ECE459 Final W2013

Question 1

a) One compiler optimization that could apply is function inlining. We can inline the r->rand() function. This also involves devirtualization. For this optimization to be safe, we need that r always points to an S object, and that rand() has no side effects.

   Another compiler optimization we could apply is loop-invariant code motion. Instead of computing (double)r->rand()/RAND_MAX every time, we can move it outside of the loop. For this optimization to work, we need that rand() has no side effects and returns the same value every time.

b)
```
unsigned long int montecarlo(unsigned long int iterations) {
        unsigned long int i, c = 0;
        double xx = 5.0/RAND_MAX * 5.0/RAND_MAX;

        for (i=0;i<iterations;++i) {
                if (xx + xx*i*i <= 1.0) {
                        ++c;
                }
        }
        return c;
}
```

Question 2

a) When find() is called, it acquires the lock m1. When traversing child nodes, it calls find() recursively without releasing the lock. This results in the second find() call attempting to acquire the lock and being blocked. To fix this, we can unlock m1 before the recursive call.

b) We use one lock to lock the entire tree during traversal, instead of having smaller locks for nodes or subtrees. Coarse-grained locking can significantly reduce opportunities for parallelism.

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

}
```

For small trees, this implementation may introduce more overhead as there are many locks, as well as locking and unlocking. As the tree size and number of threads increasing, this implementation allows more threads to traverse different sections of the tree simultaneously; there is reduced contention as threads only block each other when attempting to access the same node.

Question 3

a) One thread to traverse and calculate arith, one for geom, and one for m. The main thread then calculates the final result with these.
b) The parallelized implementation is likely not a 3x speedup from the original. There is overhead for thread creation and cleanup, and each of the threads still has to traverse the entire linked list. There may only be a 1.5x speedup. With 4 physical CPUs, there may be better performance as each CPU has its own memory and cache, reducing memory contention.
c) Not covered.

Question 4

Not covered.

Question 5

Everything in CUDA instead of OpenCL.

a) read-only: haystack, needle
write-only: result, array of size haystack_length - needle_length

b) (haystack_length-needle_length+1)/256 blocks, 256 threads/block

c) __global__ void mystrstr(const char* haystack, const char* needle, int* result, int haystack_length, int needle_length) {

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx < haystack_length - needle_length) {
    int match = 1;
    for (int j = 0; j < needle_length; j++) {
        if (haystack[idx+j] != needle[j] {
            match = 0;
            break;
        }
    }
    result[idx] = match ? 1 : 0;
}
```

}

Question 6

Not covered, Gustafson's Law.

1. Since we have 20 seconds, half the time is sequential setup and half is parallelizable. Using Gustafson's Law, we have that speedup = 0.5 + 0.5*4 = 2.5x.
2. Using Gustafson's Law, we have that speedup = 0.01 + 0.99*4 = 3.97x.
3. The parallelizable fraction is 0.5. Using Amdahl's Law, we have that 3.97 = 1/(0.5+0.5/N). Solving for N, we get that it is negative. It is not possible to achieve such a speedup.