THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

Parallel and Distributed Computing PROJECT I

ID: 202110725

Name: Ahmad Sabbah

Section: 1

Total	/100

Good Luck!

Mr.HusseinYounis

Introduction

In this project, I selected a problem related to finding the two closest prime numbers in a given range. This problem is computationally intensive when repeated for many queries, making it suitable for parallelization. The core of the algorithm relies on generating primes and checking them within a range, which can be independently divided among multiple threads.

Sequential Implementation

The sequential version was written in C++ using basic loops and a prime-checking function. I used the `<chrono>` library to measure the execution time for multiple input sizes. The input is read from a file, and for each range, the two closest primes are calculated and printed.

```
auto start = chrono::high_resolution_clock::now();
// function calls here
auto end = chrono::high_resolution_clock::now();
chrono::duration<double, milli> diff = end - start;
cout << "Time: " << diff.count() << " ms" << endl;
```

Parallelization Strategy

To parallelize the solution, I divided the queries across multiple threads. Each thread receives a chunk of the queries and processes them independently. I used `pthread_create`, `pthread_join`, and structs to pass arguments to each thread.

The results are collected in separate arrays for each thread to avoid race conditions. Finally, all partial results are merged and printed.

Experiments

Hardware:

- CPU: Intel Core i7
- Cores: 8
- OS: Ubuntu 22.04 LTS

Input Sizes and Threads:

- Queries tested: 1,000 – 100,000
- Thread counts: 1 (sequential), 4, 7, and 8 (max)
- Each test case was run 5 times and averaged.

Results

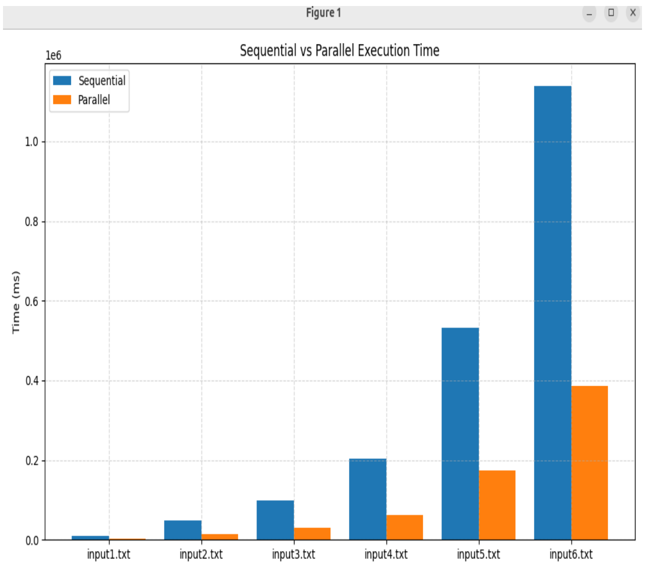
Execution Time :

Num of Thread	Queries	Time (ms)	Speedup
Seq	1000	9802.49	
Seq	5000	49298.13	
Seq	10000	99608.23	
Seq	20000	203285.31	
Seq	50000	531640.16	
Seq	100000	1138508.31	
4	1000	2634.53	3.72
4	5000	13815.87	3.57
4	10000	29521.97	3.37
4	20000	63118.08	3.22
4	50000	174279.09	3.05
4	100000	194279.09	2.96
Seq	1000	12436.25	
Seq	5000	65784.58	
Seq	10000	165905.16	
Seq	20000	294853.64	
Seq	50000	548032.38	
Seq	100000	548032.38	
7	1000	2448.34	5.08
7	5000	15373.63	4.82
7	10000	28669.25	5.79
7	20000	57106.72	5.16
7	50000	147290.16	3.72
7	100000	331328.49	3.43
Seq	1000	10011.48	

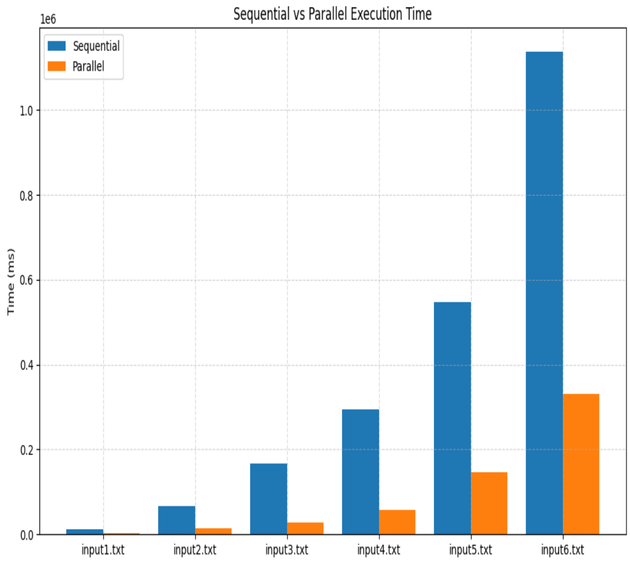
Seq	5000	49650.7	
Seq	10000	100565.97	
Seq	20000	204157.66	
Seq	50000	535321.69	
Seq	100000	1138861.72	
all	1000	11298.11	4.93
all	5000	11298.11	4.93
all	10000	23210.88	4.33
all	20000	52280.83	3.91
all	50000	140683.64	3.81
all	100000	295383.04	3.86

Graphs :

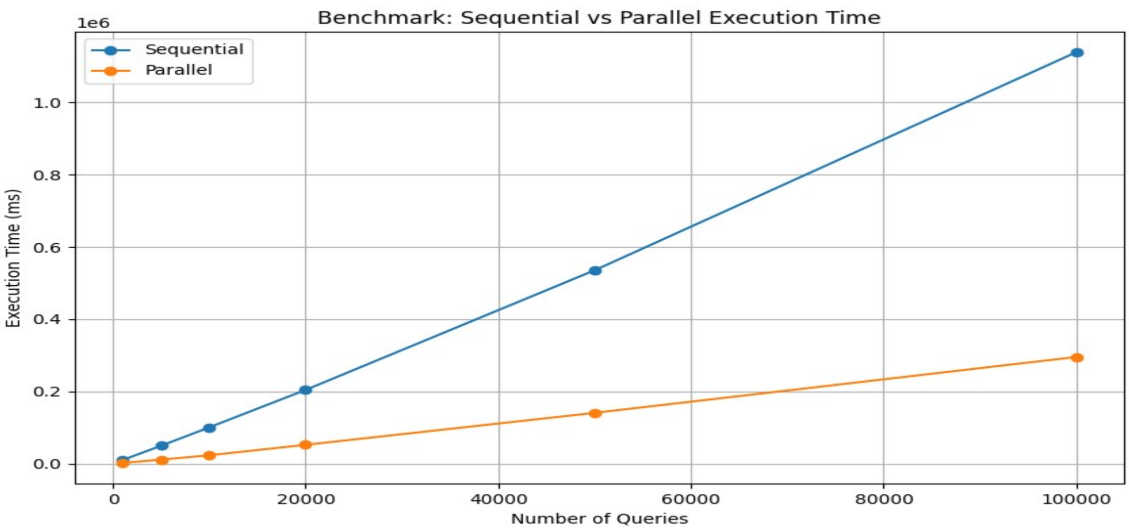
4 treads vs sequential



7 threads vs sequential



max threads vs sequential



Discussion

The speedup achieved was significant, especially with larger input sizes. However, perfect linear speedup was not achieved due to:

- Thread creation and joining overhead.
- Shared memory synchronization (though minimal).
- Amdahl's Law: some parts of the code remain sequential.

In general, 4–7 threads provided the best balance between performance and CPU utilization.

Conclusion

Through this project, I learned:

- How to analyze and split work among threads.
- The importance of minimizing shared data.
- How to test and measure performance using real workloads.

The parallel version provided up to 5x speedup and demonstrated the power of multithreading for CPU-bound problems.