



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:

1. Initialization:

The algorithm begins by initializing a Boolean vector where each index represents whether the number is prime. assume all numbers are prime at the beginning:

```
9      std::vector<bool> is_prime(n + 1, true);
10     is_prime[0] = is_prime[1] = false;
11
```

Use a vector of size $n+1$ to cover all numbers from 0 to n , and 1 are marked as non-prime explicitly 0.

2. Main Logic – Sieve Loop:

The core of the algorithm uses nested loops. The outer loop selects each number p starting from 2, and the inner loop marks all multiples of p as non-prime.

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

The loop runs only up to \sqrt{n} because any composite number above that will already have been marked.

The inner loop starts from $p * p$ as all smaller multiples would have been marked by smaller primes.

3. Counting the Prime Numbers:

After marking non-primes, now count how many true values remain in the `is_prime` vector

```

20     int count = 0;
21     for (int i = 2; i <= n; ++i) {
22         if (is_prime[i]) {
23             count++;
24             if (print_primes) std::cout << i << " ";
25         }
26     }

```

This count is used to compare with the result of the parallel version.

4. Time Measurement:

Use the `<chrono>` lib to measure the execution time of the entire algorithm.

```

46     auto start = std::chrono::high_resolution_clock::now();
47     int prime_count = sequentialSieve(n);
48     auto end = std::chrono::high_resolution_clock::now();
49
50     std::chrono::duration<double, std::milli> duration = end - start;
51

```

This allows us to capture the time taken in milliseconds and later compare it to parallel version.


5. Sample Output:

For `n = 1000000`, the output may look like this:

```

PS C:\Users\abual> cd "c:\Users\abual\OneDrive\حطس
ial.cpp -o sequential } ; if ($?) { .\sequential }
Sequential Sieve for n = 1000000
Found 78498 primes.
Execution time: 69.226 ms

```



Parallelization Strategy:

1. Shared Data and Thread Struct:

Define a struct `ThreadData` to encapsulate all the necessary information that each thread needs to operate independently:

```

8     struct ThreadData {
9         int id;
10        int num_threads;
11        int n;
12        std::vector<bool>* is_prime;
13    };
14

```

`id`: the thread index (from 0 to `num_threads - 1`).

`is_prime`: a shared pointer to the boolean vector used for marking primes.

2. Initial Sequential Phase (up to \sqrt{n})

The first part of the algorithm is done sequentially. compute all primes up to \sqrt{n} , which are then used by the threads to mark multiples:

```

50     for (int p = 2; p <= sqrt_n; ++p) {
51         if (is_prime[p]) {
52             for (int i = p * p; i <= n; i += p) {
53                 is_prime[i] = false;
54             }
55         }
56     }

```

This ensures threads will only work with already known primes (avoiding complex synchronization).

3. Block Decomposition:

Each thread is responsible for marking a block (range) of numbers:

```

int block_size = (n - 1) / num_threads;
int start = 2 + id * block_size;
int end = (id == num_threads - 1) ? n : start + block_size - 1;

```

The full range $[2, n]$ is divided approximately evenly across all threads.
The last thread takes the remainder to avoid missing any numbers.

4. Thread Worker Function:

Each thread runs the following function, marking the multiples of known primes in its assigned block:

```

16 void* sieveWorker(void* arg) {
17     ThreadData* data = (ThreadData*)arg;
18     int n = data->n;
19     int id = data->id;
20     int num_threads = data->num_threads;
21     std::vector<bool>& is_prime = *(data->is_prime);
22     int sqrt_n = static_cast<int>(sqrt(n));
23
24     int block_size = (n - 1) / num_threads;
25     int start = 2 + id * block_size;
26     int end = (id == num_threads - 1) ? n : start + block_size - 1;
27
28     for (int p = 2; p <= sqrt_n; ++p) {
29         if (is_prime[p]) {
30             int first = (start / p) * p;
31             if (first < start) first += p;
32             if (first < p * p) first = p * p;
33
34             for (int i = first; i <= end; i += p) {
35                 is_prime[i] = false;
36             }
37         }
38     }
39     pthread_exit(NULL);
40 }

```

5. Thread management:

Threads are created and synchronized using `pthread_create` and `pthread_join`:

```
pthread_t threads[num_threads];
ThreadData data[num_threads];

for (int i = 0; i < num_threads; ++i) {
    data[i] = {i, num_threads, n, &is_prime};
    pthread_create(&threads[i], NULL, sieveWorker, (void*)&data[i]);
}

for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
}
```

Each thread independently and exits using `pthread_exit`.

All threads must complete before proceeding to the final count.

6. Final Prime Count and Timing:

After threads complete, we perform a final pass to count how many numbers remained marked as prime:

```
int count = 0;
for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) count++;
}
```

The full function is timed using `<chrono>` for later performance analysis:

```
auto start = std::chrono::high_resolution_clock::now();
int count = parallelSieve(n, threads);
auto end = std::chrono::high_resolution_clock::now();
```

7. Sample Output:

A typical output for $n = 1000000$ using 4 threads:

```
[Running] cd "/home/kali/Desktop/project1/" && g++ project.cpp -o
project && "/home/kali/Desktop/project1/"project
Running parallel sieve for n = 1000000 using 4 threads.
Found 78498 prime numbers.
Execution time: 128.836 ms
```



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:

Execution Times:

The average execution times (in milliseconds) for each configuration over 5 runs are summarized below.

N	Implementation	Threads	Avg Time (ms)
100,000	Sequential	0	0.447
100,000	Parallel	1	3.725
100,000	Parallel	2	2.744
100,000	Parallel	4	3.536
100,000	Parallel	8	4.295
1,000,000	Sequential	0	5.457
1,000,000	Parallel	1	10.663
1,000,000	Parallel	2	9.511
1,000,000	Parallel	4	11.630
1,000,000	Parallel	8	9.932
10,000,000	Sequential	0	61.742
10,000,000	Parallel	1	110.029
10,000,000	Parallel	2	92.177
10,000,000	Parallel	4	78.075

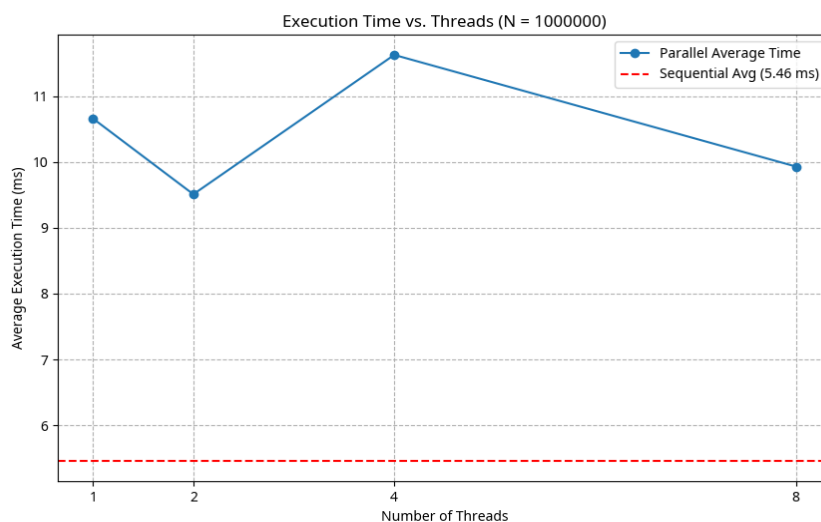
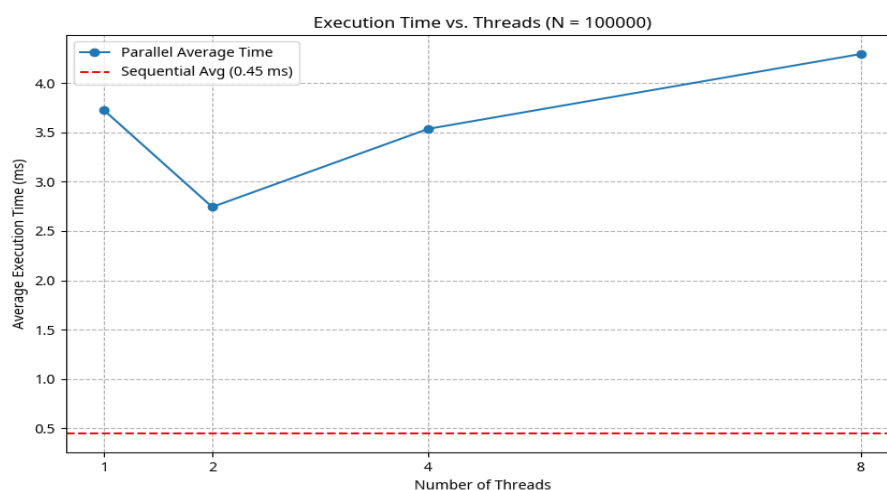
Speedup Calculation:

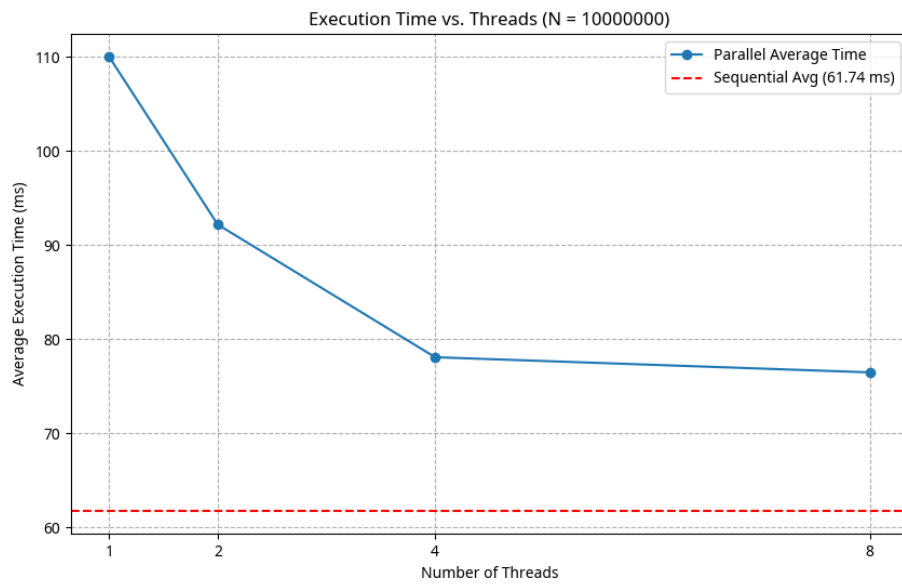
Speedup is calculated as $\text{Speedup} = (\text{Average Sequential Time}) / (\text{Average Parallel Time})$. The calculated speedup values for each parallel configuration are:

N	Threads	Sequential Time (ms)	Parallel Time (ms)	Speedup
100,000	1	0.447	3.725	0.120
100,000	2	0.447	2.744	0.163
100,000	4	0.447	3.536	0.126
100,000	8	0.447	4.295	0.104
1,000,000	1	5.457	10.663	0.512
1,000,000	2	5.457	9.511	0.574
1,000,000	4	5.457	11.630	0.469
1,000,000	8	5.457	9.932	0.549
10,000,000	1	61.742	110.029	0.561
10,000,000	2	61.742	92.177	0.670
10,000,000	4	61.742	78.075	0.791
10,000,000	8	61.742	76.464	0.807

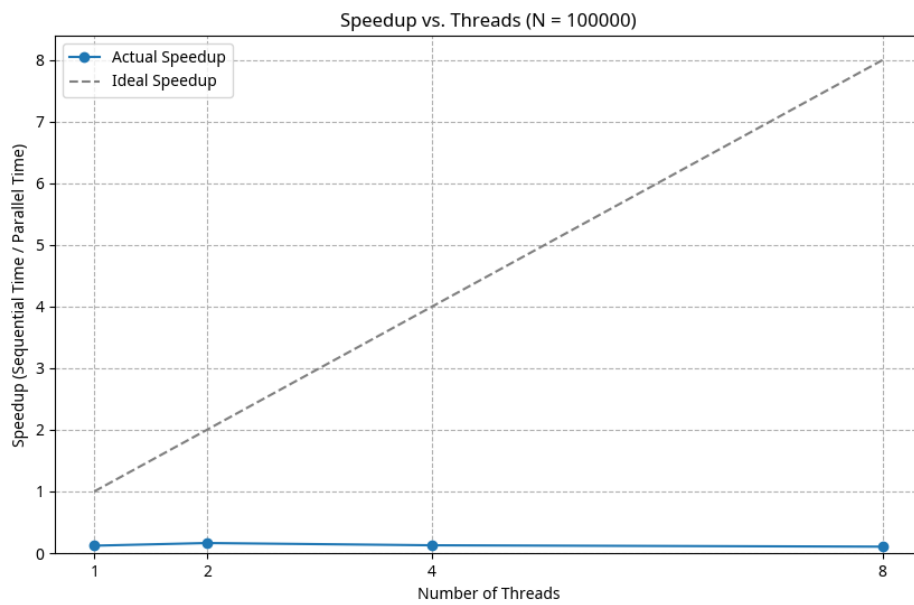
Graphs:

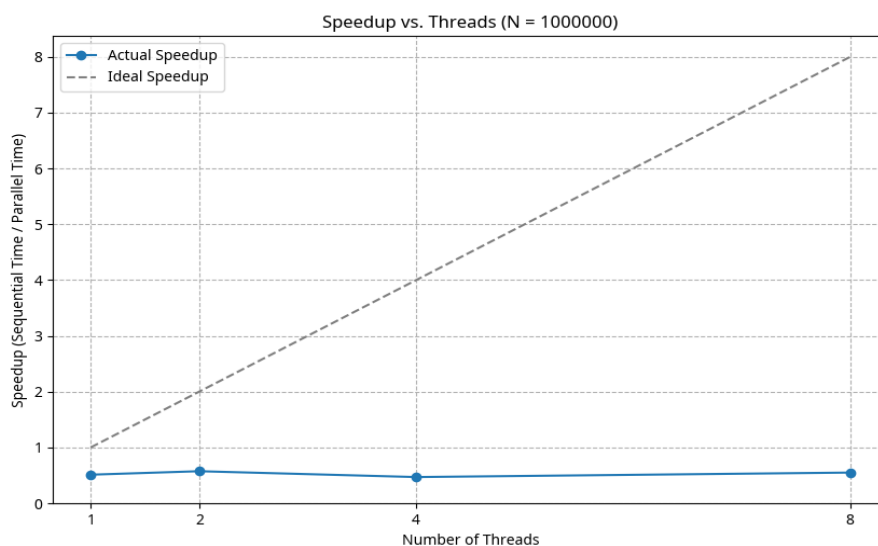
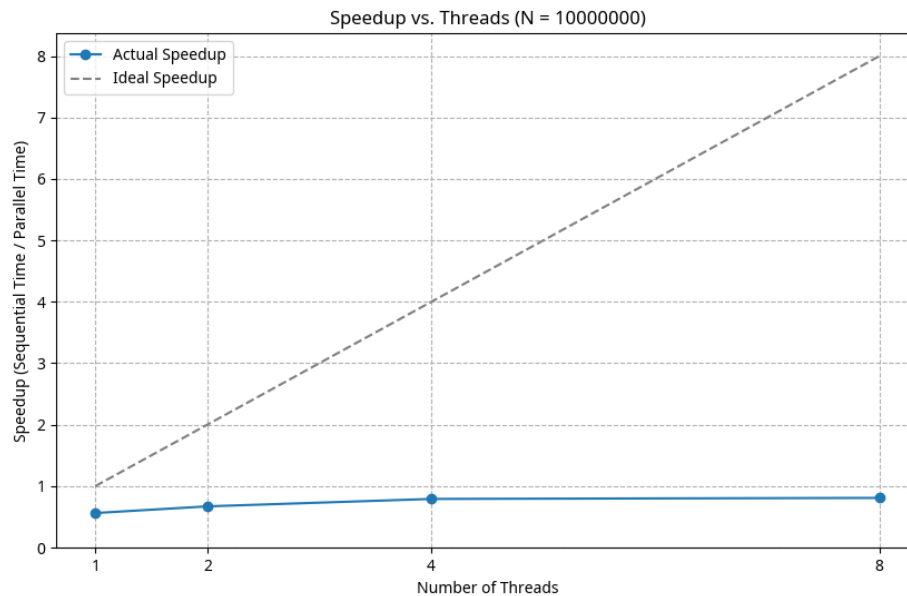
Execution time vs. thread count:





Speedup vs. thread count:



**Tools Used:**

Compiler:

Language:

Parallelization Library:

Timing:

Text Editor/IDE:

OS:

Ms office:

Virtual machine:

g++ (11.4.0)

C++

Pthreads

C++ <chrono>

VS CODE

Kali (Linux Kernel 6.1), Windows 11 pro

Excel, Word

VMware workstation pro

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 speed up 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.