will be responsible for analyzing data, making graphs, and outputting quantitative data to compare the performance of locks.

Test plan:

I will need to test my implementations of serial and parallel.c for the following capabilities:

1. being able to generate correct checksums. In assignment 2, I demonstrated to myself the ability to generate deterministic streams of input from the packet generator interface. For that assignment, I was able to use this determinism to generate amalgamated checksum counters that I could use to assure myself that my serial implementation was consistent and that my parallelized implementations were working as they should. In other words, if my serial and parallel implementations agreed on their outputs for given values of N, W, s, and T, then if I was sure that my serialized implementation was working, I could be assured my parallelized implementation was working.
   For this assignment however, as there are no guarantees on how many packets ought to be processed across different implementations, this method of implementation validation won't be valid. Rather, prior to experimental testing, I will have to assure myself that my serial and parallel implementations agree on output for a set number of packets. This can be done by creating test instances of *chksum_serial*() and *chksum_parallel*() which, rather than generating packets for M milliseconds, generate some value T packets from N sources, before terminating. This should replicate the deterministic and consistent outputs of serial.c and parallel.c from HW2.

2. being able to use my timing and signaling system to promptly exit from loops and terminate performance tests. This is most salient when considering the operation of our parallel implementation, which uses inter-process signaling in order to inform worker threads to exit. This leaves the possibility open that a worker thread processes packets after a term of M milliseconds – this would affect our experimental measures of throughput. In order to measure and account for this effect, I will need to see if signaling the closing of threads without stopping the dispatching of packets has an affect on the output of parallelized implementations.

3. being able to rely on lock.c and queue.c: as these implementations were tested in previous assignments and remain, by and large, unchanged, I do not plan on testing their correctness.

Expected Performance (Performance Hypotheses):

Idle Lock Overhead:

The purpose of this test is to measure the overhead of using locks in the HomeQueue strategy in comparison to the LockFree strategy. Between these two strategies, we should expect the HomeQueue strategy to entail greater overhead in its use of a lock around the worker thread's invocation of *dequeue*(). This follows from our observations in HW3a – in our counter tests, for $n = 1$, both the Mutex lock and Anderson's lock displayed 10x slowdown over ideal, LockFree performance.
However, for larger values of W, we should expect the relative overhead of HomeQueue over LockFree to decrease. This follows from our observations in HW2 – larger values of W entailed longer packet processing times, which gave more opportunities for parallelism and speedup. In this case, larger values of W will otherwise outweigh the increased overhead entailed in acquiring locks, reducing the relative overhead of HomeQueue over LockFree.
As to the relative performance of these tests across the use of our different locks: both the Mutex and our Anderson locks showed similar performance for use with a single acquiring thread. Therefore we should expect similar curves from our uses of these locks.

Speedup with Uniform Load:

Larger values of W imply more opportunity to parallelize the computation of checksums, for a given value of n. Therefore, while using both our LockFree and HomeQueue strategies, we should expect increased speedup of parallel.c over serial.c for greater values of W. The same might be said for larger values of n given a fixed value for W, in the sense that a greater number of worker threads might be able to work together more efficiently than a single thread. However, larger values of n also imply larger amounts of overhead in managing threads and queues. Furthermore, the once n exceeds the number of cores, we can expect to see drop offs in speedup, as observed in our results for HW2.

Therefore, in comparing the observed speedup for different values of W, we should expect increased speedup for larger values of W. For a given value of W, we should expect to see greater speedup for greater values of n – as observed in the results for HW2, we can expect speedup to appear linear large enough values of W and $n \leq 13$. However, this is accompanied by diminishing returns on speedup for larger values of n, especially $n > 13$.

Between load balancing strategies, we should expect to see consistently faster performance from LockFree, as opposed to HomeQueue, due to the overhead entailed in using locks. We should expect to see levels of overhead similar to those of the previous experiment, as, though there are n worker threads to consider and manage, each lock used by the threads is uncontested. Likewise, between the use of Mutex and Anderson's locks, we should expect to see similar levels of performance for this same reason.

Speedup with Exponential Load:

We adopt the same reasoning from the prior experiment. However, because we employ a exponential packet generator, the amount of work each thread must perform is constantly increasing at an exponential rate. This may make it seem like we should experience even greater opportunities for speedup even as we continue to process packets. However, as different sources generate packets with different curves that describe the work needed to generate a checksum for the i-th packet, all of which merely share an average amount of work, this leaves the possibility of some queues becoming overburdened with large packets as the test goes on, creating performance bottlenecks. Therefore, while greater values of W should entail greater opportunities for speedup, without load balancing, this effect will not be as noticeable as when using an Uniform packet generator.

Like the previous experiment, in comparing the observed speedup for different values of W, we should expect increased speedup for larger values of W. For a given value of W, we should expect to see greater speedup for greater values of n – as observed in the results for HW2, we can expect speedup to appear linear for large enough values of W and $n \leq 13$. However, this is accompanied by diminishing returns on speedup for larger values of n, especially $n > 13$.

Likewise, we should expect to see consistently faster performance from LockFree, as opposed to HomeQueue, due to the overhead entailed in using locks. We should expect to see levels of overhead similar to those of the previous experiment, as, though there are n worker threads to consider and manage, each lock used by the threads is uncontested. Likewise, between the use of Mutex and Anderson's locks, we should expect to see similar levels of performance for this same reason. We may expect to see less overall slowdown of HomeQueue over LockFree in this experiment, as the bottlenecks posed by exponential packets without load balancing may mitigate overhead.

Speedup with Awesome

For my final experiment, I propose to redo the prior experiment but using $S \in \{LockFree, Awesome\}$.

Like the previous experiment, in comparing the observed speedup for different values of W, we should expect increased speedup for larger values of W. For a given value of W, we should expect to see greater speedup for greater values of n – as observed in the results for HW2, we can expect speedup to appear linear for large enough values of W and $n \leq 13$. However, this is accompanied by diminishing returns on speedup for larger values of n, especially $n > 13$.

Unlike the previous experiment, while the speedup of our LockFree implementation will suffer due to the use of an exponential packet generator, the speedup of our Awesome implementation will not. Rather, because worker threads are able to search for work to do, they can alleviate uneven burdens put on the implementation's various queues. This is the situation under which the Awesome strategy should succeed; otherwise, if there was an even distribution of work among queues, there would be no performance to be gained from making threads search for work. In this test then, the Awesome strategy will outperform the LockFree strategy.
This increase in performance from load balancing, however, will be mitigated because locks can now be in contention among different worker threads, they can be a greater source of slowdown than they were when used in the HomeQueue strategy. Performance between the two locks, however, should be similar, if not with the Mutex lock performing a bit better – this is confirmed by our curves from HW3a, Experiment 2. Furthermore, there are even more calls to lock() and unlock() in the Awesome strategy than in the HomeQueue strategy, suggesting increased levels of overhead.