portfolio_6

2023-03-02

OpenMP

OpenMP let's us carry out parallel programming in C, C++ and Fortran. We will focus on using it in conjunction with C++. First to define the number of threads used in OpenMP we type the command **export OMP_NUM_THREADS= < number of threads you want >** into bash. Let's now print the contents of the C++ file "openmp.cpp" which has some functions that carry out parallel programming using openmp. It is all very well commented and will explain how openmp works if you read it from top to bottom:

```
cat openmp.cpp
```

```
## #include <iostream>
## #include <unistd.h> // For sleep()
## #include <random> // For random number generation
##
## // The following lines are used to include the OpenMP header file
## #ifdef _OPENMP
##     #include <omp.h>
## #else
##     #define omp_get_thread_num() 0
## #endif
##
## // Here we define a function that will calculate pi using a Monte Carlo method
## // it will calculate pi in parallel using openmp
## double get_pi_critical(){
##     // This creates a random device
##     std::random_device rd;
##
##     // Define variables to store the number of points inside and outside the circle
##     int n_inside = 0;
##     int n_outside = 0;
##
##     // This defines a parallel region:
##     #pragma omp parallel
##     {
##         // Define variables to store the number of points inside and outside the circle
##         // specific to each thread
##         int pvt_n_inside = 0;
##         int pvt_n_outside = 0;
##
##         // This creates a random number generator
##         std::minstd_rand generator(rd());
##         // This creates a function that generates random numbers between 0 and 1
##         std::uniform_real_distribution<double> unif(0, 1);
##
```

```
##         // This defines a region where the for loop will be run in parallel
##         // if we have n threads available,
##         // then the iterations of the for loop will be split into n chunks
##         // and each thread will be assigned one chunk
##         #pragma omp for
##         for (int i=0; i<1000000; ++i)
##         {
##             double x = unif(generator);
##             double y = unif(generator);
##
##             double r = std::sqrt( x*x + y*y );
##
##             if (r < 1.0)
##             {
##                 ++pvt_n_inside;
##             }
##             else
##             {
##                 ++pvt_n_outside;
##             }
##         }
##         // This defines a region where the code will be run by one thread at a time
##         // ie. any time a thread enters this region,
##         // no other threads can enter this region until the first thread has exited
##         // This is useful for combining the results of the calculations for each thread,
##         // if we didn't use this,
##         // then threads would be writing to the same variables at the same time
##         #pragma omp critical
##         {
##             n_inside += pvt_n_inside;
##             n_outside += pvt_n_outside;
##         }
##     }
##     // This calculates pi using the number of points inside and outside the circle
##     double pi = (4.0 * n_inside) / (n_inside + n_outside);
##     return pi;
## }
##
## // If the number of cores is 4,
## // then the above function is essentially the same as the following function
## // here we will "manually" split the for loop into 4 sections
## // and assign each section to a different thread
## // using the sections directive
## double get_pi_sections(){
##     // This creates a random device
##     std::random_device rd;
##
##     // Define variables to store the number of points inside and outside the circle
##     int n_inside = 0;
##     int n_outside = 0;
##
##     // Define variables to store the number of points inside and outside the circle
##     // one specific to each thread
##     int pvt_1_n_inside = 0;
```

```
##     int pvt_1_n_outside = 0;
##     int pvt_2_n_inside = 0;
##     int pvt_2_n_outside = 0;
##     int pvt_3_n_inside = 0;
##     int pvt_3_n_outside = 0;
##     int pvt_4_n_inside = 0;
##     int pvt_4_n_outside = 0;
##
##     // This defines a region where the code will be run in parallel:
##     #pragma omp parallel
##     {
##         // This creates a random number generator
##         std::minstd_rand generator(rd());
##         // This creates a function that generates random numbers between 0 and 1
##         std::uniform_real_distribution<double> unif(0, 1);
##
##         // This defines a region where we can define sections
##         // the code in each section will be run by a different thread
##         // to parallelise the code
##         // Here we simply split the for loop into 4 sections
##         // and assign each section to a different thread
##         #pragma omp sections
##         {
##             // This defines a section
##             #pragma omp section
##             {
##                 for (int i=0; i<250000; ++i)
##                 {
##                     double x = unif(generator);
##                     double y = unif(generator);
##
##                     double r = std::sqrt( x*x + y*y );
##
##                     if (r < 1.0)
##                     {
##                         ++pvt_1_n_inside;
##                     }
##                     else
##                     {
##                         ++pvt_1_n_outside;
##                     }
##                 }
##             }
##
##             #pragma omp section
##             {
##                 for (int i=250000; i<500000; ++i)
##                 {
##                     double x = unif(generator);
##                     double y = unif(generator);
##
##                     double r = std::sqrt( x*x + y*y );
##
##                     if (r < 1.0)
```

```
##                     {
##                         ++pvt_2_n_inside;
##                     }
##                     else
##                     {
##                         ++pvt_2_n_outside;
##                     }
##                 }
##             }
##
##             #pragma omp section
##             {
##                 for (int i=500000; i<750000; ++i)
##                 {
##                     double x = unif(generator);
##                     double y = unif(generator);
##
##                     double r = std::sqrt( x*x + y*y );
##
##                     if (r < 1.0)
##                     {
##                         ++pvt_3_n_inside;
##                     }
##                     else
##                     {
##                         ++pvt_3_n_outside;
##                     }
##                 }
##             }
##
##             #pragma omp section
##             {
##                 for (int i=750000; i<1000000; ++i)
##                 {
##                     double x = unif(generator);
##                     double y = unif(generator);
##
##                     double r = std::sqrt( x*x + y*y );
##
##                     if (r < 1.0)
##                     {
##                         ++pvt_4_n_inside;
##                     }
##                     else
##                     {
##                         ++pvt_4_n_outside;
##                     }
##                 }
##             }
##         }
##     }
##     // We sum the results of the calculations for each thread
##     n_inside = pvt_1_n_inside + pvt_2_n_inside + pvt_3_n_inside + pvt_4_n_inside;
##     n_outside = pvt_1_n_outside + pvt_2_n_outside + pvt_3_n_outside + pvt_4_n_outside;
```

```
##     // This calculates pi using the number of points inside and outside the circle
##     double pi = (4.0 * n_inside) / (n_inside + n_outside);
##     return pi;
## }
##
## // This function again calculates pi,
## // but this time it uses the reduction directive
## double get_pi_reduction(){
##     // This creates a random device
##     std::random_device rd;
##
##     // Define variables to store the number of points inside and outside the circle
##     int n_inside = 0;
##     int n_outside = 0;
##
##     // This defines a parallel region,
##     // where we use a reduction clause,
##     // it says that when the "+" operator is applied to the variables n_inside and n_outside,
##     // that these results will hold the global results of the thread private calculations,
##     // ie. the results of the calculations for each thread will be combined into a single result
##     #pragma omp parallel reduction(+ : n_inside, n_outside)
##     {
##         // This creates a random number generator
##         std::minstd_rand generator(rd());
##         // This creates a function that generates random numbers between 0 and 1
##         std::uniform_real_distribution<double> unif(0, 1);
##
##         // Again we define a region where the for loop will be run in parallel
##         #pragma omp for
##         for (int i=0; i<1000000; ++i)
##         {
##             double x = unif(generator);
##             double y = unif(generator);
##
##             double r = std::sqrt( x*x + y*y );
##
##             if (r < 1.0)
##             {
##                 // This increments the number of points inside the circle
##                 // notice that we directly increment the global variable n_inside
##                 // here as we are using the reduction clause
##                 ++n_inside;
##             }
##             else
##             {
##                 // This increments the number of points outside the circle
##                 // notice that we directly increment the global variable n_outside
##                 ++n_outside;
##             }
##         }
##     }
##     // This calculates pi using the number of points inside and outside the circle
##     double pi = (4.0 * n_inside) / (n_inside + n_outside);
##     return pi;
```

```
## }
##
## // This is the main function,
## // we will now call some of the functions we have defined above,
## // to see if they work!
## int main(int argc, const char **argv)
## {
##     // We are going to start by calculating pi using a Monte Carlo method,
##     // get_pi_critical(), get_pi_sections() and get_pi_reduction()
##     // are all functions that calculate pi
##     // using the Monte Carlo method, but they use different OpenMP directives
##     // the first function uses the for and critical directives
##     // the second function uses the sections directive
##     // the second function uses the for and reduction directives
##
##     // We call get_pi_critical() to get a value for pi
##     double pi = get_pi_critical();
##     std::cout << "Using critical, Pi is approximately " << pi << std::endl;
##
##     // We call get_pi_sections() to get a value for pi
##     pi = get_pi_sections();
##     std::cout << "Using sections, Pi is approximately " << pi << std::endl;
##
##     // We call get_pi_reduction() to get a value for pi
##     pi = get_pi_reduction();
##     std::cout << "Using reduction, Pi is approximately " << pi << std::endl;
##
##     return 0;
## }
```

Let's now run the script to see if our functions all work:

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

```
## Using critical, Pi is approximately 3.14185
## Using sections, Pi is approximately 3.14263
## Using reduction, Pi is approximately 3.14072
```

Yay! it looks like everything is working!

When using OpenMP it is important that we think about how we are using it. To make good use of parallelization we have to think carefully about how we are carrying out computation and what the best way of dividing that computation up into threads is.

A typical style of parallel calculation is a map/reduce style calculation. This is where we split something up and map something (eg. elements of an array) to a function, calculate everything in parallel and then reduce all the outputs of your parallel computation into your result.

To maximize performance when working with OpenMP it is important to keep the following in mind: avoid working with global variables whenever possible (as it is either unpredictable or slow (or both!)), do as much as you can using thread private variables (ie. try combine results from threads only at the end), avoid critical regions as much as possible (slows your code down) and finally try and use OpenMP reductions instead of writing your own. Some side notes are: benchmark your code (time how fast it is and see how modifications to your code affect run time) and compare run times for different numbers of threads (as you increase the number of threads you may get speed ups, but there will reach a point where the overhead of dealing with more threads becomes burdensome and there are no computational gains to increasing the

number of threads).

A useful tool for benchmarking is **hyperfine**, if you just add it before the bit of code you want to run it will track computation.

Finally, before you use any of this OpenMP first go and try running your code with the following flag: **-O3**. Which will run your code in an optimized way, if its fast enough then there might be no need for manually paralyzing the code yourself!