ECE 459 W16 Final Solutions

J. Zarnett

April 15, 2016

(1) (a) The advantage is faster execution (less function overhead) but increased binary size (and therefore more risk of cache misses for that).

(b) In an open system, arrivals are independent of departures (in a closed system, as soon as a job leaves, a new one arrives immediately).

(c) This avoids the overhead in creating and destroying threads repeatedly and can improve response time since it is not necessary to wait for thread creation.

(d) Nonblocking I/O reduces the risk of deadlock or race conditions, and reduces the amount of threads (kernel resources) necessary. It also gives us more control over the scheduling (whereas threads are run by the OS scheduler).

(e) Coarse grained locks force more serialization on the program (threads spend lots of time waiting for the lock); fine grained locks increase the risk of deadlock or other synchronization issues.

(f) With try lock, a thread is not blocked if it does not acquire the lock. Therefore, it can release any locks it does have, so no deadlock will occur.

(g)

```
int tmp;
void swap( int*x, int* y) {
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Cute and minimalist answer:

```
void foo() {
 printf("Hi");
}
```

(h) Missing points can be filled in with interpolation (i.e. average some adjacent values, for example). This requires that the function be "nice" – continuous in the missing segment (and differentiable, if we want to think back to first year calc).

(i) The arrival rate represents the rate at which jobs (requests/etc) arrive at the system. Throughput is how much work the system accomplished work in a certain period of time. If the arrival rate is less than the maximum throughput, the arrival rate limits the throughput.

(j) The `mfence` instruction performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the mfence instruction. This serializing operation guarantees that every load and store instruction that precedes in program order the mfence instruction is globally visible before any load or store instruction that follows the mfence instruction is globally visible. (So says Intel!)

(2a)

Any loop where different iterations take varying amounts of time. The following loop is ridiculous (why would you ever do this...) but for higher values of i then more work is done...

```
for (int i = 0; i < 10000; ++i) {
    for ( int j = 0; j < i; ++j) {
        x += j;
    }
}
```

(2b)

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid)
```

(3 marks) Starts a parallel region, declaring variables a, b, nthreads, locka, lockb as shared between all threads and tid as private to each thread.

```
#pragma omp master
```

The following block is executed only by the master thread.

```
#pragma omp barrier
```

Threads wait at this point until all threads reach that point.

```
#pragma omp sections nowait
```

(2 marks) Declares the beginning of parallel sections and indicates that sections don't need to wait for one another to finish before the code can proceed.

```
#pragma omp section
```

Defines a parallel section.

(2c)

```
void *store_wrapper( void* arg );
void *update_wrapper( void* arg );

void post_process( void* message ) {
    int* error_code;
    pthread_t store_thread, update_thread;

    pthread_create(&store_thread, NULL, store_wrapper, message);
    pthread_create(&update_thread, NULL, update_wrapper, message);

    pthread_join( store_thread, (void**) &error_code );
    if ( 0 != *error_code ) {
        pthread_kill( update_thread, 9);
        pthread_create(&update_thread, NULL, update_wrapper, message);
    }
    pthread_join( update_thread, NULL);
    free( error_code );
    pthread_exit(NULL);
}

void *store_wrapper( void *arg ) {
    int *retval = malloc( sizeof ( int ) );
    *retval = store_message( arg );
    pthread_exit( retval );
}

void *update_wrapper( void *arg ) {
    update_statistics( arg );
    pthread_exit( NULL );
}
```

You could put `error_code` in a global variable and avoid the complexities of getting the value out of the join call. That's fine too. If you didn't write the function prototypes at the top above `post_process` that's also fine (the compiler will moan about it though). Remember to deallocate all memory you allocated!

(3a) The slow operation is the SHA256 hash operation, therefore the hash computation would take place in the kernel. The host code sets up the problem. The possible inputs could be generated in host code, but it would make sense to generate them in an OpenCL kernel as well. So the input password state as well as the hashed password result are given in to the kernel. Once the setup is run, the SHA256 computation runs in parallel and compares the computed hashes against the password's has. A way of indicating a successful result is needed; the result can be sent back as a buffer, or, simply thrown to the console with a printf.

(3b) A: read only; B: read only; C: write only.

```
__kernel void mat_mul( const int Mdim, const int Ndim, const int PDim,
                       __global float *A, __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    for ( k = 0; i < Pdim; k++ ) {
        C[i*Ndim+j] += A[i*Ndim+k] * B[k*Pdim+j];
    }
}
```

The problem as written doesn't use Mdim correctly as part of the computation of the index into C. It doesn't cause any errors in the program if Mdim = Ndim which is the case in the host code that goes with this. Correcting that on the exam is okay, but not necessary.

(4a)

Steps: The code is compiled, run with instrumentation to find out what is likely to happen (branches taken/not taken, for example), and then the code is recompiled with this additional data to optimize the code.

```
if (x < 0) {
   foo();
} else {
   bar();
}
```

At compile time, it's not clear which function is more likely to be invoked. With profiling data, we may find that bar() is called 90% of the time, so the code will then be optimized for the case that is most likely to happen (bar).

(4b)

Function prologue and epilogue take nonzero time; this is the setup like allocating space on the stack for declared variables or moving return values off the stack to the calling function. If a sample takes place during those times, it gets charged to the function but not to any actual executable statement thereof, hence it appears at the beginning or end.

(4c)

Take a CPU profiler to the code and get a breakdown by function for a sufficiently representative run of the code. Then you can examine where the code spends the majority of its time. Suppose that the runtime of the code is dominated by functions you cannot modify (in assignment 4, `execute` or `generate`). To prove that a speedup of 2.0 is impossible, imagine the runtime for all the functions you can modify is zero, and calculate the speedup for that scenario. If the speedup in that case is less than 2.0, then you know that it is impossible to achieve a speedup of 2.0 no matter how you modify the code.

(5a) 1. Bernoulli.

2. Binomial.

3. $50 = n(0.95) \to 52$. (I said round down but some said 53...?)

(5b) 500 000 per hour = 138.889 arrivals per second.

$\rho = \lambda \times s = (500000)/3600 \times 0.004 = 0.556 \to 55.6\%$

$T_q = \dfrac{s}{(1-\rho)} = 0.004/(1 - .556) = 0.009$

$W = \dfrac{\rho^2}{1-\rho} = (.556)^2/(1 - .556) = 0.696.$

Max arrival is $\rho = 1$ so $1 = \lambda \times .004 \to 250$.

Now s is .003.

500 000 per hour = 138.889 arrivals per second.

$\rho = \lambda \times s = (500000)/3600 \times 0.003 = 0.417 \to 41.7\%$

$T_q = \dfrac{s}{(1-\rho)} = 0.003/(1 - .417) = 0.005$

$W = \dfrac{\rho^2}{1-\rho} = (.417)^2/(1 - .417) = 0.298.$

Max arrival is $\rho = 1$ so $1 = \lambda \times .003 \to 333.33$.

(5c)

Throughput is simply transactions per second:

$X_1 = 25000/(8 \times 60 + 9) = 51.125$

$X_2 = 25000/(58) = 431.034$

$X_0 = (51.125 \times 431.034)/(51.125 + 431.034) = 22035.613/482.159 = 45.702.$

(5d)

The write master centralizes all writes to provide serialization and consistency – that is, all databases see writes in the same order because the write master is enforcing its version of ordering. The write master does not handle any reads, so the reads do not slow down the execution of the writes.

The purpose of the slave is for redundancy; if the master should fail. It will take over and be the write master if for some reason the master is unavailable. This improves availability and means the application or site does not go down in the event of a failure of the write master.

There are no Read-After-Read dependencies and therefore reads do not need to lock tables to execute, so a pure database read can take place on the read replicas much faster than they could on a database where writes are also taking place. If a read needs tables A and B, and a write currently has tables B and C locked, the read must wait.