

# Parallel Load Balancing

Minke Zhang

January 28, 2014

## 1 Modules

### Lamport Queue

The lamport queue will export the following functions—

```
struct lamport_queue_t {
    int depth;
} lq;
lq *lq_init(int depth);
void lq_enq(lq *q, struct packet_source_t *pkt);    // called only by the dispatcher thread
struct packet_source_t *lq_deq(lq *q);              // called only by the parent worker thread
```

### 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 remaining to supply
    lq *q;                     // the queue
} worker;
worker *worker_init(int T);    // initializes worker_t struct with p_remaining = T
void process_next_packet(worker *); // pops an item from the source lq and process data
```

### Dispatcher

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

```
struct dispatch_t {
    worker *threads;           // list of all threads
} dispatch;
dispatch *dispatch_init(int n, int T, int go_parallel); // spawns n worker threads to handle
                                                         // packets-- executes serial-queue if
                                                         // go_parallel = false void
enqueue_next_packet(int difficulty); // gets a new packet_source_t (with
                                     // difficulty get_constant_packet,
                                     // get_uniform_packet, or
                                     // get_exponential_packet), if queue
```

```
// is empty and assigns to specified
// worker
```

## Executable

The main executable should have the following format–

```
void execute(n, T, D, W, int use_serial); // spawns the dispatcher thread, or use the
// provided serial implementation
```

## 2 Hypotheses

**NB:** we define *speedup*  $\sigma$  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  $\sigma$  would approach unity as  $W \rightarrow \infty$  and as  $n \rightarrow 1$ .

## 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  $n \rightarrow \infty$ .

## 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 Testing

### Queue Integrity

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

```
struct blob_t {
    lq *q;
    packet_source_t packets[];
} blob;

void test_lq() {
    pthread_create(..., enq_handler, ...);
    pthread_create(..., deq_handler, ...);
}

void *enq_handler(void *args) {
    blob *b = (blob *) args;
    for(int i = 0; i < CONST; i++) {
        lq_enq(b->q, b->packets[i]);
        sleep(10);
    }
    pthread_exit(NULL);
}

void *deq_handler(void *args) {
    ...
    for(int i = 0; i < CONST; i++) {
        p = lq_dec(b->q);
        assert(p == b->packets[i]);
    }
    pthread_exit(NULL);
}
```

### 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 distribution 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

If the previous tests pass, then the parallel implementation workers and dispatchers will be handling the same data as the serial implementation, and without race conditions– we can thus pass the same packets to the serial implementation, and compare the output to ensure that the two results match.