

Homework 2

*Professor: Henry Hoffmann**TA: Harper Zhang*

Theory

Problems from Ch.3 from previous homework.

Programming

The original idea for this assignment came from Nir Shavit. Lamont Samuels developed helper code for C.

Please begin programming assignments early. Doing so will allow us to do a better job helping you.

The Programming Environment

We will be programming in C with POSIX threads (pthreads). There are many pthreads tutorials and overviews on the web. I prefer the one from Lawrence Livermore National Laboratory: <https://computing.llnl.gov/tutorials/pthreads/>

Pthreads provides several useful functions for mutual exclusion and synchronization. Your final codes should not make use of these unless explicitly stated. However, they might be useful for debugging. For example, if you are not sure whether you have correctly implemented mutual exclusion, you can use a pthread mutex (MUTual EXclusion) object to replace your mutual exclusion protocol in the code. If the program starts to work, then you have a better idea of where to look for bugs.

The gcc compiler provides access to hardware synchronization instructions through gcc atomic builtins. You can read more about those here: <http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html>. Please do make use of these unless we explicitly tell you not to.

Programming Projects Background

These programming assignments are based on building a firewall: we will see a stream of *packets* from various source addresses (represented as integers) and perform several operations before sending them to the destination address (also integers) specified in the header of the packet. By default, packets will be *data* packets, messages from source address to destination address with some variable-length payload. Our firewall will perform two primary functions: 1) enforce access controls and 2) filter the data payloads for evidence of *bad guys* (that the packet is from a suspicious source).

The First Programming Assignment

This assignment is all about work distribution and load balancing. Suppose that we have n different sources, each of which will supply T packets (n and T are input parameters). Your goal is to calculate a checksum on all packets. You'll be provided with a data object and associated functions that calculate the checksum and you will also be provided with an object which creates the packets (the *Packet Generator*). In order to get a parallel speedup, you will have a single thread (the *Dispatcher*) retrieve the packets from the packet generator, but distribute the work of calculating the checksums across $n - 1$ threads (the *Workers*). Because we tie the number of threads to the number of sources, we can use a Lamport queue (Figure 3.3 "WaitFreeQueue" in the text - also called a single-reader / single-writer queue) of depth D (an input parameter) to buffer packets for each thread. That is, each worker has a dedicated Lamport

queue which is written only by the Dispatcher and read only by the associated Worker, thus requiring no locks. Provided the typical work to calculate a checksum is large relative to the work to retrieve the packet and write it into the Lamport queue, we should see a reasonable opportunity for parallel speedup. We will build three different versions of the code:

- **SERIAL** The application, as described above, where a thread loops through each source and each packet to calculate the checksums serially. This version is provided - it should be used to test correctness of the other versions and as a tutorial for how the other provided components are used.
- **PARALLEL** The application, as described above, implemented with n workers calculating the checksums in parallel.
- **SERIAL-QUEUE** The application, as described above, in which a thread proceeds in much the same way as in **SERIAL**, but each packet from the i th source must be written to and then subsequently read from the i th Lamport queue. In particular, all data structures used in the **PARALLEL** version should be instantiated in this version, the only difference being that all code is performed by a single thread. This configuration allows us to ascertain the overhead in additional work we incur by parallelizing the application and communicating through the queue.

Load Balancing

Assuming that the work is dominated by the checksum calculation, we might indeed get decent parallel speedup. However, this also relies on how the distribution of work falls to each worker, since we have a fixed assignment of worker to packet source. That is, if one worker has considerably more work than the average worker, the overall runtime will suffer. In order to explore this effect, we will use three different sources of packets: *Constant*, *Uniform*, and *Exponential*. Constant packets have the same amount of work per packet, Uniform packets have the property that the expected work for every packet from every source is identical and distributed according to a uniform random variable (uniform over $[0, 2W]$), where the expected work, W , is an input parameter. Exponential packets, by contrast, have for each packet coming from source i an expected work equal to μ_i , which is distributed according to an exponential distribution with parameter $\lambda = \frac{1}{W}$. Furthermore, the packets from source i have work that is distributed according to an exponential distribution with parameter $\lambda_i = \frac{1}{\mu_i}$. The difference in average work per source and the added variability of the exponential distribution (vs. the uniform distribution) both give rise to an imbalance of work across the n workers.

Code

You should create a directory and expand the accompanying and retrieve our helper code from svn (directions will be posted on the class website shortly – it will either be through svn or a tarball posted on the class website).

In particular, the following data types are provided:

- **PACKETSOURCE_T** is a random packet generator (see `packetsource.h`) which is instantiated with three parameters:
 - *mean* is the expected amount of work that will need to be performed per packet.
 - *numSources* is the number of packet sources.
 - *seed* initializes the internal random number generator - it can be set to the trial number if you wish to corroborate your packet checksums across runs of **SERIAL**, **SERIAL-QUEUE** and **PARALLEL**.

In addition to creation and deletion, it exports three functions, `GETCONSTANTPACKET`, `GETUNIFORMPACKET`, and `GETEXPONENTIALPACKET`, which, given an input source number, returns the next packet from that source. *Note: the uniform and exponential packet streams are independent for each source - so the sequence of packets from each source is deterministic, regardless of the order that they are taken among the sources.*

- `STOPWATCH` is a simple timer (see `stopwatch.h`) which reports times accurate to microseconds.
- The function `GETFINGERPRINT` takes two **longs**, *iterations* and *startSeed* (*i.e.*, as a proxy for the packet body), and returns a **long**, which is the checksum, or fingerprint, of the packet body.

You should not need to modify this code for these assignments. If you believe that you do, please start a discussion on the class webpage, or see the instructors to discuss why you think it should be done.

Your code should be configurable by the following input parameters (`argc`, `argv`):

- n - the number of workers. In this case, it is also equal to the number of sources and the number of queues, due to the fixed mapping of work to worker - (`NUMSOURCES` in the code).
- T - the total number of packets from each source - (`NUMPACKETS` in the code).
- D - the number of entries in each Lamport queue - (`QUEUEDEPTH` in the code).
- W - the expected amount of work per packet - (`MEAN` in the code).

Next, you will perform the following set of experiments across various cross-products of these parameters. Each data point is a measurement taken on a dynamic system (a computer...) and is thus subject to noise. As a result, some care should be taken to extract representative data - we would propose running some reasonable number of experiments for each data point and selecting the median value as the representative. For Uniform packets, something like 5 data points should suffice whereas for Exponentially Distributed packets 11 may be required to get relatively smooth plots - please use your own engineering judgment to decide how many trials you require. You should set the *seed* parameter for the `PACKETSOURCE` object to the trial number (or some deterministic function thereof) to ensure that you're seeing a reasonable variation in load from each source. In each of the following experiments, we describe a plot that you should produce, analyze and discuss in the writeup.

1. **Parallel Overhead** Run `SERIAL` and `SERIAL-QUEUE` ($D = 32$) on uniformly distributed packets with $W \in \{25, 50, 100, 200, 400, 800\}$ and $n \in \{1, 8, 16\}$. Let $T \approx \frac{2^{24}}{nW}$ for each experiment. Each should take about a second. Plot the speedup (*i.e.* ratio of `SERIAL-QUEUE` runtime to `SERIAL` runtime) for each n (*i.e.* one curve for each n) on the Y -axis vs. W on the X -axis. Using the `SERIAL-QUEUE` data, derive the **Worker Rate**, the packet rate of a worker, given W .
2. **Dispatcher Rate** Run `PARALLEL` ($D = 32$) on uniformly distributed packets with $W = 1$, $n \in \{1, 2, 4, 8, 16, 32\}$, $T = \frac{2^{20}}{n}$. Plot the ratio of nT to runtime of `PARALLEL` (*i.e.* packets per second) on the Y -axis vs. n on the X -axis.
3. **Speedup with Constant Load** Run `SERIAL` and `PARALLEL` ($D = 32$) on constant load packets with $W \in \{1000, 2000, 4000, 8000\}$ and $n \in \{1, 2, 4, 8, 16, 32\}$. Plot the speedup of `PARALLEL` to `SERIAL` for each W (*i.e.* a curve for each load) on the Y -axis vs. the number of cores, n , on the X -axis. Derive an expected speedup using the **Worker Rate** and **Dispatcher Rate** from the previous experiments. How does the expected speedup compare with the measured speedup?
4. **Speedup with Uniform Load** Run `SERIAL` and `PARALLEL` ($D = 32$) on uniformly distributed packets with $W \in \{1000, 2000, 4000, 8000\}$ and $n \in \{1, 2, 4, 8, 16, 32, 64\}$. Let $T = 2^{17}$. Plot the

speedup of PARALLEL to SERIAL for each W (*i.e.* a curve for each expected load) on the Y -axis vs. the number of cores, n , on the X -axis. Derive an expected speedup using the **Worker Rate** and **Dispatcher Rate** from the previous experiments. How does the expected speedup compare with the measured speedup?

5. **Speedup with Exponentially Distributed Load** Run the same experiment as **Speedup with uniform load**, except use the exponentially distributed packets from the packet generator. How do these results compare with those with the uniform packets?
6. **Extra Credit:** Design an experiment (and hypothesis) that tests speed as a function of D . If you attempt this extra credit, your design document should describe how you will conduct this study and what results you expect.

Design and Test Document

The design for this problem is more complicated than the previous assignment. You need to spend some time thinking of how you will test your queues, workers, and dispatcher as independent (or semi-independent) entities. You also need to think about how they will work together.

Also, the hypotheses are more complicated for this assignment. You should come up with one hypothesis per experiment. You should leave yourself enough time to adjust if your hypothesis does not match the experimental results. This may require beginning the implementation before the design review. Certainly, you should plan to have results prior to the morning before the assignment is due.

The design and test document is due 1/28/2014.

Writeup

Please submit a typeset report summarizing your results from the experiments and the conclusions you draw from them. Your report should include the five plots (graphs) as specified above. Also, submit the working code for SERIAL, SERIAL-QUEUE and PARALLEL. The write up should be submitted in class, the code should be submitted in the svn repo.