Homework 2 Design

Michael Tang

April 23, 2019

1 Design

1.1 Input and Initialization

Currently planned is to parse command-line arguments in *main*() that encompass application version (default 0 for serial version, 1 for serial-queue, 2 for parallel), number of sources, Lamport queue depth (default 32), type of work distribution (constant, uniform, or parallel), number of packets per source, and program output direction. Then, a function is called to handle serial/serial-queue/parallel code pathing. Parallel and serial-queue will utilize the Lamport queue object as specified in Figure 3.3 of the textbook. The computed checksums can be printed to the directed output, paired with the packet and source number. This will be handled in a helper *output*() function, which will default output to stdout and also support muting output.

1.2 Serial

The sole thread iterates over each source, retrieving all packets from the source and outputting the checksum at each packet.

1.3 Parallel

The dispatcher thread retrieves packets from every source, looping over each of them (i.e. source 1 to 2 to 3 to 4 to 1 and so on if there are 4 sources) and gives them to the thread that matches their source to verify checksum. Each checking thread is given their packets to verify by the dispatcher via a Lamport queue, and output for each packet they dequeue. In order to ensure linearization of memory reads and writes, memory barriers and the *volatile* keyword will be utilized around the reading and writing of the tail and head pointers of the Lamport queues.

1.4 Serial-Queue

Same design as parallel except the dispatcher must dequeue from the Lamport queues and perform the checksum by itself. The same practice of looping over the sources will be done, and the dispatcher will prioritize enqueuing packets until there are no more packets to retrieve, then iterate over each Lamport queue to empty them and checksum each packet that it dequeues. If a queue is full when the dispatcher is attempting to enqueue another packet the dispatcher will save the packet, dequeue and checksum from that queue, then try enqueuing the packet again.

2 Testing

2.1 Correctness

The packet retrieval and checksum functions are trusted to be correct and will not be tested unless an error comes up while every possible other invariant is satisfied.
The other invariants are: Lamport queue correctness, linearization of reads and writes to head and tail, and dispatch of every packet to the correct queue.
Dispatch of packets to the correct queue as well as Lamport enqueueing correctness can be tested by outputting each queue's contents after every dispatch and ensuring that the packets came from the matching source for the queue.
Linearization of reads and writes for head and tail pointers, and Lamport dequeueing correctness can be tested by examining final output and ensuring that all packets have been outputted with their checksum, only once. In the case of an error here, formal tests can be drafted to test dequeueing correctness. If dequeueing is not the culprit, then linearization must be reasoned about.

2.2 Performance

To most optimally measure performance time, *stopwatch*() will measure the interval between entering a specific version of the code and the beginning of output procedures. That way the commonly shared overhead of initializing the program, parsing arguments, and outputting results will be excluded as they serve no use of comparison. Measurements can be taken as specified for each experiment by adjusting command line arguments and plotting results. Below are the hypotheses for each experiment.

2.2.1 Parallel Overhead

For each curve (each setting of number of sources), the ratio of serial-queue to serial runtime should be above 1 due to the overhead of managing the Lamport queues, and remain the same as the total amount of packets processed should remain the same as we increase W. However, the more sources there are, the lower the speedup should be, so the curve for $n = 14$ should be below $n = 9$ should be below $n = 2$.

2.2.2 Dispatcher Rate

$(n-1)T$ should remain constant for all n and parallel runtime should decrease with more threads, so the ratio of $(n-1)T$ to parallel runtime should increase as n increases. The curve will be upwards sloped.

2.2.3 Speedup with Constant Load

Optimally, the ratio of parallel to serial runtime would be $\frac{W}{n-1}$. Judging by that each curve should start around that point and then decrease, in direct proportionality to the increase of n. In increasing order of W each representative curve should have a bigger (downwards) slope.

2.2.4 Speedup with Uniform Load

The same thing as with constant load but curves start lower, because W is now distributed uniformly over all packets.

2.2.5 Speedup with Exponentially Distributed Load

Curves start higher than with constant load and the slope is also smaller (downwards), because the exponential distribution of work means that certain threads should be much slower than others, setting the upper bound for runtime and also restricting the amount of speedup additional threads can contribute.