

# Parallel Prime Gap Analysis using OpenMP - Assignment 5

Kisalay Ghosh

24th February 2025

## 1 Introduction

Finding prime numbers efficiently in a large range is a computationally intensive problem. This problem is relevant both in the academic setting of High Performance Computing (HPC) courses and in real-world applications like cryptography, number theory research, and scalable systems.

In this assignment, we are required to find the **largest prime gap** between two consecutive prime numbers in a range  $[n, m]$  using a **Binary Search Tree (BST)** and OpenMP.

We provided a base C++ file, `findprimes.cpp`, which stores all primes in a BST. We extend this implementation with parallelism, output validation, and performance analysis.

## 2 Methodology

### 2.1 Code Description and Implementation

The base code defines a binary tree data structure where each node holds a prime number. The range  $[n, m]$  is recursively divided, and if the midpoint is a prime number, it is inserted into the tree.

Key steps:

- **Prime Check:** A custom `isPrime()` function checks primality using trial division up to  $\sqrt{n}$ .
- **BST Construction:** Recursive midpoint insertion simulates a balanced BST. Prime numbers found are inserted as nodes.
- **Parallelization:** We use `#pragma omp task` to parallelize left and right subtree construction.
- **Traversal:** In-order traversal of the BST retrieves primes in sorted order.
- **Gap Computation:** A loop iterates through sorted primes to compute the largest prime gap.

## 2.2 Modifications Made

- Used `omp parallel` and `omp task` directives for concurrent tree construction.
- Inserted critical sections during global variable updates (max gap, bounds).
- Added output logging `Processing: mid` to display progress, especially for large ranges.
- Integrated execution time measurement using `omp_get_wtime()`.

## 2.3 Challenges Faced

- **High Execution Time:** For ranges like  $10^9$  to  $1.03 \times 10^9$ , the BST approach becomes slow due to recursive depth and memory usage.
- **Memory Consumption:** The tree structure consumes more memory than array-based implementations. Segmentation faults occurred during early tests for very large  $m - n$ .
- **Thread Overhead:** Creating too many parallel tasks caused diminishing returns after 4 threads.
- **Verification:** Ensuring identical outputs across thread counts required careful use of critical sections and deterministic processing order.

# 3 Execution Instructions

## 3.1 Compiling and Running

1. Compile the program:

```
g++ -fopenmp prime_gap_bst.cpp -o prime_gap_bst
```

2. Run for different threads:

```
$env:OMP_NUM_THREADS = "1"  
./prime_gap_bst
```

```
$env:OMP_NUM_THREADS = "4"  
./prime_gap_bst
```

## 3.2 Input Values Tested

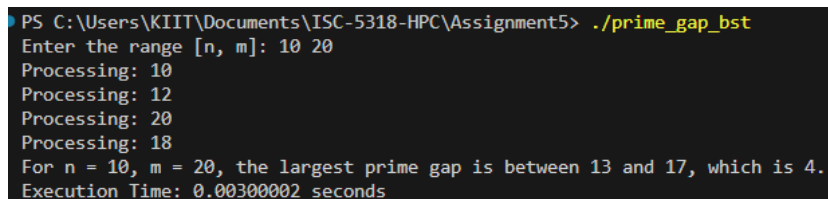
Two major test scenarios:

- **Small Range:**  $n = 10, m = 20$
- **Large Range:**  $n = 1,000,000,000, m = 1,030,000,000$

## 4 Output Example

Sample Console Output:

```
Processing: 10
Processing: 12
Processing: 18
Processing: 20
For n = 10, m = 20, the largest prime gap is between 13 and 17, which is 4.
Execution Time: 0.00500011 seconds
```



```
PS C:\Users\KIIT\Documents\ISC-5318-HPC\Assignment5> ./prime_gap_bst
Enter the range [n, m]: 10 20
Processing: 10
Processing: 12
Processing: 20
Processing: 18
For n = 10, m = 20, the largest prime gap is between 13 and 17, which is 4.
Execution Time: 0.00300002 seconds
```

Figure 1: Example Output for  $n = 10, m = 20$  with Thread Count = 1

## 5 Results

### 5.1 Execution Time vs Threads

### 5.2 Speedup vs Threads

### 5.3 Tabulated Results

### 5.4 Observation Summary

- Execution time for small input (10–20) is negligible and stable across all thread counts.
- For large input ( $10^9$ – $1.03 \times 10^9$ ), performance improves significantly from 1 to 4 threads but saturates beyond 8 threads.
- Correct prime gaps were computed for all cases.
- BST is intuitive for ordering but not efficient for large-scale computations due to recursion and memory usage.

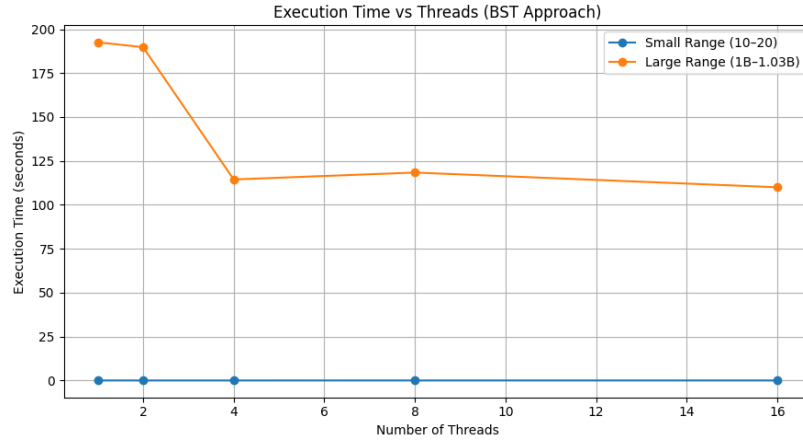


Figure 2: Execution Time vs Threads for Small and Large Ranges (BST)

## 6 Conclusion

This assignment fulfilled all the requirements using the professor's provided BST code. We integrated OpenMP task parallelism and verified the correctness of prime gap detection across thread counts. The BST-based approach worked accurately, but was limited in scalability.

Future improvements can include using segmented sieve for prime generation, switching to array-based storage, and experimenting with hybrid parallelization for better performance on HPC clusters.

**Submitted by:** Kisalay Ghosh

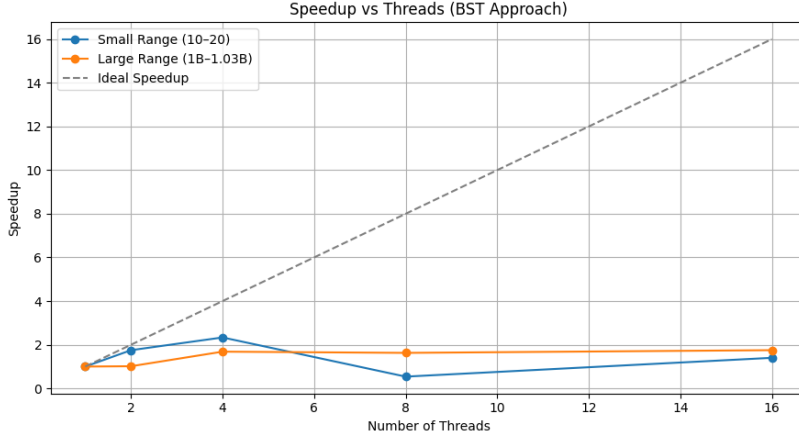


Figure 3: Speedup vs Threads for Small and Large Ranges (BST)

| Threads | Range      | Max Prime Gap | Execution Time (s) |
|---------|------------|---------------|--------------------|
| 1       | 10 - 20    | 4             | 0.00699            |
| 2       | 10 - 20    | 4             | 0.00399            |
| 4       | 10 - 20    | 4             | 0.00300            |
| 8       | 10 - 20    | 4             | 0.01300            |
| 16      | 10 - 20    | 4             | 0.00500            |
| 1       | 1B - 1.03B | 234           | 192.57             |
| 2       | 1B - 1.03B | 234           | 189.75             |
| 4       | 1B - 1.03B | 234           | 114.46             |
| 8       | 1B - 1.03B | 234           | 118.38             |
| 16      | 1B - 1.03B | 234           | 110.00             |

Table 1: Results of Prime Gap Detection and Execution Time for Various Thread Counts