CSM 61C
Fall 2019

Parallelism
Mentoring 10: November 18, 2019

1 Data Level Parallelism

__m128i _mm_set1_epi32(int i)	sets the four signed 32-bit integers to i
__m128i _mm_loadu_si128(__m128i *p)	returns 128-bit vector stored at pointer p
__m128i _mm_add_epi32(__m128i a, __m128 b)	returns vector (a0+b0, a1+b1, a2+b2, a3+b3)
__m128i _mm_mullo_epi32 (__m128 a, __m128 b)	returns vector (a0*b0, a1*b1, a2*b2, a3*b3)
__m128i _mm_cmpgt_epi32(__m128 a, __m128 b)	returns: vector((a0>b0)?0xffffffff:0, (a1>b1)?0xffffffff:0, (a2>b2)?0xffffffff:0, (a3>b3)?0xffffffff:0)
void _mm_storeu_si128(__m128i *p,__m128i a)	stores 128-bit vector a at pointer p

1.1 The following code uses loop unrolling to improve performance:

```
static void sum_unrolled (int *a) {
    int sum = 0;
    for (int i = 0; i < 16; i += 4) {
        sum += (a + i);
        sum += (a + i + 1);
        sum += (a + i + 2);
        sum += (a + i + 3);
    }
}
```

(a) Implement the following function with a single SIMD instruction:

```
static void sum_unrolled_SIMD (int *a) {
    int sum = 0;
    int result[4];
    __m128i result_v = _mm_set1_epi32(0);
    for (int i = 0; i < 16; i += 4) {
        result_v = ___________________________________;
    }
    _mm_storeu_si128((__m128i*)result, result_v);
    sum = result[0] + result[1] + result[2] + result[3];
}
```

1.2 (a) Implement the following function using SIMD:

```
// Sequential code
static int selective_sum_total (int n, int *a, int c) {
    int sum = 0;
    for (int i = 0; i < n; i += 1) {
        if (a[i] > c) {
            sum += a[i];
        }
    return sum;
}

// SIMD code
static int selective_sum_vectorized (int n, int *a, int c) {
    int result[4];
    _m128i sum_v = _______________________________;
    _m128i cond_v = _______________________________;

    for (int i = 0; i < ___; i += ___) { //Vectorized loop
      _m128i curr_v = m128i_mm_loadu_si128(______________);
        __m128i tmp = _mm_cmpgt_epi32(____________);
        sum_v = _______________________________;
    }
  _mm_storeu_si128(________________________________________);

    for (int i = ___; i < ___; i += 1) { //Tail case
       result[0] += _____________________________________;
    }
  return ____________________________________________________;
```

2 Thread Level Parallelism

Some OpenMP syntax:

`#pragma omp parallel{ ... }`	Signals the system to spawn threads
`#pragma omp for`	Split the for loop into equal-sized chunks
`int omp_get_num_threads(void);`	Returns the number of threads the system has
`int omp_get_thread_num(void);`	Returns the thread ID of the current thread

2.1 Implement the following function using openMP:

```
// Sequential code
static int selective_product_total (int n, int *a, int c) {
    for (int i = 0; i < n; i += 1) {
        a[i] = (a[i] > c)? 10 : 0;
}
```

(a) Given sizeof(a*) = sizeof(int)*16, manually distribute the iterations such that there are four contiguous chunks and no false sharing. Assume the total number of threads is a multiple of 4. int omp_get_num_threads ← returns total number of threads in a team int omp_get_thread_num ← returns current thread ID number

```
// openMP code
static int selective_product_parallelized (int n, int *a, int c) {
    #pragma omp parallel {
    for (int i = 0; i < n; i += 1) {
        if ( ____________________________________________) {
                a[i] = (a[i] > c)? 10 : 0;
            }
        }
    }
}
```

2.2 Implement the following function using openMP:

```
// Sequential code
static int selective_square_total (int n, int *a, int c) {
    for (int i = 0; i < n; i += 1) {
        if (a[i] > c) {
            a[i] *= a[i];
        }
    return product;
}
```

(a)
```
// openMP for code
static int selective_square_parallelized (int n, int *a, int c) {
    #pragma omp parallel {
    #pragma omp for
    for (int i = ___; i < ___; i += ___) {
        if (_____ > _____) {
            a[i] *= _____;
        }
    }
    return product;
}
```

2.3 Assume the code is run with the following function call on a 32-bit machine and access to 8 threads. Also assume that ptr is block-aligned:

```
selective_square_parallelized(512, ptr, 61);
```

(a) Assume the machine uses a 1 KiB direct-mapped cache with 256 B blocks. Is the code correct?

(b) Assume the machine uses a 1 KiB direct-mapped cache with 1024 B blocks. Is the code correct?

3 Data Race!

3.1 Suppose we have int *A that points to the head of an array of length len. Determine which statement (A)-(E) correctly describes the code execution and provide a one or two sentence justification.

(a) Consider the following code:

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

Is the code:
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

(a)
```
#pragma omp parallel
{
    for (int x = 0; x < len; x++){
        *(A+x) = x;
    }
}
```

Is the code:
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