Program 1

Riley Kopp

13 September 2020

Program Description:

Sieve of Eratosthenes:

This program uses the Sieve of Eratosthenes algorithm to determine every prime that is less than a given number in parallel. See the Wiki Page for a description and background of the sequential version of the algorithm.

Algorithm 1: Sieve of Eratosthenes

```
Result: All prime numbers from 2 through n
A = an array of booleans of size n all set to true;
index = 2;
while index < $\sqrt{n}$ do
    if A[index] is true then
        $j = i^2$;
        while $j < n$ do
            A[j] = false;
        end
        j += i;
    end
    index += 1;
end
```

To parallelize this algorithm, it really didn't make sense to wrap the inner loop in a parallel block since the threads would be spawned and killed for every n which would make the runtime incredibly inefficient. Thus the only logical solution was to wrap the outer loop in a parallel tag.

Monte Carlo Approximation of Pi:

This program also the Monte Carlo technique to approximate Pi by generating random points (x, y) where x and y are randomly generated within the range $[0, 1]$.

Algorithm 2: Monte Carlo Approximation of Pi

```
**Result:** Approximation of $\pi$
$n = numberofpoints$;
$pointcount = 0$;
$pointslessthanone = 0$;
**while** $pointcount < n$ **do**
    generate x and y;
    **if** $x^2 + y^2 <= 1$ **then**
        $pointslessthanone+ = 1$;
    **end**
    $pointcount+ = 1$
**end**
$approximation = 4 * (pointslessthanzero/n)$;
```

To parallelize this algorithm, I wrapped the loop in a parallel tag and used a reduction on the $pointslessthanone$ variable to sum up the 'hits' after the threads joined.

Code Break Down:

Organization of Code:

The code was logically broken down into sections: Darts, Sieve, Main. Main (main.c) included the command line argument input parsing and the calls to run the sieve and/or run the Monte Carlo algorithm. Sieve (sift.h sift.c) included the initialization and main loop for the Sieve of Eratosthenes. Darts (darts.h darts.c) included the initialization and main loop for the Monte Carlo algorithm.

Libraries Used:

There were no libraries used beyond the expected (math, time, stdio, stdlib, omp...)

How to Use the program:

To compile the program simply run the command:

```
$ make
```

Then to run program, type one of the following:

```
# Run just the Monte Carlo approximation
$ ./run_program_1 -d <number of darts>
# Run just the Sieve of Eratosthenes
$ ./run_program_1 -s <max sift value>
# run both algorithms
$ ./run_program_1 -sd <max sift value> <number of darts>
```

Testing and Verification

The sieve was tested by comparing the programs output to the list of primes less than one million provided on this site. Testing the Monte Carlo algorithm was slightly more pain intensive...

Monte Carlo Approximation Testing/Discussion Questions

How many darts are needed to reach a reasonable estimate?

At 2 billion darts, I was able to reach an estimate of 4 decimal places which, as an engineer, is about 4 decimal places of over precision.

Is there any measurable difference between using a critical statement and using a reduction clause?

Yes, with a reduction 900,000,000 darts took 1.5 seconds. Meanwhile, the critical tag took 93 seconds. I would assume the critical section takes so much longer because behind the scenes, Open MP creates a legit mutex lock so only one thread can modify the hit counter while the reduction just runs something like an $O(n)$ summation of n private copies of the hit counter once all the threads have rejoined the main thread.

Time your code using both static and dynamic schedule to determine if there is any difference in performance?

Again at 9 million darts. A static run with a chunk size of 1, took 1.96 seconds to finish while dynamic (with a chunk size of 1) took 20 seconds.

Why is Random() not thread safe?

According to random(3), random is 'MT-Safe' which in most interpretations of English, one could assume that the function random is in fact 'Thread Safe' **BUT** upon reading the GNU definition of 'MT-Safe', One learns that 'MT-Safe' Stands for Multi Thread Safe. Which again, One would assume that the function is safe to use in a multi threaded environment..... **BUT WAIT THERE'S MORE** If One continues to read the GNU definition of MT-Safe, One would read this line: "For example, having a thread call two MT-Safe functions one right after the other does not guarantee behavior equivalent to atomic execution of a combination of both functions, since concurrent calls in other threads may interfere in a destructive way." Which basically means that multiple calls to random() in a thread is the computer science equivalent of going to the beach during a pandemic. Unlike a pandemic, our issue with random() can be solved without political interference (thankfully). All we have to do is pregenerate our random numbers before we enter the parallel section of our code.