Programming for Performance (ECE459): Midterm

March 6, 2015

This open-book midterm has 4 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

Each part is worth 5 points.

Part (a). **Describe** the difference between parallelism and concurrency. (3 lines max)

Part (b). Here's a nested loop which you could use `omp parallel for collapse` with. **Write** this loop as a single loop, obviating the need for `collapse`.

```
for (int x = 0; x < 3000; x++) {
  for (int y = 0; y < 4000; y++) {
    array[x*4000+y] = x-y;
  }
}
```

Part (c). Write a loop which contains a reduction (i.e. it would be meaningful to apply the OpenMP `reduction` clause to it). Rewrite the computation so that it could be safely parallelized. You can sacrifice efficiency here and share the reduction variable between threads (as long as you avoid race conditions).

Part (d). Weather forecasting works like this: given observations of the current weather state (fit to a grid), models use fluid dynamics and thermodynamics equations to predict future weather states for each grid point. Forecasters use a number of weather models and combine the results. **Explain** how Gustafson's Law allows us get better weather forecasts, and also how Amdahl's Law limits weather forecasting.

Part (e). Assume that a horse-sized duck can waddle at 4 km/h, and it can carry 30kg at a time. On the other hand, a duck-sized horse can trot at 8 km/h, and it can carry 2kg at a time. But, of course, you have 100 such horses. Both ducks and horses make constant progress towards their destinations at the stated speed. There is no synchronization (or loading/unloading) overhead.

You need to carry 60kg of stuff from point A to point B. The distance between these points is 8km. (You can divide the stuff into smaller packets). **How long** does it take to carry this stuff from A to B for the duck and for the horses?

2 Race conditions

Consider the following OpenMP code. It contains a race condition. (5 points:) **Explain** the race condition. (15 points:) **Describe** how to fix the race condition. Don't use locks or atomics. (5 points:) **Explain** what you would do to determine whether it is profitable to parallelize the loop.

```
#pragma omp parallel for collapse(2)
{
  int r, c;
  for(r=0;r<rows;r++) {
    for(c=0;c<cols;c++,pos++) {
      int sq1 = (int)delta_x[pos] * (int)delta_x[pos];
      int sq2 = (int)delta_y[pos] * (int)delta_y[pos];
      (*magnitude)[pos] = (short)(0.5 + sqrt((float)sq1 + (float)sq2));
    } } }
```

3 Thread Cancellation

Consider the following thread main function:

```
void thread_main() {
  for (int i = 0; i < 1000; i++) {
    high_latency_operation();
  }
}
```

Sometimes `thread_main()` doesn't need to finish. We'd like to terminate it early.

Part (a). [5 points] Why is it unsafe to use something like `pthread_kill()` to forcibly terminate a thread? **Provide** a (1-line) example of a bad thing that could happen when forcibly terminating a thread.

Part (b). [20 points] Pthreads provides the notion of cancellability state. Cancellation requests are queued until a thread indicates that it is cancellable, at which point Pthreads cancels the thread.

In this question, you'll implement a simplified version of cancellation requests.

Hints: You may assume that you need to support at most `MAX_TIDS` threads ever being created. Pthreads: Remember that `gettid()` returns the `pid_t` for the current thread. C++11: `std::this_thread::get_id()` returns a `std::thread::id`. You can get the ID of a C++11 `thread` using the `get_id()` call.

(15 points:) **Write** a function `cancel_thread(tid)` that initiates a cancellation request for a particular thread and **modify** `thread_main` to implement the cancellation request. You may also provide a `init_cancellation()` method to initialize data structures. (5 points:) **Explain** why your code is free of races.

Here's a sample use of `cancel_thread`:

```
init_cancellation();
pthread_create(&tid, NULL, thread_main, NULL);
// ...
cancel_thread(tid);
```

4 More Races

Part (a). Consider the following code in function `work()`[1]:

```
int (* my_func) (int);
int foo(int x) { return x; }

void work() {
  int my_counter = counter; // read global
  if (my_counter > 5) {
    my_func = foo;
  }

  if (my_counter > 5) {
    my_func(...);
  }
}
```

We have an assignment to a function pointer, `my_func`, and then a call to the function pointed to by `my_func`. Also, `counter` is a global shared variable, and `my_counter` is a local copy. A concurrently-executing thread updates the global `counter`. **Hint:** The compiler is allowed to insert, anywhere, the new statement `my_counter = counter`.

(8 points:) **Explain** how something bad could happen inside `work()` due to compiler interference. Show me one execution trace that performs an illegal operation, and explain why it is wrong. (7 points:) **Propose** a fix for the code and explain why your fix works.

Part (b). Now consider the following code:

```
thread1() {
  count = 0;
  reg1 = count;
  count = reg1;
}
```

```
void thread2() {
  reg2 = count;
  count = reg2;
  count = 0;
}
```

Variable `count` is shared between the two threads and is initially 17. (5 points:) **Exhibit** an interleaving which results in `count` not being 0 at the end. (This is why compilers aren't allowed to insert spurious assignments `x = x`.) (5 points:) Would it help if `count` was atomic? **Explain** why or why not.

[1] `http://hboehm.info/boehm-hotpar11.pdf`