ECE 459 Final Exam Solutions

J. Zarnett

April 19, 2017

(1)

(a) Inlining decision is acceptable as an answer, or devirtualization.

(b) This option tells the compiler it does not need to follow IEEE standards in floating point calculations, enabling some optimizations that reduce the accuracy but increase the speed.

(c) If the outer loop has very few iterations it might not profitable to parallelize that loop. Example in code:

```
for (int i = 0; i < 2; i++ ) {
    for (int j = 0; j < 60000; j++ ) {
        ...
    }
}
```

(d) `#pragma omp master` defines a section of code that should be run by a single thread only, the master thread of the team.

(e) The `mfence` instruction can have this result: it triggers a pipeline flush and that is attributed, generally, to the instructions being flushed, rather than the `mfence` instruction itself.

(f) If memory is being allocated repeatedly increasing the amount of memory in use, then the profiler would have to constantly overwrite the peak snapshot after each allocation. If there has been a deallocation, however, the peak snapshot is valid and should be kept.

(g)

Thread 1:

```
lock(b);
lock(a);
/* Critical Section */
unlock(a);
unlock(b);
```

Thread 2:

```
lock(a);
lock(b);
/* Critical Section */
unlock(b);
unlock(a);
```

(h)

$s = 3$, $\lambda = 0.25$, so $\rho = .25 \times 3 = .75$, so $T_q = s/(1-\rho) = 3/(1-.75) = 12$. Average completion time is therefore 12s.

(i) Take a copy of the value of `x`:

```
int x_copy = x;
int y = f(x_copy);
int z = g(x);
```

(j)

If any detached threads have been created, a `return` statement at the end of main will terminate them, but `pthread_exit` means the program waits until those detached threads are finished.

(2)

Line #	**Code**	**What it does**	**pthread equivalent**
7, 8	`omp_lock_t`	Declares a lock	`pthread_mutex_t`
16, 21	`omp_test_lock`	Tries to acquire the lock	`pthread_mutex_trylock`
18, 23	`omp_unset_lock`	Releases a held lock	`pthread_mutex_unlock`
32, 33	`omp_init_lock`	Initialize a lock	`pthread_mutex_init`
36	`#pragma omp parallel private(id)`	Declares and begins a parallel section with variable `id` as private to each thread and uninitialized, with an implicit barrier at the end of the section	Looped `pthread_create`, followed by looped `pthread_join`
38	`omp_get_thread_num`	Retrieves the identifier of this thread	No pthread equivalent
43, 44	`omp_destroy_lock`	Clean up a lock	`pthread_mutex_destroy`

One mark each for the boxes, but the what-it-does for the OMP directive is worth two marks (one for the parallel section, one for the variable).

(3)

Part 1 Relevant parts of the host code (nothing else needs changing):

```
// Create buffers
cl::Buffer output = cl::Buffer(context, CL_MEM_WRITE_ONLY, iterations * sizeof(cl_int) );
// Write buffers
queue.enqueueWriteBuffer(output, CL_TRUE, 0, iterations * sizeof(cl_int), results );

// Set arguments to kernel
kernel.setArg(0, output);

// Run the kernel on specific ND range
cl::NDRange global( iterations );
cl::NDRange local(1);

queue.enqueueNDRangeKernel(kernel, cl::NDRange(), global, local);

// Read buffer
queue.enqueueReadBuffer(output, CL_TRUE, 0, iterations * sizeof( cl_int ), results );
```

Part 2 The kernel:

```
#define INT_MAX 4294967296

__kernel void montepi( global int * output  )  {
    int i = get_global_id(0);
    float x = (float) random( i ) / INT_MAX;
    float y = (float) random( i ^ random( i ) ) / INT_MAX;
    output[i] =  (x*x + y*y) <= 1;
}
```

(4)

The two scenarios that could cause a problem are (1):

```
swap( &a, &a );
```

and (2)

```
swap( &a, &b );
```

In thread 1 while at the same time in thread 2:

```
swap( &b, &a );
```

The solution to problem 1

```
void swap(container_t * x, container_t * y) {
    if (x == y) { // Fix for Scenario 1
        return;
    }
    // Fix for the second scenario: acquire both locks beforehand
    pthread mutex lock(&x->lock );
    while ( pthread_mutex_trylock( &y->lock ) != 0) {
        pthread_mutex_unlock( &x->lock ); // Let another thread have a chance
        pthread_mutex_lock( &x->lock );
    }
    int temp = x->data;
    x->data = y->data ;
    pthread_mutex_unlock( &x->lock );
    y->data = temp;
    pthread_mutex_unlock( &y->lock );
}
```

Any trylock solution that avoids deadlock in scenario 2 is sufficient; doesn't have to be exactly like this.

Grading notes:

- 2 marks for finding problem 1
- 4 marks for finding problem 2
- 4 marks for fixing problem 1
- 10 marks for fixing problem 2

(5A)

Device	**Data/Hour**	λ	S	V	ρ	$V \times S$
Transactions	22 500	6.25		1		
CPU	1 743 750	484.375	0.0006	77.5	0.30	0.0465
RAM Disk	765 000	212.5	0.002s	34	0.425	0.068
Solid State Drive	945 000	262.5	0.003s	42	0.78	0.126
Network	11 250	3.125	0.025s	0.5	0.078	0.0125

Bottleneck device is the one with the highest utilization: SSD

Maximum rate of transactions = 1/.126 = 7.94 transactions per second.

Average transaction time = sum of the $S_i \times V_i$ columns = 0.253 s

Values may be reported in other units as long as they are consistent.

(5B)

Both Alex and Taylor are correct if the system is ergodic. Is it? Ergodicity requires (1) irreducibility, (2) positively recurrent, and (3) aperiodic.

Point 1 is satisfied; state is the number of jobs in the system and the initial state does not matter, we can get to any state.

Point 2 is also satisfied because we can revisit the "queue empty" state infinitely often in an infinitely long run.

As for Point 3: The program is aperiodic if random number generation is sufficiently random and we can assume that it is.

Jordan is wrong, however, because this is a very small number of jobs and the system is probably not at steady state when that executes. Initial conditions have not attenuated so this is not a good approach.