Design Document

Alex Miller

April 20, 2021

Modules, data structures, interfaces, invariants:

Our code base will consist of the following modules:

- The highest ranked modules containing C code will be three files, one called serial.c, another called parallel.c, and a third called serial_queue.c
  - These modules represent our serial, parallel, and serial-queue implementations of our checksum generating scheme
  - All will take in the following input parameters: $< n >< T >< W >< D >< s >< -cue >$ where:
    - $< n >$: How many sources should we use?
    - $< T >$: How many packets per source should we ask for?
    - $< W >$: What is the expected work of one packet?
    - $< D >$: How deep should our Lamport queues be (if an implementation supports their use)
    - $< s >$: A seed number to use throughout the programs runtime. This variable will correspond to what trial we are currently running.
    - $< -cue >$: This should be a flag; each character represents what type of packet extraction method we should utilize (c is for constant, u is for uniform, e is for exponential)
  - We will be able to reference these files in our testing script (to be described later), simplifying our automated testing routines
  - Each file will describe the following workflow:
    1. Using user input, they 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.
    2. They will also bind the specified method of extracting packets from *packet_gen* to a pointer desginated *packet_method*.
    3. Lastly, they will allocate a 2D array with enough space to hold the values of all checksums to be generated. Call this array res
    4. They will then pass *packet_gen*, *packet_method*, res, as well as any necessary variables to one of two methods. The first will be known as chksum_serial.c and the second chksum_parallel.c. We will discuss the details of these functions in the context of their own modules. The first of these methods will be used by serial.c, while the second will be used by parallel.c and serial_queue.c.
    5. These methods will be responsible for calculating the checksums for packets generated by *packet_gen* using *packet_method*. They will also be the methods who's runtime we will measure to determine performance. Call this runtime measurement $< time >$
    6. These methods will also be writing the checksums they generate to res. We will discuss how this is done in more detail later.
    7. These files will, finally, use the contents of res in order to output verifiable results. For a given trial number, and for all n sources, will write a file called $res/ < n > \_ < T > \_ < W > \_ < D > \_ < s > \_ < -cue > \_ < time >$ which contains the list of T checksums generated for a specified source, using a specified seed s. We will use these files to compare results of serial.c, parallel.c, and serial_queue.c in order to judge correctness.

- chksum.c
  - This module will describe chksum_serial() and chksum_parallel, the methods that will bs used by serial.c, parallel.c, and serial_queue.c
  - chksum_serial()
    * Input: In addition to *packet_gen*, *packet_method*, and *res* this function will also take in the values n and T
    * This method simply loops through T packets for $n-1$ sources, utilizing *packet_method* to extract new packets from *packet_gen*
    * $\forall i \in [n-1], j \in [T]$ the method generates a checksum for the j-th packet of source i. This value is written to $res[i][j]$
  - chksum_parallel()
    * Input: In addition to *packet_gen*, *packet_method*, and *res* this function will also take in the values n, T, D, as well as a new value $< p >$. p describes how many worker threads should be used to generate checksums. When called by parallel.c, $p = n-1$. When this method is called by serial_queue.c, $p = 1$.
    * This method will begin by allocating space for p instances of the *queue_t* objects with depth D (see next section for details on *queue_t*). Call the thread that does this allocation t_d
    * t_d will create p threads and bind each thread to a *queue_t* instance. Each thread t will operate on it's queue q in the following manner:
      · t will need to keep track of two values: n and T. n is set by t_d for each thread t; it represents for what source index t is generating checksums. T is a counter maintained by t; initially $T = 0$.
      · While $q.done == false$ and $q.isEmpty() == false$:
      · If $q.isEmpty()$, t will wait until it contains data
      · Otherwise, t will call $q.dequeu()$ and generate a checksum for the $Packet_t$ instance it extracts. More on how this instance is placed in q in a second.
      · Once the checksum is generated, t stores the result at $res[n][T]$ and increments T by 1.
      · Once $q.done == true$ and $q.isEmpty() == true$, t should return.
    * Meanwhile, as each thread t is operating on its queue q_t, t_d is looping through T packets for $n-1$ and acting accordingly:
      · Since each thread is fixed to a given source, t_d must be responsible for populating each thread's queue with packets from a singular source. What we must keep in mind is that there might be more sources than threads (in the case of serial_queue.c) and the number of packets we can generate at any one time is limited by the depth D of our *queue_t* objects.
      · Therefore, I propose the following loop structure: $\forall t \in [p]$, let q_t designate a Queue object bound to the thread t, let n_t designate the source for which t is generating checksums, completely fill q_t with packets generated using *packet_method* from source n_t. Once this queue is full, move on to the next thread and do the same thing. Keep looping through all active threads like this until t_d has written T total packets to each q_t, at which point t_d should $write(q_t.done = true)$.
      · At this point, there may or may not be more sources for which we still must generate checksums; we take as many threads as we need (bounded by p at a time) and utilize the above mentioned strategy for the remaining sources.
    * In this way, we are able to split the work of generating checksums of T packets each from $n-1$ sources among p worker threads.
    * Invariants:
      · Each thread t computes the checksum of the ith packet from source n_t before computing the $i+1$th packet from source n_t
      · t_d never writes a packet to a queue that is full
      · Each thread t does not return until it has generated all T checksum for a given source n_t
- queue.c

  - This module will describe the construction and functionality of a *queue_t* object, containing the following attributes and utilizing the following methods:
  - Attributes:
    * D : a specified depth
    * Q : a Lamport queue with depth D
    * T : an integer designating how many packets have been queued so far in Q. This is initially set to 0.
    * *done* : a boolean designating whether or not we should keep reading from Q. This is initially set to $false$
  - Methods:
    * $enq(Tx)$: queue a value x of type T in Q. Throw an exception if Q already contains D entries
    * $deq()$: dequeue the next value from Q. Throw an exception if Q contains no entries
    * $isEmpty$: returns $true$ is Q does not contain any data, false if otherwise
    * $isFull$: returns $true$ is Q is full of data, false if otherwise

- test_script.sh
  - This testing script will consist of a shell script that utilizes serial.c, parallel.c, and serial_queue.c.
  - Because all three aforementioned executables record their resultant checksums in text files, this script can use these modules to generate experimental data and then organize that data in appropriately named folders. I can use this capability to automatically run and format my experiments using as many trials as I want.

  All in all, my code can be said to contain four modules: the first level describes the serial, parallel, and serial-queue implementations specified in the assignment, the second describes serial and parallel implementations of computing checksums of T packets from $n-1$ sources, the third describes the *queue_t* object that is needed for the parallel implementation, and the last level describes an automated testing structure that will generate and format my data.

Test plan:

Initially I will need to test my implementation of serial.c, which in turn will require I test the following capabilities:

1. Using the packet generator to generate different streams of input to chksum_serial.c; I will know I am doing this successfully when I am able to deterministically print output at the command line using all three packet retrieval methods specified in the assignment. I can verify whether or not I am able to do this by testing output for sufficiently small values of n, T, W and predetermined seed value s. This should count as a robust test on the grounds that deterministic output from these random functions using predetermined seed values should indicate that I am using these functions correctly or, at the very least generating usable inputs for my methods described in chksum.c

2. Using chksum_serial.c to generate checksums for T packets from $n-1$ sources; I will know that I am doing this successfully if I am able to deterministically output checksums at the command line using all three packet retrieval methods specified in the assignment. I can verify whether or not I am able to do this by testing output for sufficiently small values of n, T, W and predetermined seed value s. This strategy holds because the implementation of chksum_serial should be sufficiently trivial to implement and infer correctness for.

3. Using the contents of res in order to craft readable output in the form of text files; Once I am sure that chksum_serial() works, I can use its output to verify that I am writing results to text files correctly

Once I have demonstrated to myself an ability to generate checksums and record results using my serial implementation, I can use said results from my serial implementation to verify those of my parallel implementation. More specifically, I know these results should be equal for a given set of inputs for a given source given the uniformity of our seed variable within a given trial. Moreover, since threads in my parallel

implementation record results in the order in which packets for a given source are generated, we know that our representation of results in text files should be equivalent across both serial and parallel implementations.

test_script.sh can utilize this equality of results across implementations to check that our results are being generated correctly across implementations and trials; it can do this by comparing the text files generated by chksum_serial and chksum_parallel and raising an error on the appearance of any discrepancies.

As to the correctness of my *queue_t* declaration; whether or not this declaration is correct will be inferred from performance. Lamport queues are sufficiently well-known and easy to implement. The remaining methods and attributes of the *queue_t* data should also be easy to verify. We will therefore take an absence of compile errors and verifiable correctness of results to infer the correctness of this data structure as well.

Expected Performance (Performance Hypotheses):

Parallel Overhead:

I expect to see slowdown of serial_queue.c in relation to serial.c due to the extra overhead of allocating space for our singular Lamport queue and spawning an extra thread to calculate all of our checksums. Of course, however, this slowdown will be dwarfed or otherwise obscured by increasingly large values of n, T, and W; larger values for n and T imply that we will have more checksums to calculate and larger values of W imply that each checksum will require more work. Therefore, for larger values of n, T, and W we should notice the effects of slowdown less. For that same reason, I would expect that worker rate go up for larger values of W.

Dispatcher Rate:

This experiment involves distributing a fixed amount of data among varying number of packets and sources. However, because the work per packet remains fairly small throughout ($W = 1$), we should not expect to see much opportunity for parallelization. If anything, the increased overhead from having to deal with more threads should mitigate any potential speedup. For that reason I think the ratio of $(n-1) * T : runtime$ to shrink for larger values of n, even as T becomes smaller.

Speedup with Constant Load:

Because larger values of W imply more opportunity to parallelize the computation of checksums, for a given value of n, we should expect increased speedup 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. Moreover, because we employ a constant packet generator, the amount of work each thread must perform is constant, so we should experience lesser opportunities for speedup even as we continue to process packets.

Therefore, in comparing the observed speedup for different values of W, we should expect increased speedup for larger values of W. However, given a value of W, we should expect to possibly see some speedup for some values of n, if only accompanied by diminishing returns on speedup for larger values of n. The curves we might expect from these plots would be concave downward in shape.

Speedup with Uniform Load:

We adopt the same reasoning from the prior experiment. However, because we employ a uniform packet generator, the amount of work each thread must perform is constantly increasing, so we should experience greater opportunities for speedup even as we continue to process packets.

Therefore, in comparing the observed speedup for different values of W, we should expect increased speedup for larger values of W. For a given a value of W, we should expect to see something close to linear speedup for increasing values of n.

Speedup with Exponentially Distributed 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, so we should experience even greater opportunities for speedup even as we continue to process packets.

Therefore, in comparing the observed speedup for different values of W, we should expect increased speedup for larger values of W. For a given a value of W, we should expect to see something linear speedup for increasing values of n, if only because the advantage of using $n-1$ times as many threads in the parallel version than in the serial version will make itself even more apparent due to the exponential increase in work to be done.