CMSC 23010: Homework 2 Design Documentation

Will Thomas

October 20, 2022

Introduction

The goal of the following design documentation is to outline a cohesive plan for the design, correctness testing, and performance analysis of a program that will compute checksums on large amounts of packets using explicit parallelism. We will outline three versions of the program: A serial version, a parallel version, and a serial-queue version to better estimate the overhead required to compute the parallel version.

Design

We will begin by outlining our design for the serial program.

Serial:

The Serial program will be broken up into phases:

Phase 1:

Given the following program inputs: n, T, D, W, V, P where n refers to how many workers / packet generators there will be, T refers to how many packets will be created per packet generator, D refers to the depth of the queue (in this case, will be 0), W refers to the mean amount of work per packet, V refers to the version of the program (in this case V will always be 1 to signify serial) and P refers to the packet type, phase 1 of the program will consist of allocating space for an $n-1$ length array that will hold PacketSource_t pointers. Then, the PacketSource_t objects will be created and pointers to them will be placed in the array.

Phase 2:

Next, we will pass this array to a helper function that will iterate through every PacketSource_t object and then repeat the following T times:

1. Request a packet
2. Compute a checksum for said packet

The checksum output file will be closed, the helper function will terminate.

Phase 3:

Back in the main function, we will free all the allocated space, and the program will terminate.

Parallel

The Parallel program will also be broken up into phases.

Phase 1:

Much like the Serial program, the Parallel program will begin with an array of $n-1$ pointers to PacketSource_t objects being created.

Phase 2:

From here, a pthread will be created and sent, holding the packet source array as an argument, to a function called dispatch. Dispatch will work as follows:

1. First, dispatch will create an $n-1$ length array each holding the value T. This array corresponds to how many packets have been offered this worker.
2. Next, a while loop will be entered, the condition being that all cells in the array are not empty.
   (a) From there, we will iterate through the worker array and find a non-zero entry. If the value is still equal to T, the worker has been offered no jobs and the corresponding thread has not yet been created. As such, we will create a Lamport queue object, create a thread, and send the thread to execute a function 'worker' that we will define later.
   (b) In the case that the cells are not equal to T, meaning that the threads have already been created (as well as the corresponding queue objects), search for the first non-zero cell in the array. Then, check if the queue is full or not. If it is, move on to the next non-zero element in the array. If not, generate a packet and en-queue it, decrement the value stored in the array by one, and then move on to the next non-zero element in the array.
3. The above loop will terminate when every worker has been given T jobs. From there, wait for each worker to join.

Phase 3:

Back in main, join the dispatcher thread and free all of the space allocated.

Worker Function:

The function 'worker' will take as input a Lamport queue pointer as well as the value T. The function will implement a while loop that continues as long as T is not 0. In each iteration of the while loop, if the queue is not empty, the packet is de-queued and a checksum is computed, and the counter is decremented. If the queue is empty, do not decrement the counter and try again. The queue should be volatile so that it will be repeatedly checked. Once the counter reaches 0, join the thread.

Serial Queue:

Phase 1:

Phase 1 will be the same as the other two versions of the program.

Phase 2:

Like in the parallel implementation, a dispatcher thread will be created and sent to execute a function dispatch. From there, it will function very much like the parallel version of the program except it will only use one worker thread. It will do this by looping through the array of packet sources and creating a pthread each time to compute the worker function. It will simply wait until that single worker has computed T checksums and then it will join the thread and continue on to the next packet source.

Phase 3:

The dispatcher thread will be joined, the allocated space will be freed.

Invariants

The following invariants will inform our correctness testing:

1. A worker does not start working until its queue has been created
2. A job is not even created if the recipient worker's queue is full.
3. A worker does not attempt to dequeue from an empty queue.
4. A worker stops working after T jobs.
5. Each worker completes its work in FIFO order.
6. For each packet generator, there will be exactly T packets generated; no more and no fewer.
7. The dispatcher will never enqueue more than one job at a time for a single worker. This is so that it will not starve other workers completely filling one worker's queue.
8. Once every worker has been given T jobs, the dispatcher stops creating and enqueueing new jobs.

Correctness Testing

For the purpose of testing and only testing, I will introduce locks into the code. The reasoning for this is as follows: In order to show that the workers complete all T jobs, it would be very helpful to have them write the computed checksum as well as their thread/worker id into an output file. As such, the locks would exist only to prevent concurrent file writes. Given this, the testing plan is as follows:

1. Run the serial version on the program to create on small numbers of n and T and W so that the correctness of the checksums can be easily verified.
2. Create larger test sets with various values of n and T and W. Do this for each packet type.
3. Run the parallel and the serial-queue programs with the same parameters and check the outputs against the test sets. If the outputs are the same, we know that the checksums are correct and that the proper amount of checksums are being computed. Furthermore, verifying that everything is completed in the correct order would be very simple.
4. The other listed invariants can be ensured to be true with checks and asserts, as well as unit tests to ensure that the dispatch function terminates once the work array holds only 0 for all cells.

Performance Hypothesis and Testing Plan

1. **Parallel Overhead**

   I suspect that for each value of $n \in \{2, 9, 14\}$, we will see the speedup approach 1 as W grows large. This is because the overhead required to create and join each thread will become smaller compared to the overall amount of work that needs to be done to compute each checksum. However, I think that we will see speedup approach 1 at a faster rate for smaller values of n due to fewer threads needing to be created.

2. **Dispatcher Rate**

   I hypothesize that we will see packets per second decrease slightly for small values of n until the overhead can be overcome. I believe for values larger than $n = 9$, packets per second will be increasingly monotonically with n.

3. **Speedup with Constant Load**

   I hypothesize that for each graph, as was the case in parallel overhead, when the packets are more work intensive the speedup will be more pronounced. Because each packet will be taking more time, the benefit from doing them concurrently will begin to outweigh the downside of parallel overhead. I believe that as the number of cores increases we will see a slowdown for smaller amounts of packet work but a more significant speedup for larger amounts of packet work.

4. **Speedup with Uniform Load**

   I suspect that overall we will get similar graphs as we did with the constant load test, as the mean is the same and because we will be running numerous (at least 5) trials. That is, I expect better performance for larger values of both n and W.

5. **Speedup with Exponentially Distributed Load**

   I believe that the exponentially distributed packets will lead to serious imbalances in work, especially as W gets large. I think we will get the best performance from moderately small values of W and moderately small values of n, but even then it is far from clear how exactly this will play out. I do believe that we will get very bad performance from large values of both n and W due to the compounded negative effects of large amounts of parallel overhead and extremely work intensive packets mixed in that may effect certain workers disproportionately.