



Article

Efficient Sequential and Parallel Prime Sieve Algorithms

Hazem M. Bahig^{1,2,*} , Mohamed A. G. Hazber¹, Khaled Al-Utaibi³, Dieaa I. Nassr²  and Hatem M. Bahig²

¹ Information and Computer Science Department, College of Computer Science and Engineering, University of Ha'il, Ha'il 81481, Saudi Arabia

² Computer Science Division, Department of Mathematics, Faculty of Science, Ain Shams University, Cairo 11566, Egypt

³ Computer Engineering Department, College of Computer Science and Engineering, University of Ha'il, Ha'il 81481, Saudi Arabia

* Correspondence: h.bahig@uoh.edu.sa

Abstract: Generating prime numbers less than or equal to an integer number m plays an important role in many asymmetric key cryptosystems. Recently, a new sequential prime sieve algorithm was proposed based on set theory. The main drawback of this algorithm is that the running time and storage are high when the size of m is large. This paper introduces three new algorithms for a prime sieve based on two approaches. The first approach develops a fast sequential prime sieve algorithm based on set theory and some structural improvements to the recent prime sieve algorithm. The second approach introduces two new parallel algorithms in the shared memory parallel model based on static and dynamic strategies. The analysis of the experimental studies shows the following results. (1) The proposed sequential algorithm outperforms the recent prime sieve algorithm in terms of running time by 98% and memory consumption by 80%, on average. (2) The two proposed parallel algorithms outperform the proposed sequential algorithm by 72% and 67%, respectively, on average. (3) The maximum speedups achieved by the dynamic and static parallel algorithms using 16 threads are 7 and 4.5, respectively. As a result, the proposed algorithms are more effective than the recent algorithm in terms of running time, storage and scalability in generating primes.

Keywords: asymmetric key cryptography; prime numbers; sieve algorithm; parallel algorithm; high-performance computing



Citation: Bahig, H.M.; Hazber, M.A.G.; Al-Utaibi, K.; Nassr, D.I.; Bahig, H.M. Efficient Sequential and Parallel Prime Sieve Algorithms. *Symmetry* **2022**, *14*, 2527. <https://doi.org/10.3390/sym14122527>

Academic Editor: Christos Volos

Received: 28 October 2022

Accepted: 22 November 2022

Published: 30 November 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Many problems in number theory and computer arithmetic play important roles in cryptography. Examples of such problems are the generation of prime numbers [1–3], primality testing [4,5], modular exponentiation [6], addition chains and sequences [7,8] and integer factorization [9–12]. Developing fast algorithms that address these problems is one of the main challenges of algorithm complexity and leads to significant improvements in various applications.

The research in this paper focuses on generating primes. Given an integer number m , the objective is to find all prime numbers p such that $1 < p \leq m$, where a number p is prime if and only if it has only two divisors, 1 and p . There are many crucial applications [13–17] for generating a prime number or all prime numbers up to an integer m , for example:

1. Creating one or more large primes is a step in the key generation algorithms of some asymmetric key cryptosystems, such as the Rivest, Shamir and Adleman (RSA) cryptosystem [13]; protocols, such as the Diffie–Hellman exchange protocol [14]; and digital signature algorithms, such as ElGamal [15].
2. Error-correcting codes are defined over the Galois field of prime power order [17], where Galois' theory originated in the study of symmetric functions, i.e., the coefficients of a monic polynomial are the elementary symmetric polynomials in the roots.

3. The security of many asymmetric key cryptosystems, such as RSA, is based on factoring a large composite (nonprime) integer into its prime factors. One efficient method to find a small prime factor for any composite integer is by using prime sieving.

Several algorithms have been proposed to generate prime numbers up to an integer m using different techniques and platforms. The sieve of Eratosthenes, set theory, and wheel factorization are all examples of different ways to generate prime numbers up to the integer m . On the other hand, different platforms are used to design the solution, such as sequential and parallel computers.

The oldest and simplest algorithm to generate prime numbers up to an integer m is the sieve of Eratosthenes [1]. The algorithm works by repeatedly marking the multiples of each prime as a composite number, beginning with the first prime number, 2. There are two main disadvantages of the Eratosthenes sieve algorithm. First, the run time of the algorithm is not linear, $O(m \log \log m)$. Second, the algorithm uses a large storage size, $O(m)$.

Based on the two disadvantages of the Eratosthenes sieve algorithm, different algorithms have been proposed. Some of these algorithms [18–22] reduced the running time to a linear time by testing each composite number exactly once. Marison [18], theoretically, reduced the time complexity of prime sieving using a double linked list. In [19], the authors achieved linear time complexity based on assuming that the multiplication of integers that are less than or equal to m can be performed in a constant time. The Pritchard sieve [20] avoided considering near-composite numbers by generating gradually larger wheels that represent the sequence of numbers that are not divisible by any of the primes already processed. In [21], the author introduced a practical improvement on the Eratosthenes sieve algorithm. In [22], the author reduced the time of the sieving algorithm to a sublinear time using the wheel method.

Another technique for improving the prime sieve is the segmented sieve [23–26], which aims to reduce the memory consumption of the sieve method. The technique is based on dividing the range, 2 to m , into m/Δ subintervals, each of size Δ . Then, the method sieves one subinterval at a time by marking the multiples of each prime in the subinterval. Experimentally, the value of Δ is equal to \sqrt{m} . In [26], the author reduced the memory consumption to $O(m^{1/3} (\log m)^{2/3})$ as a theoretical study.

Recently, in 2021, the authors of [1] proposed an algorithm based on set theory to generate a set of all prime numbers less than or equal to an integer $m \geq 9$. The proposed algorithm outperforms the Eratosthenes and Sundaram algorithms in their run time, where the Sundaram sieve algorithm has a weaker performance than the Eratosthenes sieve algorithm [1].

On the other hand, there are several parallel implementations for prime sieve algorithms. In [27], two different distributed sieves of Eratosthenes algorithms on a hypercube computer, NCUBE/10, were introduced. Sorenson and Parberry [28] presented and analyzed two theoretical parallel sieve algorithms on an exclusive read exclusive write parallel random-access machine. The authors of [29] presented a parallel version of the Eratosthenes sieve algorithm on a cluster machine consisting of eight nodes. The proposed algorithm uses static and dynamic strategies to achieve load balancing between the processors. Two distributed algorithms were theoretically proposed in [2,30]. They are based on scheduling by a multiple edge reversal (SMER) graph and the wheel technique.

For three reasons, the work in this paper is based on the algorithm proposed by Abd-Elnaby and El-Baz [1], denoted as the AE algorithm. First, the AE algorithm is a recent algorithm that was published in 2021. Second, Abd-Elnaby and El-Baz showed that the AE algorithm is more efficient in its run time than the Eratosthenes and Sundaram algorithms by 81% and 97%, respectively [1]. Third, there is no parallel algorithm for the AE algorithm. In addition, the main drawback of this algorithm is that its run time and storage are high when the size of m is large.

In light of the above reasons, this paper introduces three algorithms for prime sieving based on two different platforms: sequential and multicore systems. The first algorithm is sequential. The other two algorithms are parallel. The experimental results of implementing

the AE algorithm and the three proposed algorithms on different data sets show the following contributions.

1. The proposed sequential algorithm (Section 3) is more efficient than the AE algorithm by 98% on average.
2. The proposed sequential algorithm reduces the memory consumption used by the AE algorithm by 80% on average.
3. The two proposed parallel algorithms are more efficient than the proposed sequential algorithm by 72% and 67%, respectively, on average. The proposed parallel algorithms are scalable.

The structure of the paper consists of an introduction and five sections. In Section 2, the AE algorithm is described. The first proposed algorithm for a single processor is presented in Section 3. In Section 4, two parallel algorithms for prime sieving are introduced. The experimental studies for the three proposed algorithms are presented in Section 5. Finally, Section 6 contains the conclusion and open research questions.

2. AE Algorithm

Abd Elnaby and El-Baz [1] proposed an algorithm to generate the set, $P(m)$, of all prime numbers less than or equal to an integer $m \geq 9$, based on set theory. The idea of the algorithm is based on generating k sets:

$$A_i = \{(2i + 1)(2i + 2n_i + 1) : 0 \leq n_i \leq \lfloor (m - (2i + 1)^2) / (4i + 2) \rfloor\}, \quad (1)$$

where $1 \leq i \leq k$. If $n_{k+1} < 0$ and $A_{k+1} = \Phi$, then the set of all prime numbers less than or equal to m is

$$P(m) = \{2\} \cup (B - A), \quad (2)$$

where $B = \{2j + 1 : 1 \leq j \leq \lfloor (m - 1) / 2 \rfloor\}$.

The AE algorithm can be divided into four main steps, as follows.

- Step 1: An auxiliary array B of size $\lfloor (m - 1) / 2 \rfloor$ is initialized, with odd numbers only. Array B is used to determine the final set P . This step represents Lines 1–2.
- Step 2: The number of sets k is determined, which can be generated and used for Step 3. This step represents Lines 3–6 and aims to find the value of k such that $n_{k+1} < 0$.
- Step 3: The set of all nonprime odd numbers A is determined. This step represents Lines 7–13 and aims to generate k sets, where, in each iteration, i , the algorithm generates one set A_i . The k sets are collected in A , where $A = \cup_{i=1}^k A_i$ (see Line 12).
- Step 4: The prime numbers are determined only as in Line 14. This step can be computed by calculating the difference between two sets, the set of all odd numbers and the set of all nonprime numbers.

The complete pseudocode for the AE algorithm is shown in Algorithm 1.

Algorithm 1: AE.**Input:** A positive integer m .

1. for $i = 1$ to $\lfloor (m - 1)/2 \rfloor$ do
2. $B[i] = 2i + 1$
3. $k = i = 1$
4. while $i > 0$ do
5. $i = \lfloor (m - (2k + 1)^2)/(4k + 2) \rfloor$
6. $k = k + 1$
7. $i = 1; n = 0$
8. while $i \leq k$ do
9. $d = \lfloor (m - (2i + 1)^2)/(4i + 2) \rfloor$
10. while $n \leq d$ do
11. $X[n +] = (2i + 1)(2i + 2n + 1)$
12. Add X to the array A
13. Empty X
14. $P = \{2\} \cup (B - A)$

Output: A set of all prime numbers less than or equal to m **3. The Proposed Sequential Algorithm**

This section describes the main idea behind the new prime sieve sequential algorithm. This section also presents the pseudocode of the proposed algorithm that aims to reduce the amount of time and storage for the AE algorithm.

3.1. Main Idea

The AE algorithm consists of four main steps, and the proposed algorithm focuses on modifying these steps.

For Step 1, there is no need to store the odd numbers $2i + 1$ into array B . Initially, all odd numbers can be marked as a prime number by setting the value 1 at position i in array B , where position number i represents an odd number $2i + 1$.

For Step 2, the value of k , which is used for Step 3, can be computed in a single statement without making a loop as follows. The while loop of Step 2 terminates when

$$\frac{m - (2k + 1)^2}{4k + 2} \leq 0 \quad (3)$$

Therefore, the question now is as follows: what is the value of k that satisfies inequality (3)? The value of k can be computed by solving inequality (3), as follows:

$$\begin{aligned} m - (2k + 1)^2 &\leq 0 \\ 2k + 1 &\geq \sqrt{m} \\ k &\geq \frac{\sqrt{m} - 1}{2} \end{aligned} \quad (4)$$

Therefore, instead of computing k by executing a number of iterations, the value of k is computed directly by inequality (4). The main advantage of this modification is reducing the computation time from $O(k) = O(\sqrt{m}/2)$ to $O(1)$.

For Step 3, the AE algorithm collects all multiples of the prime numbers and assigns them to the auxiliary array X . This step can be modified by ignoring the collection of these numbers, multiples of primes, in array X , so there is no need to use array X . Thus, array B can be modified directly as follows.

Since the integer $(2i + 1)(2i + 2n + 1)$ is odd but not prime, and since the location of this integer in array B is $\lfloor (2i + 1)(2i + 2n + 1)/2 \rfloor$, the algorithm marks the integer $(2i + 1)(2i + 2n + 1)$ as a composite number by modifying array B directly, as follows: $B[\lfloor (2i + 1)(2i + 2n + 1)/2 \rfloor] = 0$.

The main advantages of this modification are as follows.

1. Reducing the auxiliary storage by a factor $O(d) = O(\lfloor (m-9)/6 \rfloor)$ in the worst case. The worst case occurs when $i = 1$. Therefore, the storage of the modified AE algorithm is reduced by a factor $O(|A| + |X|)$.
2. Reducing the run time of assigning array X to the auxiliary array A (see Line 12 in Algorithm 1).
3. There is no need to compute the difference between the two arrays as in Step 4 (see Line 14 in Algorithm 1).

For Step 4, the AE algorithm computes the difference between two arrays. The first array B contains all odd numbers, and the other array X represents all nonprime odd numbers.

Based on the improvement in Step 3, all prime numbers are found by the following simple test: if the value of $B[i]$ does not change to 0, i.e., $B[i] = 1$, then the integer $2i + 1$ is prime, and thus the algorithm stores the integer $2i + 1$ to the output set P . Otherwise, the algorithm ignores the integer $2i + 1$. The pseudocode for this step is as follows: if $B[i] = 1$, then $P[j++] = 2i + 1$, where j represents the location of the prime number $2i + 1$ at the array P and starts with 1.

The main advantage of this step is reducing the time from $O(|A| + |B|)$ to $O(|B|)$ if A is a sorted array in the best case; otherwise, the time is reduced from $O(|A|^2)$ to $O(|B|)$.

3.2. The Algorithm

From the previous comments on the AE algorithm and the suggested improvements, the pseudocode of the modified AE algorithm, denoted as MAE, is given in Algorithm 2. Steps 1, 2, 3 and 4 represent Lines 1–2, 3, 4–9 and 10–14, respectively.

Algorithm 2: MAE.

Input: A positive integer m .

1. for $i = 1$ to $\lfloor (m-1)/2 \rfloor$ do
2. $B[i] = 1$
3. $k = \lfloor (\sqrt{m}-1)/2 \rfloor$
4. for $i = 1$ to k do
5. $d = \lfloor (m - (2i+1)^2)/(4i+2) \rfloor$
6. $n = 0$
7. while $(d \neq 0)$ and $(n \leq d)$ do
8. $B[(2i+1)(2i+2n+1)] = 0$
9. $n = n + 1$
10. $P[0] = 2; j = 1$
11. for $i = 1$ to $\lfloor (m-1)/2 \rfloor$ do
12. if $B[i] = 1$ then
13. $P[j] = 2i + 1$
14. $j = j + 1$

Output: A set P of all prime numbers less than or equal to m .

4. The Proposed Parallel Algorithms

This section presents two new parallel algorithms for prime sieving by parallelizing the MAE algorithm on a shared memory parallel model. In this model, all processors communicate via shared memory and assume that the model can run t threads at the same time.

The parallelization of the MAE algorithm can be performed by parallelizing each step in the MAE algorithm, except for the second step, since it is a simple statement. Two versions of parallelization, PMAE-1 and PMAE-2, are introduced. Both versions are similar except for the parallelization of the third step of the MAE algorithm. The steps of the PMAE-1 algorithm, Algorithm 3, are as follows.

For the first step, the sequential loop can be parallelized easily by dividing it into t subranges of equal size, and each thread works independently on one subrange (see Lines

1–8 in the PMAE-1 algorithm). In this step, there is no need to read the same element by many threads and to write in the same cell concurrently by more than one thread.

For the second step, there is no need to parallelize it because it consists of one simple step (see Line 9 of the PMAE-1 algorithm).

For the third step, the parallelization of this step is performed dynamically because the number of iterations for the inner loop is based on the value of d , and this value varies for each new iteration of the outer loop, i.e., it depends on the value of i . For example, when $i = 1$, the value of $d = \lfloor (m - 9)/6 \rfloor$, whereas the value of $d = \lfloor (m - 25)/10 \rfloor$ when $i = 2$. It is clear that the value of the numerator of d decreases and that the value of the denominator increases. Since each iteration of the outer loop in Step 3 of the MAE algorithm is done individually, we parallelize the outer loop while keeping the inner loop sequential (see Lines 10–14 in the PMAE-1 algorithm).

For the fourth step, the parallelization of this step can be performed by using three substeps. In the first substep, each thread t_i counts the number of primes in the subrange

$$r_i = \left[\frac{i \lfloor (m - 1)/2 \rfloor}{t}, \frac{(i + 1) \lfloor (m - 1)/2 \rfloor}{t} \right], 0 \leq i < t, \quad (5)$$

and stores this value, i.e., the number of primes, in the variable $count[i]$, where

$$count[i] = |\{x : x \text{ is a prime and } x \in r_i\}|, 0 \leq i < t \quad (6)$$

Algorithm 3: PMAE-1.

Input: A positive integer m and t threads.

1. $m_1 = \lfloor (m - 1)/2 \rfloor$
2. for $i = 0$ to $t - 1$ do parallel
3. if $(i \neq t - 1)$ then
4. for $j = (i \times m_1)/t + 1$ to $((i + 1) \times m_1)/t$ do
5. $B[j] = 0$
6. else
7. for $j = (i \times m_1)/t + 1$ to m_1 do
8. $B[j] = 0$
9. $k = \lfloor (\sqrt{m} - 1)/2 \rfloor$
10. for $i = 1$ to k do parallel (Dynamically)
11. $d = \lfloor (m - (2i + 1)^2)/(4i + 2) \rfloor$
12. if $(d \neq 0)$ then
13. for $n = 0$ to d do
14. $B[(2i + 1)(2i + 2n + 1)/2] = 0$
15. for $i = 0$ to $t - 1$ do parallel
16. $count[i] = 0$
17. if $(i \neq t - 1)$ then
18. for $j = (i \times m_1)/t + 1$ to $((i + 1) \times m_1)/t$ do
19. if $B[j] \neq 0$ then
20. $TempP[i, count[i]] = 2 \times j + 1$
21. $count[i] = count[i] + 1$
22. Else
23. for $j = (i \times m_1)/t + 1$ to m_1 do
24. if $B[j] \neq 0$ then
25. $TempP[i, count[i]] = 2 \times j + 1$
26. $count[i] = count[i] + 1$
27. $ps[0] = 0$
28. for $i = 1$ to $t - 1$ do
29. $ps[i] = ps[i - 1] + count[i - 1]$
30. for $i = 0$ to t do parallel
31. for $j = 0$ to $count[i] - 1$ do
32. $P[ps[i] + j] = TempP[i, j]$

Output: A set P of all prime numbers less than or equal to m .

The thread t_i stores the j -th prime number in the range r_i at position $j - 1$ in the auxiliary array $TmpP[i, j]$. This means that all prime numbers in the subrange r_i exist in $TmpP[i, -]$. This substep represents Lines 15–26 of the PMAE-1 algorithm.

In the second substep of Step 4, the algorithm computes the prefix sums ps for the integers $0, count[0], count[1], \dots, count[t - 2]$. The objective of this step is to compute and store the start positions of all prime numbers that exist in the subrange r_i in the output array P . The start position of each subrange is given by

$$ps[i] = \sum_{j=0}^{i-1} count[j], \forall 0 < i < t \quad (7)$$

where $ps[0] = 0$.

If the value of t is small, as in the experimental study, array ps is computed sequentially (see lines 27–29). Otherwise, the algorithm uses the binary tree strategy [31,32] to compute ps .

In the third substep, each thread t_i starts to copy the prime numbers from the auxiliary array $TmpP[i, -]$ to the output array P starting from position $ps[i]$ (see Lines 30–32).

The run time for the PMAE-1 algorithm can be computed as follows. Step 1 requires $O(m_1/t)$, whereas Step 2 requires $O(1)$. Step 4 requires $O(m_1/t + t)$ when ps is computed sequentially and $O(m_1/t + \log t)$ when ps is computed in parallel. Step 3 is based on the number of elements in each set A_i , and its run time is approximately equal to $O(m_1/t)$ in the average case. Therefore, the overall time complexity of the PMAE-1 algorithm is $O(m_1/t + t) = O(m_1/t)$ or $O(m_1/t + \log t) = O(m_1/t)$, since $m_1 \gg t$.

The steps of the PMAE-2 algorithm, Algorithm 4, are similar to those of the PMAE-1 algorithm except for the third step. Step 3 of the PMAE-2 algorithm is based on parallelizing the inner loop, whereas the outer loop is sequential. The inner loop consists of $d + 1$ concurrent iterations that are distributed to t threads. Therefore, Lines 10–14 of the PMAE-1 algorithm can be rewritten as follows.

Algorithm 4: PMAE-2.

Input: A positive integer m and t threads.
 // Similar to Algorithm 3, Lines 1–9
 10. for $i = 1$ to k do
 11. $d = \lfloor (m - (2i + 1)^2) / (4i + 2) \rfloor$
 12. if ($d \neq 0$) then
 13. for $n = 0$ to d do parallel
 14. $B[(2i + 1)(2i + 2n + 1)/2] = 0$
 // Similar to Algorithm 3, Lines 15–32

The run time for Step 3 of the PMAE-2 algorithm is $\sum_{i=1}^k d_i/t \approx O(m_1/t)$, where d_i/t is the run time for the inner loop using t threads, and d_i represents the value of d at iteration i from the outer loop. Therefore, the run time of the PMAE-2 algorithm is $O(m_1/t)$.

5. Experimental Studies

This section experimentally shows the superiority of the proposed sequential and parallel algorithms over the AE algorithm. The experiments were based on implementing all sequential algorithms using the C++ programming language. Open multiprocessing (OpenMP) was used to implement the parallel algorithms on a multicore system. The multicore computer consisted of two Hexa core processors, each of speed 2.6 GHz. The global and cache memories of the computer were of sizes 16 GB and 15 MB, respectively. The computer was operated with the Windows 10 operating system. All run times for implementing the algorithms in this paper were measured in seconds.

5.1. Sequential Algorithms Comparison

The comparison between the two sequential algorithms, AE and MAE, used the same dataset used in [1]: $10^7, 2 \times 10^7, 3 \times 10^7, 4 \times 10^7, 5 \times 10^7, 6 \times 10^7, 7 \times 10^7, 8 \times 10^7, 9 \times 10^7$ and 10^8 . Table 1 shows that the MAE algorithm had better performance than the original algorithm, AE. The percentage of improvement was almost 98%. The two modified steps, 3 and 4 in Algorithm 2, significantly improved the AE algorithm by approximately 90% of the total time.

Table 1. Run time, in seconds, for the AE and MAE algorithms.

m	10^7	2×10^7	3×10^7	4×10^7	5×10^7	6×10^7	7×10^7	8×10^7	9×10^7	10^8
AE Algorithm	9.9	24	44	59.9	83.3	109	127.2	152.2	170.8	192.3
MAE Algorithm	0.18	0.35	0.62	0.96	1.44	1.63	2.1	2.3	2.5	2.9
% of Improvement	98.2	98.5	98.6	98.4	98.3	98.5	98.3	98.5	98.5	98.5

Figure 1 shows a comparison between the AE and MAE algorithms in terms of memory consumption. It is clear that the MAE algorithm used fewer auxiliary arrays than the AE algorithm. The improvement in memory consumption was approximately 80%.

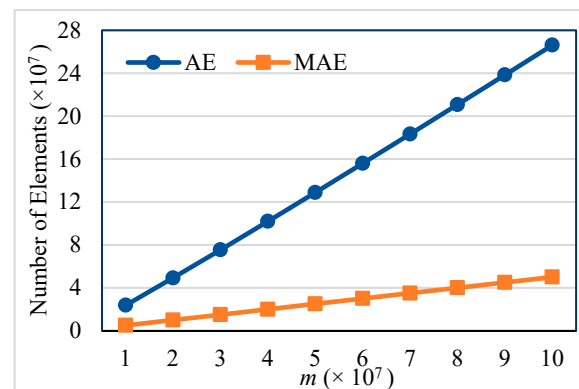


Figure 1. Memory consumption for the AE and MAE algorithms.

5.2. Parallel Algorithms Comparison

This section presents the implementation of the two proposed parallel AE algorithms, PMAE-1 and PMAE-2, and compares them to the MAE algorithm. PMAE-1 and PMAE-2 were implemented using different numbers of threads: 2, 4, 8 and 16. The data set used in the implementation was: $10^8, 2 \times 10^8, 3 \times 10^8, 4 \times 10^8, 5 \times 10^8, 6 \times 10^8, 7 \times 10^8, 8 \times 10^8, 9 \times 10^8$ and 10^9 . Figures 2–5 and Table 2 show comparisons among the run times of the PMAE-1, PMAE-2 and MAE algorithms. From the figures and the table, one can conclude the following.

1. Both algorithms, PMAE-1 and PMAE-2, are faster than the sequential algorithm, MAE, using a minimum number of concurrent threads $t = 2$ (see Figure 2a).
2. The difference between the run times of the PMAE-1 and MAE algorithms (similarly for the PMAE-2 and MAE) increase with an increasing the number of threads (see Figure 2a–d).
3. The run time for the PMAE-1 algorithm (similarly for PMAE-2) decreases with an increasing number of threads; see Figure 3 (similarly see Figure 4).
4. Table 2, Column 2 shows the percentage of improvement for the PMAE-1 algorithm compared with the MAE algorithm with an increasing number of threads. Similarly, the percentage of improvement for the PMAE-2 algorithm compared with the MAE algorithm is less than that for the PMAE-1 algorithm (see Table 2, Column 3). From

Table 2, the two proposed parallel algorithms outperform the modified sequential algorithm, MAE, with improvements of 72% and 67%, respectively, on average.

- Figure 2a–d show that the PMAE-1 algorithm has a better performance than the PMAE-2 algorithm. The percentage of improvement of the PMAE-1 algorithm compared with the PMAE-2 algorithm is 19.7% on average (see Table 2, Column 4).
- The run time for the PMAE-1 algorithm (similarly for PMAE-2) is the sum of the run times for the four steps. This time is analyzed, and the main step that takes a large amount of time is Step 3, which requires more than 80% of the total time. This percentage increases slightly as the value of m increases (see Figure 5).

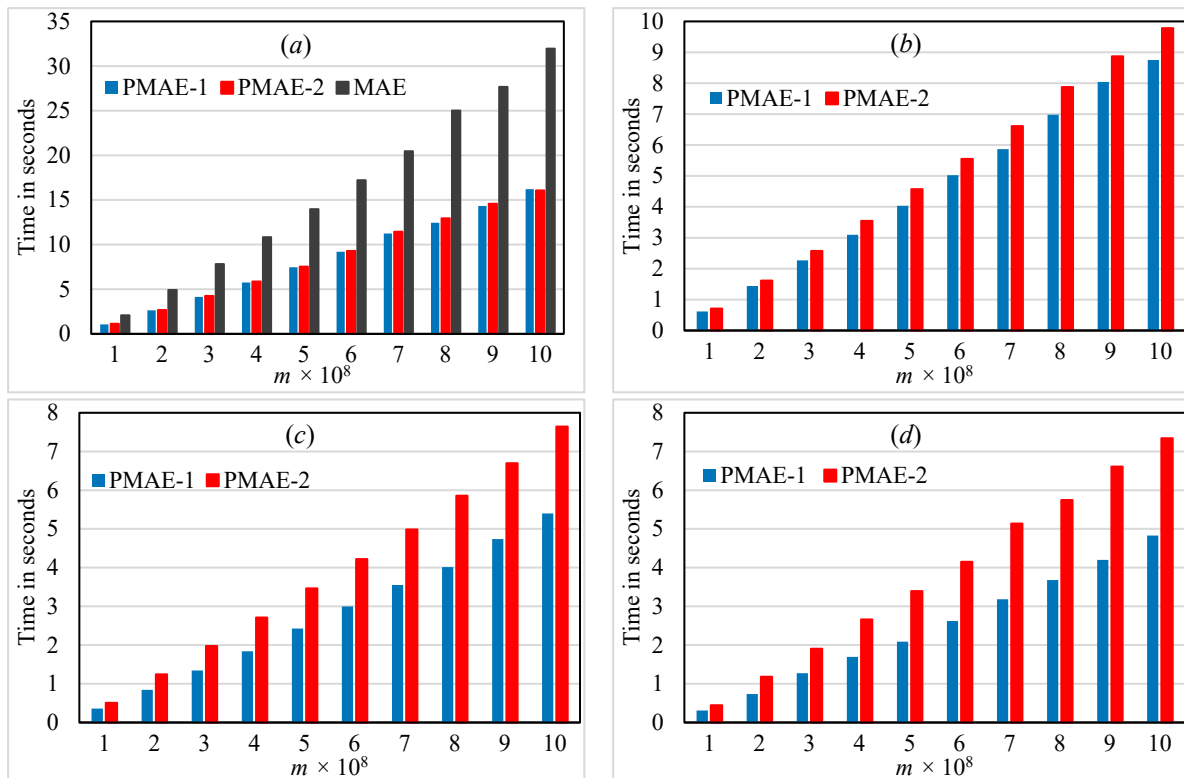


Figure 2. Time comparisons among MAE, PMAE-1, and PMAE-2 algorithms. (a) The running times of MAE using $t = 1$, and PMAE-1 and PMAE-2 using $t = 2$. (b) The running times of PMAE-1 and PMAE-2 using $t = 4$. (c) The running times of PMAE-1 and PMAE-2 using $t = 8$. (d) The running times of PMAE-1 and PMAE-2 using $t = 16$.

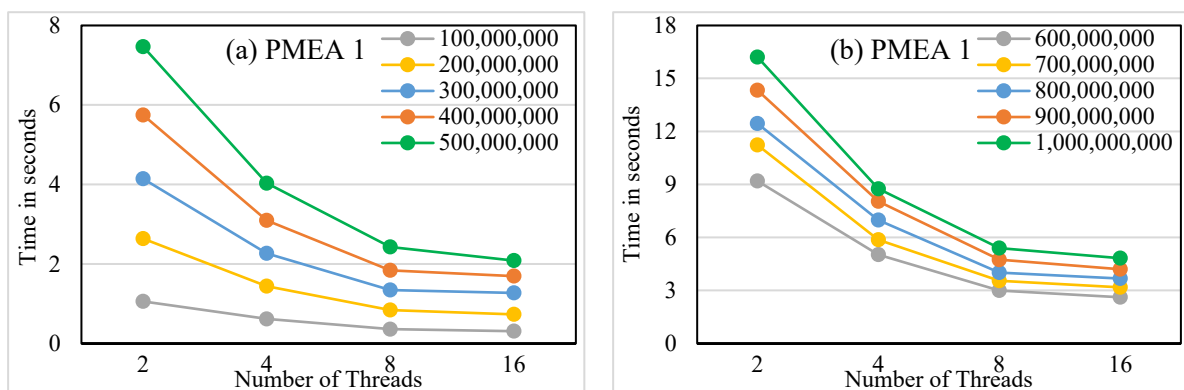


Figure 3. Run times for the PMAE-1 algorithm using different threads for different data n . (a) $n = 10^8, 2 \times 10^8, 3 \times 10^8, 4 \times 10^8, 5 \times 10^8$. (b) $n = 6 \times 10^8, 7 \times 10^8, 8 \times 10^8, 9 \times 10^8, 10^9$.

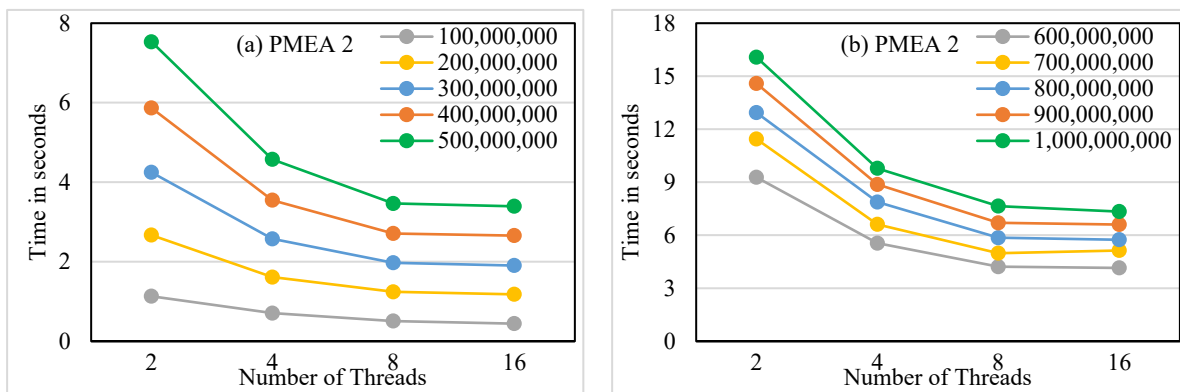


Figure 4. Run times for the PMAE-2 algorithm using different threads for different data n. (a) $n = 10^8, 2 \times 10^8, 3 \times 10^8, 4 \times 10^8,$ and 5×10^8 . (b) $n = 6 \times 10^8, 7 \times 10^8, 8 \times 10^8, 9 \times 10^8,$ and 10^9 .

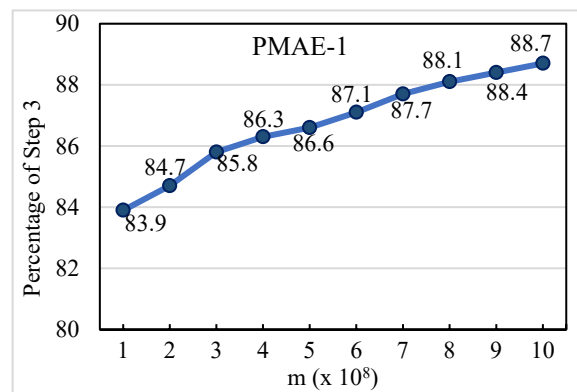


Figure 5. Percentage of Step 3 compared with the total time of the PMAE-1 algorithm.

Table 2. Percentage of improvement for the proposed algorithms.

Threads	Percentage of Improvement (on Average)		
	PMAE-1 vs. MAE	PMAE-2 vs. MAE	PMAE-1 vs. PMAE-2
2	47.5%	46.4%	2.1%
4	71.7%	68.2%	11.2%
8	83.5%	76.3%	30.4%
16	85.7%	78.0%	34.9%
Average	72.1%	67.2%	19.7

5.3. Speedup of Parallel Algorithms

Speedup is a criterion used to measure how frequently a parallel algorithm solves the same problem faster than its sequential counterpart. The speedup of the proposed parallel algorithm is the ratio of the run time of the MAE algorithm to the time of the parallel algorithm, PMAE-1 or PMAE-2.

Table 3 shows the speedup of the PMAE-1 and PMAE-2 algorithms using different numbers of threads: $t = 2, 4, 8$ and 16. The results indicate the following observations.

1. The speedup of the PMAE-1 algorithm is better than that of the PMAE-2 algorithm, except for $t = 2$. The reasons for this performance are that (i) the mechanism of dynamic parallelization for looping is better than that of the static mechanism, in particular when the inner loop is not fixed, and (ii) the mechanism of parallelizing two nested loops, where the outer loop is parallel and the inner is sequential, has better performance than that of when the outer loop is sequential and the inner is parallel.

- Increasing the number of threads affects the performance of the PMAE-1 algorithm, whereas increasing the number of threads from 8 to 16 slightly affects the performance of the PMAE-2 algorithm.

Table 3. Speedup of the two parallel algorithms PMAE-1 and PMAE-2.

Threads	Speedup	
	PMAE-1	PMAE-2
2	1.9	1.9
4	3.5	3.1
8	6.1	4.2
16	7.1	4.5

6. Conclusions and Future Work

Generating prime numbers is important in designing some cryptosystems, such as in [13,33]. In this paper, generating prime numbers up to an integer number m on two platforms, sequential and parallel, is addressed. The developed works for prime sieving are based on set theory. The first proposed sequential algorithm improves the run time of the best known algorithm by 98%. Moreover, two proposed parallel algorithms based on two strategies, static and dynamic, are introduced. The two parallel algorithms surpass the sequential algorithm, with improvements of 72% and 67% on average. Additionally, the maximum speedup achieved by the best parallel algorithm using 16 threads is 7.

Based on the discussion of the results, there are some open questions regarding future works. First, how can a technique that is more efficient than Step 3 of the proposed parallel algorithms be developed? Second, what is the effect of increasing the number of threads and data size on the proposed parallel algorithms? Third, since the performance of PMAE-1 increases with increasing the number of threads, and since recent graphic processing units (GPUs) have thousands of threads, how can the PMAE-1 algorithm be implemented on a single GPU or multi-GPUs?

Author Contributions: Conceptualization, H.M.B. (Hazem M. Bahig) and M.A.G.H.; methodology, H.M.B. (Hazem M. Bahig); software, H.M.B. (Hazem M. Bahig) and M.A.G.H.; validation, H.M.B. (Hazem M. Bahig) and H.M.B. (Hatem M. Bahig); formal analysis, H.M.B. (Hazem M. Bahig), M.A.G.H. and H.M.B. (Hatem M. Bahig); investigation, K.A.-U., D.I.N. and H.M.B. (Hatem M. Bahig); resources, H.M.B. (Hazem M. Bahig), M.A.G.H., K.A.-U. and D.I.N.; data curation, H.M.B. (Hazem M. Bahig), M.A.G.H., K.A.-U., D.I.N. and H.M.B. (Hatem M. Bahig); writing—original draft preparation, H.M.B. (Hazem M. Bahig), M.A.G.H. and H.M.B. (Hatem M. Bahig); writing—review and editing, H.M.B. (Hazem M. Bahig), M.A.G.H., K.A.-U., D.I.N. and H.M.B. (Hatem M. Bahig); visualization, H.M.B. (Hazem M. Bahig), M.A.G.H., K.A.-U. and D.I.N.; supervision, H.M.B. (Hazem M. Bahig); project administration, H.M.B. (Hazem M. Bahig); funding acquisition, H.M.B. (Hazem M. Bahig). All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Scientific Research Deanship at the University of Ha'il, Saudi Arabia, through project number RG-21 124.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to acknowledge the support provided by the Scientific Research Deanship at the University of Ha'il, Saudi Arabia, through project number RG-21 124. The authors also would like to thank the referees for their valuable comments.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Elnabya, A.A.; El-Baz, A.H. A new explicit algorithmic method for generating the prime numbers in order. *Egypt. Inf. J.* **2021**, *22*, 101–104.
- Paillard, G.; Franca, F.M.; Lavault, C. A distributed wheel sieve algorithm. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 619–626.

3. Aiazzi, B.; Baronti, S.; Santurri, L.; Selva, M. An Investigation on the prime and twin prime number functions by periodical binary sequences and symmetrical runs in a modified sieve procedure. *Symmetry* **2019**, *11*, 775. [[CrossRef](#)]
4. Agrawal, M.; Kayal, N.; Saxena, N. Primes is in p. *Ann. Math.* **2004**, *160*, 781–793. [[CrossRef](#)]
5. Ishmukhametov, S.T.; Mubarakov, B.G.; Rubtsova, R.G. On the number of witnesses in the Miller-Rabin primality test. *Symmetry* **2020**, *12*, 890. [[CrossRef](#)]
6. Fathy, K.; Bahig, H.; Farag, M. Speeding up multi-exponentiation algorithm on a multicore system. *J. Egypt. Math. Soc.* **2018**, *26*, 235–244. [[CrossRef](#)]
7. Bahig, H.M. A fast optimal parallel algorithm for a short addition chain. *J. Supercomput.* **2018**, *74*, 324–333. [[CrossRef](#)]
8. Bahig, H.; Kotb, Y. An efficient multicore algorithm for minimal length addition chains. *Computers* **2019**, *8*, 23. [[CrossRef](#)]
9. Tahir, R.R.; Asbullah, M.A.; Ariffin, M.R.; Mahad, Z. Determination of a good indicator for estimated prime factor and its modification in Fermat’s factoring algorithm. *Symmetry* **2021**, *13*, 735. [[CrossRef](#)]
10. Bahig, H.M.; Nassr, D.I.; Mahdi, M.A.; Bahig, H.M. Small private exponent attacks on RSA using continued fractions and multicore systems. *Symmetry* **2022**, *14*, 1897. [[CrossRef](#)]
11. Bahig, H.M.; Mahdi, M.A.; Alutaib, K.A.; AlGhadhban, A.; Bahig, H.M. Performance analysis of Fermat factorization algorithms. *Int. J. Adv. Comput. Sci. Appl.* **2020**, *11*, 340–352. [[CrossRef](#)]
12. Somsuk, K. An Efficient variant of Pollard’s p-1 for the case that all prime factors of the p-1 in B-Smooth. *Symmetry* **2022**, *14*, 312. [[CrossRef](#)]
13. Rivest, R.L.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* **1978**, *21*, 120–126. [[CrossRef](#)]
14. Diffie, W.; Hellman, M. New directions in cryptography. *IEEE Trans. Inf. Theory* **1976**, *22*, 644–654. [[CrossRef](#)]
15. Elgamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory* **1985**, *31*, 469–472. [[CrossRef](#)]
16. Jo, H.; Park, H. Fast prime number generation algorithms on smart mobile devices. *Cluster Comput.* **2017**, *20*, 2167–2175. [[CrossRef](#)]
17. Cayrel, P.L.; Alaoui, S.M.E.Y.; Hoffmann, G.; Véron, P. An improved threshold ring signature scheme based on error correcting codes. In Proceedings of the International Workshop on the Arithmetic of Finite Fields, Bochum, Germany, 16–19 July 2021; pp. 45–63.
18. Mairson, H.G. Some new upper bounds on the generation of prime numbers. *Commun. ACM* **1977**, *20*, 664–669. [[CrossRef](#)]
19. Gries, D.; Misra, J. A linear sieve algorithm for finding prime numbers. *Commun. ACM* **2014**, *21*, 999–1003. [[CrossRef](#)]
20. Pritchard, P. Linear prime-number sieves: A family tree. *Sci. Comput. Progr.* **2005**, *9*, 17–35. [[CrossRef](#)]
21. Luo, X. A practical sieve algorithm finding prime numbers. *Commun. ACM* **2010**, *32*, 344–346. [[CrossRef](#)]
22. Pritchard, P. A sublinear additive sieve for finding prime number. *Commun. ACM* **2007**, *24*, 18–23. [[CrossRef](#)]
23. Dunten, B.; Jones, J.; Sorenson, J. A space-efficient fast prime number sieve. *Inf. Process. Lett.* **2010**, *59*, 79–84. [[CrossRef](#)]
24. Pritchard, P. Fast compact prime number sieves (among others). *J. Algorithms* **2000**, *4*, 332–344. [[CrossRef](#)]
25. Bays, C.; Hudson, R.H. The segmented sieve of eratosthenes and primes in arithmetic progressions to 10^{12} . *BIT* **1999**, *17*, 121–127. [[CrossRef](#)]
26. Helfgott, H.A. An improved sieve of eratosthenes. *Math. Comput.* **2020**, *89*, 333–350. [[CrossRef](#)]
27. Wainwright, R.L. Parallel sieve algorithms on a hypercube multiprocessor. In Proceedings of the 17th conference on ACM Annual Computer Science Conference, New York, NY, USA, 21–23 February 2009; pp. 232–238.
28. Sorenson, J.; Parberry, I. 2 fast parallel prime number sieves. *Inf. Comput.* **2014**, *114*, 115–130. [[CrossRef](#)]
29. Hwang, S.; Chung, K.; Kim, D. Load balanced parallel prime number generator with sieve of eratosthenes on cluster computers. In Proceedings of the 7th IEEE International Conference on Computer and Information Technology, Aizu-Wakamatsu, Japan, 16–19 October 2007; pp. 295–299.
30. Paillard, G.; Lavault, C.; Franca, F.A. Distributed prime sieving algorithm based on scheduling by multiple edge reversal. In Proceedings of the 4th International Symposium on Parallel and Distributed Computing, Villeneuve-d’Ascq, France, 4–6 July 2005; pp. 139–146.
31. Blleloch, G.E. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*; Reif, J.H., Ed.; Morgan Kaufmann: San Francisco, CA, USA, 1993; pp. 35–60.
32. Bahig, H.M.; Fathy, K.A. An efficient parallel strategy for high-cost prefix operation. *J. Supercomput.* **2021**, *77*, 5267–5288. [[CrossRef](#)]
33. Li, C.; Dong, M.; Li, J.; Xu, G.; Chen, X.; Liu, W.; Ota, K. Efficient medical big data management with keyword-searchable encryption in healthchain. *IEEE Syst. J.* **2022**; *Early Access*. [[CrossRef](#)]