Concurrency Project  
Prime Numbers

## 26th February 2021

# Bachelor Of Science (Honours) Software Development

**Graphical user interface

Description automatically generated**

**Liliana O’Sullivan**

C00227188

**Joseph Kehoe**

Lecturer

# Table of Figures

[Figure 1 Sieve of Eratosthenes array creation 4](#_Toc64912347)

[Figure 2 Sieve of Eratosthenes post-marking 2 4](#_Toc64912348)

[Figure 3 Sieve of Eratosthenes post-marking 3 5](#_Toc64912349)

[Figure 4 Sieve of Eratosthenes post-marking 5 5](#_Toc64912350)

[Figure 5 macOS relative and absolute speedup 6](#_Toc64912351)

[Figure 6 Linux relative and absolute speedup 7](#_Toc64912352)

[Figure 7 macOS time taken increasing Threads 8](#_Toc64912353)

[Figure 8 Linux Mint VM time taken with increasing threads 9](#_Toc64912354)

# Table of Contents

[Table of Figures 2](#_Toc65146736)

[Algorithm Overview 4](#_Toc65146738)

[Speedup Results 6](#_Toc65146739)

[Scalability 8](#_Toc65146740)

[Additional Information 12](#_Toc65146741)

# Algorithm Overview

This project uses the “Sieve of Eratosthenes” algorithm for the discovery of prime numbers. This algorithm marks multiples of each prime as none prime, continually until all multiples have been tried. This algorithm is efficient in finding small primes, as it exchanges memory usage for low computational complexity. Once a large N is provided, the memory complexity required can become too large of a process.

The steps taken by the algorithm can be summaries as follows

1. Create an array from one to N, where N is the number to find primes up to
2. Beginning at the number two, eliminate all multiples of two that are less than N
3. Choose the next none-eliminated number, and remove all multiples of it less than N
4. Once all none-marked elements have been checked for elimination, the remaining elements after one are all primes.

If the algorithm is tasked with an N of thirty, it will begin by creating an array of 30 elements, as shown below.

A picture containing text, white

Description automatically generated

Figure Sieve of Eratosthenes array creation

It first eliminates multiples of two, without including two. The result can be seen below in lilac.

A picture containing text, furniture, white, chest of drawers

Description automatically generated

Figure Sieve of Eratosthenes post-marking 2

The next number in the list to be found is the three; All multiples of three are removed, without including the three. This is shown in orange.

Chart, waterfall chart

Description automatically generated

Figure Sieve of Eratosthenes post-marking 3

Five is next to be removed, with only twenty-five being removed due to previous iterations removing overlapping multiples of five; This is shown in red.

Chart

Description automatically generated

Figure Sieve of Eratosthenes post-marking 5

At this point, the remaining digits will be checked using the same approach, though all primes have been calculated. Not including the initial one, the primes up to 30 are 2,3,5,7,11,13,17,19,23 and 29.The pseudocode can be seen below for this algorithm.

def sieve\_of\_eratosthenes(N):

sieve = [True for every number in range 0 .. N]

i = 2

for i=2; i\*i <= N; ++i:

if sieve[i] == True:

for j=i\*2; j<N ;j+=i:

sieve[j] = False

return sieve

Once the prime numbers have been calculated, a count of the twin primes can begin.

def twin\_prime\_count(sieve):

count = 0

for i=3; i<len(sieve)-2 ;i+=2:

if sieve[i] == True and sieve[i+2] == True:

count+=1

return count

To achieve concurrency within the project, OpenMP (Open MultiProcessing) was used. It is enabled by passing a -fopenmp flag to the compiler. OpenMP predominantly uses compiler directives to add parallelism to the application. It does this with simplistic syntax and allows the transition of sequential code to parallel code with relative ease through the additional of OpenMP pragmas.

OpenMP allows memory-sharing among its threads; This is fundamental to the execution of the algorithm used in this project. A sieve array had to be shared across multiple threads to concurrently decide which numbers needed to be marked as none-prime.

The schedule type of a loop can be set through the pragmas. OpenMP specifically provides (Corner, 2016)

* Static
* Dynamic
* Guided
* Auto
* Runtime

All of the above were tested, and Dynamic yielded greatly decreased execution time overall in the marking of primes; Dynamic provided lower execution time than auto and runtime.

# Speedup Results

Relative and absolute speedup is calculated using the formulae below

An example of both are as follows, to calculate it for 16 threads, the following variables would be inserted into the formulae

Applying this to every recorded thread count within the Mac environment provides the following graph

Figure macOS relative and absolute speedup, N = 1 Billion

Applying the same formulae to the Linux environment yields the following graph

Figure Linux relative and absolute speedup, N = 1 Billion

# Scalability

To test the scalability of the code, the codebase was tested in two environments. The first run on native macOS running Big Sur 11.2.1. The hardware specifications can be found in the table below.

|  |  |
| --- | --- |
| Model | MacBook Pro 16-inch 2019 |
| Processor | Intel Core i9 9880H |
| Processor Speed | 2.3 GHz, boosting up to 4.8 GHz |
| Core Count | 8 Cores/16 Threads with Intel Hyperthreading |
| RAM | 16 GB 2667 MHz DDR4 |
| Hard Drive | Apple 1 TB NVME |
| GPU | AMD Radeon Pro 5500M 4 GB |

A virtual machine (VM) running Linux Mint was additionally created running on the hardware above. VirtualBox was used to emulate Linux Mint 20.1 – Cinnamon 64 bit. This virtual machine was allocated 8 CPU Threads, 10171 MB of RAM, 128 MB of video memory and 30 GB of hard drive space. All benchmark results are averages from three executions on each platform. The macOS environment can be seen in the graph below.

Figure macOS time taken increasing Threads, N = 1 Billion

Peak performance here is attained at 16 threads, the maximum the tested hardware supported. Performance leading up to the 16 threads steadily increased with a decrease once passed the peak execution time. As macOS natively uses Grand Central Dispatch for task parallelism, the use of OpenMP might be causing additional overhead.

Within the emulated Linux environment, the following results were obtained.

Figure Linux Mint VM time taken with increasing threads, N = 1 Billion

Within the Linux Mint virtual environment, increasing the number of threads net a positive impact on the execution time of the application continuously.

Additionally, results have been gathered for N = 1 Million and N = 500,500,000, a halfway point between 1 Million and 1 Billion. The curves attained from these additional values of N mimicked the curvature of N = 1 Billion on both environments.

Figure macOS time taken with increasing threads, N = 500,500,000

Figure Linux Mint time taken with increasing threads, N = 500,500,000

Figure macOS time taken with increasing threads, N = 1 Million

# Additional Information

## Largest N Tested

A screenshot of a computer

Description automatically generated with medium confidenceText

Description automatically generatedA picture containing text, black, screenshot, silver

Description automatically generatedAs part of the testing, the largest N that was tested on the hardware specified within the scalability section was an N of 13 Billion. This was run on macOS, and counted a total of 6,718,032 Twin Primes, running with 16 Threads and totalling 2 hours and 18 minutes of execution time; Within user-space time, this totals to almost 19 hours. It used 13.97 GB of physical RAM, with a total of 18.16 GB of virtual memory.

A screenshot of a computer

Description automatically generated with medium confidenceAt an N of 1 Trillion, macOS reports the process reaching 935.52 GB of Virtual Memory usage, with it swiftly killing the process.

## Recreating in Python

A screenshot of a computer

Description automatically generated with medium confidenceThe C++ codebase was recreated in Python, in the hope of the Python interpreter possibly being capable of performing its’ own memory swapping, or another form of memory optimisation to enable it to execute an N of 1 Trillion, however, MacOS swiftly killed the process too, once its virtual memory size of 7.28 TB began allocation.

The Python equivalent codebase is included with the main project as main\_asPy.py

## Segmentation Fault

At initial development and testing of the algorithm, the array used as a sieve was originally allocated on the stack. Once large N numbers, such as 1 Billion, were used a segmentation fault was being raised. My understanding is a stack overflow was occurring that caused the segmentation fault. Changing the array to be stored on the heap through the use of the new keyword resolved the issue. The method named “sequential\_stack” remains in the main.cpp file.

## Linux Scalability Trendline

Within the Linux Mint VM environment for an N of 1 Billion, results were gathered for 2048 and 4096 threads. This was excluded from the scalability sections as for other N values results were not gathered for consistency as the macOS environment struggled to compute at 1024 threads. However, this has been graphed below, with an exponential trendline.

All benchmark results are available at:

https://www.dropbox.com/sh/okug5zww9gyry5d/AAA9K355umlE27qVP-X\_5lGla?dl=0

# Bibliography

Corner, J., 2016. *Jaka's Corner.* [Online]   
Available at: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html  
[Accessed 23rd February 2021].