Programming for Performance (ECE459): Midterm

February 26, 2014

This open-book midterm has 3 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.

(a) What can the compiler assume about a `restrict`-qualified pointer?

(b) Write down a Write-after-Write dependency. Rewrite your code, eliminating the write-after-write dependency. (You may, of course, introduce a different dependency).

(c) You are running a simple web server 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? (Be explicit with your assumptions.)

(d) OpenMP will not parallelize this loop properly. Propose an equivalent for loop which will parallelize.

```
double * array = malloc(sizeof(double) * 20);
for (double d = 0.0; d < 10.0; d += 0.5) {
  array[(int)(d*2)] = sin(d);
}
```

(e) Will you ever get a race condition from converting an OpenMP shared variable into a private variable? Why or why not?

(f) Gustafson's Law differs from Amdahl's Law because it allows what to vary?

(g) What is one problem with keeping a bunch of joinable threads around indefinitely?

(h) Say you have 300,000 potentially-active incoming connections open, but only 5 of them are ever active at once. Would threads or nonblocking I/O be better? Why?

(i) Which parallelization pattern most closely corresponds to a bank of subway turnstiles all controlling access to the subway in parallel?

(j) Give an example where you would use OpenMP tasks rather than sections. Explain why sections don't work in that case.

2 Locking

Louis Reasoner is working on the following tree implementation.

```c
struct node
{
  struct node *left, *right;
  int key;
  int *data;
};

struct node *root;

int find_and_increment(int key)
{
  pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;

  struct node *n = root;
  pthread_mutex_lock(&global_lock);
  while (n != NULL) {
    if (key == n->key) {
      *n->data++;
      pthread_mutex_unlock(&global_lock);
      return *n->data;
    }
    if (key < n->key)
      n = n->left;
    else if (key > n->key)
      n = n->right;
  }
  pthread_mutex_unlock(&global_lock);
  return NULL;
}
```

(a, 5 points) This code does not actually lock accesses to the tree. Why not? Propose a fix which properly locks accesses to the tree.

(b, 20 points) Make the following assumptions: (i) the `data` pointers may be shared among nodes and may be changed (as we see in the example); (ii) the structure of the tree (`key`, `left` and `right` fields) never changes after the tree is initialized. Now, propose changes to `struct node` and `find_and_increment` which permit two threads to concurrently call the function, while avoiding races on the `data` fields. Explain why your changes are correct.

3 Reductions

If you ask a compiler to parallelize the following loop, it will tell you that it found a reduction.

```
double sum(double[] array, int N) {
  double accum = 0.0;
  for (int i = 0; i < N; i++)
    accum += array[i];
  return accum;
}
```

Assume that there is a `NUM_THREADS` constant. You have to use that number of threads (part a) or tasks (part b). For simplicity, assume that `N % NUM_THREADS == 0`.

(a, 10 points) Rewrite this loop using `pthread` primitives to implement the reduction. (Casting `int` to and from `void *` is OK here.)

(b, 10 points) Rewrite this loop using OpenMP directives (no reduction).

(c, 5 points) Describe the source of overheads for autoparallelized reduction and one of pthreads/OpenMP. As the array gets larger, which implementation is fastest?

4 Dependences and parallelization

Consider these two code fragments.

```
void methodA(double * x, double * y) {
  for (int i = 0; i < 101; ++i)
    for (int j = 0; j < 100; ++j)
      x[j] = y[i+1];
}
```

```
void methodB(double * x) {
  for (int i = 0; i < 1000; ++i)
    for (int j = 0; j < 100; ++j)
      x[j] += j;
}
```

For each of these code fragments:

(a, 2.5 points) Describe the dependency in the code fragment. Be specific.

(b, 10 points) Will a compiler auto-parallelize this loop fragment? Why or why not? If the answer is “no”, propose transformations which will allow the loop fragment to be auto-parallelized.