Programming for Performance (ECE459): Midterm
February 27, 2013

This open-book midterm has 5 pages and 4 questions, worth 25 points each. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

1 Short Answer

Answer these questions using at most three sentences. Each question is worth 2.5 points.

1. Which is faster, `#pragma omp single` or `#pragma omp master`? Why?
2. You are running a Monte Carlo calculation on an otherwise-unloaded 8-core machine. Explain why performance goes down when you start 9 threads instead of 8.
3. You are running a simple web server, again on an otherwise-unloaded 8-core machine. The web server works as follows: when a main thread accepts a connection, it dispatches a thread from a thread pool to respond to the request. Do you expect better throughput from a pool with 8 or 9 threads? Why?
4. Consider a C list: `typedef struct s { struct s* next; void * data; } S;`. How can you leverage parallelism in traversing this linked list?
5. What does the `volatile` qualifier ensure? What are two problems that it does not protect against?
6. Let `x = y = 0`. T1 contains `x = 1; r1 = y;` and T2 contains `y = 1; r2 = x;`. Show me one possible result from a weak consistency model that is not possible in sequential consistency.
7. Will you ever get a race condition from converting an OpenMP shared variable into a private variable? Why or why not?
8. If you change an OpenMP shared variable into a private variable, can the behaviour of the program change? If not, why not? If so, show me an example. ("Three sentences" applies to the explanation of the example.)
9. Here's a function prototype: `void foo(int * restrict p, int * restrict q);`. Write a call to `foo()` which does not respect the `restrict` qualifier, including necessary context.
10. Write down one condition when you'd parallelize a middle or inner loop rather than the outermost loop.

2 Zeroing a Register

Let's say that you are working at the x86 assembler level and you want to zero register `eax`. The most straightforward way is to write:

```
mov eax, 0
```

It turns out that this is fairly efficient: it takes one cycle to execute this statement and the throughput is 3 per cycle, which is the maximum integer throughput possible on a Sandybridge.

An alternative way of zeroing a register is by exclusive-oring it with itself:

```
xor eax, eax
```

Question (2a) (5 points). Why is there a performance advantage from using `xor` rather than `mov`? The cycle count and throughput are exactly the same. (Hint: what slows down modern processors?)

Now consider the following code fragment:

```
1       add eax, 1
2       mov ebx, eax        ; ebx <- eax
3       xor eax, eax
4       add eax, ecx        ; eax <- eax + ecx
```

Question (2b) (10 points). We'd like to issue instructions out-of-order. Data dependencies prevent that. Identify all data dependencies between instructions (1) through (4). Based on your list of dependencies, say why each pair of instructions could or could not execute out of order.

Question (2c) (10 points total). How can the processor execute instructions (1) and (3) out-of-order anyway? (5 points) Rewrite the list of four instructions to permit out-of-order execution. You may use new registers. (Optionally, explain what's going on at an architectural level behind-the-scenes). (5 points) Identify the dependencies in the rewritten instructions and say why each pair of instructions could or could not execute out of order.

3 Dynamic Scheduling using Pthreads

The goal of this question is to rewrite the following OpenMP code using Pthreads-like constructs. You may use the following primitives: 1) a queue data structure, which is not itself thread-safe (so you can enqueue and dequeue objects from the queue); 2) Pthreads mutual exclusion primitives; 3) Pthreads thread create, join and detach primitives.

I am looking for the use of the proper concepts. For full marks, use the appropriate Pthreads primitives in your solution. I'm not looking for the syntax to be exactly right, but I'm looking for the appropriate choice of primitives. Proper problem decomposition along with a handwavy pseudocode solution will get you about 15 marks out of 25—that depends on how precise your solution is.

Please do not hardcode the work distribution among threads. You can hardcode the number of worker threads to 4, if that makes your life easier (it shouldn't.)

```
1  double calc(int count) {
2    double d = 1.0;
3    for (int i = 0; i < count*count; i++) d += d;
4    return d;
5  }
6
7  int main() {
8    double data[200][200];
9    int i, j;
10   #pragma omp parallel for private(i, j) shared(data) schedule(dynamic, 50)
11   for (int i = 0; i < 200; i++) {
12     for (int j = 0; j < 200; j++) {
13       data[i][j] = calc(i+j);
14     }
15   }
16 }
```

4 Race Conditions and Memory Barriers

Consider the following (inefficient) function to determine primality. Given an input v, it iterates up to $\lfloor\sqrt{v}\rfloor + 1$, searching for factors of v.

```
1  // shared array; assume initialized to all 1's.
2  int pflag[LARGE];
3
4  int is_prime(int v) {
5      int i;
6      int bound = floor(sqrt ((double)v)) + 1;
7
8      for (i = 2; i < bound; i++) {
9          /* No need to check against known composites */
10         if (!pflag[i])
11             continue;
12         if (v % i == 0) {
13             pflag[v] = 0;
14             return 0;
15         }
16     }
17     return (v > 1);
18 }
```

Question (4a) (15 points total). Assume that function `is_prime()` is called concurrently from two threads. (5 points) Identify the race condition. (5 points) Using locks or otherwise, eliminate the race condition. (5 points) Argue that the race is benign: it never affects the output of the function.

Memory Barriers

Now consider the following C++ code.

```
1  class Singleton {
2  public:
3    static Singleton* instance() {
4      if (pInstance == 0) {
5        lock(l);                    // this does what you think it does
6        if (pInstance == 0) {
7          pInstance = new Singleton;
8        }
9        unlock(l);
10     }
11     return pInstance;
```

```
12    }
13
14  private:
15    static Singleton* pInstance = 0;
16    Lock l;
17  };
```

Question (4b) (10 points total + 1 bonus). This code attempts to implement a `Singleton` design pattern, but it doesn't really work in the presence of multiple threads. In particular, there is only supposed to be one `Singleton` object ever. (5 points) Show me one execution trace that gives the wrong result, and explain why it is wrong. (Hint: look at the title of the question.) (5 points) Propose a fix for the code and explain why it works. Anything correct will get you full points; it does not have to be efficient at all. (1 bonus point) Propose an efficient fix for this code.