# Operating Systems–1: Autumn 2023

# Programming Assignment 2 - REPORT

**NAME: GHOLAP SIDDHESH ASHOK**

**ROLL NO: AI22BTECH11007**

## 1. Overview:

The provided C program is designed to find vampire numbers within a given range (N) using multiple threads (M). Vampire numbers are numbers that can be factored into two numbers with half the number of digits, with the same digits as the original number.

## 2. Low-Level Design:

### ❖ Thread Data Structure:

- The program defines a struct thread_data to encapsulate thread-specific data. It includes the thread ID (thread_id), the number of elements assigned to the thread (num_count), and an array (numbers) to store the assigned numbers.

### ❖ Thread Safety:

- The program uses a mutex (count_mutex) to ensure thread safety when updating the global variable vampire_count. This prevents race conditions that may occur when multiple threads try to increment the count simultaneously.

### ❖ Vampire Number Checking:

- The function isVampireNumber checks whether a given number is a vampire number.
- It employs a permutation-based approach to find all possible combinations of factors for a given number.
- The logic iterates through permutations of digits to find valid factor pairs and increments the vampire_count accordingly.

### ❖ File Handling:

- The program reads input values (N and M) from an input file (InFile.txt) and writes results to an output file (OutFile.txt).
- The input file is read to determine the range of numbers to be processed.
- Output, including the numbers found by each thread, is appended to the output file.

### ❖ Thread Creation and Joining:

- The main function creates an array of thread_data structures and initializes them based on the number of threads (M).
- Threads are created using pthread_create, and the main thread waits for their completion using pthread_join.

### ❖ Partitioning Logic:

- Numbers are partitioned among threads in a **round-robin fashion**.
- The program calculates the number of elements each thread will process (num_per_thread), and any remaining elements are distributed to the first few threads.
- The partitioning logic ensures an even distribution of the workload among the threads.

### ❖ Output Reporting:

- The program generates a summary in the output file, including the values of N and M.
- Each thread reports the vampire numbers it finds along with its thread ID.
- The total count of vampire numbers across all threads is reported at the end.

## 3. Complications and Considerations:

### ❖ Memory Management:

- The program dynamically allocates memory for thread-specific data (thread_info), and care is taken to free this memory at the end of the program to prevent memory leaks.

### ❖ Non-Trivial Values Of N or M:

- If M>N, i.e. if number of threads are greater than numbers, then only N number of threads are used.
- If any of N or M is zero, then program terminates.

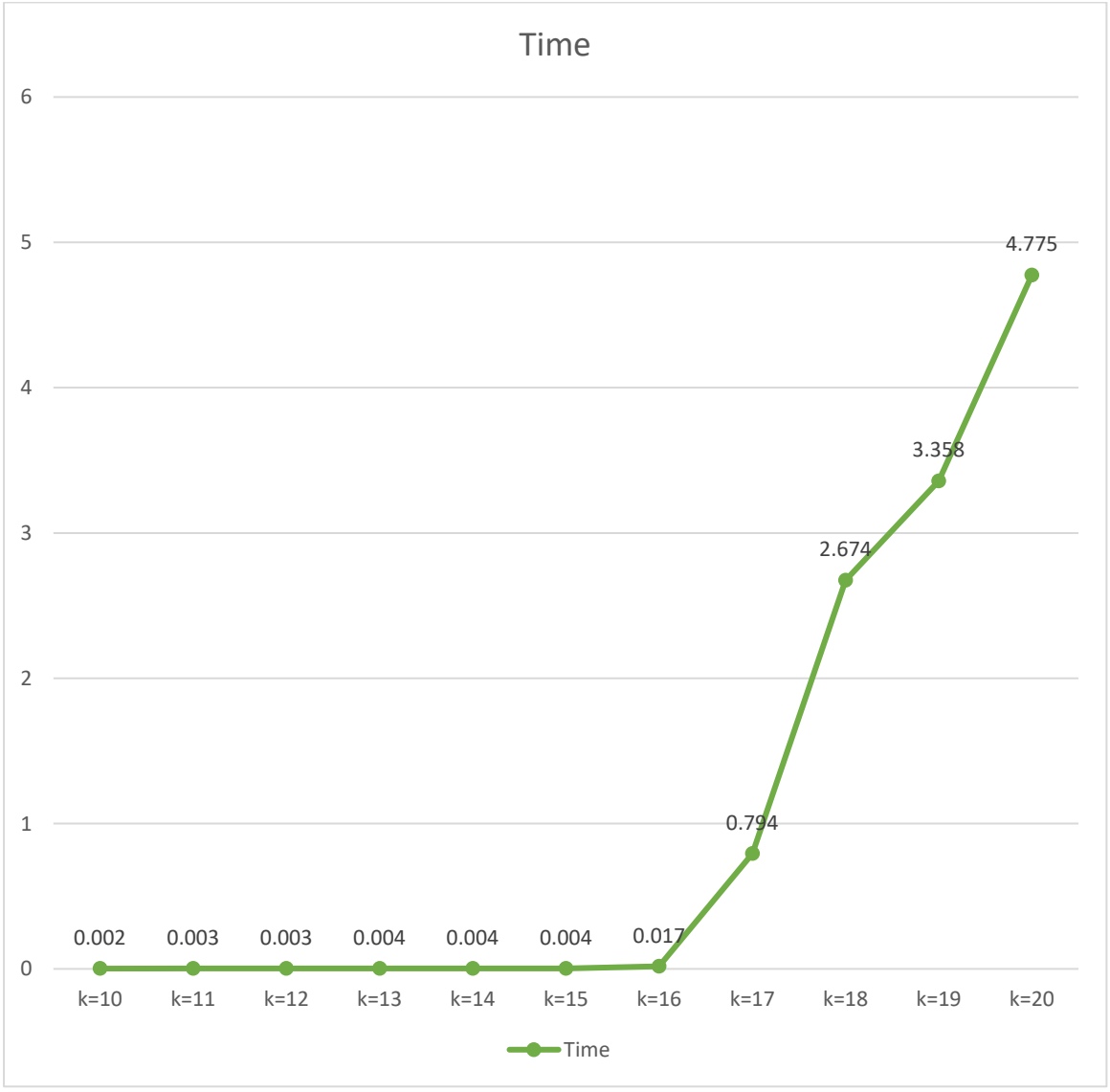
❖ File Handling:

➢ Proper file handling is implemented to check for errors in file opening and closure.

❖ Thread Safety:

➢ The use of a mutex ensures the correct incrementing of the global vampire_count variable, avoiding data corruption.

## 4. Graph-1 (time vs size):



### Graph 1: Time vs Size (N) with Fixed Threads (M=8)

In this graph, the time taken by the algorithm is plotted against the size of the problem, where the size is determined by the value of N (number of numbers to be processed). The number of threads (M) is fixed at 8 for all experiments.

# Analysis:

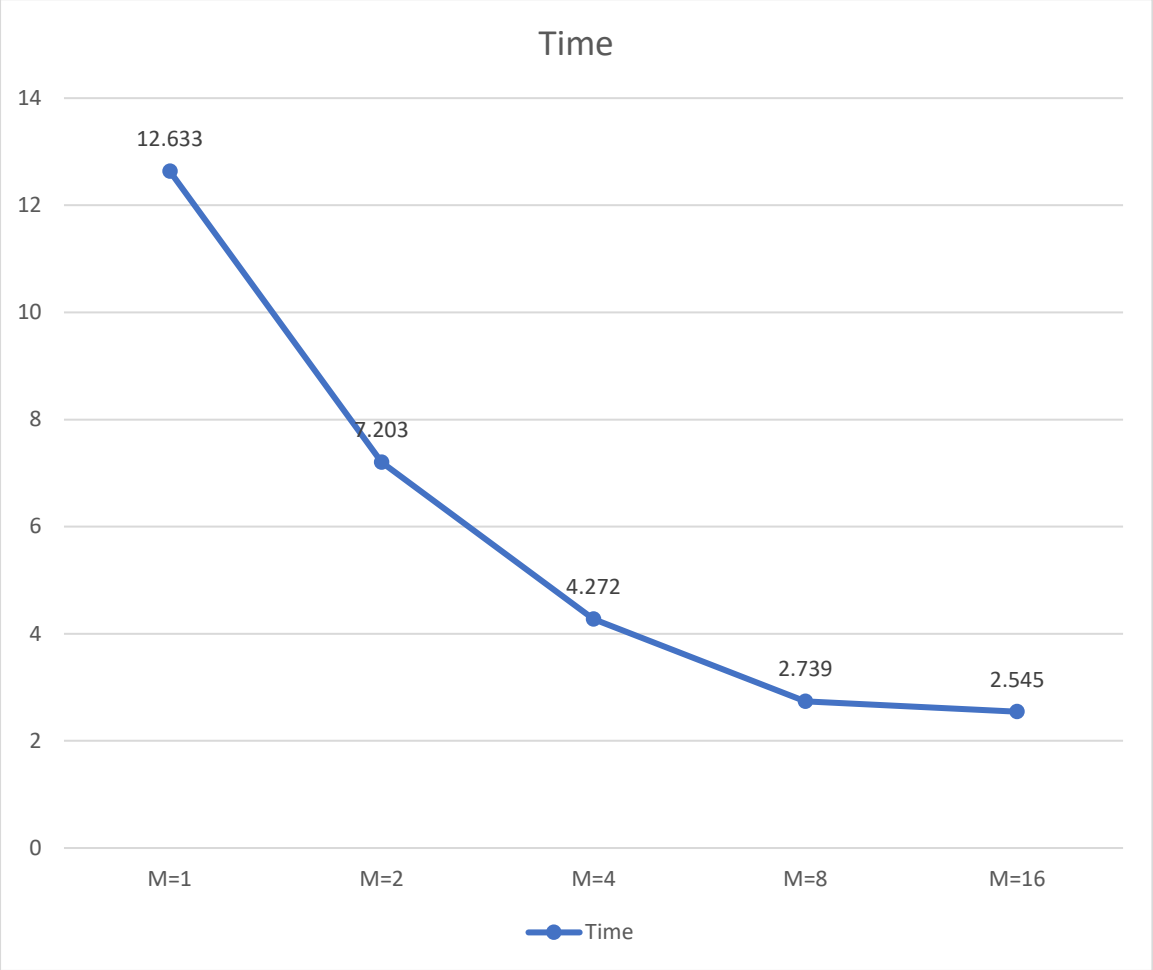
**Exponential Growth:** As observed in the time values, the time increases significantly as the problem size (N) grows. This is consistent with the expected exponential growth in computation time for larger values of N.

**Parallel Efficiency:** With a fixed number of threads (M=8), the graph demonstrates diminishing returns in parallel efficiency. Notably, the time jumps from 0.004s to 0.017s when moving from N=2^14 to N=2^16. This suggests that beyond a certain point, the overhead of managing multiple threads starts to outweigh the benefits of parallelization.

**Optimal Range:** The optimal range for parallel execution seems to be between N=2^10 and N=2^14, where the parallel execution significantly reduces the overall processing time.

**Inflection Point:** The graph suggests an inflection point around N=2^14, beyond which the time taken increases more rapidly. This indicates the threshold at which the increase in problem size has a more pronounced impact on computation time.

## 5. Graph-2 (time vs no. of threads):




**Graph 2: Time vs Number of Threads (M) with Fixed Size (N=1000000)**

In this graph, the time taken by the algorithm is plotted against the number of threads (M), with the size of the problem fixed at N=1000000 (10 lakhs).

# Analysis:

**Parallel Scalability:** The time values indicate improved parallel scalability as the number of threads increases, up to a certain point. Notably, the time decreases from 12.633s to 2.545s as the number of threads increases from 1 to 16.

**Diminishing Returns:** The diminishing returns become evident as the number of threads increases. While going from 1 to 2 threads results in a substantial reduction in time, further increases in the number of threads yield diminishing improvements.

**Optimal Thread Count:** The optimal number of threads for this problem seems to be around 8, as it yields a relatively low time of 2.739s. Beyond this point, the gains from additional threads diminish.

**Saturation Point:** The time value for M=16 (2.545s) indicates that the performance improvement starts to saturate, and the overhead of managing additional threads becomes a limiting factor.