Programming for Performance (ECE459): Final

April 25, 2013

This open-book exam has 5 non-cover pages & 6 questions, worth 25 points each. Answer in your answer book. You may consult any printed material (books, notes, etc).

1 Compiler Optimizations

Here is a simplified version of some code from Assignment 1.

```
#include <stdlib.h>

class R {
public:
    virtual double rand();
};

class S : public R {
    unsigned int seed;
public:
    virtual double rand() { return 5.0; }
};

unsigned long int montecarlo(unsigned long int iterations) {
    unsigned long int i, c = 0;
    double x, y, z;
    R * r = new S();

    for (i = 0; i < iterations; ++i) {
        x = (double)r->rand()/RAND_MAX;
        y = (double)r->rand()*i/RAND_MAX;
        z = x*x + y*y;
        if (z <= 1.0) {
            ++c;
        }
    }
    return c;
}
```

(part a, 10 points each) Describe two compiler optimizations which could apply to this code and write out the resulting code after optimization. Briefly summarize the conditions that need to hold for the code to be safe. I don't think that any additional qualifiers would help in this case, but you can add them if you want.

(part b, 5 points) Given this implementation of function `montecarlo()`, how far could an all-powerful compiler go in optimizing it[1]? Write down the most optimized possible version of this function that you can think of.

[1] $\sum^{n} i = \frac{n(n+1)}{2}$.

2 Fine-grained Locking

Here is some tree traversal code.

```
#include <stdio.h>
#include <pthread.h>

struct tree * find(int k, struct tree * t);

struct tree {
    int entry;
    struct tree * left, * right;
};

pthread_mutex_t m1;

int main() {
    pthread_mutexattr_t m1_attr;

    pthread_mutexattr_init(&m1_attr);
    pthread_mutex_init(&m1, &m1_attr);

    struct tree * t = 0; // ...

    printf("%d\n", find(17, t)->entry);
}

struct tree * find(int k, struct tree * t) {
    if (!t) return NULL;
    pthread_mutex_lock(&m1);
    if (k == t->entry) {
        pthread_mutex_unlock(&m1);
        return t;
    }
    else if (k < t->entry) {
        struct tree * r = find(k, t->left);
        pthread_mutex_unlock(&m1);
        return r;
    }
    else if (k > t->entry) {
        struct tree * r = find(k, t->right);
        pthread_mutex_unlock(&m1);
        return r;
    }
}
```

(part a, 5 points) What happens if you run this code on a tree of nonzero size? How can you fix that problem?

(part b, 5 points) Why would we say that this code uses *coarse-grained locking*? What is the disadvantage of coarse-grained locking?

(part c, 15 points) Write down versions of `struct tree` and `find()` which use fine-grained locks. Assume that `find()` may be called from a number of threads on an multicore machine and that nodes may be added as children concurrently while holding lock `m1`. Discuss the relative performance of the original and your new version in relevant contexts.

3 Parallelization

Consider the following linked list traversal.

```
#include <math.h>

struct elem {
    double entry;
    struct elem * next;
};

double calculate(struct elem * root) {
    double arith = 0, geom = 1, m = root->entry;

    struct elem * n = root;
    int count = 0;

    while (n) {
        count++;
        arith += n->entry;
        geom *= n->entry;
        if (n->entry > m) m = n->entry;
        n = n->next;
    }
    return arith/count + pow(geom, 1/count) + m;
}
```

(part a, 15 points) Write down a pthreads parallelization of function `calculate()` which uses 3 cores. Your parallelization should, of course, return the same value.

(part b, 5 points) Let's say that the linked list has 2 million elements. (a) Assume that you have a single processor with 8 cores. Estimate relative runtimes for the original version and the parallelized version and describe the side effects of the transformation. (Is it going to be faster or slower? Roughly speaking, by how much?) Support your estimate. (b) Same question, but for a machine with 4 physical CPUs: estimate runtimes, say why.

(part c, 5 points) You can change anything you want. How can you parallelize the code? What speedup would you expect from your parallelization?

4 OpenMP

(part a, 10 points) Consider again your solution from Question 3 part (a). This time, use OpenMP tasks to parallelize the code. (Again, you should be aiming for something that uses 3 cores). Describe how your code avoids race conditions and visibility problems.

(part b, 15 points) Now consider the following code.

```
struct ZMAT {
    int m, n;
    double **me;
};

double zm_norm1(struct ZMAT * a) {
    double maxval, sum;

    if (!a)
        return 0;

    maxval = 0.0;

    for (int j = 0; j < a->n; j++) {
        sum = 0.0;
        for (int i = 0; i < a->m; j++)
            sum += abs(a->me[i][j]);
        maxval = max(maxval, sum);
    }

    return maxval;
}
```

Parallelize this code with OpenMP pragmas. Indicate data types for all 3 non-loop variables. Argue that your parallelization is safe.

5 OpenCL

Here is a naïve implementation of a substring search similar to C's `strstr` function.

```
int mystrstr(const char * haystack, const char * needle) {
    for (int i = 0; haystack[i]; i++) {
        int match = 1;
        for (int j = 0; needle[j]; j++) {
            if (haystack[i+j] != needle[j]) {
                match = 0;
                break;
            }
        }
        if (match) return i;
    }
    return -1;
}
```

Your task is to write a OpenCL code which implements the substring search in parallel.

(part a, 6 marks) Explain all the buffers that you would declare in the host code and state whether each buffer should be read-only, write-only, or read/write.

(part b, 4 marks) Describe the global range you'd pass to `enqueueNDRangeKernel`.

(part c, 15 marks) Write your OpenCL kernel implementation of `mysubstr()`. Although I won't be super picky about syntax, you have to include the important points.

6 Limitations to Parallelization

We have a problem we need to solve. We always need 10 seconds of sequential setup time. After the setup, we can perfectly parallelize the rest of the code.

(part a, 7.5 marks) We have 8 processors and 20 seconds. What is our speedup over a sequential execution?

(part b, 7.5 marks) Now we have 1,000 seconds and the same 4 processors. What is our speedup now?

(part c, 10 marks) Assume that we have a *fixed* problem size, with the same amount of work as in part (a). How many processors do we need to get the same speedup as in part (b)?