UNIVERSITY OF WISCONSIN PARKSIDE

An Analysis of two Multi-Threaded Programs

Homework 1

CSCI 370: Operating Systems

Author: *Tanner Smith*

Instructor: *Susan Lincke*
Department: Computer Science
Date: March 27, 2023

Contents

1 Introduction

Parallel programming has become a very big field in modern computing. Most modern processors that are releasing today have at least a minimum of 2 cores, and on average 4-8 cores. Designing programs that can take advantage of these abundance of cores is a really important problem for programmers and computer scientists. Moreover, how do operating systems properly assign tasks to each core to be as efficient as possible with the available hardware. The goal of this lab is to compare and analyse the problem of creating an efficient multi threaded program, compare actual speedup versus expected speedup, and see how operating systems handles managing a multi threaded program with varying threads amounts.

2 Method

2.1 About the Test Programs

In order to test multi threaded programs we will be using two multi threaded java programs, one to calculate prime numbers, and another to calculate square roots. Both of these programs are called from the same main class. The main class asks for which program to launch, how many numbers to calculate (1 to n), and how many threads should the task be split upon.
Both programs had methods of running. (See Code)

- System.out.println() each result
- Storing results to a Linked List and doing a System.out.println() of the list at the end
- Storing results to a Linked List but not printing

2.2 System Configuration

CPU: AMD Ryzen 7 2700x 8 core 16 thread Processor, 3700Mhz.
Memory: 32GB of DDR4 3000Mhz
OS: Windows 11 Pro ver 10.0.22000 Build 22000

2.3 The Tests

Both programs were tested running within the Intellij IDE. Each program was tested using 1, 2, 4, 8, 16, 32, 64 threads. This spacing of number of threads to test was chosen because the target processor had 16 hardware threads. This allows results of varying % usage of the total number of available hardware threads. For each thread range the programs were tasked to compute a range of 100 to 1 million numbers increasing by a power of 10 each time. For example... the first test would have a program calculate numbers from 0 to 100 for all thread ranges, and the last test would have the program calculate numbers from 0 to 1 million (for all thread ranges). Time recorded is wall clock time from the first thread creation to when the last thread finishes execution. In addition to these tests smaller case study tests were conducted on individual thread times in order to compare the fairness of time each thread spent working.

3 Results

Below are tables of results from running the programs. **All time is in (ms)**

3.1 Square Roots All Thread Results

Table 1: Square Roots. Printing Each Number.

	Number of Threads						
Numbers calculated	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	17	16	19	21	28	34	46
1000	26	29	26	32	35	50	69
10,000	54	71	77	78	93	95	108
100,000	253	361	396	410	417	428	435
1,000,000	1733	2574	2830	2841	2935	3181	3153

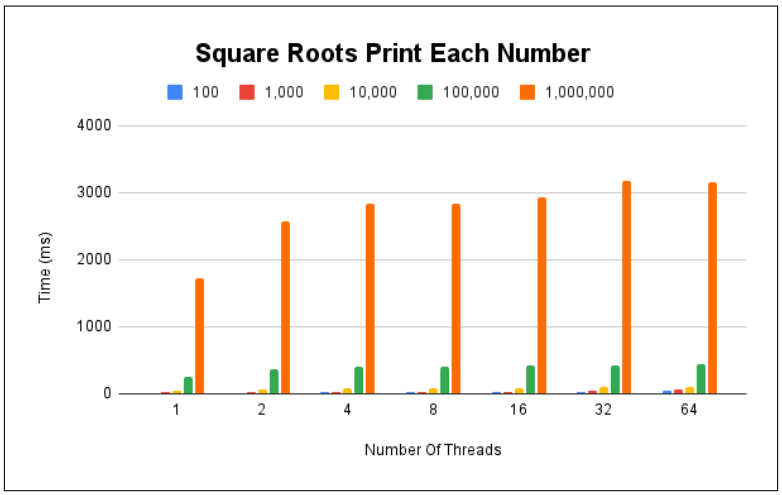


Table 2: Square Roots. Save to Linked-List. Print List at End

	Number of Threads						
Numbers calculated	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	13	15	12	14	17	22	40
1000	16	15	15	15	17	22	35
10,000	27	30	33	32	28	30	36
100,000	73	73	68	66	68	80	100
1,000,000	697	468	327	286	286	340	323

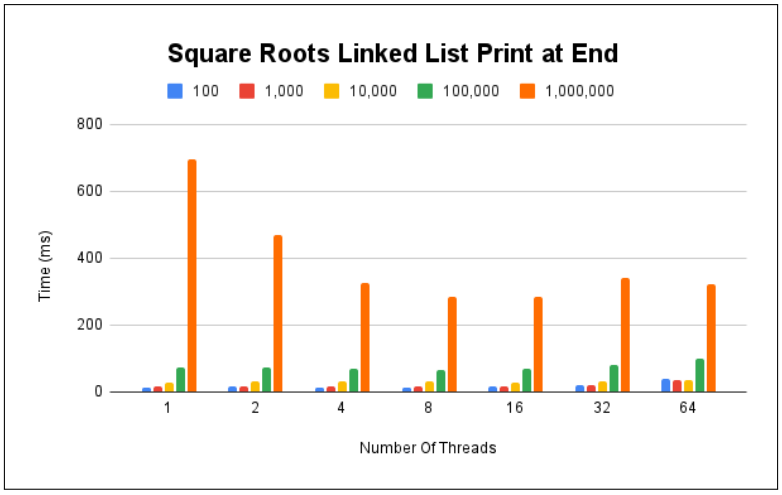
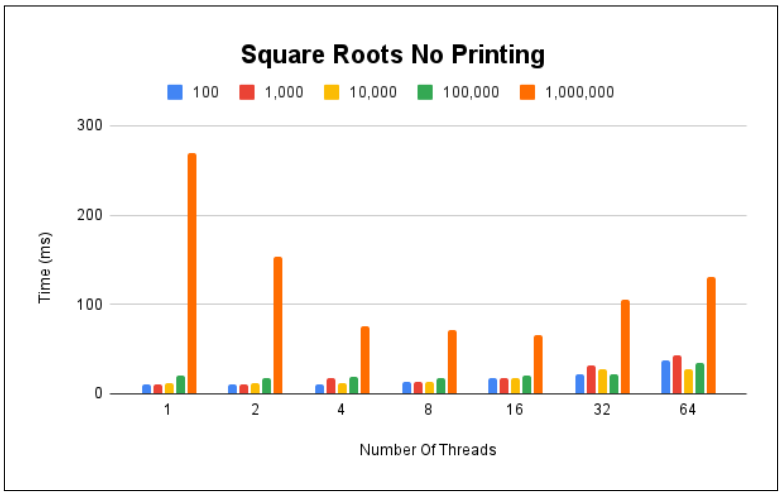


Table 3: Square Roots. Save to Linked-List. No Printing

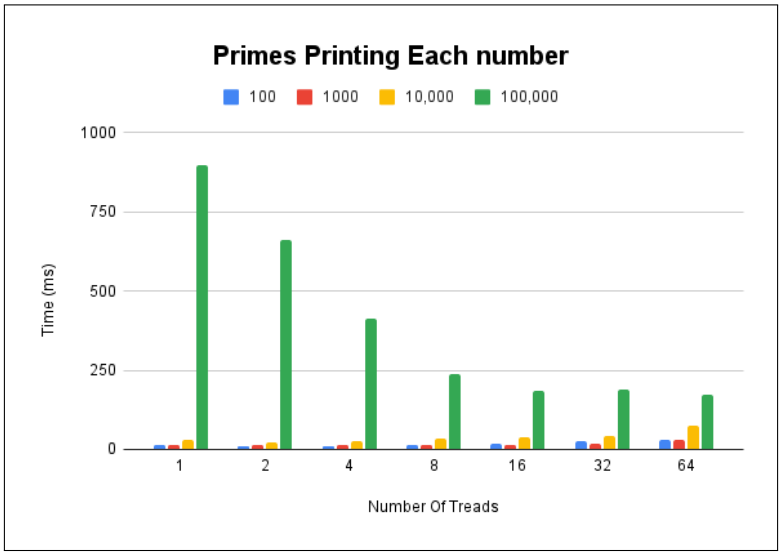
	Number of Threads						
Numbers calculated	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	10	10	11	13	17	22	38
1000	11	10	17	14	18	32	43
10,000	12	12	12	14	17	27	28
100,000	20	17	19	18	21	22	34
1,000,000	270	154	75	72	65	105	131
100,000,000	32652	17149	5482	3493	2833	2962	2707



3.2 Primes All Threads Results

Table 4: Primes. Print Each Result

	Number of Threads						
Numbers Calculated	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	16	11	12	14	17	25	30
1000	14	13	14	15	16	19	31
10,000	31	24	27	36	37	44	77
100,000	899	663	415	237	185	188	172
1,000,000	70654	52436	31040	17940	11518	9968	9625



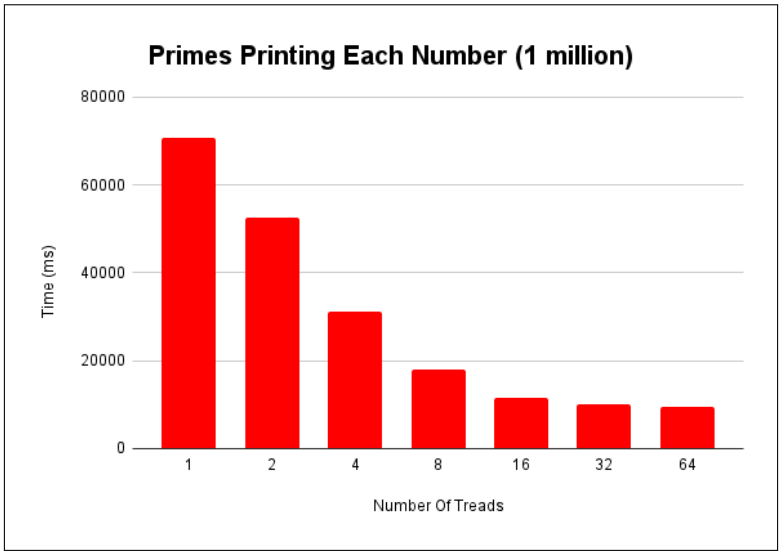
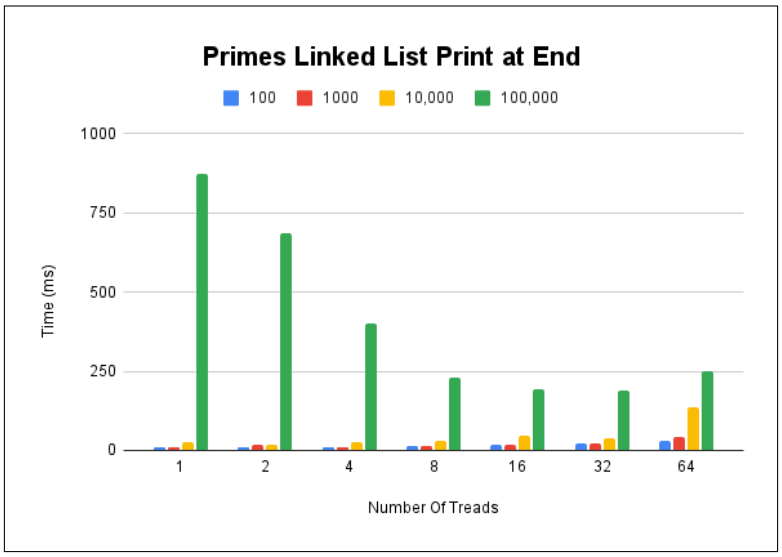


Table 5: Primes. Save to List. Print List at End

	Number of Threads						
Numbers Calculated	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	12	10	12	15	18	23	32
1000	12	17	12	13	17	22	44
10,000	25	20	27	30	48	37	135
100,000	873	688	401	228	193	188	251
1,000,000	69993	52288	31202	17956	11207	9697	9314



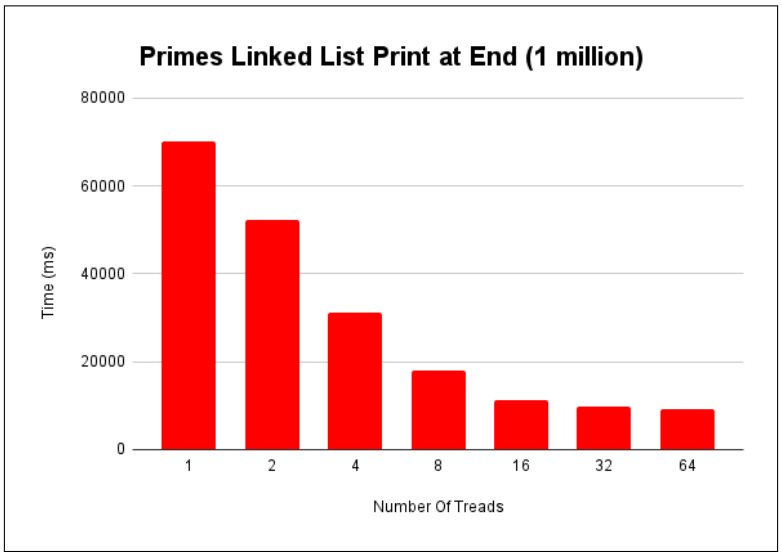
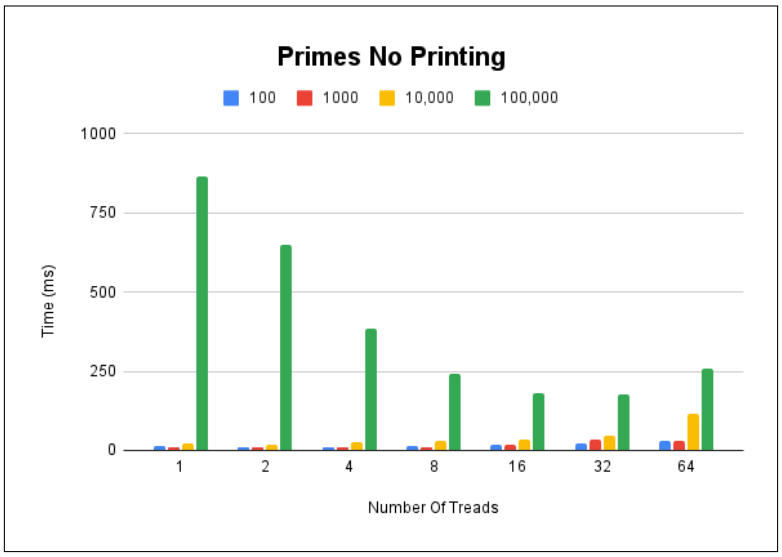
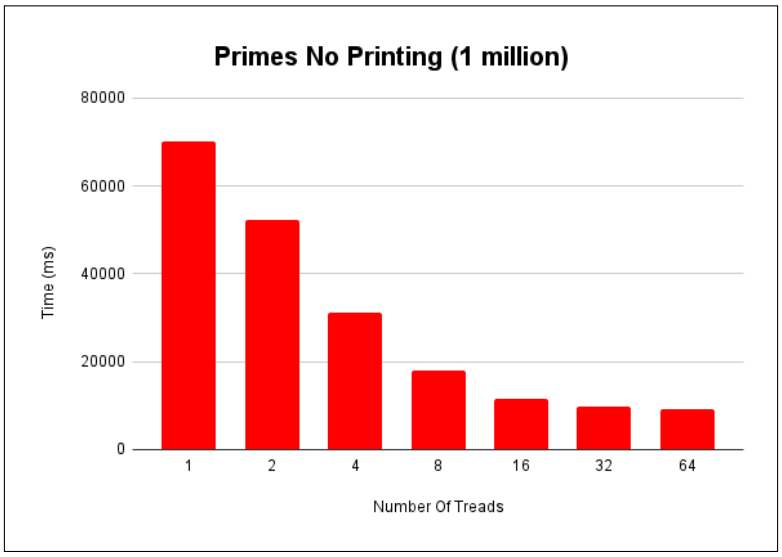


Table 6: Primes. No Printing

	Number of Threads						
Numbers Calculated	**1**	**2**	**4**	**8**	**16**	**32**	**64**
100	15	11	11	14	18	21	32
1000	12	11	12	12	18	33	29
10,000	24	20	25	30	35	45	115
100,000	867	651	384	244	182	179	258
1,000,000	69978	52266	31097	17978	11463	9893	9251





3.3 Individual Thread Test Results

Table 7: Individual Thread Times For Primes n = 1,000,000

Thread #	Number Range	Time	Thread #	Number Range	Time
0	0-62500	732	8	500000-562500	7585
1	62500-125000	1933	9	562500-625000	8285
2	125000-187500	2997	10	625000-687500	8762
3	187500-250000	3907	11	687500-750000	9584
4	250000-312500	4750	12	750000-812500	10073
5	312500-375000	5648	13	812500-875000	10442
6	375000-437500	6398	14	875000-875000	10964
7	437500-500000	6846	15	937500-1000000	11192

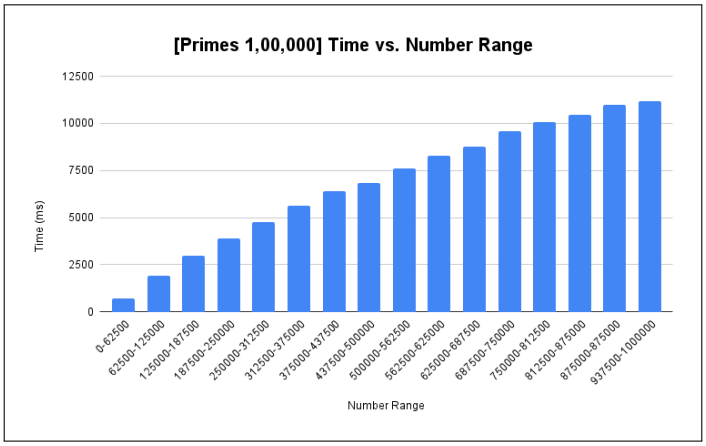
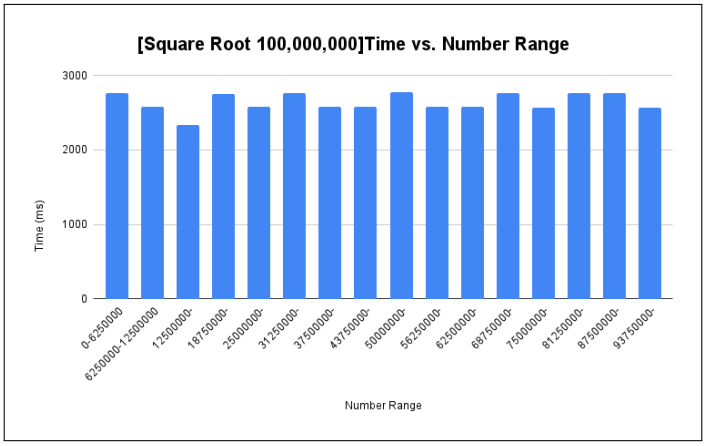
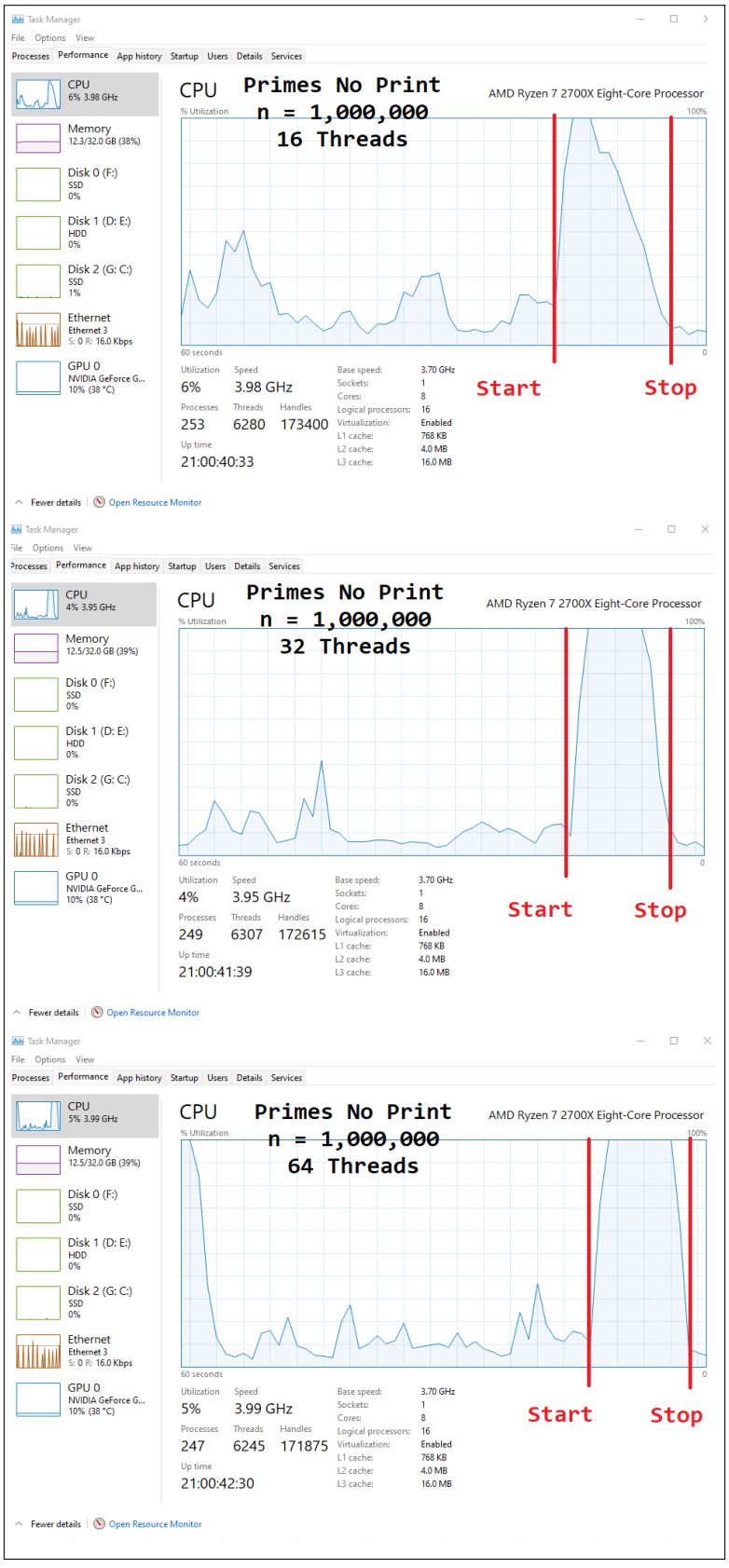


Table 8: Indiviual Thread Times for SqrRoot n = 100,000,000

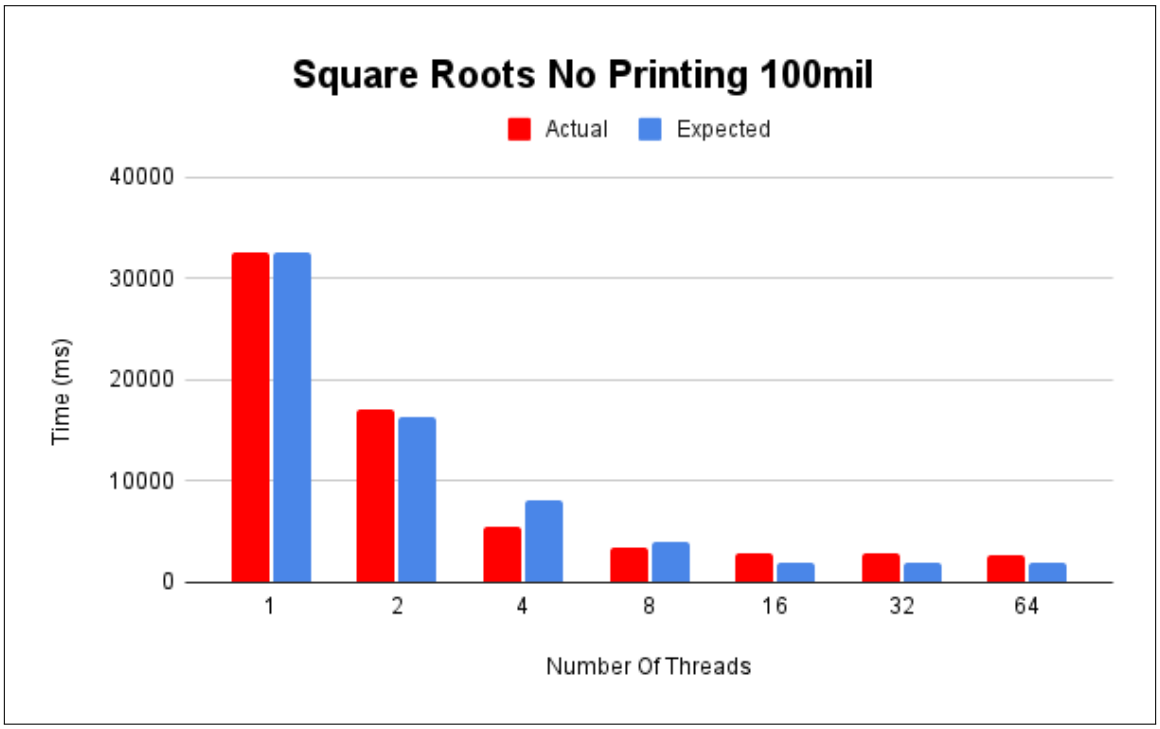
Thread #	Number Range	Time	Thread #	Number Range	Time
0	0-6250000	2761	8	50000000-56250000	2774
1	6250000-12500000	2572	9	56250000-62500000	2575
2	12500000-18750000	2329	10	62500000-68750000	2583
3	18750000-25000000	2755	11	68750000-75000000	2757
4	25000000-31250000	2584	12	75000000-81250000	2568
5	31250000-31250000	2762	13	81250000-87500000	2756
6	37500000-43750000	2574	14	87500000-93750000	2760
7	43750000-50000000	2578	15	93750000-100000000	2571

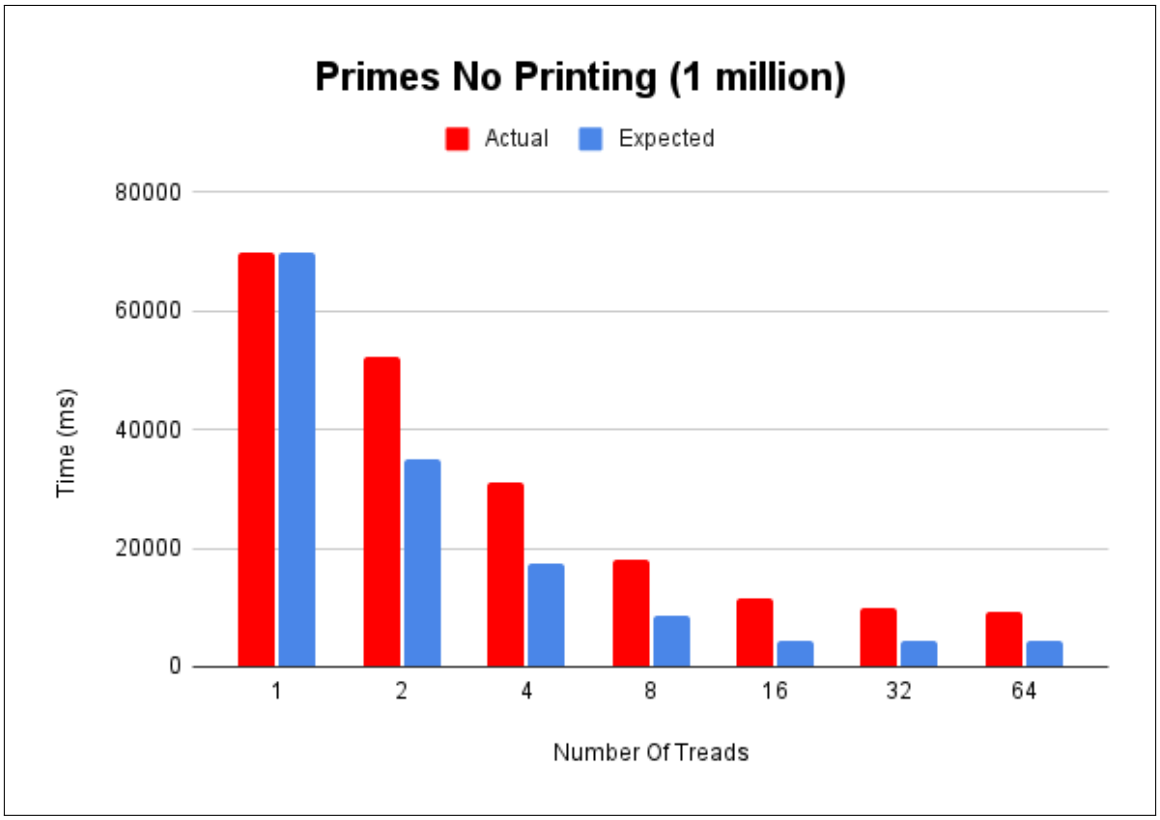


3.4 CPU Utilization on Primes No Print



3.5 Some Expected vs Actual Results





4 Analysis

4.1 Expected Speedup vs Observed

There is a lot to unpack here with tons of interesting results from these test runs. In general let's look at the observed speedup vs the theoretical speedup of increasing the number of threads. In theory every time we increase the number of threads (up to 16) the physical hardware limit, we should see a double in performance. Two threads should be twice as fast as 1 thread, 4 threads should be twice as fast as two threads and so on. The only tests that come close to matching this theoretical increase in speed are the tests runs at very large n values. The biggest jump in speed increase usually occurred going from one to two threads with diminishing returns until 16 threads (the max for this hardware). So why is this the case? There could be many reasons for this. This is real hardware running on a real operating systems. There are more than just the test programs running on the computer and the operating system needs to try and balance the work load fairly across all hardware threads. One hardware thread could happen to have more processes scheduled to run on it than another slowing down and individual threads performance for the test program. Another thing is if we were to see a two times increase in performance each time we increased our thread count by two, that would assume each thread runs equal time for the split work load. If we look at the two charts of the expected vs actual speedup between the two programs we can see the square root program more closely follows the expected speedup than the primes program. This could imply that the two programs don't equally split the work fairly. We will look into individual thread's performance in a later part of this analysis section.

4.2 Printing Each Result as They are Calculated

How did the different methods of displaying results effect the overall time as we increased threads? The longest run times were achieved by the mode to System.out.println() each result as they were calculated. In this mode on the square root program, we saw an increase in time to run as we increased thread counts. (Shown in Table 1) Why? Shouldn't increasing threads lower the run time? In contrast the prime number program saw a decrease in time to run as we increased thread counts. I believe this is happening because the square root program needs to print much more than the prime number program. For each number the square root program will print it's calculated square root. The prime number program only needs to print each time it finds a prime number. The standard output for the operating system is shared between all threads and only 1 thread can print to it at a time. If one thread needs to print but another thread is currently printing then it has to wait until the first thread is finished. With the square root program, since every number calculated gets printed, at any given time all n threads are usually attempting to print to the standard output. This causes a lot of downtime for the threads waiting to print instead of actually computing numbers. The prime number program doesn't run into this issue because not every number is prime, and each thread is only printing a limited number of times. This reduces the likely hood of two threads simultaneously needing to print.

4.3 Store to List then Print and No Printing

As we saw in the previous subsection having each thread print each result as they are calculated can cause slowdowns in the case of the square root program when many threads are attempting to print at the same time. Instead of printing each result as calculated what if we instead save them to a list and then print the entire list in one go once it finished calculated all the numbers? I ended up choosing to use a linked list over an array list for speed. Internally array lists store an array that doubles in size each time it reaches it's capacity. Which requires the old array to be copied to an entirely new array each time. A linked list lets us quickly add new results to the end at the cost of using more memory. Storing results into a list like this then printing caused the square root program to experience a speedup as threads increased instead of slowing down (Table 2). However, the primes number program did not see much of a difference in run time compared to printing each result (Table 5). In general the primes number program did not see much a difference in run times over all 3 testing methods compared to the square root program seeing an increase in speed from printing each result, to printing the list, to no printing at all. The square root program does a lot more printing than the prime program and comparing the two across these tests really shows how slow it is to print a ton of data to the standard output.

4.4 How Fair Were The Two Programs

One of the goals sated in the introduction was to observe the challenge of writing efficient multi-threaded programs. Sometimes it's difficult to split a program evenly into multiple threads such that each thread gets an equal amount of work to compute. Looking at the results of the individual thread times between both programs calculating a range on 1 million numbers we can see that the square root program split it's compute time across all threads more fairly compared to the prime number program. The square root program is much more fair because each thread is calculating the same amount of values. If the range of numbers is 75 thousand per thread then each thread in the square root program will calculate 75 thousand individual numbers. This can not be said about the prime numbers program. As the number to be determined as prime increases, so too does the time it takes to determine weather or not it is prime. The prime program needs to check from 0 to $\frac{n}{2}$ for each number it wants to determine is prime. So the thread tasked with calculated 0 to 75 thousand will take less time than the thread tasked to calculate 225 thousand to 300 thousand. There are more efficient ways to calculate prime numbers but this method easier to split into a multi threaded program. Not every program can be multi-threaded easy and we saw in this example that just because a program is multi-threaded it's an efficient multi-threaded implementation. Designing multi-threaded programs takes careful consideration and time.

4.5 Going Above the Physical Hardware Thread Count

In theory we should only get speedup up to the total amount of hardware threads the system has. For the most part we do see that. In the data as we test on 32 and 64 threads we see time increase over the 16 thread runs. Just because we can split the task to double or quadruple the hardware thread count doesn't mean we are going to get a speedup. Most cases the slowdown going to 32 or 64 threads came from a few places. The operating system needs to manage swapping between the extra threads since there are more threads per cores. Also the parent program does spent some extra time creating the threads. This extra time from the parent is more exaggerated in lower n counts. At lower counts of numbers to compute it was on often faster to have 1 or 2 threads compute then spend the extra time creating up to 16 threads just for a thread single thread to compute like 100 to 1,000 numbers. Table 1 shows this behaviour.

Interestingly there is one exception where going over the hardware thread count actually saw an even greater increase in speeds than just doing 16 threads (the max). It was faster to use 32, and 64 threads to calculate the largest amount of prime numbers. (See Table 6). I believe this is due to the unfairness of the prime number program as discussed in the previous sub section. Imagine we have 16 threads to compute prime numbers. We know and can see from the results section that the threads that calculate the lower ranges of numbers finish much faster than the threads that calculate the highest range of number. So on our 16 thread version the first thread will finish relatively fast compared to the other threads, leaving behind an empty hardware thread that is no longer being used to compute prime numbers. But in the case of 32 threads, once the quicker threads finish the remaining threads can begin taking advantage of the unused cores they leave behind. This results in a slightly faster run time because cores are on average all being used instead of some being empty. This finding made me go back and look at the CPU utilization for 16, 32, and 64 threads to confirm weather or not this analysis was right. Looking at the CPU Utilization image we can see that the 32 and 64 thread test runs stayed at 100% CPU utilization longer than the 16 threaded program confirming the hunch.

5 Conclusion

Overall there are a few key concepts in this assignment to take away. One would be that multi-threading never really sees its expected results. There's tons of co founding variables that can affect an individual threads performer. How different would my results be on the same machine running a different operating system. Task scheduling plays a huge role and recently a big update came out to the Linux kernel that is said to greatly improve task scheduling across the board. I do wish I had more time to work on this because I wanted to do more test runs on my 4 core laptop running windows 10 and compare the results to my main computer running windows 11 where I did the testing. I also wonder how difficult it would be to refactor the primes program to be more fair across threads. It took me awhile to figure out why the primes program was running faster on a thread count higher than my computer's actual amount of hardware threads. When I finally ended up digging deeper I found that the CPU was being utilized better. This assignment provided good crash course into learning how to make multi threaded programs, their benefits and drawbacks, and resource sharing bottlenecks.

6 Code Used

6.1 Main Class

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class Main
{


    public static void main(String[] args)
    {
        try
        {
            //Get input of which program to run
            Scanner input = new Scanner(System.in);
            System.out.println("Enter S for square roots and P for prime
                numbers");
            String choice = input.nextLine();

            System.out.println("How many numbers?");
            int n = input.nextInt();

            System.out.println("How many threads? Enter 0 to use max
                hardware threads (" + Runtime.getRuntime().
                availableProcessors() + ") on this PC");
            int threadAmt = input.nextInt();
            if (threadAmt < 1)
            {
                threadAmt = Runtime.getRuntime().availableProcessors();
            }

            int range = n / threadAmt;
            int begin = 0;
            Thread[] threads = new Thread[threadAmt];
            long startTime = System.currentTimeMillis();

            //Create and run # of threads
            for (int i = 0; i < threadAmt; i++)
            {
//                System.out.println(begin + "  " + (begin + range));
                Thread t;
                if (choice.equalsIgnoreCase("S"))
                {
                    SquareRoots rootCalculator = new SquareRoots(begin,
                        begin + range);
                    t = new Thread(rootCalculator);
                } else if (choice.equalsIgnoreCase("P"))
                {
                    PrimeNumbers primeCalculator = new PrimeNumbers(begin,
                        begin + range);
                    t = new Thread(primeCalculator);
                } else
                {
```

```
                    return;
                }
                t.setName("Thread " + i + " From: " + begin + ", To: " + (
                    begin + range));
                threads[i] = t;
                t.start();
                begin += range;
            }

            //Wait for all threads to finish
            for (Thread thread : threads)
            {
                thread.join();
            }

            long endTime = System.currentTimeMillis();
            long totalTime = endTime - startTime;
            System.out.println("Total time was: " + totalTime + "
                milliseconds");

        } catch (InputMismatchException | InterruptedException e)
        {
            System.err.println("INVALID INPUT. EXITING");
            System.exit(1);
        } //End Try Catch

    }//End void Main

} //End Class Main
```

6.2 PrimeNumbers Class

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class PrimeNumbers implements Runnable
{
    private int start_;
    private int end_;
    private List<Integer> resultList;

    public PrimeNumbers(int s, int e)
    {
        start_ = s;
        end_ = e;
        resultList = new LinkedList<>();
    }

    //Bug (misses 2 and 3)
    private void calculate(int start, int end)
    {
        double totalSum = 0.0;
        for (int n = start; n < end; n++) //Loop through range
        {
            for(int i = 2; i<=(n/2); i++) //Check up to half of current
                value
            {
                if(n % i == 0)
                {
                    break; //Not a prime
                }
                if(i>=(n/2))
                {
                    //Found prime
                    //System.out.println("Prime found: " + n);
                    resultList.add(n);
                }
            }
        }
        //System.out.println(Arrays.toString(resultList.toArray()));
    }

    @Override
    public void run()
    {
        long startTime = System.currentTimeMillis();
        calculate(start_, end_);
        long endTime = System.currentTimeMillis();
        long totalTime = endTime - startTime;
        System.out.println("DONE! " + Thread.currentThread().getName() + "
            TOTAL TIME: " + totalTime + " milliseconds");
    }
}
```

6.3 SquareRoot Class

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class SquareRoots implements Runnable
{

    private int start_;
    private int end_;
    private List<Double> resultList;

    public SquareRoots(int s, int e)
    {
        start_ = s;
        end_ = e;
        resultList = new LinkedList<>();
    }

    private void calculate(int start, int end)
    {
        double totalSum = 0.0;
        for (int n = start; n < end; n++)
        {
            double root = Math.sqrt(n);
            totalSum += root;
            //System Print Statement
            //System.out.println("N = " + n + ", sqrRoot = " + root);
            resultList.add(root);
        }
        //System.out.println(Arrays.toString(resultList.toArray()));
    }


    @Override
    public void run()
    {
        long startTime = System.currentTimeMillis();
        calculate(start_, end_);
        long endTime = System.currentTimeMillis();
        long totalTime = endTime - startTime;
        System.out.println("DONE! " + Thread.currentThread().getName() + "
            TOTAL TIME: " + totalTime + " milliseconds");
    }
}
```