A Parallel Approach to Prime Number Generation: Comparative Analysis of Sieve-Based Algorithms

Viktor Kostadinoski*, Vladimir Zdraveski*

*Faculty of Computer Science and Engineering, Skopje, Macedonia
viktor.kostadinoski@students.finki.ukim.mk, vladimir.zdraveski@finki.ukim.mk

Abstract

Prime number generation is a fundamental problem in number theory and cryptography, with sieve-based algorithms such as the Sieve of Eratosthenes and the Sieve of Euler being widely used due to their efficiency. This paper presents a parallel approach to these algorithms, aiming to enhance performance through multi-threading. We outline the parallelization strategy, which involves dividing the computational workload among multiple threads while ensuring correctness through a precomputed set of prime markers.

The effectiveness of our approach is evaluated by comparing execution times and improvement factors between sequential and parallel implementations. Experimental results demonstrate that while parallelization introduces overhead for smaller inputs, it yields performance gains for larger prime limits. Additionally, we compare the Sieve of Eratosthenes and the Sieve of Euler in a parallelized setting, analyzing their relative efficiency. Our findings highlight the potential of parallel computing in optimizing prime number generation.

Index Terms—Prime Numbers, Sieve of Eratosthenes, Sieve of Euler, Parallel Computing, Multi-threading, Cryptography.

1 Introduction

Prime numbers play a crucial role in various fields, particularly in number theory and cryptography. Efficiently finding prime numbers has been a longstanding challenge in computational mathematics, leading to the development of multiple algorithms over the years. Among these, the Sieve of Eratosthenes and the Sieve of Euler are two of the most well-known methods for generating prime numbers efficiently.

With the increasing demand for high-performance computing, optimizing these algorithms using parallel and distributed processing has become a significant research focus. Multi-threading and parallel computing techniques allow these sieving methods to execute faster by utilizing multiple processing units concurrently, reducing the overall computation time.

This paper explores the implementation of prime number sieving using parallel computing techniques. We compare the performance of the traditional Sieve of Eratosthenes and the Sieve of Euler in both sequential and parallel environments. Furthermore, we analyze the impact of multi-threading on the efficiency of these algorithms and discuss their potential applications in large-scale numerical computations.

The remainder of this paper is organized as follows: Section 2 presents a comprehensive overview of existing research on sequential and parallelized prime number sieving algorithms. Section 3 discusses parallelization strategies and implementation details. Section 4 presents experimental results and performance analysis. Finally, Section 5 concludes the paper and highlights future research directions.

2 Related Work

Prime number generation has a rich history, with classical algorithms such as the Sieve of Eratosthenes widely regarded for their efficiency. Recent research, such as [1], has compared various sieve algorithms, revealing the Sieve of Eratosthenes and Pritchard's wheel sieve as particularly effective for generating primes in cryptographic applications. Similarly, [2] compares multiple sieve algorithms, including the Sieve of Eratosthenes and Sundaram, especially in the context of RSA encryption, which aids in identifying the most effective methods for generating large primes.

Recent evaluations of sieve algorithms highlight the need for innovation despite theoretical advancements, as noted in [3]. This systematic review emphasizes that while many algorithms remain in use, most are primarily applied for didactic purposes. This underscores the importance of practical developments for advancing the field.

Research has also focused on enhancing existing algorithms to improve efficiency. Specifically, [4] discusses optimizations in classical sieving techniques, focusing on an enhanced version of the Sieve of Eratosthenes called Euler's Sieve, highlighting its significance in hardware performance. In addition, [5] presents innovations aimed at reducing inefficiencies in prime generation, using a simple method to substantially reduce hidden constant values for more efficient algorithms. Meanwhile, [6] proposes a recursive algorithm for generating random provable primes, demonstrating high efficiency over pseudoprime methods and ensuring uniform distribution.

Parallel and distributed methods have also gained traction. [7] showcases a parallel algorithm using the Parallel Virtual Machine library, demonstrating significant performance enhancements in prime generation through effective scalability and reduced communication latency. Additionally, [8] introduces a CUDA-optimized solution that exploits GPU capabilities, significantly reducing computation time compared to traditional CPU methods, reinforcing the trend toward leveraging parallel computing in prime generation.

The Sieve of Atkin, often overshadowed by other algorithms, has seen renewed interest. [9] provides a comprehensive review and parallel optimization of this sieve, positioning it as one of the most effective approaches available today. Furthermore, parallelization strategies for prime generation are discussed in [10], which examines two parallel sieves suited for EREW PRAM models, emphasizing their efficiency despite communication latencies.

Lastly, [11] explores load-balanced parallel algorithms for the Sieve of Eratosthenes implemented on cluster computers, further illustrating the trend toward optimizing computational resources for efficient large prime generation.

3 Solution Architecture

Prime number generation, or identifying all prime numbers less than or equal to a given number N, is essential in computational number theory. This section provides an overview of the solution architecture used for parallelizing the following sieve-based algorithms for generating primes up to the given limit N: the Sieve of Eratosthenes and the Sieve of Euler. We will begin with a brief explanation of how the sequential versions of these algorithms work.

3.1 Basic Explanation of Sequential Sieve Algorithms

The **Sieve of Eratosthenes** iterates through a list of numbers, marking multiples of each prime as non-prime. It initializes an array where each index represents a number, initially assuming all are prime. Starting from 2, it marks all multiples of each prime as non-prime. The algorithm continues until all numbers up to N have been processed.

The **Sieve of Euler** is a refinement of the Sieve of Eratosthenes. Instead of marking all multiples of a prime, it only marks numbers divisible by remaining prime numbers. This reduces unnecessary checks, resulting in potentially better performance.

3.2 Parallelization of the Sieve Algorithms

We will now explain the parallelization of both the **Sieve of Eratosthenes** and the **Sieve of Euler**. Since these two algorithms are similar, with the primary distinction being that the Sieve of Euler introduces additional constraints to rule out unnecessary checks, we will discuss them together.

3.2.1 Parallelization of the Sieve of Eratosthenes and the Sieve of Euler

Let's assume we want to find all primes less than or equal to N, and we have K threads available. First, we generate a boolean array 'primes' of length N, where each value at index i indicates whether the number i is prime. Initially, we assume all numbers from 0 to N are prime. Then, we sequentially find all primes up to $\sqrt{N}$ using the appropriate Sieve of Eratosthenes or Sieve of Euler algorithm. For each number i in the range $[0, \sqrt{N}]$, we find its multiples and mark them as non-prime. These sequentially found primes are referred to as "markers."

After identifying all the markers, we proceed with the parallelization. We divide the remaining array $[\sqrt{N}, N]$ into k equal subarrays. Each subarray and the markers are assigned to a separate thread. Each thread then eliminates all the multiples of the markers within its assigned subarray, ensuring that only primes remain. Once each thread finishes processing its subarray, the results (primes) from all subarrays and markers are accumulated into a single results array. This array will contain all prime numbers from 0 to N, which we then return. This algorithm is visually explained in Figure 1.

A natural question arises: why do we only calculate the markers up to $\sqrt{N}$? How can we be sure that in the subarrays, there won't be a prime that is also a "marker" and should be handled by other threads? The answer is based on a fundamental theorem in prime number theory: when finding prime numbers up to N, you only need to check and mark multiples of prime numbers up to $\sqrt{N}$. Any composite number greater than $\sqrt{N}$ must have at least one prime factor less than or equal to $\sqrt{N}$.

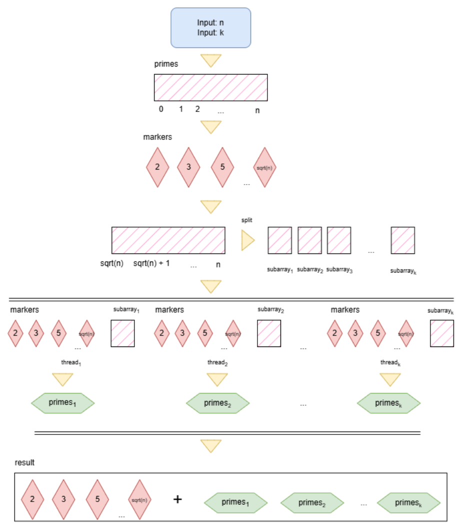

Input: n
Input: k
primes
0 1 2 ... n
markers
2 3 5 ... sqrt(n)
split
sqrt(n) sqrt(n) + 1 ... n
subarray$_1$ subarray$_2$ subarray$_3$ ... subarray$_k$
thread$_1$ thread$_2$ thread$_k$
primes$_1$ primes$_2$ ... primes$_k$
result
+

Figure 1: Parallelization of the Sieve of Eratosthenes and the Sieve of Euler

3.3 Evaluation Methods

We will compare the sequential and parallel solutions primarily through two key metrics:

- **Total Execution Time**
- **Improvement Factor (IF)** from the sequential solution, determined using the formula:

$$IF = \frac{T_S - T_P}{T_S}$$

where T_S and T_P represent the total execution time of the sequential and parallel solutions, respectively.

Since our parallelized algorithm contains a crucial sequential part (the finding of the markers), we cannot expect a linear or superlinear improvement in the total execution time. A sublinear improvement is more realistic, and for large values of N, this improvement might make a significant difference in performance.

4 Results

In this section, we present the results of the algorithm evaluations. Since the results heavily depend on the machine and its specifications, we will first describe the characteristics of the machine used for the experiments.

4.1 Machine Specifications

All experiments were conducted on a machine equipped with an Intel Core i5-1235U, a 12th-generation CPU with a base clock speed of 1.3 GHz. The CPU has 10 physical cores and 12 logical processors, making it suitable for the experiments.

The machine also features two DDR4 8GB RAM chips with a speed of 3200 MT/s.

4.2 Algorithm Evaluations

4.2.1 Sieve of Eratosthenes

With the groundwork established, we now present the results of the algorithms. As shown in Figure 2, the parallelized version of the Sieve of Eratosthenes exhibits worse execution times for smaller limits (the bounds up to which primes are generated). However, it significantly improves as the limits increase.

More specifically:

- The parallelized version with 2 threads outperforms the sequential version at the 10^5 mark.

- The parallelized version with 4 threads surpasses the sequential version at the 10^4 mark.

From these points onward, the parallelized versions (both 2 and 4 threads) consistently outperform the sequential version.

The improvement factor discussed in Section 3.3 is shown in Figure 3, while the parallelization improvement with two threads is presented in Figure 4. As evident in Figure 4 and as expected (and noted in Section 3.3), most of the parallelization improvements are sublinear. A few exceptions exhibit superlinear behavior, which are purely coincidental and may vary across runs.

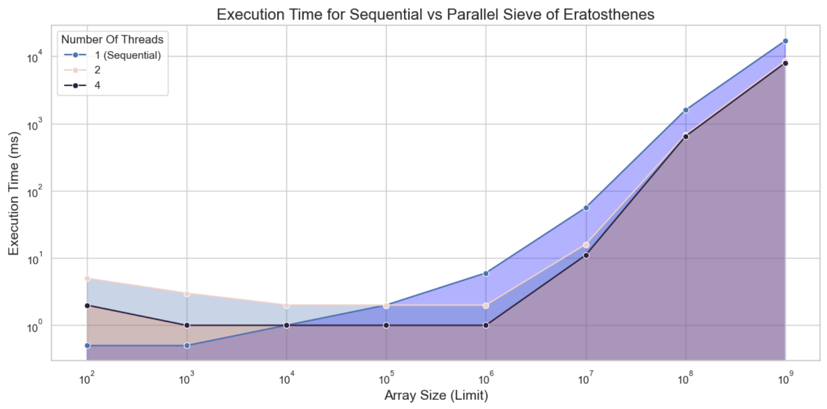
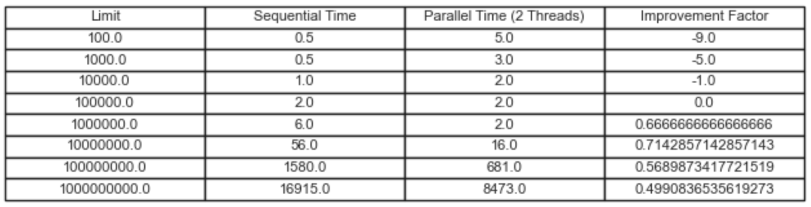
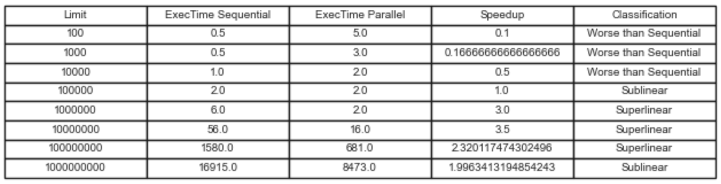


Figure 2: Execution Time for Sequential vs Parallel Sieve of Eratosthenes



Limit	Sequential Time	Parallel Time (2 Threads)	Improvement Factor
100.0	0.5	5.0	-9.0
1000.0	0.5	3.0	-5.0
10000.0	1.0	2.0	-1.0
100000.0	2.0	2.0	0.0
1000000.0	6.0	2.0	0.6666666666666666
10000000.0	56.0	16.0	0.7142857142857143
100000000.0	1580.0	681.0	0.5689873417721519
1000000000.0	16915.0	8473.0	0.4990836535619273

Figure 3: Improvement Factor for the Parallelized Sieve of Eratosthenes



Limit	ExecTime Sequential	ExecTime Parallel	Speedup	Classification
100	0.5	5.0	0.1	Worse than Sequential
1000	0.5	3.0	0.16666666666666666	Worse than Sequential
10000	1.0	2.0	0.5	Worse than Sequential
100000	2.0	2.0	1.0	Sublinear
1000000	6.0	2.0	3.0	Superlinear
10000000	56.0	16.0	3.5	Superlinear
100000000	1580.0	681.0	2.320117474302496	Superlinear
1000000000	16915.0	8473.0	1.9963413194854243	Sublinear

Figure 4: Parallelization Improvement With Two Threads for Sieve of Eratosthenes

4.2.2 Sieve of Euler

Figure 5 illustrates the execution times for the sequential and parallelized versions (2 and 4 threads) of the Sieve of Euler algorithm for various prime limits. Similar to the Sieve of Eratosthenes, the parallelized version performs worse for smaller limits but significantly improves for larger ones.

Key observations:

- The parallelized version with 2 threads outperforms the sequential version around the 10^3 mark.
- The parallelized version with 4 threads becomes faster at approximately the 10^4 mark.

The improvement factor for the Sieve of Euler is shown in Figure 6, while the parallelization improvement with two threads is detailed in Figure 7. As before, sublinear improvements are the expected outcome.

4.2.3 Sieve of Eratosthenes vs. Sieve of Euler

The comparison between the Sieve of Eratosthenes and the Sieve of Euler is presented in Figure 8. As noted earlier, the Sieve of Euler performs slightly fewer calculations by avoiding redundant marking of multiples, ensuring each composite number is marked exactly once.

However, the Sieve of Euler has slightly worse memory complexity, as it requires an additional array for composite numbers. The slightly inferior performance of the Sieve of Euler for smaller limits is expected since it requires extra time to create and populate this array. For higher limits, this overhead is negligible relative to the total execution time, resulting in better overall performance.

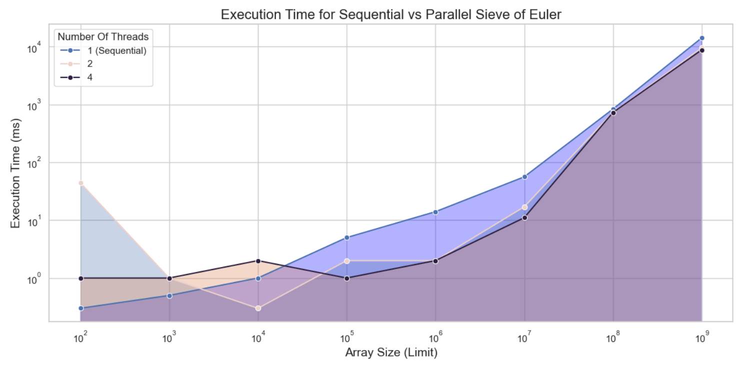
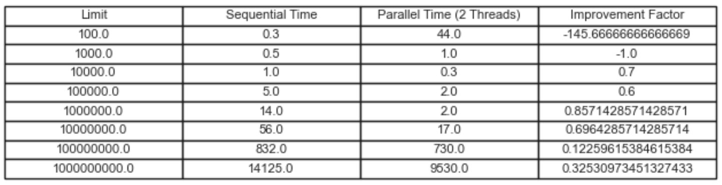
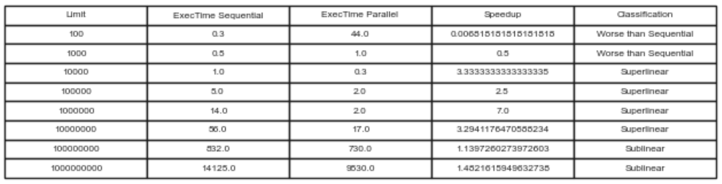


Figure 5: Execution Time for Sequential vs Parallel Sieve of Euler



Limit	Sequential Time	Parallel Time (2 Threads)	Improvement Factor
100.0	0.3	44.0	-145.66666666666669
1000.0	0.5	1.0	-1.0
10000.0	1.0	0.3	0.7
100000.0	5.0	2.0	0.6
1000000.0	14.0	2.0	0.8571428571428571
10000000.0	56.0	17.0	0.6964285714285714
100000000.0	832.0	730.0	0.12259615384615384
1000000000.0	14125.0	9530.0	0.32530973451327433

Figure 6: Improvement Factor for the Parallelized Sieve of Euler



Limit	ExecTime Sequential	ExecTime Parallel	Speedup	Classification
100	0.3	44.0	0.006818181818181818	Worse than Sequential
1000	0.5	1.0	0.5	Worse than Sequential
10000	1.0	0.3	3.3333333333333335	Superlinear
100000	5.0	2.0	2.5	Superlinear
1000000	14.0	2.0	7.0	Superlinear
10000000	56.0	17.0	3.2941176470588234	Superlinear
100000000	832.0	730.0	1.1397260273972603	Sublinear
1000000000	14125.0	9530.0	1.4821615949632738	Sublinear

Figure 7: Parallelization Improvement With Two Threads for Sieve of Euler

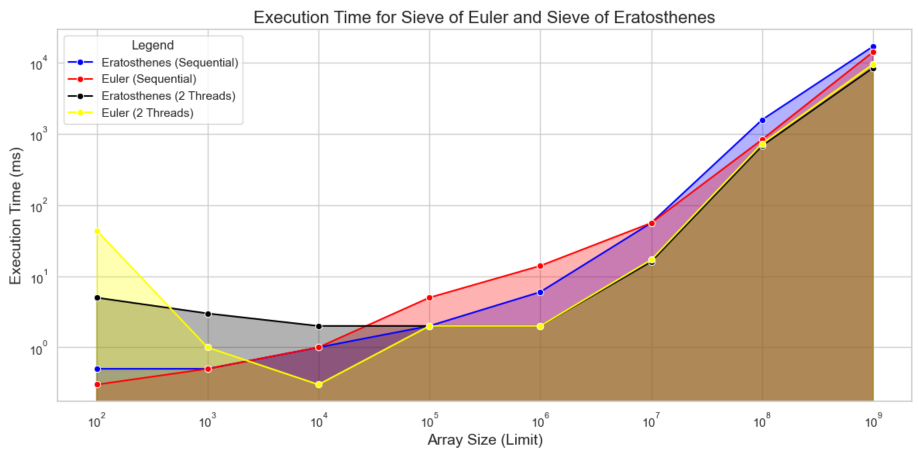


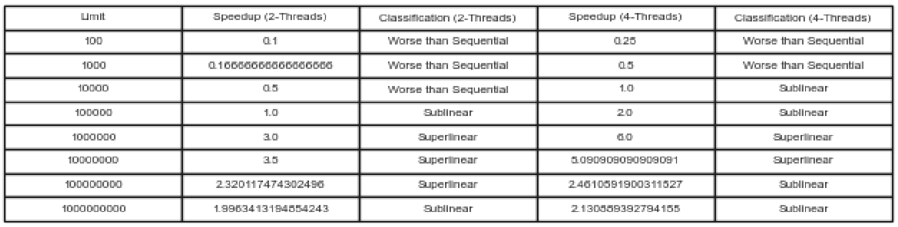
Figure 8: Execution Time for Sieve of Eratosthenes and Sieve of Euler

5 Conclusion and Future Work

In conclusion, the results obtained align well with our expectations. As outlined in Section 3, both algorithms contain a critical sequential component (the identification of markers), which limits the potential for parallelization improvements. As shown in Figure 9 and Figure 10, both algorithms exhibit slight superlinear speedup improvements between 10^5 and 10^7 mark, after which the speedup becomes sublinear. Furthermore, after the 10^7 mark, the speedup for both algorithms stabilizes at around a factor of 2. Based on these findings, we can conclude that parallelization using 2 threads is optimal for both algorithms. This is because the overhead associated with parallelization increases linearly as the number of threads grows.

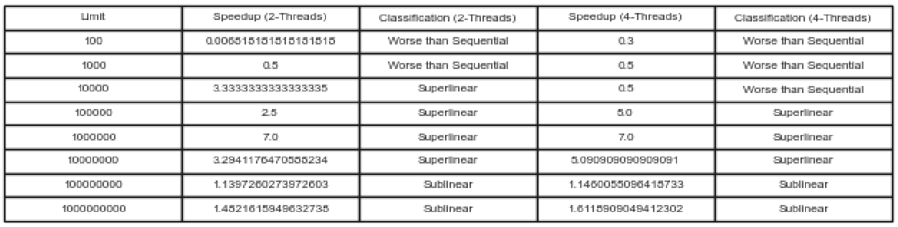
Nevertheless, for larger limits ($>10^7$), the parallelized versions of both the Sieve of Eratosthenes and the Sieve of Euler demonstrate improvements in execution time, highlighting their effectiveness in computationally intensive scenarios.

While the current implementation optimizes prime number sieving, further improvements can be explored. Future work includes implementing GPU-based parallelization to exploit massive parallelism, and extending the sieving techniques to distributed computing environments. Additionally, investigating hybrid approaches that combine elements of both algorithms could lead to even more efficient solutions.



Limit	Speedup (2-Threads)	Classification (2-Threads)	Speedup (4-Threads)	Classification (4-Threads)
100	0.1	Worse than Sequential	0.25	Worse than Sequential
1000	0.16666666666666666	Worse than Sequential	0.5	Worse than Sequential
10000	0.5	Worse than Sequential	1.0	Sublinear
100000	1.0	Sublinear	2.0	Sublinear
1000000	3.0	Superlinear	6.0	Superlinear
10000000	3.5	Superlinear	5.090909090909091	Superlinear
100000000	2.320117474302496	Superlinear	2.4610591900311527	Sublinear
1000000000	1.9963413194554243	Sublinear	2.130559392794155	Sublinear

Figure 9: Parallelization Improvement With Two and Four Threads for Sieve of Eratosthenes



Limit	Speedup (2-Threads)	Classification (2-Threads)	Speedup (4-Threads)	Classification (4-Threads)
100	0.006515151515151515	Worse than Sequential	0.3	Worse than Sequential
1000	0.5	Worse than Sequential	0.5	Worse than Sequential
10000	3.3333333333333335	Superlinear	0.5	Worse than Sequential
100000	2.5	Superlinear	5.0	Superlinear
1000000	7.0	Superlinear	7.0	Superlinear
10000000	3.2941176470555234	Superlinear	5.090909090909091	Superlinear
100000000	1.1397260273972603	Sublinear	1.1460055096415733	Sublinear
1000000000	1.4521615949632735	Sublinear	1.6115909049412302	Sublinear

Figure 10: Parallelization Improvement With Two and Four Threads for Sieve of Euler

References

[1] Chakrabarty, Alok, and Bipul Syam Purkayastha. "An Analysis of Some Prime Generating Sieves." *IUP Journal of Computer Sciences* 4.1 (2010).

[2] Umadevi, V., and K. Sivakami. "Comparison and Performance Evaluation of Prime Number Generation Using Sieves Algorithm." *Journal of Computational and Theoretical Nanoscience* 17.4 (2020): 1839-1841.

[3] Ghidarcea, Mircea, and Decebal Popescu. "Prime Number Sieving—A Systematic Review with Performance Analysis." *Algorithms* 17.4 (2024): 157.

[4] Salvo, Ivano, and Agnese Pacifico. "Three Euler's Sieves and a Fast Prime Generator (Functional Pearl)." *arXiv preprint arXiv:1811.09840* (2018).

[5] Joye, Marc, Pascal Paillier, and Serge Vaudenay. "Efficient generation of prime numbers." *Cryptographic Hardware and Embedded Systems—CHES 2000: Second International Workshop Worcester, MA, USA, August 17–18, 2000 Proceedings 2*. Springer Berlin Heidelberg, 2000.

[6] Maurer, Ueli M. "Fast generation of prime numbers and secure public-key cryptographic parameters." *Journal of Cryptology* 8 (1995): 123-155.

[7] Al-Asadi, E. A. "Finding N prime numbers using distrusted computing PVM (parallel virtual machine)." *Int. J. Eng. Technol* 5.11 (2015): 578-588.

[8] Nezarat, Amin, Mahdi Raja, and G. H. Dastghaibifard. "A new high performance gpu-based approach to prime numbers generation." *World Applied Programming* 5.1 (2015): 1-7.

[9] Ghidarcea, Mircea, and Decebal Popescu. "Sieve of Atkin Revisited." *Sci. Bull. Univ. Politeh. Buchar* 86 (2024): 15-26.

[10] Sorenson, Jonathan, and Ian Parberry. "2 Fast Parallel Prime Number Sieves." *Information and Computation* 114.1 (1994): 115-130.

[11] Hwang, Soonwook, Kyusik Chung, and Dongseung Kim. "Load balanced parallel prime number generator with sieve of eratosthenes on cluster computers." *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*. IEEE, 2007.