

Programming Assignment 1

Author: Gabriel Hofer

CSC-410 Parallel Computing

Instructor: Dr. Karlsson

Due: September 18, 2020

Computer Science and Engineering
South Dakota School of Mines and Technology

Sieve of Eratosthenes

Listing 1: Non-parallelized Sieve of Eratosthenes (prime.c)

```
void erat(long int n, long int * pcnt){
    for(int i=2;i<=n;i++)
        sieve[i]=1;
    for(int i=2;i*i<=n;i++)
        if(sieve[i])
            for(int j=i*i;j<=n;j+=i)
                sieve[j]=0;
    *pcnt=0;
    for(int i=2;i<=n;i++)
        if(sieve[i])
            primes[( *pcnt)++]=i;
}
```

Listing 2: Parallelized Sieve of Eratosthenes (prime.c)

```
void erat2(long int n, long int * pcnt){
    #pragma omp parallel for
    for(int i=2;i<=n;i++)
        sieve[i]=1;
    int sqrtn = sqrt((double)n);
    for(int i=2;i <= sqrtn;i++)
        if(sieve[i]){
            #pragma omp parallel for
            for(int j=i*i;j<=n;j+=i)
                sieve[j]=0;
        }
    *pcnt=0;
    for(int i=2;i<=n;i++)
        if(sieve[i])
            primes[( *pcnt)++]=i;
}
```

Listing 3: Measuring Runtime Performance (prime.c)

```
void main(){
    long int n, pcnt;
    double start, end;
    scanf("%li",&n);
    n--;

    pcnt=0;
    start = omp_get_wtime();
    erat(n,& pcnt);
    end = omp_get_wtime();
    print(pcnt);
    printf("Elapsed_time = %.6f seconds\n\n", end-start);

    // reset primes and sieve.
    for(int i=0; i<(1<<30); i++){
        sieve[i]=0;
        primes[i]=0;
    }

    pcnt=0;
    start = omp_get_wtime();
    erat2(n,& pcnt);
    end = omp_get_wtime();
    print(pcnt);
    printf("Elapsed_time = %.6f seconds\n\n", end-start);
}
```

Listing 4: Output in Terminal from prime sieve program (prime.c)

```
gcc prime.c -lm -fopenmp
```

Monte Carlo Method

Listing 5: Non-parallelized Monte Carlo Method (monte.c)

```
double monte(long long n){
    long long hits=0;
    double x, y, pi;
    for(int i=0; i<n; i++){
        hits += sq((double)rand()/((double)RANDMAX)) +
               sq((double)rand()/((double)RANDMAX))
               <= 1.0 ? 1 : 0;
    }
    pi = 4.0*hits/(double)n;
    return pi;
}
```

Listing 6: Parallelized Monte Carlo Method (monte.c)

```
double monte2(long long n){
    long long hits=0;
    double x, y, pi;
    #pragma omp for
    for(int i=0; i<n; i++){
        hits += sq((double)rand()/((double)RANDMAX)) +
               sq((double)rand()/((double)RANDMAX))
               <= 1.0 ? 1 : 0;
    }
    pi = 4.0*hits/(double)n;
    return pi;
}
```

Listing 7: Measuring Runtime Performance (monte.c)

```
void main(){
    long long n;
    double start , end , _PI_;
    scanf("%lld", & n);

    printf("Monte_Carlo_Method_NON-parallelized\n");
    start = omp_get_wtime();
    _PI_=monte(n);
    end = omp_get_wtime();

    printf("PI: %f\n" , _PI_);
    printf("Elapsed_time = %f seconds\n\n" , end-start);

    printf("Monte_Carlo_Method_parallelized\n");
    start = omp_get_wtime();
    _PI_=monte2(n);
    end = omp_get_wtime();

    printf("PI: %f\n" , _PI_);
    printf("Elapsed_time = %f seconds\n\n" , end-start);
}
```

Listing 8: Output in Terminal (monte.c)

```
gcc monte.c -lm -fopenmp
```

Description of Programs

Functions `erat` and `erat2` compute all of the primes between 1 and $n-1$, as specified, where $n > 1$. Functions `monte` and `monte2` compute an approximation of PI . `erat` and `monte` are non-parallelized while `erat2` and `monte2` are parallelized.

Description of Algorithms and Libraries:

I used OpenMP for both programs.

The prime sieve program is essentially two nested for loops. Initially, we start with an array, called `sieve`, containing a `True` value in every cell. Then, we iterate `i` from 2 to \sqrt{n} . For each of these iterations, we also iterate `j` from $i*i$ to n in steps of `i`. At each step, we set the current cell in `sieve` to `False`, because we know the number isn't prime.

The Monte Carlo Algorithm is a single for loop. Each iteration of the for loop is analogous to throwing a dart at a dart board. In our code, we use a circle of radius 1 as our dart board. We call the C `rand()` function twice for the `x` and `y` dimensions. Pythagora's Theorem is used to calculate whether the dart is more than 1.0 from the center of the board. If not, we increment our count variable, `hits`. Otherwise, we continue to the next iteration of the loop and throw another dart. We use `hits` and number of throws to calculate PI .

Description of Functions and Program Structure:

I created two files called `prime.c` and `monte.c`. The prime sieve is written in `prime.c` and, the Monte Carlo method is written in `monte.c`.

How to compile and use the programs

On Linux (Ubuntu):

```
$ gcc filename.c -fopenmp -lm
```

Description of the Testing and Verification Process:

I viewed the output of both `prime.c` and `monte.c` to check the correctness and verified that the algorithms for both programs are correct. For

testing the runtime of the programs and algorithms, I used the function `omp_get_wtime()`.

Description of Deliverable:

I submitted a zip folder with the following:

1. this pdf document
2. `prime.c`
3. `monte.c`