Chapter 3: Multithreading with OpenMP

Introduction

We have already used OpenMP parallel for loops in Chapter 2, but let us now explore OpenMP in more detail.

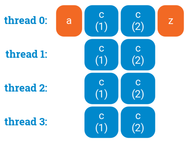
Recall that OpenMP is readily available in the GCC compiler; to use it, we just need to add the right `#pragma omp` directives and compile and link with the command line switch `-fopenmp`.

Basic multithreading construction: parallel regions

Any useful OpenMP program has to use at least one `parallel` region. This is a construction that tells OpenMP that we would like to create multiple threads:

```
a();
#pragma omp parallel
{
    c(1);
    c(2);
}
z();
```

If we execute the above program, the timeline of the execution might look e.g. like this (in the figure, time goes from left to right):



The number of threads is automatically chosen by OpenMP; it is typically equal to the number of hardware threads that the CPU supports, which in the case of a low-end CPU is usually the number of CPU cores. In the examples of this chapter we use a 4-core CPU with 4 threads.

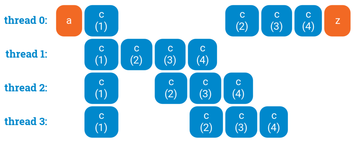
Everything that is contained inside a `parallel` region is executed by all threads. Note that there is no synchronization inside the region unless you explicitly ask for it. Here, for example, different threads might execute at slightly different speeds, and some might start with `c(2)` already while some other threads are still processing `c(1)`. However, OpenMP will automatically wait for all threads to finish their work before continuing the part that comes after the loop — so in the above example we know that when `z()` starts, all threads have finished their work.

Naturally, this example is completely useless by itself as all threads do the same work! We will very soon see how to do work-sharing, i.e., how to tell different threads to do different things. Before getting there, let us first have a look at the main coordination primitive: critical sections.

Critical sections

Whenever we modify a shared resource, we must take care of proper synchronization between the threads. The simplest synchronization primitive is a critical section. A critical section ensures that at most one thread is executing code that is inside the critical section. For example, here only one thread is running `c(2)` at any point of time:

```
a();
#pragma omp parallel
{
    c(1);
    #pragma omp critical
    {
        c(2);
    }
    c(3);
    c(4);
}
z();
```



Note that other threads are free to execute e.g. `c(3)` while another thread is running `c(2)`. However, no other thread can start executing `c(2)` while one thread is running it.

As we will discuss later in more detail, critical sections also ensure that modifications to shared data in one thread will be visible to the other threads, as long as all references to the shared data are kept inside critical sections. Each thread synchronizes its local caches with global memory whenever it enters or exits a critical section.

Shared vs. private data

Any variable that is declared outside a `parallel` region is shared: there is only one copy of the variable, and all threads refer to the same variable. Care is needed whenever you refer to such a variable.

Any variable that is declared inside a `parallel` region is private: each thread has its own copy of the variable. Such variables are always safe to use.

If a shared variable is read-only, you can safely refer to it from multiple threads inside the `parallel` region. However, if any thread ever writes to a shared variable, then proper coordination is needed to ensure that no other thread is simultaneously reading or writing to it.

Here is an example of the use of shared and private variables:

```
static void critical_example(int v) {

    // Shared variables
    int a = 0;
    int b = v;

    #pragma omp parallel
    {
        // Private variable - one for each thread
        int c;

        // Reading from "b" is safe: it is read-only
        // Writing to "c" is safe: it is private
        c = b;

        // Reading from "c" is safe: it is private
        // Writing to "c" is safe: it is private
        c = c * 10;

        #pragma omp critical
        {
            // Any modifications of "a" are safe:
            // we are inside a critical section
            a = a + c;
        }
    }

    // Reading from "a" is safe:
    // we are outside parallel region
    std::cout << a << std::endl;
}
```

If you run `critical_example(1)` on a machine with 4 threads, it will output "40". Each thread will set its own private variable `c` to 1, then it will multiply it by 10, and finally each thread will increment the shared variable `a` by 10. Note that all references to `a` — both reading and writing — are kept inside a critical section.

Other shared resources

Note that e.g. I/O streams are shared resources. Any piece of code that prints something to stdout has to be kept inside a critical section or outside a `parallel` region.