Parallel Load Balancing

Minke Zhang

February 4, 2014

1 Modules

Lamport Queue

The lamport queue will export the following functions–

```
struct lamport_queue_t {
  int size;
  Packet_t **elem;
  ...
} q;

q *q_init(int depth);
void enq(q *q, Packet_t *pkt);          // called only by the dispatcher thread
Packet_t *deq(q *q);                    // called only by the parent worker thread
int is_full(q *q);
int is_empty(q *q);
```

Workers

Each worker is responsible for processing data from one source, which is stored in the associated queue–

```
struct worker_t {
  int p_remaining;                      // packets which have yet to have been supplied
                                        //   by the dispatcher
  q *q;
  int is_done;                          // !p_remaining && is_empty(q)
  ...
} worker;

worker *init_worker(int q_depth, int T);  // initializes the worker and the associated queue
void *execute_worker(void *args);         // thread handler for the worker
long process_packet(worker *);            // blocks until w->q is not empty and processes
                                          //   next packet
```

Dispatcher

The dispatcher thread is responsible for relaying data from the provided `packet_source_t` provider–

```
struct dispatcher_t {
  worker **workers;                        // list of worker data structs
  ...
} dispatcher;

dispatcher *init_dispatcher(...);
void *execute_dispatcher(void *args);      // thread handler for the dispatcher
                                           // loops through the list of workers and assigns
                                           //   non-full queues with an appropriate packet
                                           // skips fully-queued workers
                                           // exits when sum(w->is_done) equals the number
                                           //   of workers
```

Executable

The main executable should have the following format–

```
struct result_t {
  float time;                              // the main return result
  long fingerprint;                        // the fingerprint of the data generated -- used for
                                           //   correctness testing
  ...
}

result *(*firewall_execute) (int T, int n, long W, int is_uniform, short id);
```

2 Hypotheses

NB: we define *speedup* σ in this document as the relative execution speed of the respective execution method versus the serial (reference) implementation. Thus, if we say that the speedup of implementation X was 0.8, we are stating that implementation X is slower than the serial implementation under identical executable parameters $\mathcal{P} = (n, T, D, W)$.

Parallel Overhead

As was the case with the Floyd-Warshall parallel implementation, we would expect a certain `pthread` overhead generated– the relative contributing factor of the overhead however, will depend on how much time the executable will spend processing actual packet data. As such, we would expect that σ would approach unity as $W \to \infty$ and as $n \to 1$.

Dispatcher Rate

Here we are testing the efficiency of the dispatcher when tasked with distributing packets across multiple workers. In the case that a worker queue is full, the expected behavior of the dispatcher is to skip over the worker and attempt to distribute the appropriate packet to the next worker– as the `worker` data structure has the `p_remaining` property which counts the number of packets still left from the global packet number, the dispatcher can simply refuse to decrement the counter and will know to add extra packets to the appropriate worker once the queue is free.

If the dispatcher is fully parallelized, we expect that the throughput to increase linearly with the number of threads. However, due to the serial nature of the dispatcher, we will expect an asymptotic falloff away from the linear relationship as $n \to \infty$.

Speedup with Uniform Load

The worker rate represents work which is parallelizable, whereas the dispatcher rate represents work which cannot be parallelized. We can thus use Ahmdal's law to approximate the expected speedup as a function of thread count. However, we expect that the expected speedup is an *upper* bound to the measured curve, as thread overhead is nontrivial in the cases to be tested.

Speedup with Exponentially Distributed Load

We would expect the speedup to be significantly less than the serial implementation, as the load imbalance would mean unlucky threads are spending far more time over their equally-distributed number of packets than the threads with lower tids.

3 Data & Analysis

Parallel Overhead

We see that the relative speedup of the serial queue implementation (representing the parallel overhead) increases towards unity as $W \to \infty$, as is expected– the relative amount of work done by `getFingerprint` increases as W increases, which leads to the overall decreased contribution of time allotted in initializing the parallel data structures.

Dispatcher Rate

The dispatch rate represents the rate at which the dispatcher thread is able to hand out packets to the workers; we see that the logarithmic nature of the graph implies there are diminishing returns for increasing the number of threads, as the serial nature of the dispatch thread means we cannot process packets faster than the dispatcher can hand out– the bottleneck here limits the fundamental utility of our program.

Speedup with Uniform Load

We see that the spread of speedup ranges at $n = 32$ is much wider than the spread at $n = 1$– this is expected, as the total amount of time spent in the $getFingerprint$ call becomes a significant portion of the running time of the program for high W, which as a result, benefits more parallelization.

Given the dispatcher rate dW and the worker rate rW, the max rate at which packets can be processed is $\min(rW(W), dW(n))$; plugging the appropriate values of W and n, we arrive at the predicted speedups

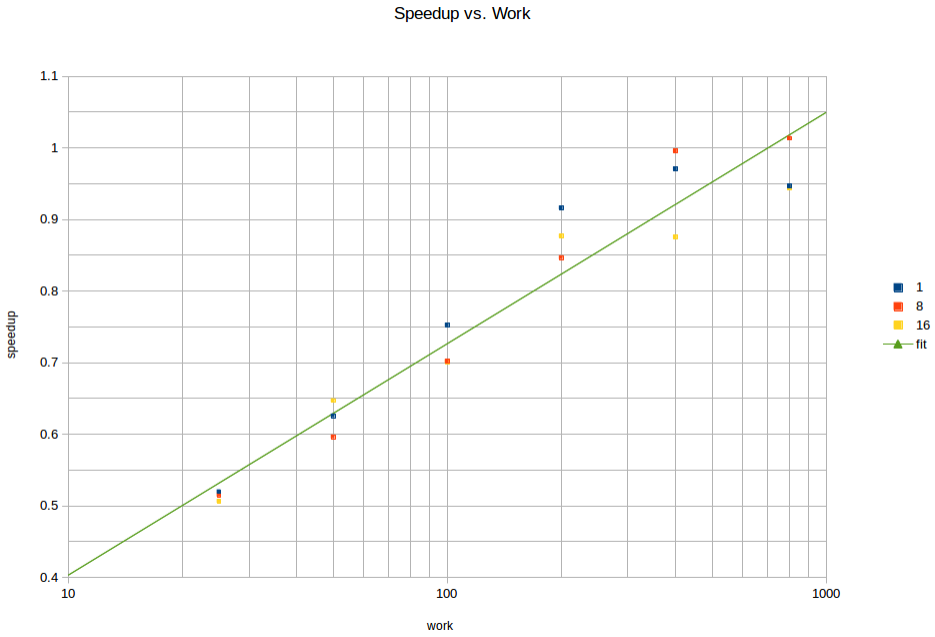

Figure 1: The parallel overhead is demonstrated by comparing the running time of `serial_queue` vs. `serial`. As more time is spent on processing the packet (work), we see the parallel data struct overhead contribute to less of the running time.

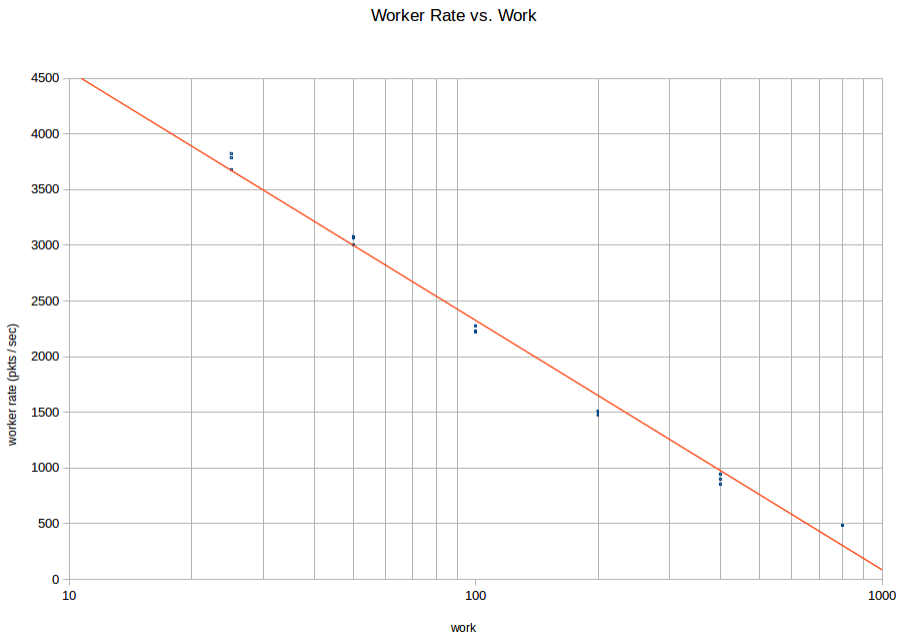

Figure 2: Total packets processing rate is plotted against the work involved with processing each packet. Line of best fit

$$wR(W) = 0.1404 \ln(W) + 0.0797$$

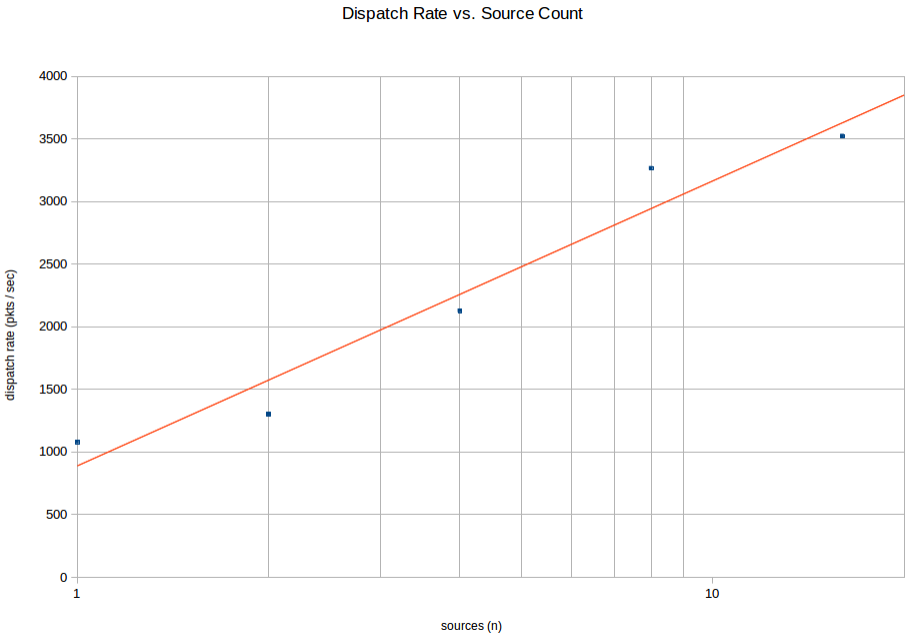

Figure 3: Packet rate being dispatched as a function of the number of threads. We see that the rate levels off (as is the characteristic of a logarithmic function) as $n \to \infty$. Line of best fit

$$dR(n) = 987.8244 \ln(n) + 890.3917$$

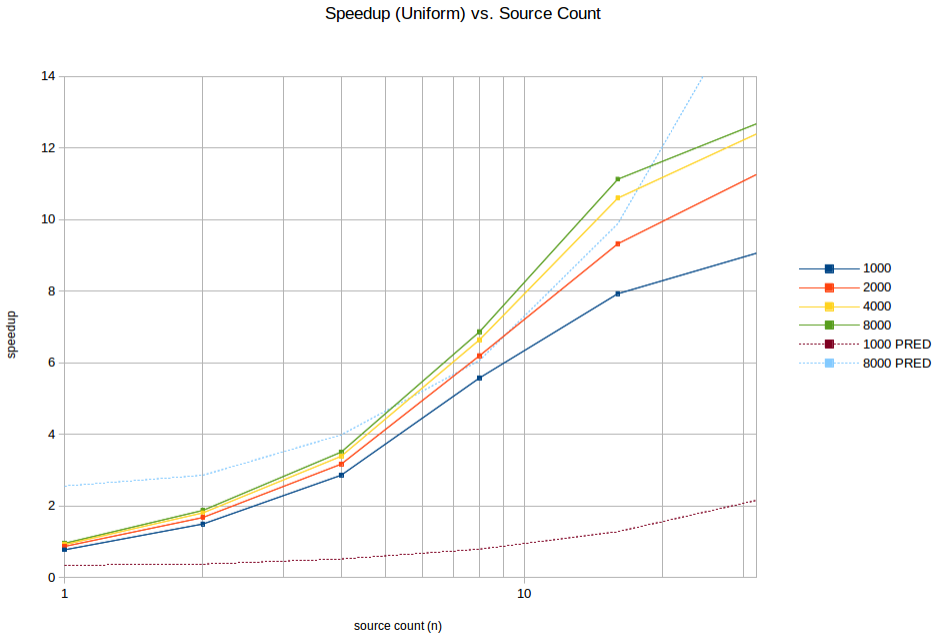

Figure 4: Parallel execution speedup (uniform packet distribution) as a function of source count.

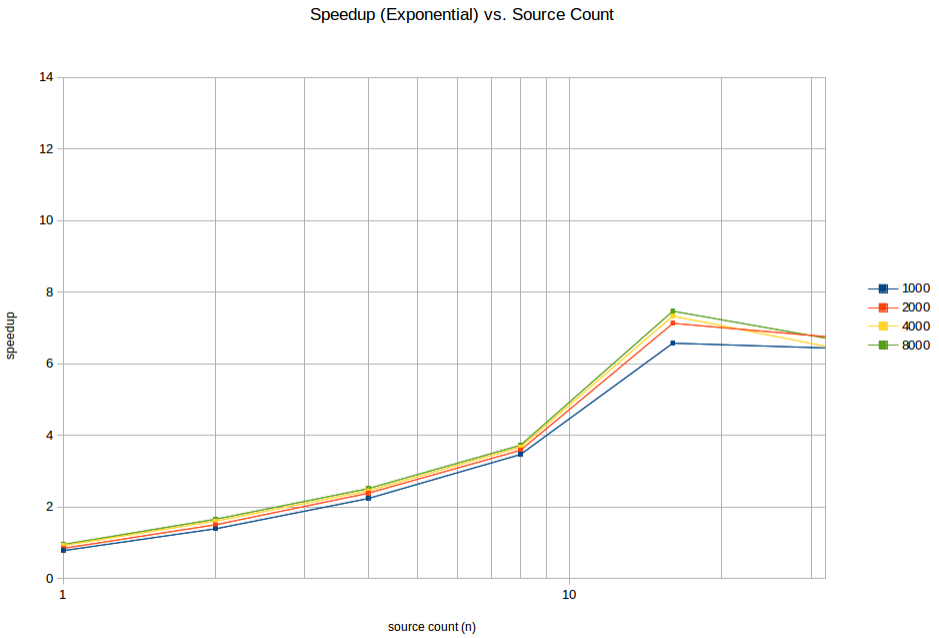

Figure 5: Parallel execution speedup (exponential packet distribution) as a function of source count.

as shown by the dotted lines in Fig. (4).

Speedup with Exponentially Distributed Load

Comparing Fig. (5) with Fig. (4), we see that the speedup rates for an exponentially-distributed workload to be much less than that of the uniformly-distributed workload– at $n = 16$, the relative speedup ratio between exponential and uniform distributions is $\sim 77.8\%$; furthermore, we see that the relative speedup ratio is $\sim 70\%$ at $n = 32$. This is to be expected, as was stated in the hypothesis, as load balancing in our parallel implementation cannot handle the workloads given to us.

4 Testing

Queue Integrity

We would like to ensure that the queue can be safely modified simultaneously by an enqueuing thread and a dequeuing thread– as such we would like to implement a simple test model–

```
struct blob_t {
  q *q;
  Packet_t packets[DIM];
} blob;

typedef void *(*func) (void *) thread_handler

void test_q(thread_handler) {
```

```
  pthread_create(..., thread_handler, ...);
  for(int i = 0; i < DIM; i++) {
    Packet_t p = deq(b->q);
    ASSERT(getFingerprint(p->...) == getFingerprint(b->packets[i]->...));
    ...
  }
}

void *enq_handler(void *args) {
  blob *b = (blob *) args;
  for(int i = 0; i < DIM; i++) {
    q_enq(b->q, b->packets[i]);
  }
  ...
}

void *enq_handler_sleep(void *args) {
  blob *b = (blob *) args;
  for(int i = 0; i < DIM; i++) {
    enq(b->q, b->packets[i]);
    sleep(1);
  }
  ...
}
```

Packet Distribution

The packets generated by calls within the dispatcher are initialized with the destination worker id as an input– we can thus test packet distrubution by

1. generate packets using exponential workload,
2. assign the *wrong* packets (that is, the hardest packets to a thread which should otherwise expect an easy load),
3. use `stopwatch` to clock the execution times of each thread,
4. see if the expected (wrong) thread is hit with a performance loss

Worker & Dispatcher Quality Assurance

We will use the fingerprints of the processed results to identify if the results given are correct; we will check `parallel_firewall` and `serial_queue_firewall` against the results of `serial_firewall`–

```
void *test_correctness(void *args) {
  ...
  result *p = serial_firewall(t, n, w, uniform_flag, i);
  result *q = parallel_firewall(t, n, w, uniform_flag, i);
  result *r = serial_queue_firewall(t, n, w, uniform_flag, i);
  ASSERT(p->fingerprint == q->fingerprint);
  ASSERT(q->fingerprint == r->fingerprint);
  ...
}
```

5 Appendix

Optimization After consultation with Jake Whitaker, I implemented the worker thread behavior to sleep instead of busy wait; this significantly sped up the runs for $n \geq 16$, as the core is not being burdened with running threads which did nothing.