CS 61C Summer 2016 Discussion 10 – SIMD and OpenMP

SIMD PRACTICE

A.SIMDize the following code:

```
void count (int n, float *c) {
        for (int i= 0 ; i<n; i++)
                c[i] = i;
}
```

Enter your solution by filling in the spaces provided. Assume n is a multiple of 4. (__mm__set1__ps(x) returns a __m128 with all four elements set to x.)

```
Void countfast( int n, float *c) {
        float m[4] = { _____, _____, _____, _____};
        __m128 iterate = __mm_loadu_ps( m );
        for (int i= 0; i< _____; i++)  {
                __mm__storeu__ps ( _____, iterate);
                iterate  = __mm__add__ps( iterate, __mm__set1__ps(_____));
                }
        }
```

b. Horner's rule is an efficient way to find the value of polynomial $p(x) = c_0x^{n-1} + c_1x^{n-2} + \ldots + c_{n-2}x + c_{n-1}$:

```
float poly( int n, float *c, float x) {
        float p = 0;
        for (int i = 0; i < n; i++ )
                p = p *x + c[i];
        return p;
}
```

Complete the following SIMD solution by filling in the blanks. Assume n is a multiple of 4.

```
Float fastpoly( int n, float *c, float x) {
        __m128 p = __mm__setzero__ps( );
        for  (int i= 0 ; i< n; i+=4) {
                p = __mm__mul__ps ( p, __mm__set1__ps (_____));
                p = __mm__add__ps ( p, __mm__loadu__ps( _____));
        }
        float m[4] = { _____, _____, _____, _____);
        p = __mm__mul__ps( p, __mm__loadu__ps(m));
        __mm__storeu__ps(m, p);
        return _____;
}
```

Thread Level Parallelism

```
#pragma omp parallelism
{
        /* code here */
}

#pragma omp parallel for
for (int i = 0; i < n; i++) {
        /* code here */
}
```

*Each thread runs a copy of code within the block
*Thread scheduling is non-deterministic

Same as:

```
#pragma omp parallel
{
        #pragma  omp for
        for (int i =0; i < n; i++) {...}
}
```

1. For the following snippets of code
below, circle one of the following to indicate what issue, if any, the code will experience. Then provide a short justification. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume arr is an int array with length n.

OPENMP Practice

a)

```
//Set element i to arr to i
#pragma omp parallel
for (int i = 0; i < n; i ++)
        arr[i] = I;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

b)

```
//Set arr  to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i ++)
        arr[i] = arr[i-1] + arr[i-2];
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

c)

```
//Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (int i = 0; i < n; i ++)
        arr[i] = 0;
```

Sometimes incorrect Always incorrect Slower than serial Faster than serial

2. Consider the following code:

```
// Square element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel
{
        int threadCount = omp_get_num_threads();
        int myThread = omp_get_thread_num();
        for (int i = 0; i < n; i++) {
                if (i% threadCount  == my Thread)
                        arr[i] * = arr[i];
        }
}
```

What potential issue can arise from this code?

3.Consider the following function:

```
void transferFunds(struct account *from, struct account *to, long cents) {
        from  -> cents  -= cents;
        to-> cents += cents;
}
```

a. What are some data races that could occur if this function is called simultaneously from two (or more) threads on the same accounts? (Hint: If the problem isn't obvious, translate the function into MIPS first)

b. How could you fix or avoid these races? Can you do this without hardware support?

OPEN MP Final Questions:

1.Which of the following choices correctly describes the given code? (Circle one). Explain your choice.

```
omp_set_num_threads(8);
#pragma omp parallel

{
     int thread_id = omp_get_thread_num();
     for(int count=0; count < ARR_SIZE/8*8; count+=8){
          arr[(thread_id%4)*2 + (thread_id/4) + count] = thread_id;
     }

}
```

A) Always correct, slower than serial
B) Always correct, speed depends on caching scheme C) Always correct, faster than serial
D) Sometimes incorrect
E) Always incorrect

2.Suppose we have **`int *A`** that points to the head of an array of length **`len`**. Below are 3 different attempts to set each element to its index (i.e. **`A[i]=i`**) using OpenMP with **`n>1`** threads. Determine which statement (A)-(E) correctly describes the code execution and provide a one or two sentence justification.

a) #pragma omp parallel for

```
for (int x = 0; x < len; x ++) {
        *A = x;
        A++;
```

```
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always**Correct**,FasterthanSerial

b) #pragma omp parallel {

```
for (int x = 0; x < len; x ++) {
        *(A+x) = x;
        }
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always **Correct**, Faster than Serial

c) #paragma omp parallel

```
{   for (int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {

        A[x] = x;

}}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always **Correct**, Faster than Serial