CMSC 23010 Homework 2 Design
Ruolin Zheng
April 16, 2019

0.1 Design

Modules
There will be three modules for this assignment, `lamport_queue`, `dispatcher`, and `worker`.
- `lamport_queue` is implemented with a circular array of length D. Each entry of the queue is of type `void*` which could be a pointer to a data structure, an integer, and so on. The `head` and `tail` attributes are declared as `volatile`; the gcc memory fence `asm volatile ("" : : : "memory");` precedes any read or write to these two variables. The Lamport queue is a FIFO lock-free concurrent object that supports one reader and one writer simultaneously. The queue returns 0 on a successful call to `enqueue` and 1 if the queue is already full; it returns the first entry `void*` on a successful call to `dequeue`, or `NULL` if the queue is empty.
- `dispatcher` retrieves packets from the packet generator, and writes to the corresponding queue associated with the specific Worker and packet source.
- `worker` reads from its exclusive queue, and calls `get_fingerprint`.

Interface
The program can be run with these parameters: `./checksum -n NUM_THREADS -t NUM_PACKETS -d DEPTH -w WORK`
Putting the three modules together, `main` will create a `packet_source` for `lamport_queue` for each `worker` and initiate the `dispatcher`; as the `dispatcher` starts to write into each `lamport_queue`, the associated `worker` reads and computes the checksum. If the i-th queue happens to be full, the dispatcher will skip over it to give a packet from source $i+1$ to the $i+1$-th queue. Each worker keeps count of the packets it has computed and exits once the count reaches T.

Data Structure
SERIAL does not require any special data structure. For PARALLEL and SERIAL-QUEUE, each Worker is associated with one Lamport queue as described in **Modules**.
Synchronization and Data Sharing
Since each Worker is supposed to read only its assigned queue, we can enforce this in our implementation by pass a pointer to its exclusive queue to each worker thread. Each queue is shared between the Dispatcher and one unique Worker.
Assignment of Tasks to Threads
In SERIAL, the main thread sequentially computes the checksum for each packet from each source. In SERIAL-QUEUE, the main thread acts both as the Dispatcher and the Worker; for each packet, it first writes to the queue then reads from the queue and computes the checksum. In PARALLEL, the main thread acts as the Dispatcher, and there are $n-1$ Workers; each worker is associated with a unique queue and a unique source; for a given packet, the Dispatcher writes to the corresponding worker's queue and that particular worker has exclusive access to read and compute the checksum.

0.2 Testing

Correctness Testing Plans
- Lamport Queue: We test both methods, `enqueue` and `dequeue` for error condition and correctness first in a **sequential** setting. We allocate a queue of length 4 and assume the entries of the queue are integers.

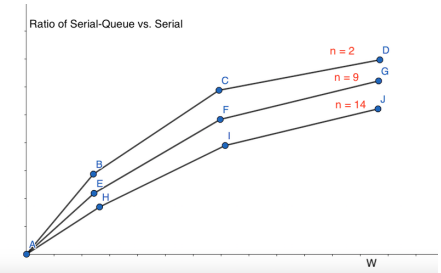
- Test `dequeue` on an empty queue; we expect a `NULL` return value.
- Test `enqueue` on a full queue of length D; we expect a return code of 1.
- Test the following sequence of instructions sequentially: `enqueue(1); dequeue(); enqueue(2); enqueue(3); enqueue(4); enqueue(5); dequeue(); dequeue(); dequeue(); dequeue();`
  We expect the following return values (including return codes): `0, 1, 0, 0, 0, 0, 2, 3, 4, 5`

- Test the queue concurrently with two threads A and B; assume no two reads or writes occur simultaneously but we allow one reader and one writer to perform concurrently. We've ascertained sequential correctness from the above tests, and we can show concurrent correctness by **linearization**.
  Test the following instructions (textbook p.52, figure 3.6) where `enqueue_A` means A performs an `enqueue`: A will execute these two instructions in order `enqueue_A(1); dequeue_A();` and B will execute `enqueue_B(2); dequeue_B();`
  We should be able to get back `2` from `dequeue_A()` and `1` from `dequeue_B()` for some execution.

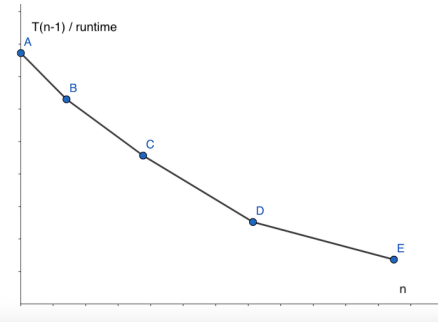
- Dispatcher: We test that for each source i, the Dispatcher only writes packets to the i-th queue. We also test that a Dispatcher skips over the i-th queue if it is full and writes a packet from source $i+1$ to the $i+1$-th queue.
- Worker: We test that each Worker is only able to read from its exclusive queue (we've enforced this in the code in **Synchronization**.) We also test that a Worker's count is exactly T when it exits.
- Invariant for SERIAL-QUEUE **and** PARALLEL**:** The invariant is that FIFO order must hold; an item cannot be dequeued before it is enqueued; dequeue always return the head element in the queue, and enqueue always append to the tail.

Performance Testing Hypothesis and Hypothesized Plots

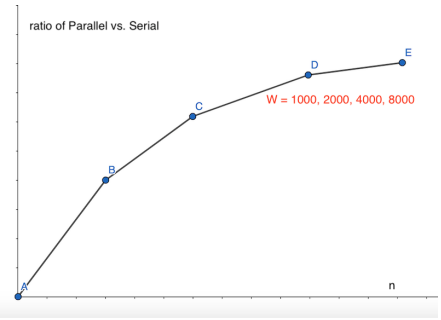
1. Parallel Overhead The ratio of SERIAL-QUEUE runtime to SERIAL runtime should be consistently lower than 1.0 as we incur an overhead of creating and communicating over the queue. As W increases, the increase in overall runtime downplays the effect of the overhead and the ratio should tend to 1.0. This is to say, for each n curve, the ratio increases as W increases. Moreover, when we fix W, the total number of packets across all sources is approximately $T(n-1) \approx 2^{24}/W$; with larger n, SERIAL-QUEUE needs to perform more reads and writes, incurs a larger overhead and thus suffers in runtime. Therefore, the curve $n = 2$ should lie above $n = 9$, and $n = 9$ should lie above $n = 14$.
Since the total number of packets is $2^{24}/W$, the **Worker Rate** should be $2^{24}/(W * runtime)$; therefore, the greater the expected amount of work per packet, the lower the worker packet rate.



2. Dispatcher Rate The total amount of work is approximately $T(n-1) = 2^{20}$, hence the **Dispatcher Rate** is $2^{20}/runtime$. Since the overhead of creating and joining $n-1$ threads, and of creating and communicating over $n-1$ queues increases in proportion to n, the runtime should increase as n increase; therefore, as n increases, the dispatcher rate becomes lower.

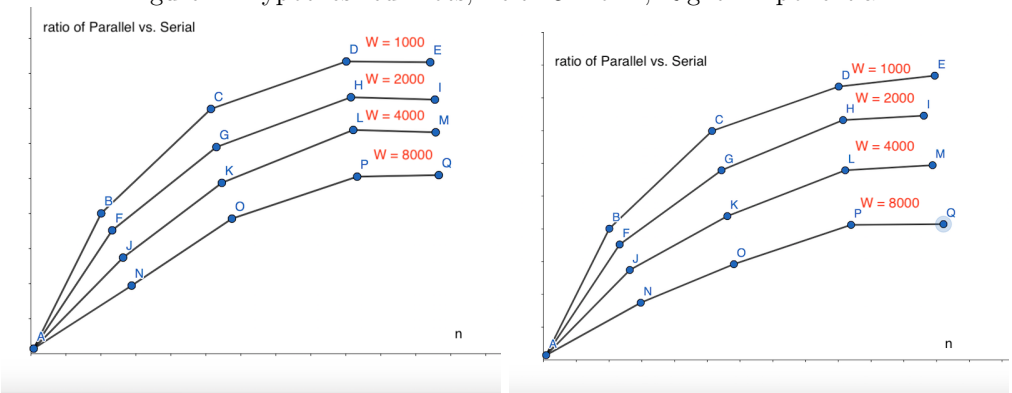


3. Speedup with Constant Load Ideally, we would expect an $n-1$-fold speedup as a result of using $n-1$ threads instead of one to handle the $n-1$ sources. However, in reality, the overhead is proportional to n and the actual measured speedup should be lower than expected. As the n increases, PARALLEL becomes significantly more efficient compared to SERIAL, and hence the curves are upward-sloping. Moreover, for a fixed n, since W is constant, the amount of work per thread is exactly WT; assume PARALLEL runtime is a multiple of WT, then SERIAL runtime should be a multiple of $WT(n-1)$ and the ideal ratio should be $1/(n-1)$. Hence, there shouldn't be much variability among different values of W, and their curves should be approximately overlapping.



4. Speedup with Uniform Load The uniform distribution adds some variability to the system; for example, it is possible that one thread repeatedly gets heavy packets and increases the runtime of the entire system. With larger W, the range $[0, 2W]$ is wider, and the variability increases. (It may be worth noting that the probability shouldn't be too large when T is as great as 2^{17}.) Overall, as in **Speedup with Constant Load**, the ratio still increases with n, but the W curves may be different from each other; the curves of larger W might lie below those of smaller W because it is more probable for threads to get uneven loads and increase the runtime of the system. Ideally the speedup should still be $n-1$-fold, but the measured speedup should be lower due to the unbalanced load problem.

Figure 1: Hypothesized Plots, Left: Uniform, Right: Exponential



5. Speedup with Exponentially Distributed Load The exponential distribution introduces even more variability; and it is very likely that some thread constantly gets heavy packets and drastically increases the runtime of the entire system. With larger W, $\lambda = 1/W$ becomes smaller, and the individual W can vary more in value. Therefore, the curves of larger W should lie below those of smaller W because of this added inbalance of load distribution. The measured speedup should be a lot slower than $n-1$-fold.