Chapter 3: Multithreading with OpenMP

OpenMP memory model: manipulating shared data

Any multithreaded programming environment has to define a **memory model**. This is a **contract between programmer and the environment**: the system will assume that the programmer is following the rules, and if this is indeed the case, the system will do what the programmer expects it to do.

Temporary view vs. memory

The OpenMP memory model consists of two components: there is a global memory shared by all threads, and each thread has its own temporary view of the memory.

Each read and write operation refers to the temporary view. The temporary view may or may not be consistent with the global memory. Consistency is guaranteed only after a "flush" operation. Flush makes sure that whatever you have written to your own temporary view will be visible in the global memory, and whatever someone else has flushed to the global memory will be available for reading in your own temporary view.

OpenMP performs a flush automatically whenever you enter or leave a `parallel` region. It also performs a flush whenever you enter or leave a critical section.

Memory model is an abstraction

It is important to keep in mind that memory models are abstractions. "Temporary view" is just an abstract way to describe how the system is guaranteed to behave. This does not refer to any specific piece of hardware or software.

In practice, "temporary view" refers to the combination of the following elements (and many more):

- What kind of optimizations the compiler might do.
- How the cache memory system works in the CPU.

For example, if you modify some shared data in your code, the compiler might choose to keep the current value in CPU registers until it has to be flushed, or the CPU might choose to keep the current value in L1 cache until it has to be flushed.

An example

Assume that we run the following code, and we have got two threads.

```
int a = 0;
#pragma omp parallel
{
    #pragma omp critical
    {
        a += 1;
    }
}
```

If thread 0 happens to enter the critical section first, we can reason about the timeline of the global memory and the temporary views of the threads as follows:

Memory	Thread 0	Thread 1	Explanation
a = ?	a = ?		
a = ?	a = 0		Thread 0 sets `a = 0`
a = 0	a = 0	a = 0	Threads 0 and 1 enter `parallel` region, both flush and everybody has a consistent view of everything
a = 0	a = 0	a = 0	Thread 0 enters critical section, flush (that does nothing)
a = ?	a = 1	a = ?	Thread 0 modifies `a`, we do not know if the change is visible in the global memory or in the temporary view of thread 1
a = 1	a = 1	a = ?	Thread 0 leaves critical section, flush, modification visible in the global memory, but thread 1 has not flushed its temporary view yet
a = 1	a = 1	a = 1	Thread 1 enters critical section, flush, the latest value of `a` is now visible in its temporary view
a = ?	a = ?	a = 2	Thread 1 modifies `a`, we do not know if the change is visible in the global memory or in the temporary view of thread 0
a = 2	a = ?	a = 2	Thread 1 leaves critical section, flush, modification visible in the global memory, but thread 0 has not flushed its temporary view yet
a = 2	a = 2	a = 2	Threads 0 and 1 leave `parallel` region, flush, now also thread 0 sees the latest value
a = 2	a = 2		Only thread 0 running, it can now read `a` and see what we would expect

Rules of thumb

Reasoning with the formal memory model is cumbersome. In practice, we use the following rules of thumb whenever we access shared data. We classify each shared data element X in the following categories:

- **Only one thread — easy:** There is only one thread that reads or writes X inside the `parallel` region. Everything is fine, no synchronization needed. The correct value of X is all the time available in the temporary view of one thread throughout the duration of the `parallel` region, and it will be flushed only at the very end.
- **Read-only variable — easy:** Nobody writes to X inside the `parallel` region, all references to it are reads. Everything is fine, no synchronization needed. The flush at the beginning of the `parallel` region makes X visible to everyone and it remains valid.
- **Anything else — care is needed:** At least one thread writes to X and some other thread also reads or writes X. Synchronization is necessary. You can read or write X only inside a critical section or an equivalent synchronization primitive.

Granularity

Here a "shared data element" is a single scalar variable or a single element of an array. It is perfectly fine to have an array in which e.g. thread 0 manipulates element 0 and thread 1 manipulates element 1. Care is needed only if multiple threads try to manipulate the same element.

This is one place in which we can see that the memory model is indeed an abstraction, not a description of the hardware. From the perspective of the hardware, cache memory would like to keep track of full cache lines, which are 64-byte units. If thread 0 manipulates element `x[0]` and thread 1 manipulates an adjacent element `x[1]` that is in the same cache line, this gets tricky to handle for the hardware: now if we simply update `x[0]` in the local cache of core 0 and we update `x[1]` in the local cache of core 1, then what do we do when we want to flush changes to the main memory? Cache memory can usually only keep track of which cache lines have changed, not which individual bytes inside a cache line have changed.

What happens in practice is that the CPU cores will have to communicate with each other to ensure no data is lost, and this may lower the performance of the program. This phenomenon is called **false sharing**.

It is important to note here that the system will keep its promise: according to the memory model specification, it is permitted that two threads manipulate adjacent array elements, and therefore hardware will have to make sure everything works correctly, even if it happens to be inconvenient for this particular hardware implementation.

Atomic operations

Critical sections are slow. If you have a performance-critical part in which you would like to modify some shared variable, there are also faster alternatives: atomic operations.

Let us have a look at a concrete example; we would like to calculate a histogram of the values that the following function returns, for all positive integers `x` less than 10 million (incidentally, this is the **Collatz process**, but that is not important for us):

```
static int collatz(long long x) {
    int i = 0;
    while (x != 1) {
        if (x % 2) {
            x = 3 * x + 1;
        } else {
            x = x/2;
        }
        ++i;
    }
    return i;
}
```

A straightforward sequential implementation would look like this; here we accumulate a histogram of width `w` in array `h`:

```
static void histogram(int* h, int w, int n) {
    for (int x = 1; x < n; ++x) {
        int i = collatz(x);
        assert(i < w);
        ++h[i];
    }
}
```

This takes approx. 3 seconds to run for n = 10000000.

The above code is not entirely straightforward to parallelize. We cannot just wrap this in an OpenMP parallel for loop, as we might have multiple threads manipulating the same element of the shared array `h`. We could in principle protect all references to `h` with a critical section:

```
static void histogram_critical(int* h, int w, int n) {
    #pragma omp parallel for
    for (int x = 1; x < n; ++x) {
        int i = collatz(x);
        assert(i < w);
        #pragma omp critical
        {
            ++h[i];
        }
    }
}
```

However, the performance is very poor; the running time is approx. 40 seconds, more than ten times slower than the sequential solution!

One solution would be to create a separate local histogram for each thread, and only at the very end of the loop merge the local histograms into a global histogram. But there is also a simpler solution — we can use atomic operations instead of critical sections:

```
static void histogram_atomic(int* h, int w, int n) {
    #pragma omp parallel for schedule(dynamic,10000)
    for (int x = 1; x < n; ++x) {
        int i = collatz(x);
        assert(i < w);
        #pragma omp atomic
        ++h[i];
    }
}
```

Now the performance is much better, less than 0.8 seconds!

Atomic operations are like tiny critical sections that only apply to one operation that refers to one data element. They do a flush, but only for this specific data element. Modern CPUs have direct hardware support for atomic operations, which makes them much faster than critical sections.

The compiler should give a friendly error message if you try to apply the `atomic` directive to some operation that cannot be made atomic.