Writeup Document

Alex Miller

April 29, 2021

Changes

Changes to the design

The following addresses changes I have made to my projects design, broken down by module:

- Top level modules (serial.c, parallel.c, serial-queue.c)
  - These files will still take in the same inputs. Moreover, they ways in which they are called and incorporated into the deriving experimental results is functionally the same.
  - However, these files will no longer utilize a 2D array in which to store verifiable experimental results. Instead, worker threads called in these files will individually maintain counters. These counters will be computed by adding up the values of checksums derived by the worker thread. The results of these counters will then be added together and written to the appropriately named file, instead of the checksums themselves (as specified in our previous design). We can use the results of these counters in order to compare the results of parallel, serial, and serial-queue implementations using the same number of sources and packets and the same seed.

    We can say that our implementations are consistent if they generate the same final counter value. This isn't necessarily proof that they produced equivalent checksums, in the sense that different implementations' computation of a common counter value could occur with widely varied checksum values. Strictly speaking, this method of verification is more dubious than our previous design, in the sense that our previous design inferred correctness from actual output (that is otherwise lost in our program) while out current design infers correctness indirectly. However, the odds of a false positive verification is low, considering that checksums are expressed as *longs*, and there are on the order of 2^{64} distinct values that final counters could take (accounting for overflow).

    The purpose of this is to minimize the amount of time in the critical section spent writing to memory; of course, this method still requires reading and writing memory. However, by having each thread operate on a single memory location, rather than O(T) total locations, we can minimize cache misses and get a clearer picture of performance.
  - Additionally, serial_queue.c will no longer reference chksum_parallel(). Instead it will utilize chksum_serial_queue(), details on this new function to follow. This change is due to the necessity of generating and dispatching packets in a round-robin manner, and the added run-times associated with called pthread_create(). These factors complicate the re-use of the steps outlined chksum_parallel(), but with only one thread, for the purposes of testing serial_queue.c. So instead, chksum_serial_queue() will utilize the same basic structure and data structures of chksum_parallel(), but utilize a framework more suited to a single worker thread. This will give us a better idea of how the dispatcher is operating.
- Algorithm implementation level (chksum.c)
  - First off, as our use of a 2D array in which to store results is deprecated, methods described in these files won't reference *res* anymore. Instead, they will use the values of checksums they generate in order to arrive at an amalgamated checksum value.

- Secondly, we must alter our design of our dispatcher, as specified in the method chksum_parallel.c(). Initially, we described a loop structure in which the dispatcher generates packets for a source until it's worker thread's queue is full, before moving on to populate the queue of another worker thread. This does not implement a "round-robin" loop structure, as required by the assignment. Moreover, this does actually simulate real-world use of our firewall, as packets from various sources would be arriving at our dispatcher simultaneously.

  Instead, for a given worker thread, the dispatcher will wait until its queue contains space for another packet, write to the thread's associated queue, and then move on to the next worker thread. It will repeat this process T times, or as long as there are packets to dispatch to worker threads.
- Thirdly, as mentioned above, serial_queue.c will utilize a new method, chksum_serial_queue(). This method will be defined in the chksum.c module.

  The key distinction between chksum_serial_queue() and chksum_parallel() is that rather than multiple threads maintaining single queues, a single thread will cycle through all the queues, dequeing packets and calculating checksums in the same order that the dispatcher places packets into the queues. Because of this, the single worker thread does not utilize the *done* field of its *packet_queue_t* structs, but instead exits once it has processed T packets from $N-1$ sources.

  This requires the use of a new data structure in order to pass the values of N and T to the single thread. The impact of this structure on performance is negligible, considering that is relatively small (the struct contains a pointer to a pool of *packet_queue_t* instances, as well as two integers to hold the values of N and T) and is only passed to one thread.

- Queue data structure (queue.c)
  - My $queue_t$, now known as $packet_queue_t$ declaration now defines the following struct members and methods:
    - $head, tail$: Volatile integers specifying the locations of the head and tail of the queue
    - $packets$: a Lamport queue with depth D
    - $done$: a Volatile boolean designating whether or not we should keep reading from Q. This is initially set to $false$
    - $final\_count$: As described above, each worker thread generates a combined checksum value. This value is incremented by this value every time its associated worker generates said variable.
  - Methods:
    - $enq(Packet_t x)$: queue a $Packet_t$ instance in the Queue. Return an error code if the queue is full.
    - $deq()$: dequeue the next $Packet_t$ from Q. Return a NULL value if the queue is empty
    - $create\_queue\_pool()/destroy\_queue\_pool()$: initialization/destruction methods for a block of queues
- Testing script (test_script.sh)
  - Here, I would like less to discuss a change, so much as discuss a design choice that arose in the writing my testing script; the assignment specifies that we should generate our plots using median data from our experiments. This can take be done in two ways. The first involves generating median speedup data by choosing median data points for all trials of a given test. The second involves calculating speedup for all trials, and taking the median value as a data point. My implementation opted for the latter option.

    This choice entails two advantages. Primarily, it reduces the necessary complexity of my testing script; rather than having to keep track of, store, and operate on multiple data points for a single test, I can collapse that information into a single data point per test. This makes reasoning about the data in an excel spreadsheet much easier. Secondly, this method has the added bonus of associating tests within a given trial, rather than comparing runtimes for trials that may utilize wildly different data sets.

- These changes remain consistent with my previously stated invariants

Changes to Test plan:

Instead of relying on an "absence of compile errors and the verifiable correctness of results" to infer the correctness of my *queue_t* implementation, I will test the performance of queue explicitly. I will do this by testing the performance of a *queue_t* against important edge cases; two that come to mind are the cases in which a writer is adding to a queue faster than a reader can extract values and the inverse, a reader extracting values from the queue faster than a writer can add to it. This behavior can be enforced through the use of the *sleep*() method in C. Under these tests, we will take an absence of queue corruption to imply correctness of implementation.

Results

The performance data for experiments 1 through 5 are stored in a folder called $hw2/exp\_data$. This folder contains the .csv files that describe performance data for a given experiment. This data was collected by running implementations for given number of trials, calculating measurements per trial, and taking median values. As such there is no other collected data other than median speedup, worker rates, and dispatcher rates. That data was deleted as it did not serve the purpose of analysis and would've have complicated my data storage scheme.

As per how many trials I utilized; for experiments utilizing uniform and constant packets, I used 7 trials. For experiments calling for exponential packets, I used 13 trials. I did more trials than recommended to assure myself that I would not experience jumps in data; using less yielded less than ideal data. I did not want to use more than this in fear of exaggerated testing run times.

I took this data and plotted it using Google sheets. This file can be found here. A copy of this data is also available in a .pdf that can be located at $hw1/hw1\_result\_data.pdf$. The following graphs represent this data as well.

Analysis

Theory Questions

Book problems 25, 29, 30, 31, 32

Problem 25

Dropping condition L2 would be equivalent to violating the requirements of program order; H would no longer have to abide by a legal sequential ordering.

Problem 29

No. The property leaves no guarantee that a thread can access x in a finite amount of steps. Given an infinite amount of time to return, it is not clear why an infinite number of method calls could not each wait an infinite amount of time. Therefore the following property does not describe a wait free object.

Problem 30

Since all method calls completed in an infinite amount of time, the object must be lock-free. There is no last method call that is left waiting.

Problem 31

Consider an infinite history H in which m is called an infinite amount of times, i. On the $i+1$th call, m will return after $2^{\infty+1}$ steps. Since this call neither returns in a finite number of steps nor has a finite bound on the number of steps it takes to return, it is neither wait-free nor bounded wait-free.

Problem 32

HWQueue is supposed to designate a queue; therefore it should support a FIFO ordering.

Line 15 is not an appropriate linearization point for enq() for the reason that it does not enforce proper ordering. For example, let thread-A call line 15 prior to thread-B. If line 15 was a true linearization point for enq(), then this would be reflected in the state of the queue; however this is not the case.
WLOG, let thread-A(enq(x)) $\rightarrow$ thread-B(enq(y)); more specifically thread-A calls line 15 prior to thread-B. However, say that thread-B calls line 16 prior to thread-A. Then, there is a possibility that some other thread C calls deq() after thread-B has set an item in the queue, but before thread-A has had the chance. Therefore, even though thread-A(enq(x)) $\rightarrow$ thread-B(enq(y)), thread-C(deq() = y) $\rightarrow$ thread-C(deq() = x) is a possible history as well. This violates our FIFO requirement.
Therefore, line 15 is not a linearization point because it alone does not enforce a legal queue structure; it does not make the effect of the method call visible to other method calls.

Line 16 is also not a linearization point, for much the same reason. Consider the example detailed above. If line 16 were truly a linearization point, then thread-C(deq() = y) $\rightarrow$ thread-C(deq() = x) would need to be true in all cases. However, consider the possibility that thread-C dequeues after thread-A has called line 16, in which case thread-C(deq() = x) $\rightarrow$ thread-C(deq() = y). In such a case, line 16 is not a linearization point. Though thread-B called it first, thread-A's object got dequeued first. Therefore line 16 is not a linearization point because it does not make the effect of the method call visible to other method calls.