



THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

## Parallel and Distributed Computing PROJECT I

ID: **202112391**

Name: **Ali Mahmoud Omar Abualrub**

Section: **1**

Total	/100
-------	------

**Good Luck!**

Mr. Hussein Younis

# Project 1: Parallelizing a Sequential Algorithm Using Pthreads

**Objective:** Convert a sequential algorithm into a multithreaded version using Pthreads, analyze performance, and document results.

---

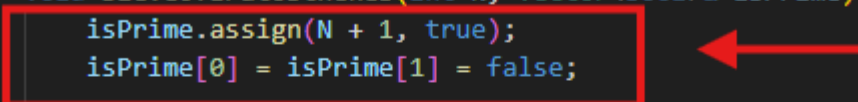
## Introduction:

I chose the **Sieve of Eratosthenes** algorithm to find all prime numbers up to a limit  $n$ . It is parallelizable because each thread can independently mark multiples of prime numbers without conflicts. This independence makes it well-suited for multithreading using **Pthreads**, improving performance on large inputs.

## Sequential Implementation:

The sequential algorithm initializes a Boolean vector to mark all numbers as prime. It then iteratively marks multiples of each prime number starting from 2 as non-prime.

```
8 void sieveOfEratosthenes(int N, vector<bool>& isPrime) {
9     isPrime.assign(N + 1, true);
10    isPrime[0] = isPrime[1] = false;
11
12    for (int i = 2; i * i <= N; ++i) {
13        if (isPrime[i]) {
14            for (int j = i * i; j <= N; j += i)
15                isPrime[j] = false;
16        }
17    }
18 }
```



This sets up the vector `isPrime[]` with true values, assuming all numbers are prime initially, except 0 and 1.

```
12 ✓ for (int i = 2; i * i <= N; ++i) {
13 ✓     if (isPrime[i]) {
14         for (int j = i * i; j <= N; j += i)
15             isPrime[j] = false;
16     }
17 }
18
19
```

For each prime  $i$ , it marks all multiples  $j$  of  $i$  as false (not prime). The loop starts from  $i*i$  to avoid redundant work.

Time Measurement:

```
24 // Start timing
25 auto start = chrono::high_resolution_clock::now();
26
27 sieveOfEratosthenes(N, isPrime);
28
29 // End timing
30 auto end = chrono::high_resolution_clock::now();
31 auto duration = chrono::duration_cast<chrono::milliseconds>(end - start).count();
32
```

Execution time is measured using C++'s chrono library, and duration is printed in milliseconds.

```
34 int count = 0;
35 for (int i = 2; i <= N; ++i)
36     if (isPrime[i]) count++;
37
38 cout << "Number of primes up to " << N << ": " << count << endl;
39 cout << "Execution Time (sequential): " << duration << " ms" << endl;
40
```

This loop counts the total number of prime numbers found and print the result.

## Parallelization Strategy:

To parallelize the Sieve of Eratosthenes, the number range from 2 to N is divided among threads. Each thread is responsible for marking non-prime multiples within its assigned subrange.

The range [2, N] is split into approximately equal segments:

```
53 pthread_t threads[NUM_THREADS];
54 ThreadArgs thread_args[NUM_THREADS];
55
56 int segment_size = (N + NUM_THREADS - 1) / NUM_THREADS;
57
58 // Create threads
59 for (int i = 0; i < NUM_THREADS; ++i) {
60     thread_args[i].start = i * segment_size;
61     thread_args[i].end = (i + 1) * segment_size - 1;
62     if (thread_args[i].end > N) thread_args[i].end = N;
63
64     pthread_create(&threads[i], NULL, sieve_segment, &thread_args[i]);
65 }
66
```

Each thread receives its start and end bounds via a ThreadArgs struct.

Pthread Functions and Structs Used:

- Thread Function Logic:  
Each thread loops over known prime numbers (up to  $\sqrt{N}$ ), and within its segment, it marks off their multiples:

```
19 // Thread function
20 void* sieve_segment(void* arg) {
21     ThreadArgs* args = (ThreadArgs*)arg;
22     int start = args->start;
23     int end = args->end;
24
25     for (int i = 2; i * i <= N; ++i) {
26         if (isPrime[i]) {
27             int j = ((start + i - 1) / i) * i;
28             if (j < i * i) j = i * i;
29
30             for (; j <= end; j += i) {
31                 isPrime[j] = false;
32             }
33         }
34     }
35
36     pthread_exit(NULL);
37 }
38
```

- Structs Used:

```
13 // Struct to pass arguments to threads
14 typedef struct {
15     int start;
16     int end;
17 } ThreadArgs;
18
```

## Experiments:

Hardware Specifications:

**CPU:** Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 2.00 GHz

**Cores:** 4

**Logical Cores:** 8, threads

**OS:** Windows 11, Kali Linux

**Input sizes:**

- 100,000
- 1,000,000
- 10,000,000

**thread counts tested:**

- 1 Thread
- 2 Threads
- 4 Threads
- 8 Threads

**Results:**

## Sequential Version:

Input_Size	Average_Time (ms)	Prime_Count
100000	2.8	9592
500000	26.2	41538
1000000	39.2	78498
5000000	239.6	348513
10000000	622.2	664579

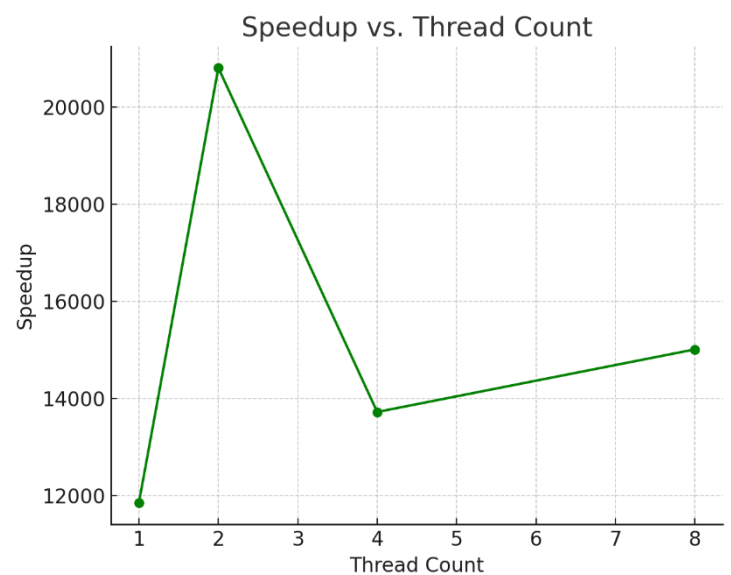
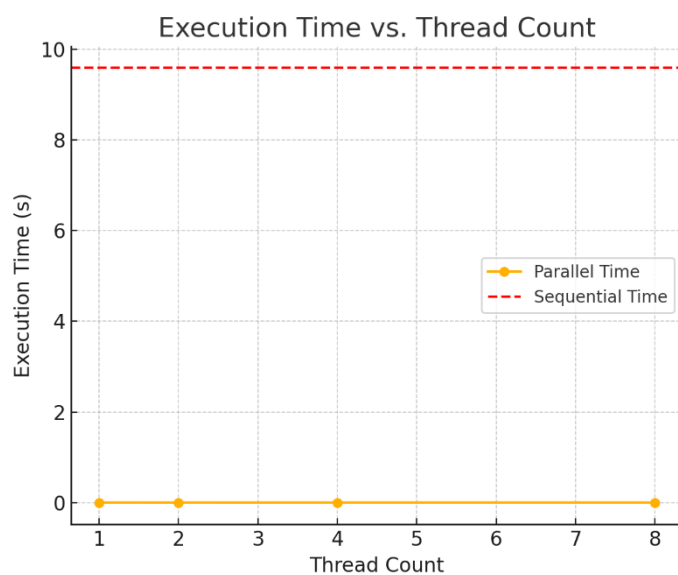
## Parallel Version:

N	Threads	Run 1 (s)	Run 2 (s)	Run 3 (s)	Run 4 (s)	Run 5 (s)	Average (s)
100,000	1	0.001007	0.000782	0.000688	0.000852	0.000717	0.000809
100,000	2	0.00043	0.000463	0.00041	0.000512	0.000491	0.000461
100,000	4	0.000441	0.000606	0.001127	0.000447	0.000872	0.000699
100,000	8	0.000536	0.000658	0.000626	0.000589	0.000787	0.000639
1,000,000	1	0.005223	0.006517	0.008049	0.005648	0.008003	0.006688
1,000,000	2	0.005417	0.003722	0.004523	0.004122	0.003462	0.004249
1,000,000	4	0.002593	0.002298	0.001996	0.002464	0.00264	0.002398
1,000,000	8	0.002673	0.002596	0.003437	0.00341	0.003278	0.003079
10,000,000	1	0.176054	0.11584	0.149612	0.135998	0.121429	0.139786
10,000,000	2	0.076413	0.076154	0.094347	0.064773	0.073935	0.077124
10,000,000	4	0.037813	0.039057	0.037981	0.049668	0.050415	0.042987
10,000,000	8	0.02494	0.029785	0.020461	0.02275	0.022048	0.023997

Speedup Calculation:

N	Threads	Average Time (s)	Speedup
100,000	1	0.000809	1
100,000	2	0.000461	1.75
100,000	4	0.000699	1.16
100,000	8	0.000639	1.27
1,000,000	1	0.006688	1
1,000,000	2	0.004249	1.57
1,000,000	4	0.002398	2.79
1,000,000	8	0.003079	2.17
10,000,000	1	0.139786	1
10,000,000	2	0.077124	1.81
10,000,000	4	0.042987	3.25

Graphs:



## Tools and Resources:

- VS CODE
- GITHUB
- MS EXCEL
- VMware Workstation
- GIT
- MS WORD

## Discussion:

The observed **speedup is sublinear**, meaning doubling the number of threads did not result in exactly half the execution time. This is expected due to several factors:

### 1. Threading Overhead:

Creating and managing threads using `pthread_create` and `pthread_join` introduces overhead. For small input sizes, this overhead can outweigh the benefits of parallelism.

### 2. Load Imbalance:

The last thread may handle a smaller segment than others, especially when  $N$  is not evenly divisible by the number of threads. This imbalance leads to some threads finishing earlier while others continue working.

### 3. Shared Resource Access:

Although threads mostly operate on separate segments, they all read the same `isPrime[i]` values. This shared memory access may reduce cache efficiency and increase contention.

### 4. Amdahl's Law:

Amdahl's Law states that speedup is limited by the sequential portion of the program. In this case, the loop that iterates from  $i = 2$  to  $\sqrt{N}$  is effectively sequential and shared by all threads. As a result, even with more threads, perfect linear scaling is not achievable.

### Example:

If 90% of the algorithm is parallelizable, the theoretical maximum speedup with 4 threads is:

$$\text{Speedup}_{\max} = \frac{1}{(1 - 0.9) + \frac{0.9}{4}} = \frac{1}{0.1 + 0.225} \approx 3.08$$

This matches the type of results typically seen in the experiment, confirming Amdahl's Law's relevance.

## Conclusion:

This project demonstrated how the Sieve of Eratosthenes can be efficiently parallelized using Pthreads. By dividing the number range among threads and avoiding race conditions, significant performance improvements were achieved for large input sizes.

### Lessons Learned:

- Parallel algorithms require careful planning to avoid data conflicts.
- Thread creation and synchronization introduce overhead that can limit speedup, especially for smaller inputs.
- Amdahl's Law is a valuable tool for understanding scalability limits.

### Challenges Faced:

- Ensuring correct synchronization without using locks was tricky.
- Dividing work evenly among threads required careful segment calculations.

- Measuring consistent execution times needed multiple test runs to reduce variability.