Portfolio 5

Sam Bowyer

2023-03-21

Parallelisation in C++ with OpenMP

Intro

Parallelisation is good makes fast compute across multiple systems can do this on one system by parallelising over CPU cores and/or over threads within a single core (which isn't technically parallelisation—the threads run sequentially on the same core, but the OS switches between them quickly enough that it looks like they're running in parallel)

One popular way to do this in C++ is with OpenMP.

Basic OpenMP (Hello World)

First we need to set an environment variable to tell the compiler to use OpenMP. This is done with the command `export OMP_NUM_THREADS=4` (or whatever number of threads you want to use). This will set the number of threads to 4. You can check that this has worked by running `echo $OMP_NUM_THREADS` and it should print 4.

```
export OMP_NUM_THREADS=4
echo $OMP_NUM_THREADS
```

```
## 4
```

Now we can compile and run a simple OpenMP program. The program is in the file `hello_openmp.cpp` and is as follows:

```
#include <iostream>

int main(int argc, const char **argv)
{
    #pragma omp parallel
    {
        std::cout << "Hello OpenMP!\n";
        std::cout << "1\n";
        std::cout << "2\n";
        std::cout << "3\n";

    }

    return 0;
}
```

The `#pragma omp parallel` line tells the compiler to parallelise the code that follows. The `std::cout` lines are executed in parallel by the 4 threads.

We compile the program with the command `g++ -fopenmp hello_openmp.cpp -o hello_openmp`. This tells the compiler to use OpenMP and to compile the program into an executable called `hello_openmp`. (Note that we also add on the command to set `OMP_NUM_THREADS` to 4 before we run the program—this is because the environment variable is only set for the current terminal session and in compiling this document each code block is run in a new terminal session.)

```
g++ -fopenmp hello_openmp.cpp -o hello_openmp
export OMP_NUM_THREADS=4; ./hello_openmp
```

```
## Hello OpenMP!
## Hello OpenMP!
## 1
## 2
## 3
## 1
## 2
## 3
## Hello OpenMP!
## 1
## 2
## 3
## Hello OpenMP!
## 1
## 2
## 3
```

We can see that the output is not in the order that we expect. This is because the threads are running in parallel and the OS is switching between them quickly enough that it looks like they're running in parallel. If we want to make sure that the output is in the order that we expect, we can use a `#pragma omp critical` line to make sure that only one thread is executing the code at a time.

As an example of this, consider the following program, which makes use of the `omp_get_num_threads()` and `omp_get_thread_num()` functions to print out the number of threads and the thread ID. The program is in the file `hello_threads.cpp` and is as follows:

```
#include <iostream>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_num_threads() 0
    #define omp_get_thread_num() 0
#endif

int main(int argc, const char **argv)
{
    std::cout << "I am the main thread.\n";

    #pragma omp parallel
    {
        #pragma omp critical
        {
            int nthreads = omp_get_num_threads();
            int thread_id = omp_get_thread_num();

            std::cout << "Hello. I am thread " << thread_id
```

```
                    << " out of a team of " << nthreads
                    << std::endl;
        }
    }

    std::cout << "Here I am, back to the main thread.\n";

    return 0;
}
```

We compile the program with the command `g++ -fopenmp hello_threads.cpp -o hello_threads` and run it with `./hello_threads`, which produces an un-jumbled output:

```
g++ -fopenmp hello_threads.cpp -o hello_threads
./hello_threads
```

```
## I am the main thread.
## Hello. I am thread 15 out of a team of 16
## Hello. I am thread 0 out of a team of 16
## Hello. I am thread 1 out of a team of 16
## Hello. I am thread 14 out of a team of 16
## Hello. I am thread 13 out of a team of 16
## Hello. I am thread 12 out of a team of 16
## Hello. I am thread 2 out of a team of 16
## Hello. I am thread 3 out of a team of 16
## Hello. I am thread 5 out of a team of 16
## Hello. I am thread 8 out of a team of 16
## Hello. I am thread 7 out of a team of 16
## Hello. I am thread 10 out of a team of 16
## Hello. I am thread 11 out of a team of 16
## Hello. I am thread 6 out of a team of 16
## Hello. I am thread 4 out of a team of 16
## Hello. I am thread 9 out of a team of 16
## Here I am, back to the main thread.
```

An effectively parallelised program

Of course, the previous program does not really need to be parallelised, but it is a good example of how to use the `#pragma omp critical` line. We can consider another example, which computes a simple Monte Carlo estimate of π by generating points within a unit square uniformly at random and multiplying the proportion of points that fall within a unit circle by 4. The program is in the file `pi.cpp` and contains the following code:

```
#include <cmath>
#include <random>
#include <iostream>

int main()
{
    int n_inside = 0;
    int n_outside = 0;

    std::random_device rd;

    #pragma omp parallel
    {
        int pvt_n_inside = 0;
        int pvt_n_outside = 0;

        std::minstd_rand generator(rd());
        std::uniform_real_distribution<> random(-1.0, 1.0);

        #pragma omp for
        for (int i=0; i<1000000; ++i)
        {
            double x = random(generator);
            double y = random(generator);

            double r = std::sqrt( x*x + y*y );

            if (r < 1.0)
            {
                ++pvt_n_inside;
            }
            else
            {
                ++pvt_n_outside;
            }
        }

        #pragma omp critical
        {
            n_inside += pvt_n_inside;
            n_outside += pvt_n_outside;
        }
    }

    double pi = (4.0 * n_inside) / (n_inside + n_outside);

    std::cout << "The estimated value of pi is " << pi << std::endl;

    return 0;
}
```

Note that we make the most of parallelisation by only using the `#pragma omp critical` block when we are adding to the variables `n_inside` and `n_outside`. This is because adding to variables is not an atomic operation and so if multiple threads try to add to the same variable at the same time, the result could easily be wrong. The way that we get around this is by using private variables `pvt_n_inside` and `pvt_n_outside` which are only used by a single thread. We then add the values of these variables to the global variables `n_inside` and `n_outside` in a critical block. This ensures that the values of the global variables are correct.

Meanwhile, the random number generator is thread-safe, so we can use it in parallel without any problems.

We compile the program with the command `g++ -fopenmp pi.cpp -o pi` and run it with `./pi`, which we do three times to obtain three different estimates of π:

```
g++ -fopenmp pi.cpp -o pi
./pi
./pi
./pi
```

```
## The estimated value of pi is 3.14165
## The estimated value of pi is 3.14131
## The estimated value of pi is 3.1416
```