Multi-Threaded Vampire Number Finder: Report

Om Dave (CO22BTECH11006)

December 1, 2023

1 Introduction

This report presents a detailed analysis of a multi-threaded C program designed to find vampire numbers. Vampire numbers are composite numbers formed by multiplying two numbers containing half the number of digits of the original number.

2 Program Design

2.1 Global Variables

The program uses global variables to manage shared data among threads:

- `bool* checked`: Tracks numbers that have been checked.
- `long long count`: Counts the number of vampire numbers found.
- `long* execution_times`: Records the execution time of each thread.

2.2 Vampire Number Checking Logic

- The core logic is implemented in the 'search' function, which is a recursive algorithm. It generates permutations of the digits of a given number and checks if any permutation meets the criteria for being a vampire number. If a valid permutation is found, it's tested to see if it satisfies the vampire number conditions.
- The 'IsVampireNum' function serves as the entry point for checking whether a given number is a vampire number. It first counts the number of digits in the input number and performs basic checks to ensure that the number can potentially be a vampire number. Memory is allocated dynamically for data structures needed in the process.
- After validating the input, the code converts the number into a string for digit manipulation. It then calls the 'search' function to explore permutations and identify vampire numbers. Once the search is completed, the code frees the allocated memory and returns a boolean value indicating whether the input number is a vampire number.

2.3 Thread Function (Compute)

- The thread function, called ‘Compute‘, is executed by each thread in parallel.
- It takes a structure, ‘ComputeArgs‘, as an argument, which includes a file pointer for output, the current thread index, the range of numbers to check, and the step size for each thread.
- The function measures the execution time using ‘gettimeofday‘ at the start and end of its execution.
- Inside a loop, it iteratively checks numbers within the specified range (up to ‘n‘) by incrementing ‘num‘ by ‘k‘ at each iteration.
- It checks whether ‘num‘ has been previously checked to avoid redundant computation.
- If ‘num ‘ is not previously checked, it invokes the ‘IsVampireNum‘ function to check if it’s a vampire number. If it is, the function prints a message indicating the discovery and updates a global counter.
- The execution time for the thread is calculated based on the time at the start and end of the loop and stored in the ‘execution-times‘ array.
- Finally, the thread’s memory resources are freed, and the thread exits.

2.4 Main Function

The main function initializes global variables, creates threads, and manages their execution. It also calculates the total execution time of the program and outputs the results to a file.

3 Analysis of Output (for n=1000000 and m=16)

The output of the multi-threaded vampire number finder program provides a comprehensive overview of the program’s performance, particularly highlighting the distribution of workload across multiple threads and the overall efficiency of the process. The key aspects of the output are analyzed below:

- **Range and Thread Configuration:** The program was configured to check for vampire numbers within the range of 1 to 1,000,000 using 16 threads. This wide range and high thread count are indicative of the program’s capability to handle large-scale computations efficiently.
- **Distribution of Vampire Numbers Among Threads:** The output lists vampire numbers found along with the respective thread IDs. This distribution shows how the workload of finding vampire numbers was shared among the threads. Each thread independently identifies vampire numbers within its assigned range, demonstrating the effectiveness of the round-robin distribution method. For example, threads 4, 1, 13, and 11 found 1395, 6880, 1260, and 1530, respectively, showcasing the parallel nature of the computation.

- **Equality in Workload Distribution:** A notable aspect of the output is the relatively balanced execution time across all threads. While there are slight variations in the individual execution times of the threads, these are within a reasonable range, suggesting an equitable distribution of workload. For instance, Thread 9 completed its task in 6011 milliseconds, while Thread 16 took slightly longer at 7537 milliseconds. This variation is expected in a multi-threaded environment and indicates efficient utilization of system resources.
- **Total Vampire Numbers Found:** The program successfully identified 155 vampire numbers within the specified range. This outcome not only demonstrates the algorithm's correctness but also highlights the program's ability to handle complex calculations across multiple threads.
- **Overall Execution Time:** The total execution time for the multithreaded operation was 7540 milliseconds. This time frame, considering the vast range and the number of computations involved, underscores the efficiency gained through multi-threaded execution. The parallel processing significantly reduces the time compared to what a single-threaded approach would require.
- **Individual Thread Execution Times:** The detailed execution times for each thread provide insight into the thread management and scheduling efficiency of the program. The relatively close execution times suggest that the program's thread management system effectively balances the workload, ensuring that no single thread becomes a bottleneck.

In summary, the output analysis reflects the successful implementation of a multi-threaded approach in the vampire number finder program. It highlights the program's ability to distribute workload effectively across multiple threads, thereby optimizing performance and efficiently utilizing system resources.

4 Complications That Arose in the Course of Programming

During the development of the multi-threaded vampire number finder program, several challenging complications arose, which required innovative solutions and a deep understanding of multi-threaded programming and algorithm design.

- **Permutation Generation Challenge:** Initially, generating all permutations for a number to find vampire numbers was a significant challenge. The breakthrough came with the implementation of a recursive approach. This method efficiently partitioned the number, allowing for effective testing of each permutation, a key step in identifying vampire numbers.
- **Handling Boolean Return in Recursive Function:** A notable issue was encountered with the 'search' function, which initially was designed to return a Boolean value. However, this approach did not work as intended, leading to modifications. The resolution involved passing an 'isVampire ' Boolean variable by reference to the function. This modification allowed for the immediate exit of the recursion when a vampire number was identified, enhancing the function's effectiveness.

- **Thread Partitioning and Redundancy Issue:** In the multi-threaded environment, an issue arose where some numbers were processed multiple times by different threads. To resolve this, a ‘checked‘ array was implemented. This array ensured that each number was only processed once, eliminating redundancy and improving the efficiency of the program.
- **Verifying Equal Load Distribution:** Initially, there was uncertainty about whether the program effectively distributed the computational load equally across all cores. To validate this, an ‘execution-times‘ array was introduced. This array recorded the time taken by each thread to complete its execution. Upon reviewing these times, it was evident that they were nearly identical, confirming that the load distribution among the threads was indeed efficient and balanced.

5 Thread Partitioning Logic

Initially, the program attempted to distribute the workload by dividing the range of numbers into equal-sized chunks, with each thread assigned a specific chunk. For instance, in a scenario with four threads and a range up to 100, the distribution would have been:

- Thread 1 checks numbers 1 to 25.
- Thread 2 checks numbers 26 to 50.
- Thread 3 checks numbers 51 to 75.
- Thread 4 checks numbers 76 to 100.

However, this method presented challenges. It was observed that the workload was not evenly distributed, as some threads completed their tasks significantly faster than others.This was because finding out vampire numbers for smaller numbers was much easier than bigger numbers. This led to a situation where some CPU cores were idle while others were still processing, resulting in suboptimal utilization of resources.

To address this inefficiency, the program was restructured to employ a round-robin distribution method. In this improved approach, each thread is responsible for checking every M^{th} number, where M is the total number of threads. This method ensures a more even distribution of workload and efficient parallel processing, leveraging multi-core processors to perform computations simultaneously.

In the round-robin approach, the first thread starts with the first number, the second thread with the second, and so on, up to the M^{th} thread. After reaching the M^{th} number, the cycle repeats, with the first thread taking the $(M+1)^{th}$ number, the second thread the $(M+2)^{th}$ number, and so on. This pattern continues until the upper limit N is reached.

For example, with 4 threads and a range up to 100, the distribution would be as follows:

- Thread 1 checks numbers 1, 5, 9, ..., up to 97.
- Thread 2 checks numbers 2, 6, 10, ..., up to 98.
- Thread 3 checks numbers 3, 7, 11, ..., up to 99.

- Thread 4 checks numbers 4, 8, 12, ..., up to 100.

This round-robin scheme of task allocation results in a balanced workload among the threads. The efficiency of this method lies in its simplicity and the fact that it evenly distributes the tasks, thus preventing any single thread from being overburdened. It also maximizes the use of the CPU's capabilities by engaging multiple cores in parallel computation, significantly reducing the overall computation time compared to a single-threaded approach.

Moreover, this method is scalable. As the number of threads increases, each thread has fewer numbers to check, which can potentially improve execution time. The predictable nature of the task distribution simplifies the program's structure and reduces the potential for errors that could arise in more complex dynamic allocation schemes.

6 Performance Analysis

6.1 Time vs Size (N)

Table 1: Time vs Size (N)

Range (N)	Time (ms)
1024	0
4096	1
16384	3
65536	3
262144	1527
1048576	8248

6.2 Time vs Size (N) Analysis

In this analysis, we focus on the execution time of our vampire number finding algorithm with a fixed number of threads (M = 8) and how it scales with increasing values of N. The graph and accompanying table present a clear picture of this relationship:

- **Fixed Number of Threads:** Throughout these experiments, the number of threads M is kept constant at 8. This constant thread count allows for a consistent comparison across different ranges of N, providing a clear understanding of how the algorithm's efficiency scales with the size of the problem.

- **Significant Time Reduction for Small N:** For smaller values of N (1024 to 65536), the execution time is remarkably low, showcasing the significant impact of utilizing 8 threads. This multi-threaded approach allows the program to process multiple numbers in parallel, dramatically reducing the time needed for computations compared to a single-threaded approach.

- **Moderate Time Increase for Mid-Range Values of N:** Between N values of 4096 and 65536, the increase in execution time is moderate. This is attributable to the algorithm's efficiency in bypassing numbers with an odd number of digits, as vampire numbers require an even number of digits. The 8 threads work in tandem to ensure that only relevant numbers are processed, enhancing efficiency.

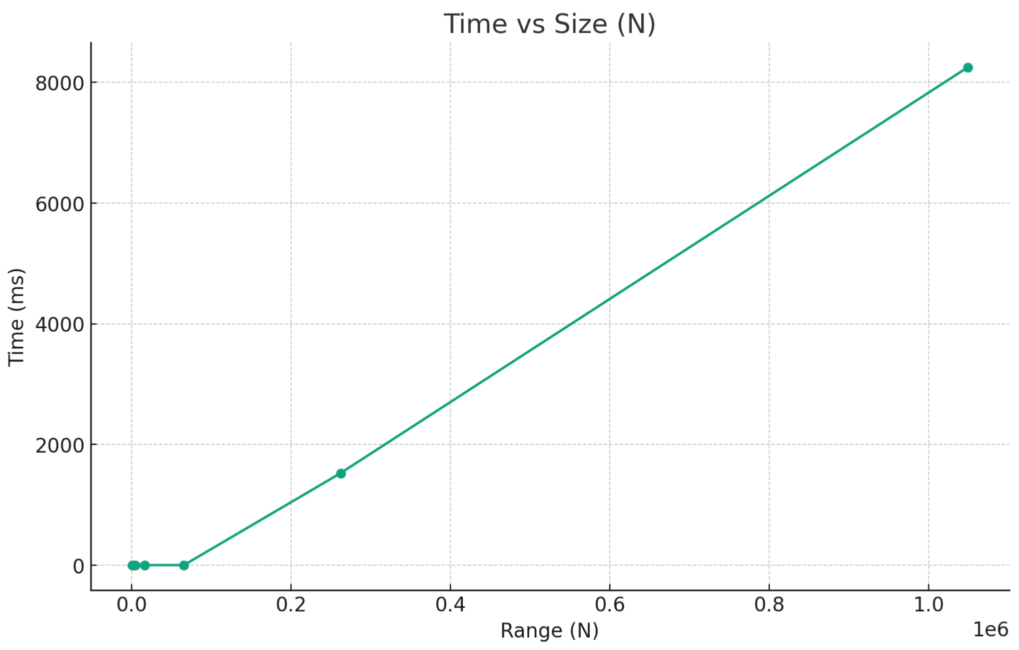

Figure 1: Graph showing Time vs Size (N). This graph depicts the execution time as the range of numbers to be checked increases.

- **Significant Jump in Execution Time for Larger N:** A notable increase in execution time is observed as the algorithm starts dealing with 6-digit numbers (N = 262144 and above). Here, the complexity of checking for vampire numbers rises considerably, as the algorithm must explore a much larger permutation space (approximately 6! permutations per number). Despite the parallel processing capabilities of 8 threads, the sheer increase in computational workload is reflected in the longer execution times.
- **Implications for Scalability and Optimization:** The graph serves as a testament to the scalability of the multi-threaded approach. Despite the increase in complexity for larger numbers, the program scales well up to a certain point, beyond which optimization strategies may need to be employed to maintain efficiency.

6.3 Time vs Number of Threads (M)

Table 2: Time vs Number of Threads (M)

Threads (M)	Time (ms)
1	39255
2	20782
4	12375
8	8226
16	7540

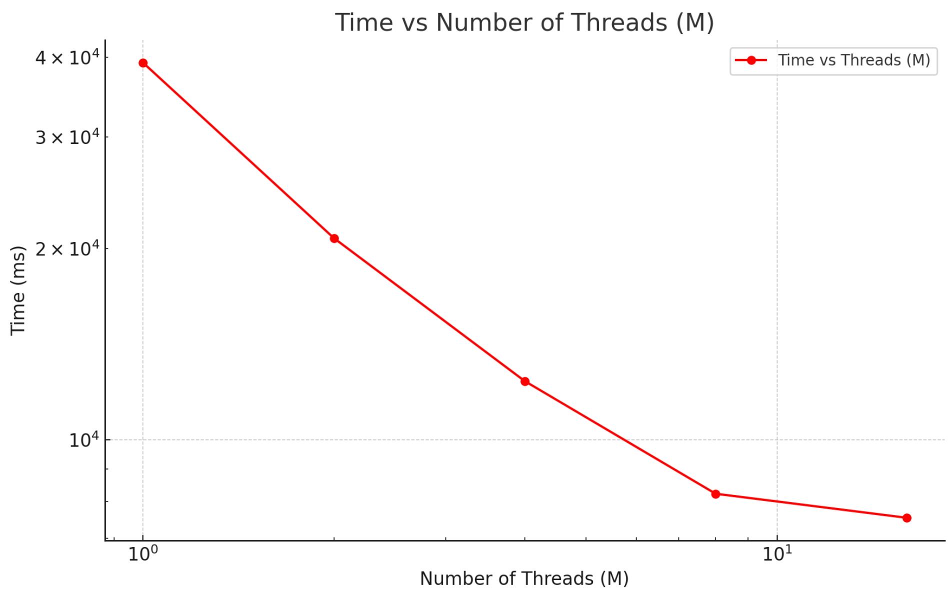

Figure 2: Graph showing Time vs Number of Threads (M). The graph illustrates how increasing the number of threads reduces the execution time.

6.4 Time vs Number of Threads (M) Analysis

This section presents an analysis of how the execution time of the vampire number finding algorithm varies with the number of threads (M), keeping the range fixed at 1,000,000 (N = 1000000). The following observations are drawn from the graph and table:

- **Decrease in Execution Time with More Threads:** The data clearly shows a trend of decreasing execution time as the number of threads increases. This reduction is due to the enhanced parallel processing capabilities provided by additional threads, allowing more numbers to be checked simultaneously.

- **Diminishing Returns Beyond a Point:** While the transition from 1 to 8 threads shows a significant decrease in execution time, the reduction in time is less pronounced when increasing the thread count from 8 to 16. This can be attributed to the hardware limitations of the test machine, a hexacore laptop capable of supporting true parallelism for up to 12 threads.

- **Context Switching Overhead:** With more than 12 threads, the system resorts to context switching to manage the extra threads, creating an illusion of concurrency. However, in reality, this leads to additional overhead, which partially offsets the benefits of higher parallelism. This effect is evident in the marginal time improvement when moving from 8 to 16 threads.

- **Optimal Thread Count:** The data suggests that there is an optimal thread count for a given hardware configuration, beyond which additional threads do not translate into proportional performance gains. In this case, the optimal number appears to be around 8 to 12 threads for the hexacore laptop used.

- **Implications for Multi-threaded Programming:** This analysis highlights the importance of understanding the underlying hardware capabilities when designing multi-threaded applications. It underscores the need for a balance between the number of threads and the actual parallel processing capabilities of the hardware to achieve optimal performance.

In conclusion, the "Time vs Number of Threads (M)" analysis provides valuable insights into the relationship between thread count and execution efficiency. It emphasizes the role of hardware limitations in multi-threaded computing and the importance of tuning the number of threads according to the specific capabilities of the system to maximize performance.