****

**De La Salle University • College of Computer Studies**

**Concurrent Prime Number Application**

(Design and Evaluation of Its Performance)

Name (last name first) : Fernandez, Ryan Austin

Poblete, Clarisse Felicia M.

San Pedro, Marc Dominic

Tan, Johansson E.

Section : G01

Date of Submission : January 19, 2016

**I. Source Code**

import java.util.Scanner;

public class Prime {

static volatile boolean isPrime = true;

static long num = 111111151111111L;

static long sqrt = (long) Math.sqrt(num);

static long nThreads;

static volatile long threadCtr = 0;

public static void main(String args[]) {

System.out.print("ENTER: ");

Scanner sc = new Scanner(System.in);

int exp = sc.nextInt();

nThreads = (long) Math.pow(2, exp);

sc.close();

long ave = 0;

for(int x = 0; x < 5; x++ ) {

threadCtr = 0;

long time = System.currentTimeMillis();

long range = sqrt / nThreads;

long curStart = 2;

long curEnd = range;

for(int i=0; i<nThreads; i++){

new Thread(new Checker(curStart, curEnd)).start();

curStart = curEnd+1;

curEnd += range;

}

System.out.println();

while(true){

if( threadCtr == nThreads) {

break;

}

}

System.out.println(isPrime);

time = System.currentTimeMillis() - time;

System.out.println((time / 1000.0) + " seconds");

ave += time;

}

System.out.println("Average: " + (ave / 5000.0 ) + " seconds\n");

}

public static synchronized void update(boolean result) {

Prime.isPrime = Prime.isPrime && result;

Prime.threadCtr++;

}

static class Checker implements Runnable {

private long start;

private long end;

public Checker(long start, long end) {

this.start = start;

this.end = end;

}

public void run() {

boolean tempIsPrime = true;

for(long i = start; i <= end; i++){

if(num % i == 0){

tempIsPrime = false;

break;

}

}

update(tempIsPrime);

}

}

}

**II. Data Gathering**

The following section with illustrate the findings in terms of time and memory usage.

|  |  |
| --- | --- |
| **x** | **Average Execution Time out of Five Trials for 2x Threads (in seconds)** |
| 0 | 0.5402 |
| 1 | 0.2952 |
| 2 | 0.2642 |
| 3 | 0.2818 |
| 4 | 0.2162 |
| 5 | 0.2416 |
| 6 | 0.2392 |
| 7 | 0.3902 |
| 8 | 0.4742 |
| 9 | 0.6036 |
| 10 | 0.6478 |
| 11 | 1.0128 |

**Table 1 – Average Execution Time For Various Thread Counts**

After testing for 1 to 2048 threads, sequentially doubling the number of threads, with the average of five trials being taken, it was found that the trend in generally the more threads, the more time is spent in processing. Computing the correlation coefficient gives 0.6505 which denotes a relatively weak but positive correlation between number of threads and time. The full results are shown in Table 1.

**Figure 1 – Graph of Average Execution Time For Various Thread Counts**

However, there is a section wherein time is optimized, as seen in Figure 1, since, for a few exponentiationspast a non-threaded implementation, the time was decreasing, being optimized at 23 or eight threads, finishing the computation with the average time of 0.2162 seconds.

|  |  |
| --- | --- |
| **x** | **Average Memory out of Five Trials for 2x Threads (in MB)** |
| 0 | 2 |
| 1 | 2 |
| 2 | 2 |
| 3 | 2 |
| 4 | 2.4 |
| 5 | 2.8 |
| 6 | 2.8 |
| 7 | 3 |
| 8 | 3 |
| 9 | 3.2 |
| 10 | 4 |
| 11 | 6 |

**Table 2 – Average Memory For Various Thread Counts**

Proceeding with a similar test structure for memory, a positive correlation was also found, with a correlation coefficient of 0.8498, implying a strong positive correlation. The results can be seen in Table 2.

**Figure 2 – Graph of Average Memory Usage For Various Thread Counts**

For a few exponentiations after a non-threaded implementation, the memory was neither increasing nor decreasing, but once 24 or sixteen threads were used, the memory used increased by a significant 400 MB.

**III. Analysis and Conclusion**

Having more threads increases the speed of the program up to a certain point, after which having a greater number of threads starts to have the opposite effect.

In terms of memory usage, having more threads does not affect the memory usage at all, again, up to a certain point wherein it starts to have a negative effect, increasing memory usage.

In general, having a greater number of threads is useful in decreasing execution time, with little to no effect on memory usage, up to a certain point wherein time is already being wasted allocating memory and processing threads that have no distinct tasks to do.