Parallel Computing Project 2

Rachel Hwang

February 4, 2014

1 Final Design

My final design differed significantly from my original design as I realized that had misunderstood the theory behind Lamport Queues, and had intitally be writing code which would require ordinary locks. My new implementations center around an enqueue function and a dequeue function as shown below:

```c
// Enqueue a single packet onto q
int enqueue(int *count, SerialList_t *q, int numPackets, int depth, volatile
    Packet_t *packet) {
  if (q_len(q) >= depth) {
    return 0;
  }
  add_list(q, *count, packet);
  (*count)++;

  if (*count == numPackets) {
    (*count) = DONE;
    return 1;
  }
  return 0;
}

// Dequeue a single packet from the end of q
void dequeue(SerialList_t *q, long int *fingerprint) {
  Item_t *curr = q->head;
  Item_t *prev = NULL;
  volatile Packet_t *tmp;
  if(!curr || !q) {
    return;
  }
  if (q->head != q->tail) {
     while (curr->next) {
      prev = curr;
      curr = curr->next;
    }
    tmp = curr->value;
    *fingerprint += getFingerprint(tmp->iterations, tmp->seed);
    remove_list(q, curr->key);
    q->tail = prev;
  }
  return;
}
```

- *count* is an array of integer for all data sources/queues which keeps track of how many packets have been enqueued by the dispatch thread. *count*[*i*] is set to value DONE when numPackets have been enqueued from one source i.

- $fingerprint$ is an array of fingerprint accumulators where checksums are added during dequeue.
- $enqueue()$ adds a single packet to the head of a queue. It will not enqueue if the maximum queue depth has been reached, or if ($count == DONE$). All queue infrastructure is build off the provided SerialList linked list library provided.
- $dequeue()$ removes a single packet from the tail of a queue. If given a valid list, it will not dequeue if ($q \rightarrow tail == q \rightarrow head$), as this means that that list is either a single element long or the list is currently being accessed by both enqueue and dequeue. Priority is thus always given to the dispatcher. Since the list length may fall prey to race conditions, the length of a queue is always obtained by traversing the list rather than simply reading the value stored in the struct. In essesnce, this program uses a very minimalist application of the Lamport algorithm to consistently give priority to the Dispatch thread, but ensure that a thread will not exit until it has dequeued the proper number of packets.

In both the serial-queue and parallel programs, the enqueue and dequeue functions are thus designed never to interfere with each other. All other important values, are touched by only one handler. $count$ is only every touched by the dispatcher via enqueue and $fingerprint$ is only ever touched by dequeue.

Overall, both the parallel and serial-queue versions are split into several logical modules: the dispatcher, the dequeue and the enqueue. Enqueue is always called in the a series of nested loops and conditionals in the dispatcher, as below. The invariant for the outermost loop checks to see if any more sources have more packets to contribute. The next loop cycles through all sources, managed by a conditional which will only allow a call to $enqueue()$ if count is below numPackets for that source and if the queue is not already full. These conditions are managed in a dispatcher function, $squeue\_firewall()$ and $parallel\_firewall()$ respectively.

```c
// from parallel_firewall()
while (done < numSources) {
    for( i = 0; i < numSources; i++ ) {
        if ((count[i] < numPackets+1) && (q_len(queues[i]) < queueDepth)) {
            volatile Packet_t *packet = getExponentialPacket(packetSource,i);
            done += enqueue(count+i, queues[i], numPackets+1, queueDepth, packet);
        }
    }
  }
}
```

In the serial-queue implementation, dequeue is called directly from the Dispatcher. In the parallel implementation, $dequeue$ is called from the $thr\_dequeue()$ function. This function loops continuously, trying to dequeue from its list whenever ($q \rightarrow tail! = q \rightarrow head$). Once count is set to DONE, meaning that all packets have been sent, and the queue is empty, all packets from the current source must have been processed. However, because this condition is needed to preserve the mututal exclusion an extra packet must be enqueued (see above), such that ($q \rightarrow tail! = q \rightarrow head$) is always true when a thread wants to dequeue a list. This is acceptable given that the program models a firewall, which would loop continously to recieve a endless number of packets, in which case this post-condition would never be necessary or applicable.

```c
// Thread function. Dequeues in a loop
void *thr_dequeue(void *arg) {

  thr_data_t *data = (thr_data_t *)arg;
  int *count = data->count;
  SerialList_t *q = data->q;
  long int *fp = data->fp;
  while (1){

    dequeue(q, fp);

    if ((*count == DONE) && (q->head == q->tail)) {
      pthread_exit(NULL);
    }
  }
}
```

2 Testing

All original code was subjected to unit testing (code may be found in firewall_tests.c). Since the serial-queue implementation and the parallel implementation were structurally and logically very similar, most tests could be shared with the exception of the thread function tests. To check correctness, output of both the serial-queue and parallel implementations were checked against the output of the given serial program.

The test module includes the following functions

- TESTcreate(int n)
  Tests that *create_queue*() creates the correct number of queues.
- TESTenqueue(int n)
  Checks behavior of *enqueue*() for length n.
- TESTdequeue(int n)
  Checks behavior of *dequeue*() for length n.
- TESTserial_queue(int numPackets, int numSources, int uniformFlag, short experimentNumber)
  Checks behavior of serial-queue against serial implementation.
- TESTthread(int n)
  Checks that thread() spawns the correct number of threads.
- TESTthr_dequeue(int n)
  Checks behavior of the thread loop function on given input length.
- TESTparallel(int numPackets, int numSources, int uniformFlag, short experimentNumber)
  Correctness test. Checks parallel implementation against serial implmentation.

3 Experiments

3.1 Parallel Overhead

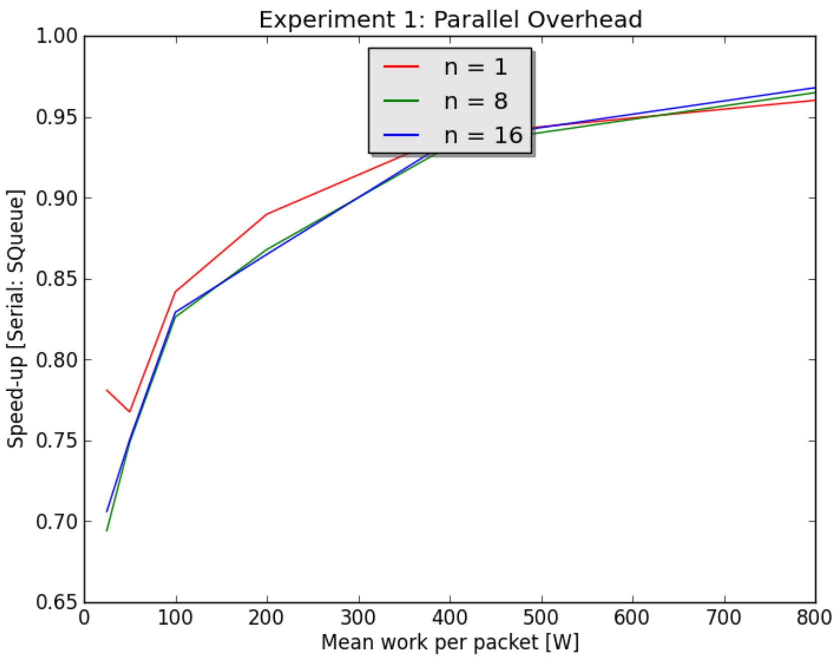

Figure 1: This figure illustrates the improvements in the speedup as the mean work per packet increases.

Hypothesis: Speedup will remain relatively fixed (but a slight decrease) as n increases, and will improve as W increases. Serial/serial-queue ≈ 0.7 when $W = 25$, and for any values of n.On the otherhand, as W increases

Results: My hypothesis was correct in that speedup is about .7 when $W = 25$ and that n does not significantly effect performance. W has by far the largest effect on speedup because the greater the mean work, the smaller the percentage of time spent on the additional infrastructure needed to implement queues. The serial and serial-queue program are functionally identical, but the serial-queue program must take the time to transition between queues rather than just enqueueing and dequeuing in one continuous stream. However, once the infrastructure has been added, the cost of adding more sources/queues is minimal compared to jump in cost from no queues to 1 queue. Thus, we observe that the performance of each of these n values is similar, and that the disparity between $n = 8$ and $n = 16$ is less noticable than that between $n = 8$ and $n = 1$. I cannot explain precisely why the discrepancies vary between $W = 100$ and $W = 200$, but once again, it seems logical that n values will have a larger effect on performance for small W values, since the queue work is a higher proportion, and more visible in the overall results. For large W values, the effect of n values looks negligible. The peak speedups for all n values were about .95 at $W = 800$.

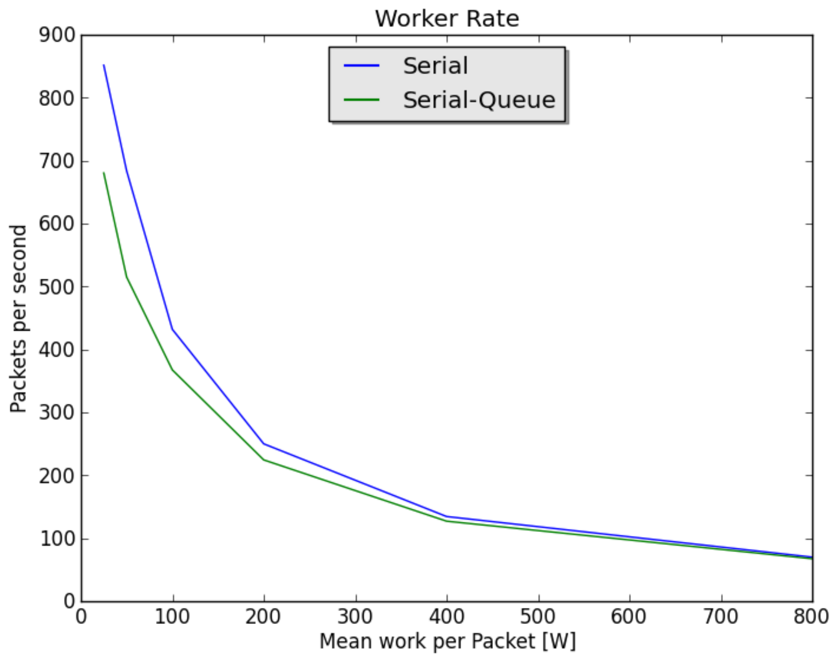

Figure 2: This figure illustrates the relationship between packet rate per worker and mean work per packet.

Worker rate shows a negative correlation with W, which is intuitive given that the larger the packet size, the less packets can be processed in a finite ammount of time. Worker rate describes the ammount of work a worker thread can expect to do per millisecond.

3.2 Dispatcher Rate

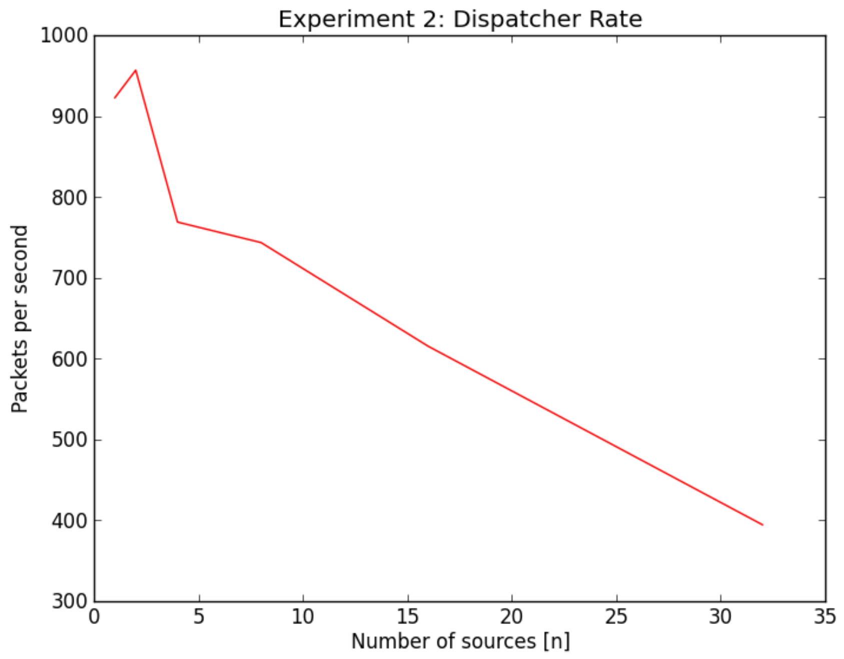

Figure 3: This figure illustrates the effects of the number of sources on dispatcher performance.

Hypothesis:[Edited from original write-up] The dispatcher rate should decrease as the number of sources increases.

Results: This is a simple, but important relationship that shows that the greater the number of sources, the more slowly the dispatcher is able to distribute packets. This makes sense given that increasing the number of sources to retreive data from complicates the work that the dispatcher does. Here we observe a fairly linear relationship between packet rate and n. The max rate was about 950 packets/ms at $n = 2$ and the minimum was about 400 at $n = 32$. The slope of the line is approximately $(-100p/10n) = 10p/n$. This linear behavior also makes sense given the way my dispatcher handles queues, in order, in a manner which scales work almost linearly with n after the inital overhead.

3.3 Speedup with Uniform Load

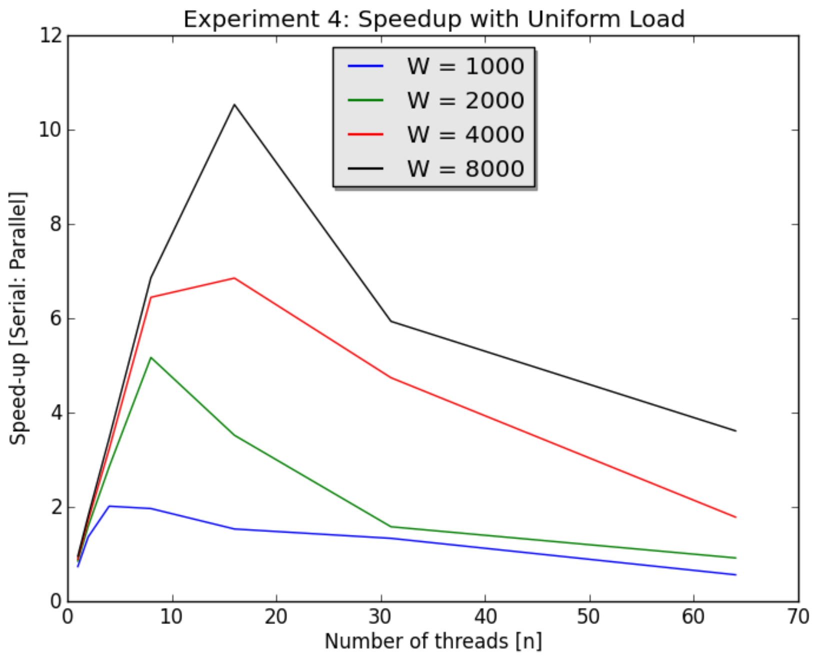

Figure 4: This figure illustrates the relationship between speedup, mean work and number of threads.

Hypothesis: Speedup will increase with n, the higher the W value, the sharper the increase.

Results: Speedup did indeed increase with n, up until $n = 16$ (as the test machine has 16 cores), then dropping off as the overhead of more queues detracts from the improvements of using all cores. W had a dramatic effect on the speedup as well, with $W = 8000$ out-performing $W = 1000$ by about a factor of 5. The disparities between W values are more exaggerated at $n = 16$ and much smaller for extreme values since optimal n values are better equipped to handle heavy work loads, whereas at small and large n values, the overhead costs overshadow parallel benefits. Maximum speedup was about 10 with $W = 8000$, $n = 16$. The minimum speedup was $W = 1000$, $n = 64$, with a speedup of about .7, which is consistent with obsrved performance in serial-queue. This means that large n values eventually provide no benefit whatsoever, as the dominant influence on performance becomes the added overhead time of working with queues.

To derive an expected speedup, consider that performance is limited by both the dispatch rate and the worker rate. A program can perform no better than the slowest of these two. Thus Parallel rate = min(WR, DR). Since in an ideal world, speedup would be n, expected speedup should be $PR = n \cdot (SR/PR) = n \cdot (SR/min(DR, WR))$.

3.4 Speedup with Exponential Load

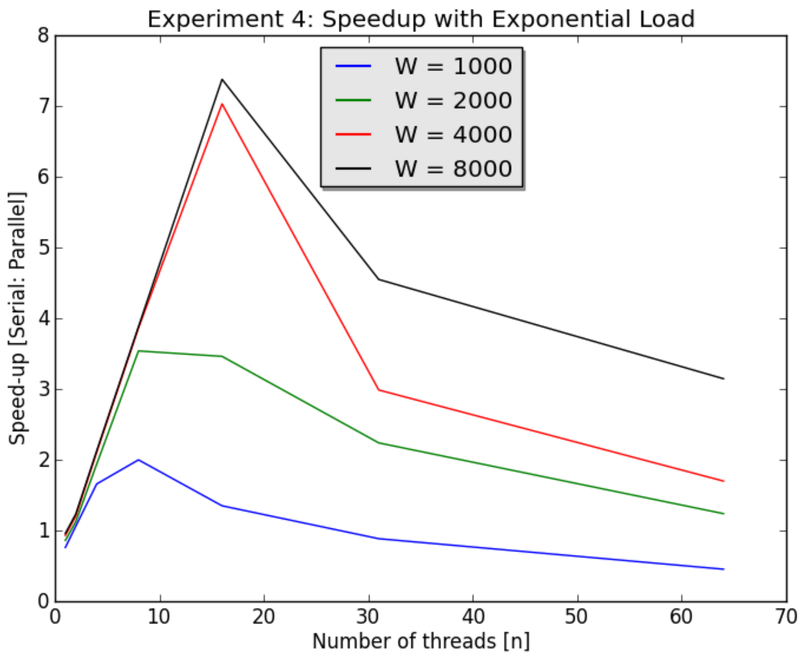

Figure 5: This figure illustrates the relationship between speedup, mean work and number of threads.

Hypothesis: This experiment will yield similar results to experiment 4, but with smaller speedups.

Results: The hypothesis is consistent with the data, which looks very similar to that of the uniform load test. The peak is again at $n = 16$ although the peak speedup is now about 8 times. Results for $W = 4000$ appear mostly unchanged, while the other W values showed a noticable decrease. It makes sense that the greatest W value would show the largest decrease given that exponential loads make it harder to load balance, and the greater the packet cost, the most costly the inefficient balancing. Once again, the minimum value is at $W = 1000$, $n = 64$. However, the minimum is far worse this time, with a speedup of under .5. However, although larger W values are more sharply affected by receiving an exponential load, the larger W values still perform strictly better than the lower, which shows that even with an unbalanced load, the benefits of parallelizing a large ammount of work offset the complications of using more threads.

4 Theory questions

- **Exercise 25**
  Yes. L2 specifies that method calls actually occur in their original order, while L1 requires only that the ordering preserves the original functionality of H. Removing L2 as a requirement leaves us with the ability to order method calls however we choose, so long as the resulting ordering brings about the same effects. This is simply sequential consistency.

- **Exercise 29**
  No. This statement in no way guarentees that an object is wait-free. An object is said to be wait-free if it is guarenteed that that it will return in a finite number of steps. In this case, any method call in H of x could take an infinite number of steps to return, thus the object cannot be wait free.

- **Exercise 30**
  Yes. This property guarentees that an object is lock-free. If an object is guarenteed to return an infinite number of method calls in an infinite history, some method must be completed infinitely often. In the worst case, this may be just one method returning again and again to the exclusion of all other calls. However, even this worst case satisfies the lock-free condition, which is that some method finishes in a finite number of steps. Thus, this property ensures lock-freedom.

- **Exercise 31**
  Since i is given to be a finite number 2^i must be a finite number as well. Let j be the maximum value of i. For any call, it is guarenteed to complete in $\leq 2^j$ steps. Thus this method must be bounded-wait-free.

- **Exercise 32**
  Line 15: Consider a case in which two simultaneous, or near simultaneous calls are made to enqueue(). If both reach line 15 at the same time, there is no guarentee that the tail will be properly incremented. Rather than being incremented twice, tail may only be incremented once. Now both threads believe they have stored space for themselves when there is in fact only room for one of them.

  Line 16: If tail is not properly incremented, both threads will try to enqueue to the same location. This may result in one value being overwritten and lost, with niether thread aware of the error. Alternatively, even if both increments go through, there is no guarentee of the order that each thread will enqueue sucessfully. If a thread calls enqueue first, it may still be only the second to have its data enqueued.