CS 61C Summer 2018 Discussion 10

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_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

1. Implement the following function, which returns the product of two arrays assuming the input values are small enough, so we only need to keep the lower 32-bit value of the product result:

```
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 = __mm_set1_epi32(1);

        for (int i = 0; i < n/4 * 4; i += 4) {   // Vectorised loop
                prod_v = __mm_mullo_epi32(prod _v, __mm_loadu_si128((__m128i *) (a + i)));
        }

        _mm_storeu_si128((__m128i *) result, prod_v);

        for (int i = n/4 * 4; i < n; i++) {  // Handle tail case
                result[0] *= a[i];
        }
        return result[0] * result[1] * result[2] * result[3];
}
```

2. 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.

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

Slower than serial – there is no for directive, so every thread executes this loop in its entirety. n threads running n loops at the same time will actually execute in the same time as 1 thread running 1 loop. Despite the possibility of false sharing, the values should all be correct at the end of the loop. Furthermore, the existence of parallel overhead due to the extra number of threads could slow down the execution time.

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

Always incorrect (if n>4) – Loop has data dependencies, so the calculation of all threads but the first one will depend on data from the previous thread. Because we said "assume no thread will complete before another thread starts executing," then this code will always be wrong from reading incorrect values.

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

Faster than serial – the for directive actually automatically makes loop variables (such as the index) private, so this will work properly. The for directive splits up the iterations of the loop into continuous chunks for each thread, so no data dependencies or false sharing.

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?

False sharing arises because different threads can modify elements located in the same memory block simultaneously. This is a problem because some threads may have incorrect values in their cache block when they modify the value arr[i], invalidating the cache block. A fix to this will be discussed in lab.

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 arr
double fast_product(double *arr, int n) {
        double product = 1;
        #pragma omp parallel for
        for (i = 0; i < n; i++) {
                product *= arr[n];
        }
        return product;
}
```

The code has the shared variable product. We can fix this with a critical as follows:

```
double fast_product(double *arr, int n) {
        double product = 1;
        #pragma omp parallel for
        for (i = 0; i < n; i++) {
                #pragma omp critical
                product *= arr[i];
        }
        return product;
}
```

OR

```
double fast_product(double *arr, int n) {
        double product = 1;
        #pragma omp parallel
        {
                thread_prod = 1;
                #pragma omp for
                for (i = 0; i < n; i++) {
                        thread_product *= arr[i];
                }
                #pragma omp critical
                product *= thread_product;
        }
        return product;
}
```

Or we can fix it with reduction as follows:

```
double fast_product(double *arr, int n) {
        double product = 1;
        #pragma omp parallel for reduction(*: product)
        for (i = 0; i < n; i++) {
                product *= arr[i];
        }
        return product;
}
```