Lab 8

61C Summer 2023

Thread level parallelism

- We saw data level parallelism in lab7 where we used Intel Intrinsics to do vector register operations
- In this lab, we will be using OpenMP to run blocks of code with many different threads

OpenMP Hello World

```
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("hello world from thread %d\n", thread_id);
    }
}
```

- #pragma omp parallel indicates the start of the parallel section, meaning that the code inside the brackets will be ran by all threads
- omp_get_thread_num() gives us the number/ID of the current thread
- What will the code above output?
    - hello world from thread 3
      hello world from thread 2
      ...
    - The threads can execute in any order

For loops

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel for
    for(int i=0; i<ARRAY_SIZE; i++)
    {
        z[i] = x[i] + y[i];
    }
}
```

With the #pragma omp parallel for header, the for loop will be split up between threads

Data Races and Critical Sections

```
int sum = 0;
#pragma omp parallel
{
sum = sum + omp_get_thread_num();
}
```

Why might this cause issues?

Data Races and Critical Sections

```
int sum = 0;

#pragma omp parallel

{

sum = sum + 1;

}
```

Why might this cause issues?

- sum(or any variable declared outside of parallel sections) are considered public variables which means all threads access the same variable
- Threads will “race” towards writing to sum which can cause the wrong value to be written in sum

Data Races and Critical Sections

```
int sum = 0;

#pragma omp parallel

{

sum = sum + omp_get_thread_num();

}
```

sum = sum + (any number); is really 3 instructions

- lw (accessing sum on the rhs)
- add (sum + (any number))
- sw (sum = ...)

Different interweavings of these 3 instructions can cause sum to behave non deterministically

Why might this cause issues?

- sum(or any variable declared outside of parallel sections) are considered public variables which means all threads access the same variable
- Threads will “race” towards writing to sum which can cause the wrong value to be written in sum

Data Races and Critical Sections

```
int sum = 0;

#pragma omp parallel

{

#pragma omp critical
sum = sum + omp_get_thread_num();

}
```

Adding the critical section will make sure a single thread finishes the entire section before another thread can start

- This will introduce further overhead and cause our code to slow down

Reductions

- Critical sections are slow so the reduction keyword can be used to increase the parallelizability of your code
- To make critical sections faster without having to use reduction, you can increase the number of private variables (private sums) and aggregate them in a public variable (public sums) in a critical section at the very end.