ECE 459 W18 Midterm Solutions

P.Lam, J. Zarnett

February 12, 2018

(1)

(a) With good luck, the answer is 80 000. With bad luck, it will be some other value 79 999 or less.

(b) Moving time is 0.5 hours, or $30 \times 60 = 1800s$. At each stop light, expected waiting time is 0s if green (analogous to a cache hit), 5s if red. The miss rate is $1/6$. The expected number of misses is $10/6$ and multiplying by $5s$, we see an expected wait time of $50s/6 = 81/3s$. Hence the expected total time is $18081/3s$.

(c) Reading a value from a sensor: the latest value is the one that is desired; a a newer value should not be overwritten by an older one.

(d) The correct answer, according to `computers-are-fast.github.io` is 342 000 000. So we would accept any answer between 100 000 000 and 1 000 000 000.

(e) This code allocates the `struct params` variable on the stack. When the next iteration of the loop is executed the same stack location could be re-used meaning that the thread could be accessing memory that's been overwritten when it runs. To fix it, allocate the memory for `p` on the heap (ie with `malloc()`.

(f) The `single` directive means that the code following it will be executed only by one (single) thread. But it could be any of the created threads. If it was instead `master`, that block of code could be executed only by the main thread of the program. (We didn't explicitly ask for a comparison to `master`, so just the first sentence is required.)

(g) You use a deterministic program where the number of branch statements is known to be n. If every branch is predicted correctly the program will take b cycles, each mispredicted branch comes with a cost of x cycles, and therefore a measured runtime will be $(b + m \times x)$, where k is the number of mispredicted branches. Solve this equation to find k and the misprediction rate: k/n.

(h) Gustafson's law allows the use of more fine-grained grids. Amdahl's law says that the minimum time to make a prediction for a given grid density is bounded.

(i) Extra unused threads consume extra resources, e.g. stacks and return value. [Not deadlock!]

(j) Valgrind will report an invalid write (probably on `p->z`) because `sizeof(struct point*)` is the size of the pointer and not the size of the structure type to which `p` is supposed to point. Fix it by changing the parameter given to `sizeof` to be `struct point` (NO $*$!) instead.

(2.1)

Start by writing a deterministic program that takes time L and runs for a sufficiently long time. Make sure that L is embarassingly parallel with negligible serial part, e.g. something that computes on a large array with no dependences between elements. Now create L_{2N} which divides the same work as L over $2N$ threads; the system must switch threads when running L_{2N} in either user-level or kernel-level threading models. We can trivially distinguish user-level and kernel-level threading by seeing whether L_{2N} takes elapsed time close to L or time close to $\frac{L}{2N}$. I'll abuse notation to mean elapsed times when I write L and L_{2N} below.

Under the user-level threading model, we would expect the difference in elapsed times $L_{2N} - L$ to be the time spent switching threads. Divide by # thread switches to get a time per thread switch.

Under the kernel-level threading model, and for L as described, the elapsed time for L_{2N} should be $\frac{L}{2N}$ + thread switching time. So this time, compute per-switch time as

$$\frac{L_{2N} - L/(2N)}{\text{\# switches}}.$$

(2.2)

Part 1. Problems with this loop:

- The loop bounds are not constant. Fix by making length parameter an int and not a pointer, or dereferencing it into a temporary variable at the beginning of the function.
- There is the possibility that `c` overlaps `a` or `b`. Fix it by declaring all three as `restrict`.

```
void add_arrays( int * restrict c, int * restrict a, int * restrict b, int * length ) {
  int l = *length;
  for( int i = 0; i < l; ++i ) {
    c[i] = a[i] + b[i];
  }
}
```

Part 2. No, this solution is not sufficient. Suppose the loop that we want to work on has pointers `a`, `b` passed to `bar` as parameters and neither are modified. It is, however, still possible that function `bar` has arbitrary side effects. If `bar` updates some other data structure (e.g., a linked list where a global variable pointer is used to reach it) and concurrent modifications of that linked list can lead to inconsistent state. Therefore, the compiler should refuse to automatically parallelize this, because doing so is unsafe.

(If you are interested: if the compiler can instead work out that a function `bar` is *pure* – has no side effects – then autoparallelization should be able to proceed.)

If you instead said that the promise as specified is not transitive – that `bar` can call another function `baz` which *does* modify, for example, `*a`, that is worth some (but not all of the) marks. Analyzing the call stack is possible, although could be difficult in a complex program.

(3)

```
void main() {
  add_cb(START, start);
  wait_for_events();
}

void start(int m, void * p) {
  if (m != START) abort();
  print ("start");
  clear_cb(START);
  add_cb(VISIT, visit);
}

void visit(int m, void * p) {
  if (m == END) {
    print ("end");
    clear_cb(VISIT);
    return;
  }
  print ("visit␣", (int)p);
}
```

(4)

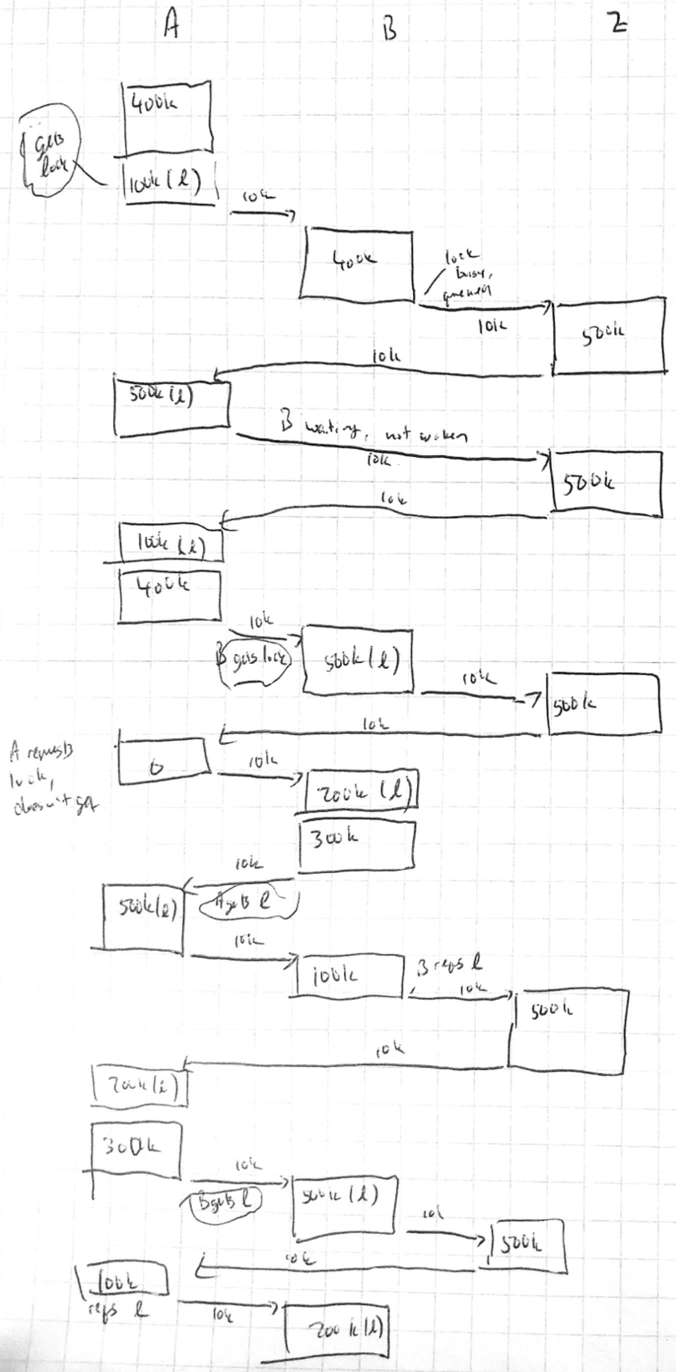
```
int main( int argc, char** argv ) {

  /* Initialize Parallel Section */
  #pragma omp parallel
  {
    /* This part should only be done by the master thread (Dobby) */
    #pragma omp master
    {
      struct job* jobs = prep_tasks();

      for( int i = 0; i < NUM_TASKS; ++i ) {
        /* Dispatch a task; the untied part is nice but optional */
        #pragma omp task untied
        {
            do_work( jobs[i] );
            /* Avoid race condition by atomic operation */
            #pragma omp atomic
            completed++;
        }
      }
      /* Wait for all created tasks to be finished */
      #pragma omp taskwait
      free( jobs );
    }
  }
  return 0;
}
```

It would also be OK to busy-loop comparing `completed` with `NUM_TASKS` as long as you avoid race conditions.

(5)

You may include additional switches to Z as desired.
You don't have to start with A (but why wouldn't you?)

(l) = holds lock



Context switches = 17 x 10k = 170k

thread Z = 2500k

thread A = 2600k

thread B = 2200k