OpenMP Integration

Dr. Christopher Marcotte — *Durham University*

Calculating π

Previously, we looked at using `critical` regions, an `atomic` update, or using a `reduction` clause to safely compute a shared variable sum. We will use these new skills to compute something slightly more interesting.

Integral method

The task is to compute the integral of $f(x) = \frac{4}{1+x^2}$ from $[0, 1]$ using the rectangle method. The exact answer is known,

$$\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x = \pi \approx \sum_{n=0}^{N} \frac{4}{1+x_n^2}\frac{1}{N},$$

where $x_n = \frac{n}{N}$ so that we can check correctness of our parallel implementation. The base code is

serial_pi_integral.c

```c
#include <stdio.h>
#include <omp.h>
#include <math.h>
#include <stdlib.h>

double f(double x){
  return 4.0/(1.0 + x*x);
}

int main(int argc, char** argv) {
    // Numerical integration of f(x) = 4/(1+x^2) in [a,b] = [0,1]
    // We use a fixed stepsize ('delta x') here.
    //const long num_steps = 500000000;
    const long num_steps = (long)(atol(argv[1]));
    const double a=0;
    const double b=1;
    const double stepsize = (b-a) / num_steps; // This is 'delta x'

    double sum = 0;   // This accumulates the f(x)*'delta x' values (i.e. areas)

    double t_0 = omp_get_wtime(); // time stamp

    for (int i = 1; i <= num_steps; i++){
        double x = a + (i - 0.5) * stepsize;
        sum += f(x) * stepsize;   // evaluate f(x)*dx and add to accumulator
    }

    double t_tot = omp_get_wtime() - t_0; // duration in seconds

    printf("\n pi error with %ld steps is %2.16f in %2.16f seconds\n ", num_steps, fabs(sum-M_PI),
t_tot);

    return 0;
}
```

Hint

You may want to complete this exercise *after* we discuss data races in OpenMP.

Exercise

Your mission, should you choose to accept it, is to parallelise the computation in three ways:

- a `#pragma omp critical` region;
- an `#pragma omp atomic` update; and
- a `#pragma omp ... reduction` clause.

Compare their performance with increasing numbers of threads (strong scaling). Why might some of them be faster than others? Can you write a manual reduction for this problem? See, e.g., previous exercise on race conditions. Add an additional calculation which compares your integral to π as defined in `math.h`: `M_PI`.

Given your current level of accuracy in computing π, estimate how many iterations (i.e., `int num_steps`) you need to calculate π to (approximately) half / single / double precision using your method.

Solution

For sufficiently large N I would expect the `#pragma omp ... reduction(+:sum)` is fastest, though this largely reflects just the reduced number of steps in the tree-summation, rather than effective parallelism (because the computation is not particularly demanding). The `#pragma omp atomic` approach is likely the next fastest, due to the hardware support, and the `#pragma omp critical` will be *slow*, in comparison.

The `#pragma omp ... reduction` approach is shown below, with the performance scaling of the three methods in Figure 1, with $f \approx 0.0625$.

pi_integral-solutions.c

```c
int main() {

  // Numerical integration of f(x) = 4/(1+x^2) in [a,b] = [0,1]
  // We use a fixed stepsize ('delta x') here.

  const long num_steps = 500000000;
  const double a=0;
  const double b=1;
  const double stepsize = (b-a) / num_steps; // This is 'delta x'

  double sum = 0;   // This accumulates the f(x)*'delta x' values (i.e. areas)

  double t_0 = omp_get_wtime(); // time stamp

  #pragma omp parallel for default(none) shared(a,b,stepsize,num_steps) reduction(+:sum)
  for (int i = 1; i <= num_steps; i++){
    double x = a + (i - 0.5) * stepsize;
    sum += f(x) * stepsize;   // evaluate f(x)*dx and add to accumulator
  }

  double t_tot = omp_get_wtime() - t_0; // duration in seconds

  printf("\n pi with %ld steps is %lf in %lf seconds\n ", num_steps, sum, t_tot);

  return 0;
}
```

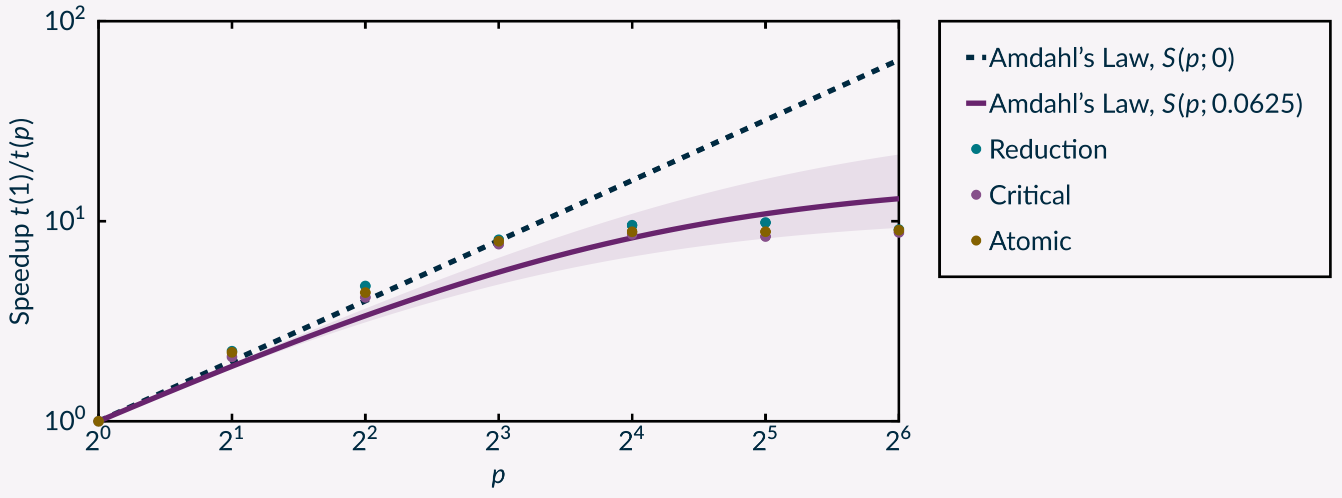

Figure 1: Speedup of reduction against the single-threaded code for the reduction, atomic, and critical clauses.

Interlude: ***Summing a random array***

We will need to figure out how to generate random numbers for the next part, so let's look at that by filling a large array with random numbers, so that $a_i \sim U[0,1)$, i.e., each element of an array `a` is sampling from the uniform distribution such that $0 \leq a_i < 1$.

Consider the following snippet:

demo.c

```c
#define _XOPEN_SOURCE
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    const size_t N = 5000000;
    double a[N];
    srand48(42);
    for (size_t i=0; i < N; i++){
        a[i] = drand48();
    }
    printf("a[%li] = %f\n", N-2, a[N-2]);
    return 0;
}
```

It should be clear that this code fills the array `a[N]` with uniformly distributed random numbers. The macro `#define _XOPEN_SOURCE` specifically permits the use of the `drand48()` function for random number generation. We use `srand(int)` to set the seed for the random number generation.

If we `#include <omp.h>` and `#pragma omp parallel for` the loop, we will suddenly find the snippet runs significantly worse. This is because `drand48()` effectively serialises the execution because it has an internal hidden state which is reset on every call — thread contention makes this even worse. Instead, we will write[*] a portable OpenMP-safe random number generator. This is comprised of several steps:

1. Initialize a base seed

[*]We will not write it ourselves, instead, we will use John Burkardt of FSU's ***random_openmp.c*** to motivate things and extract important structures and functions.

2. Initialize `private` seed variants for each OpenMP thread
3. Call our thread-safe RNG with each `private` seed on each thread to fill the array

In the following function `random_value(int *seed)` we have some obfuscating arithmetic to turn a pointer to an integer `int *seed` into a `double r`, where $r \sim U[0,1)$.

rand_fill.c

```c
double random_value( int *seed ){
  double r;

  *seed = ( *seed % 65536 );
  *seed = ( ( 3125 * *seed ) % 65536 );
  r = ( double ) ( *seed ) / 65536.0;

  return r;
}
```

Hint

All random numbers that come from computers are *pseudorandom*, it is beyond the scope of this course to discuss just how random these sequences are, but it is a good thing to keep in mind that truly random numbers are a fiction. That said... the `random_value(...)` code is not very random, even by the standards of computer made random numbers.

The `random_value(...)` function is, in turn, called by `fill(...)`:

rand_fill.c

```c
void fill( int n, int *seed ){
  int i;
  int my_id;
  int my_seed;
  double x[n];

  #pragma omp parallel private ( i, my_id, my_seed ) shared ( n, x )
  {
    my_id = omp_get_thread_num ( );
    my_seed = *seed + my_id;
    #pragma omp for
    for ( i = 0; i < n; i++ ){
      x[i] = random_value ( &my_seed );
    }
  }
  // Note: this is a lot of printing for large n!
   // You should add a check so it only prints if n is small.
  for (i = 0; i < n; i++){
      printf ( "x[%d] = %lf\n", i, x[i] );
  }

  return;
}
```

Which handles the creation of thread-private seed `int my_seed`, and filling the array. The requisite `int main(...)` function only sets the value of `int n`, the base seed `int seed`, and calls `fill(n, &seed)`. See `int main(void)` below:

rand_fill.c

```c
int main( void ){
    int n = 100;
    int seed = 123456789;
    fill( n, &seed );
    return 0;
}
```

Exercise

Add calls to `omp_get_wtime()` to time the parallel random filling and a simple serial loop filling array `double y[n]` using `drand48()`.

Increase the value of n to something very large (i.e. large enough to measure the cost of filling the array, itself), and verify that the execution time decreases as you add more threads. At what size n and number of threads p does the parallel code actually run faster than the serial code?

Convince yourself that the numbers you get are actually (pseudo-)random; this may be by computing the partial sums, $P_n = \sum_{i=0}^{n} a_i$ and verifying that $\lim_{n\to\infty} P_n \to \mathbb{E}(U[0,1)) = 0.5$, or by plotting the histogram for a sufficiently large filled array `double x[n]`.

Solution

The code is listed below:

rand_fill_solution.c

```c
int main( int argc, char** argv ){
  int q = atoi(argv[1]);
  int n = 1<<q;
  int seed = 123456789;
  double t0 = omp_get_wtime();
  int p = omp_get_max_threads();
  fill( n, &seed );
  double tpar = omp_get_wtime() - t0;
  t0 = omp_get_wtime();
  double x[n];
  for (int i=0; i < n; i++){
    x[i] = drand48();
  }
  double tser = omp_get_wtime() - t0;
  if (p > 1 && tser > tpar){
    printf("%i,%i,%2.6f,%2.6f\n", n, p, tser, tpar);
  }
  return 0;
}
```

[†] E.g., using

```bash
for (( q = 10 ; q <= 23 ; n+=2 )); do
  for (( p = 2; p <= 64; p*=2 )); do
        OMP_NUM_THREADS=$p ./rand_fill $q
  done
done
```

And if we run this from the terminal, invoking it within a double-loop iterating across (m, n) pairs mapping to $p = 2^m$ and $q = 2^n$, then we will find that the serial code is faster in most cases for small q.[†] I find that for $q \geq 2^{13}$, I can reliably get a faster solution using multiple threads than with a single thread.

Monte-Carlo method

A simple method to calculate π uses the Monte-Carlo method, named for the reliance on chance. In the typical case, you define upper and lower bounds for the evaluation of your integrand (i.e., the extremal values $f(x)$ may take) over the region of interest, and then sample the function to determine if you are under or over the curve. This is a very literal understanding of integration, but it works! We will do something slightly different than the above because we solved the integral in the first part of this exercise directly, just to make it interesting.

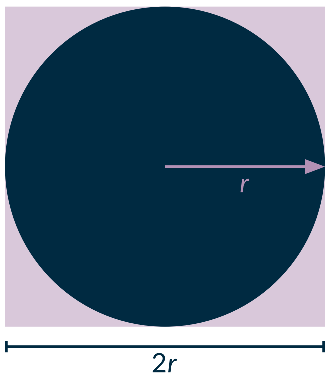

Figure 2: Square with inset circle of radius r.

Recall the area of a circle is $A_{\circ} = \pi r^2$, and the square that it circumscribes has area $A_{\square} = (2r)^2 = 4r^2$. Therefore, $\frac{A_{\circ}}{A_{\square}} = \frac{\pi}{4}$, and so if we have an efficient means of distinguishing whether some point in a square is outside of the circle, we can estimate π.

To do so, we will randomly sample the square $(x, y) \in [0, r] \times [0, r]$, and determine whether the point (x, y) is within r of the origin.

Challenge

We have simplified the set up of this problem by exploiting the symmetry of the problem. This is a common and often necessary practice in Scientific and High-Performance Computing. Can you think of other problems with symmetries that might be exploitable?

Hint

Combining random number generation with threading is deep magic. Using the usual `drand48` in a multithreaded environment is discouraged. You will get not great answers and it will be slower than the serial version. Complete the above *first*.

Exercise

Using the `random_value(...)` function above, implement another function which generates pairs of doubles `x` and `y` which are each sampled from $U[0, 1)$ in parallel on each thread.

Using a `#pragma omp ... reduction` clause, increment an `int count` when `x*x + y*y < 1.0`.

Set up a `int main(...)` function which uses your code to estimate the value of π according to the geometry described above.

Verify that the execution time decreases proportionally when using more OpenMP threads.

Split the domain, such that, for thread p, it only samples in the region $\frac{p-1}{N} \leq x < \frac{p}{N}$ and $0 \leq y < 1$. Does this harm or enhance your parallelization?

Solution

The code is shown below:

openmp_pi_montecarlo.c

```c
int calculate_pi( int n, int *seed ){
  int i, my_id, my_seed, count;
  double x,y;

  #pragma omp parallel private ( i, my_id, my_seed, x, y ) shared ( n ) reduction(+:count)
  {
    my_id = omp_get_thread_num();
    my_seed = *seed + my_id;
    #pragma omp for
    for ( i = 0; i < n; i++ ){
      x = random_value( &my_seed );
      y = random_value( &my_seed );
      if (x*x + y*y < 1.0){
        count++;
      }
    }
  }
  return count;
}
```

The domain splitting is left as an exercise for the reader; so long as you effectively map the random samples into the interval, you should still get the correct answer eventually, and the impact on your parallelisation should be minimal (because the mapping is entirely determined by the thread index).

Aims

- Introduction to parallel reductions using OpenMP
- Comparison of deterministic and stochastic integration methods
- Convergence and speedup testing of real algorithms