Chapter 2: Case study

OpenMP

The obvious drawback of the baseline implementation that we have is that it only uses one thread, and hence only one CPU core. To exploit all CPU cores, we must somehow create multiple threads of execution.

About threads

Any modern operating system makes it possible to create threads. If we create, for example, 4 threads, and we run the code on a computer with 4 CPU cores, we can usually count on the operating system to do the sensible thing: it will typically assign each thread to a separate CPU core, and hence, if all goes well, we can do (almost) 4 times as much useful work per second as previously.

There are many ways of creating threads. For example, Unix-like operating systems have a low-level interface called pthreads. The C++ programming language introduced the thread support library in version C++11; this provides a bit higher-level interface on top of the threading primitives provided by the operating system.

However, using such libraries directly to speed up computations in our application would take a nontrivial amount of code. In this course we will use a convenient high-level interface called OpenMP.

OpenMP: multithreading made easy

OpenMP is an extension of the C, C++, and Fortran programming languages. It is standardized and widely supported. For example, the GCC compiler has a built-in support for it. To enable OpenMP support, we will just need to use the command line switch `-fopenmp` both to compile the code and to link the code. So our compilation command changes from

```
g++ -g -O3 -march=native -std=c++17
```

to

```
g++ -fopenmp -g -O3 -march=native -std=c++17
```

Code written with OpenMP looks a bit strange at first, but once you get used to it, it is very convenient and quick to use. The basic idea is that you as programmer add `#pragma omp` directives in the source code, and these directives specify how OpenMP is supposed to divide work among multiple threads.

OpenMP parallel for loops

We will look at OpenMP in much more detail later in this course, but for now it is enough to be aware of one convenient directive: `#pragma omp parallel for`. You can use this directive right before a for loop that you want to parallelize. Here is a very simple example:

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

Without the `#pragma`, the code would be executed as follows:

```
Thread 1:  c(0); c(1); c(2); c(3); c(4); c(5); c(6); c(7); c(8); c(9);
```

However, the `#pragma omp parallel for` directive instructs the compiler to generate code that splits the iterations of the loop among multiple threads. For example, if we have 4 CPU cores, OpenMP typically uses 4 threads. The end result looks like this:

```
Thread 1:  c(0); c(1); c(2);
Thread 2:  c(3); c(4); c(5);
Thread 3:  c(6); c(7);
Thread 4:  c(8); c(9);
```

OpenMP will take care of creating threads for you. It will maintain a pool of threads that is readily available whenever needed. It will also take care of synchronization: for example, in the above example the program does not continue until all threads have completed their part of the parallel for loop.

Scheduling directives

You can also control how to split the iterations among multiple threads with the `schedule` directive:

```
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

Now OpenMP will let the first thread handle the first iteration, the second thread handle the second iteration, etc.:

```
Thread 1:  c(0); c(4); c(8);
Thread 2:  c(1); c(5); c(9);
Thread 3:  c(2); c(6);
Thread 4:  c(3); c(7);
```

This might be a good approach e.g. if the amount of work done by `c(i)` depends on `i`. With the standard schedule, thread 1 might get all small jobs and thread 4 might get all large jobs, while in the above scheme all threads would have both small and large jobs.

Warning! Stay safe!

Whenever you ask OpenMP to parallelize a for loop, it is your responsibility to make sure it is safe.

For example, parallelizing the following loop is safe if you can safely execute the operations `c(0)`, `c(1)`, …, `c(9)` simultaneously in parallel:

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
```

But exactly when can we execute two operations X and Y simultaneously in parallel? We will discuss this in more detail later, but for now the following rules of thumb are enough:

- There must not be any shared data element that is read by X and written by Y.
- There must not be any shared data element that is written by X and written by Y.

Here a "data element" is, for example, a scalar variable or an array element.

For example, the following loop cannot be parallelized; iteration 0 writes to `x[1]` and iteration 1 reads from the same element:

```
for (int i = 0; i < 10; ++i) {
    x[i + 1] = f(x[i]);
}
```

The following loop cannot be parallelized either; iteration 0 writes to `y[0]` and iteration 1 writes to the same element:

```
for (int i = 0; i < 10; ++i) {
    y[0] = f(x[i]);
}
```

But parallelizing the following code is perfectly fine, assuming `x` and `y` are pointers to distinct array elements and function `f` does not have any side effects:

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    y[i] = f(x[i]);
}
```

Back to our application

Now we are ready to parallelize the sequential baseline implementation; recall that our current program code looks like this:

```
void step(float* r, const float* d, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = std::numeric_limits<float>::infinity();
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = std::min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

Here it is perfectly safe to parallelize the outermost for loop:

- Variable `j` is local (introduced inside the loop), not shared.
- Variable `n` is read-only.
- Array `d` is read-only.
- Array `r` is write-only.
- Different iterations of the loop write to different elements of array `r`; the same element is never written by two different iterations.

Hence parallelizing the loop is as easy as adding a single pragma in the right place:

```
void step(float* r, const float* d, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float v = std::numeric_limits<float>::infinity();
            for (int k = 0; k < n; ++k) {
                float x = d[n*i + k];
                float y = d[n*k + j];
                float z = x + y;
                v = std::min(v, z);
            }
            r[n*i + j] = v;
        }
    }
}
```

This is really everything that we need to do! We can compile this with the `-fopenmp` flag, run it, and it will make use of all CPU cores. For example, when `n = 4000`, using a computer with 4 cores, each thread will run 1000 iterations of the outermost for loop.

Results

Using our sample platform, we can measure substantial improvements in the running time. The original code took 99 seconds; the multithreaded version only 25 seconds. We get a speedup of factor 3.9.

However, we are definitely not done yet!