Concurrency Project

James Nolan – C00226267

Distributed and Concurrent Device Development

Tutor: Dr. Joseph Kehoe

Date: 26/02/2021

Table of Contents

Pseudocode

```
//Find all Prime Numbers
SieveOfEratosthenes(n)
    // Create a boolean array
    // "prime[0..n]" and initialize
    // all entries it as true.
    // A value in prime[i] will
    // finally be false if i is
    // Not a prime, else true.
```

```
sequential(n)
{
    bool *arr = new bool[n];
    int count = 0;
    for ( i = 0; i < n; i ++){
        arr[i] = true;
    }
    // mark false values
    for ( i = 2; i * i <= n; i++){
        if(a[i]){
            for ( j =i*2; j <= n; j += i){
                arr[j] = false;
            } } }
for (i = 3; i < n-2; i += 2){
        if(arr[i] && arr[i+2])
        count ++;
         print pairs;
    }
    delete[] arr;
    return count;
}
```

```
concurrent(n)
{
    bool *arr = new bool[n];
    int count = 0;

//set threads and sqrt
    // divide loop iterations for each thread
    // assigning true to all values
    for (i = 0; i < n; i ++){
        arr[i] = true;
    }
    // dynamically distribute loop iterations between the threads
    for ( i = 2; i <= sqrt; i++){
        if(arr[i]){
            for (j =i*2; j <= n; j += i){
                arr[j] = false;
            } } }
    // divide loop iterations for each thread
    for ( i = 3; i < n-2; i += 2){
        if(arr[i] && a[i+2])
        {
            count ++;
            print pairs;
        } }
    delete[] a;
    return count;
}
```

Speedup Results

Note: In order to increase speed, the program only prints the number of twin primes, not the pairs. When printing all pairs on individual lines with brackets and commas, the time was nearly doubled in some cases.

For all runs, 1000,000,000 was chosen used to test the code.

Sequential Code

The sequential code was ran 3 times with the following results to create a fair median time:

1. 21.347 sec
2. 21.669 sec
3. 21.576 sec

The median time was 21.531 seconds. This is the time that was used for getting the Absolute Speedup where:

$$S_n = \frac{Ts}{Tp(n)\cdot}$$

Parallel 1 Thread

1. 9.529 sec
2. 10.891 sec
3. 10.402 sec

The median time was 10.274 seconds. This is the time that was used for getting the Relative Speedup.

Absolute Speedup: 21.531/10.274 = ~2.096

Parallel 2 Threads

1. 6.779 sec
2. 6.774 sec
3. 7.125 sec

The median time was 6.892 seconds.

Absolute Speedup:

21.531/6.892 = ~3.124

Relative Speedup: 10.274/6.892 = ~1.491

Parallel 4 Threads

1. 6.872 sec
2. 6.856 sec
3. 6.791 sec

The median time was 6.839 seconds.

Absolute Speedup: 21.531/6.839 = ~3.148

Relative Speedup: 10.274/6.839 = ~1.502

Parallel 8 Threads

1. 6.811 sec
2. 6.833 sec
3. 6.841 sec

The median speed was 6.828 seconds.

Absolute Speedup:

21.531/6.828 = ~3.153

Relative Speedup:

10.274/6.828 = ~1.505

Parallel 16 Threads

1. 6.496 sec
2. 6.245 sec
3. 6.575 sec

The median time was 6.438

Absolute Speedup: 21.531/6.438 = ~3.444

Relative Speedup: 10.274/6.438 = ~1.596

Parallel 32 Threads

1. 6.312 sec
2. 6.257 sec
3. 6.351 sec

The median time was 6.306 seconds.

Absolute Speedup: 21.531/6.306 = ~3.414

Relative Speedup: 10.274/6.306 = ~1.629

Parallel 64 Threads

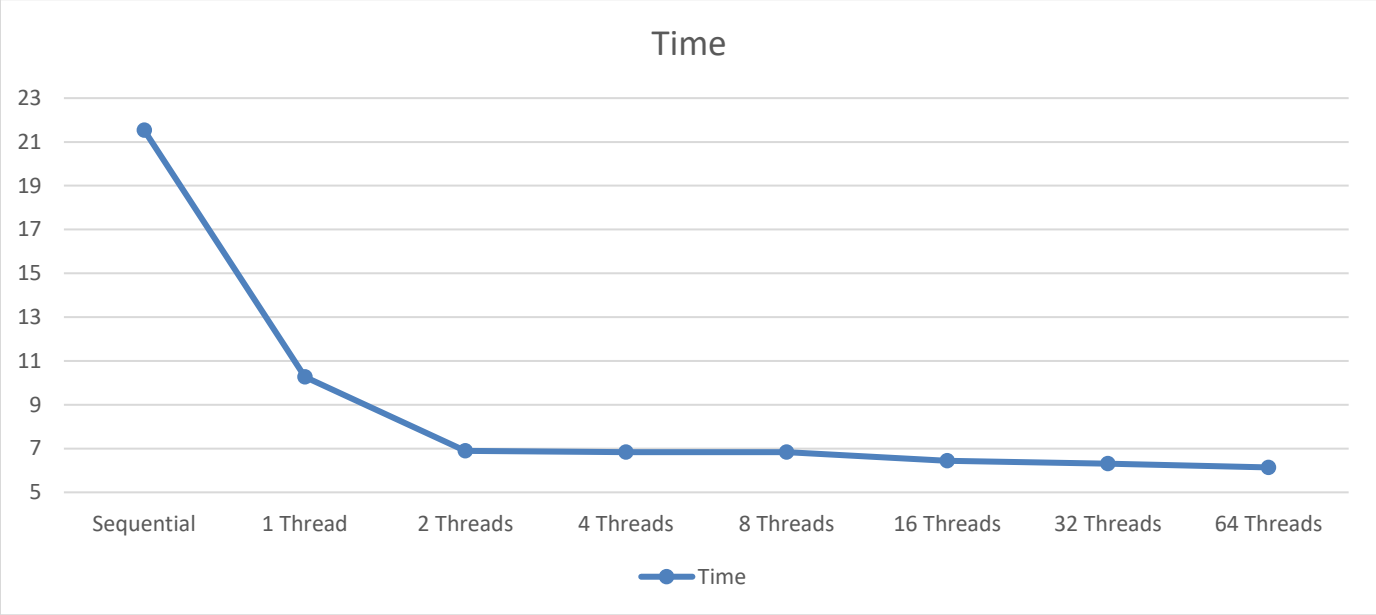
1. 6.167 sec
2. 6.153 sec
3. 6.098 sec

The median time was 6.139 seconds.

Absolute Speedup: 21.531/6.139 = ~3.507

Relative Speedup: 10.274/6.139 = ~1.674

Time Graph



Scalability

To test scalability, a number was tested using the sequential program and then doubled.

Number	Time
50,000	0.003 sec
100,000	0.005 sec
200,000	0.008 sec
400,000	0.025 sec
800,000	0.049 sec
1,600,000	0.109 sec
3,200,000	0.223 sec
6,400,000	0.424 sec
12,800,000	0.818 sec
25,600,000	1.539 sec
56,200,000	3.369 sec
128,400,000	7.822 sec
256,800,000	16.084 sec