OpenMP

Rachel Wood

2023-03-10

Introduction to OpenMP

OpenMP provides a way to write C++ code which uses multiple cores to run a program.

We control the number of threads using the global `OMP_NUM_THREADS` variable in a bash shell:

```
export OMP_NUM_THREADS=4
```

For example, the `hello_openmp.cpp` file contains:

```
#include <iostream>

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

    return 0;
}
```

where the `#pragma omp parallel` line indicates the contents of the curly brackets below form a parallel section. Then the output depends on the value of the `OMP_NUM_THREADS` value.

We can now compile the code and set the thread variable as shown above, or specify it when running the code:

```
g++ -fopenmp hello_openmp.cpp -o hello_openmp

#setting parameter globally
export OMP_NUM_THREADS=4
./hello_openmp
```

```
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
```

```
#setting parameter inline
OMP_NUM_THREADS=10 ./hello_openmp
```

```
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
## Hello OpenMP!
```

Directives

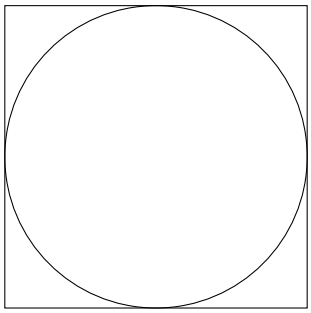
We use directives in OpenMP (e.g. the `parallel` directive above) to indicate how we want to use threads. Some basic examples include:

- `sections`: Used for code which can be run in parallel by different threads
- `for`: Does what `sections` does but for the contents of a loop
- `parallel`: Indicates a block to be executed by a team of threads.
- `critical`: Used when code can only be executed by one thread at a time.

We now use the information in this section to demonstrate how OpenMP can be useful in a concrete example.

Monte Carlo Example

A common algorithm for estimating π uses Monte Carlo methods and a unit circle (which has area π) inside a square with side length 2 (which has area 4):



Then if we generate points randomly inside the square, we expect the proportion of points inside the circle to be approximately $\frac{\pi}{4}$.

We can use a parallel loop to generate a large quantity of random numbers, determine which are in the circle and use the `reduce` directive to sum these counts.

We can generate random numbers using the `<random>` header. We can set the seed for the program by writing `std::random_device` before the `parallel` directive.

For each iteration we can then get a new generator and define a uniform distribution on the interval $[-1, 1]$ using these two lines of code:

```
std::default_random_engine generator(rd());
std::uniform_real_distribution random(-1.0, 1.0);
```

Then, if the count variables for points inside and outside the circle are called `count_in` and `count_out` respectively, the generating code under a `for` directive would be:

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

  if (r < 1.0){
    ++count_in;
  }
  else{
    ++count_out;
  }
}
```

We can then use the `reduction` directive to combine the `count_in` and `count_out` variables from all the threads to estimate π.

Putting these parts together we obtain the solution:

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

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

    std::random_device rd;

    #pragma omp parallel reduction(+ : n_inside, n_outside)
    {
        int count_in = 0;
        int count_out = 0;

        std::default_random_engine 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)
            {
                ++count_in;
```

```
            }
            else
            {
                ++count_out;
            }
        }

        n_inside += count_in;
        n_outside += count_out;
    }

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

    std::cout << "pi is approximately " << pi << std::endl;

    return 0;
}
```

Now supposing the above is contained in a file called `pi.cpp` we can compile and run it in the terminal as follows:

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

```
## pi is approximately 3.13989
```