CS 61C Summer 2018 Discussion 10

1. Data Level Parallelism

The idea central to data level parallelism is vectorized calculation. Thanks to the Intel intrinsics (Intel proprietary technology), we can use specialized instructions to deal with multiple data with one instruction.

`__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_mullo_epi32(__m128 a, __m128 b)`	returns vector (a0*b0, a1*b1, a2*b2, a3*b3)
`void _mm_storeu_si128( __m128i *p, __m128i a )`	stores 128-bit vector a at pointer `p`

Note: For `_mm_mullo_epi32`, only the low 32 bits of the results are stored in the destination register.

Implement the following function, which returns the product of two arrays:

```
static int product_naive(int n, int *a) {
      int product = 1;
      for (int i = 0; i < n; i++) {
            product *= a[i];
      }
      return product;
}

static int product_vectorized(int n, int *a)  {
      int result[4];
      __m128i prod_v = ____________________________;

      for (int i = 0; i < _____; i += ___) {      // Vectorized loop
            prod_v =_____________________________________________;
      }
      __mm_storeu_si128(__________________, ___________________);

      for (int i = ______; i < ____; i++) {       // Handle tail case
            result[0] *= ________;
      }
      return ____________________________________;
}
```

2. Thread Level Parallelism

As powerful as data level parallelization is, it provides rather inflexible functionalities. The use of thread is much more powerful and versatile in many areas of programming. And OpenMP provides the hassle-free, plug-and-play directives to use of threads. Some examples of OpenMP directives:

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

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

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

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.

a)

```
// Set element i of 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 (i = 0; i < n; i++)
      arr[i] = 0;
```

Sometimes incorrect Always incorrect

Slower than serial Faster than serial

2. Consider the following code:

```
// Decrements 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 == myThread)
                  arr[i] *= arr[i];
      }
}
```

What potential issue can arise from this code?

3. Determine what is wrong with the following code and fix it, first using `critical` (e.g. `#pragma omp critical`), and then using `reduction` (e.g `#pragma omp reduction(operation: var)`):

```
// Assume n holds the length of double array arr passed in
double fast_product(double *arr, int n) {
      double product = 1;
      #pragma omp parallel for
      for (i = 0; i < n; i++) {
            product *= arr[i];
      }
      return product;
}
```