Parallel Load Balancing

Minke Zhang

February 3, 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.

Speedup with Increased Queue Depth

We shall measure the speedup of the parallel implementation at a set T, W, n, but with varying D– we shall use the uniform workload packets, as they are the most predictable in terms of parallelizability.

The key function of the queue is to act as a buffer so that the worker threads will not have to wait for the (serialized) dispatcher to hand out packets constantly– thus we would expect that as D increases, the dispatcher can simply fill up the queue and run idle (that is, no longer contribute to the non-serializable work load in Amdahl's law), and thus would *increase* the speedup, when compared to the same results from **Speedup with Uniform Load**.

3 Data & Analysis

Parallel Overhead

Dispatcher Rate

Speedup with Uniform Load

Speedup with Exponentially Distributed Load

Speedup with Increased Queue Depth

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);
  ...
```