

Article

Enhanced Integer Factorization Method: Sequential and Parallel Approaches

Ehab T. Alnfwawy ^{1,2} , Hazem M. Bahig ³ , Hatem M. Bahig ^{4,*}  and Reda Elbarougy ^{5,6} 

¹ Department of Information Security, College of Computer Science and Engineering, University of Ha'il, Ha'il 81481, Saudi Arabia; eha.alnfwawy@uoh.edu.sa

² Department of Computer Science and Information Systems, Sadat Academy, Maadi, Cairo 2222, Egypt

³ Department of Information and Computer Science, College of Computer Science and Engineering, University of Ha'il, Ha'il 81481, Saudi Arabia; h.bahig@uoh.edu.sa

⁴ College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh 13318, Saudi Arabia

⁵ Department of Artificial Intelligence and Data Science, College of Computer Science and Engineering, University of Ha'il, Ha'il 81481, Saudi Arabia

⁶ Department of Information Technology, Faculty of Computer and Artificial Intelligence, Damietta University, New Damietta 34517, Egypt

* Correspondence: hmibrahim@imamu.edu.sa

Abstract

Integer factorization plays a foundational role in asymmetric cryptography systems, notably the Rivest, Shamir, and Adleman (RSA) cryptosystem. This paper presents an improvement of the integer factorization from both sequential and parallel computational perspectives. The algorithm is based on polynomial evaluation and the greatest common divisor. The objectives of these improvements are to decrease the execution time and memory consumption associated with the process of finding prime factors. Experimentally, we use different values of parameters (1) the number of bits n , (2) the difference between two factors, and (3) the number of processors in the parallel model. The experimental results indicate that both proposed methods, sequential and parallel, yield significant improvements regarding running time and memory usage when the difference between the two factors is $n^{1/3}$ and $n^{1/4}$. The average improvement observed is 99% in running time, with memory consumption reduced to a constant. This characteristic is important for limited hardware devices. Furthermore, the proposed parallel method demonstrates scalability and achieves sublinear speedup.

Keywords: cryptography; integer factorization; Strassen's algorithm; parallel algorithm



Academic Editors: Simon Gluzman, Calogero Vetro and Theodore E. Simos

Received: 30 March 2026

Revised: 23 April 2026

Accepted: 27 April 2026

Published: 2 May 2026

Copyright: © 2026 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 \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Integer factorization (IF) is the problem of decomposing a composite number into a product of smaller integers, ideally primes. More formally, given a positive integer N , the goal is to find its prime factors $p_1, p_2, \dots, p_\alpha$ such that $N = p_1^{e_1} p_2^{e_2} \dots p_\alpha^{e_\alpha}$, where $e_1, e_2, \dots, e_\alpha$ are positive integers. The IF is a hard problem on a classical computer, particularly when N is the product of two large prime numbers [1], i.e., there is no polynomial algorithm for IF until now.

The problem plays a central role in the design and security of several public-key asymmetric cryptosystems, such as the Rivest, Shamir, and Adleman (RSA) system [2]. RSA encryption uses modular exponentiation with the public key (N, e) . On the other hand, decryption requires knowledge of the modular inverse using the private key d , which

is derived from Euler's totient function $\Phi(N) = (p-1)(q-1)$. Since computing $\Phi(N)$ without factoring N is infeasible, RSA remains secure as long as integer factorization is computationally difficult [3]. Finding the private key helps to factorize the modulus N [4].

Over the decades, various algorithms have been developed to solve the IF problem based on different approaches, including (1) deterministic or probabilistic methods, (2) general or specialized methods, and (3) sequential or parallel methods. The trial division method is based on testing all odd integers starting from 2 up to $\sqrt{N}$ to check if they divide the given integer N [5]. This method was optimized by constructing a “wheel” that skips multiples of the first few primes, known as the wheel factorization method [6,7]. Fermat's method is based on the fact that any composite positive integer can be represented as the difference between two squares, $N = x^2 - y^2$. Therefore, the algorithm starts with $x = \lceil \sqrt{N} \rceil$ and tries to find a factor if $x^2 - N$ is a perfect square; otherwise, it increases x by 1 [8].

The Strassen factorization (SF) method is based on utilizing batch greatest common divisor (GCD) computations and fast polynomial arithmetic to detect prime divisors of N [9]. Pollard's Rho method is a probabilistic method useful for factoring numbers with small prime divisors [10,11]. Pollard's $p-1$ method [12] finds the prime factor p if $p-1$ has only small prime divisors. The elliptic curve (EC) method generalizes Pollard's $p-1$ algorithm by operating on elliptic curves [13,14]. In the EC method, randomly chosen elliptic curves and points are used, scalar multiplications are performed, and GCD is used to extract factors.

The Quadratic sieve (QS) method is based on the idea of finding two integers x and y such that $x^2 \equiv y^2 \pmod{N}$. This congruence leads to a nontrivial factor of N [15,16]. The general number field sieve (GNFS) extends the idea of the QS method by working in algebraic number fields rather than just integers [17]. Lehman's factorization method combines trial division and a square-checking strategy inspired by Fermat's method [18]. The generalization of Lehman's method is based on finding a square identity $x^2 - kN = y^2$ while optimizing the way k is chosen and incorporating early abort techniques, preprocessing by sieving, and modular arithmetic filters to skip bad candidates.

Another strategy for IF is based on mapping the arithmetic structure of integer multiplication into manageable Boolean constraints and demonstrating that most of those equations admit compact ordered binary decision diagram representations, potentially enabling a symbolic approach to factorization [19].

In this paper, we propose an improved integer factorization algorithm based on a polynomial evaluation function and GCD. Also, the algorithm depends on the backward method for searching the prime factors. The goal of the proposed algorithm is to reduce the memory complexity of the SF algorithm to be suitable for limited hardware devices. Furthermore, the proposed algorithm can factor a number $N = pq$ such that $N^{1/4} \leq \text{bit size of } |p - q| \leq N^{1/3}$. The proposed algorithm was compared with the SF method.

The idea of the proposed algorithm is inspired by the SF algorithm without using the subproduct-tree and multipoints evaluation. The SF algorithm has several advantages as follows.

- (1) It is a deterministic algorithm.
- (2) Strassen's method was the first algorithm to break the exponential time barrier for deterministic integer factorization, achieving a sub-exponential run time of approximately $\exp((1 + o(1))\sqrt{\log N \log \log N})$, significantly outperforming the naive $O(\sqrt{N})$ trial division.
- (3) Unlike heuristic or randomized methods (e.g., EC, GNFS), Strassen's method offers provable worst-case runtime guarantees, which is valuable in theoretical cryptographic settings or formal analyses.

(4) Strassen's approach inspired subsequent theoretical improvements. Hittmeir [20] introduced a babystep–giantstep method for deterministic integer factorization with running time roughly

$$\tilde{O}\left(N^{1/4} \exp\left(-C \frac{\log N}{\log \log N}\right)\right),$$

giving the first major improvement over the long-standing Strassen/Pollard–Strassen type [21] bound by a superpolynomial factor. Hittmeir [9] developed a time–space tradeoff for Lehman's deterministic factorization method and obtained a runtime of

$$O\left(N^{2/9+o(1)}\right).$$

This was the first exponential improvement over the classical deterministic bound. Harvey [22] improved the general deterministic factorization bound to

$$N^{1/5+o(1)},$$

surpassing Hittmeir's $N^{2/9+o(1)}$ result and establishing the best general-purpose deterministic complexity known in this line. Harvey and Hittmeir [23] improved the previous result to

$$O\left(\frac{N^{1/5} \log^{16/5} N}{(\log \log N)^{3/5}}\right),$$

by improving the secondary factors while keeping the main exponent equal to $1/5$.

On the other hand, the impact of altering the difference between two factors on the running time has not been investigated experimentally for the SF algorithm. Furthermore, to the best of our knowledge, no research papers have examined the use of a high-performance system to solve the SF algorithm.

The paper includes this introduction and four sections. Section 2 outlines the basic concepts related to the SF algorithm. Section 3 explains, in details, the steps of the proposed algorithm through various approaches. Section 4 discusses the experimental studies that compare the proposed algorithms with the SF algorithm. Finally, Section 5 presents the conclusions and explores open questions related to the proposed solutions.

2. The SF Algorithm

In this section, we provide a brief overview of the SF algorithm that is used as an idea behind the proposed algorithm, with some modifications. Assume that $N = pq$, where p and q are two primes such that $p \neq q$ with $\Delta = q - p$.

The SF algorithm, introduced by Volker Strassen (1977) [24], is a deterministic method for factoring composite integers N that leverages batch GCD computations and fast polynomial arithmetic to detect prime divisors of N [25]. Unlike trial division, which tests each potential factor individually, the SF algorithm encodes a large number of candidates simultaneously into a single polynomial, significantly reducing computational complexity.

The main steps of the SF algorithm are given in Algorithm 1. Initially, the SF algorithm starts by generating two sets B and S , such that (1) the two sets are disjoint modulo N , and (2) the two sets are not disjoint modulo p or q . Hittmeir [9] constructed the two sets as follows:

$$B = \{b_i = -i : 0 \leq i \leq d - 1\}, \quad S = \{s_i = d \cdot i : 1 \leq i \leq d\}$$

where

- $d = \lceil \Delta^{1/2} \rceil$,
- $\Delta = \lceil (N/(\eta + 1))^{1/2} \rceil$,

- $\eta = \lceil N^{2/9} / \log^{1/3} N \rceil$.

The SF algorithm consists of two main phases. In the first phase (lines 6–7 of Algorithm 1), the algorithm evaluates the elements of S using a polynomial function modulo N :

$$f(x) = \prod_{j=1}^{|B|} (x - b_j) \bmod N$$

where $|B|$ is the cardinality of B . The results of computing the polynomial function on the elements of S are stored in the auxiliary array Y , where $y_i = f(s_i)$. This phase can be computed using the subproduct-tree algorithm to calculate f and implement the multipoint-evaluation technique to calculate $f(s_i)$.

In the second phase (lines 8–24), the algorithm computes the GCD of y_i and N , say g_i . Then g_i is a factor of N if $1 < g_i < N$. If $g_i = N$, this means that there exists a nontrivial factor of N , and it can be determined when the value of the GCD for $s_i - b_j$ and N is greater than 1, $\forall 1 \leq j \leq |B|$. In this case, the algorithm terminates, and there is no need to consider a new iteration for the main loop at line 8.

Algorithm 1 Strassen Factorization (SF) Algorithm

Require: A composite number N of size n bits.

Ensure: Two factors p and q of N , or no factor found.

```

1:  $\eta = \lceil N^{2/9} / \log^{1/3} N \rceil$ 
2:  $\Delta = \lceil (N/(\eta + 1))^{1/2} \rceil$ 
3:  $d = \lceil \Delta^{1/2} \rceil$ 
4:  $B = \{-i : 0 \leq i \leq d - 1\}$ 
5:  $S = \{s_i = d \cdot i : 1 \leq i \leq d\}$ 
6:  $f(x) = \prod_{j=1}^{|B|} (x - b_j) \bmod N$ 
7: Compute  $y_i = f(s_i)$ ,  $\forall 1 \leq i \leq d$ 
8: for  $i = 1$  to  $d$  do
9:    $g_i = \text{GCD}(y_i, N)$ 
10:  if  $1 < g_i < N$  then
11:     $p = g_i$ 
12:     $q = N/p$ 
13:    return  $p, q$ 
14:  else if  $g_i = N$  then
15:    for  $k = 1$  to  $d$  do
16:       $g = \text{GCD}(s_i - b_k, N)$ 
17:      if  $g > 1$  then
18:         $p = g$ 
19:         $q = N/p$ 
20:        return  $p, q$ 
21:      end if
22:    end for
23:  end if
24: end for
25: return No factor found
  
```

The time complexity of the SF algorithm is $O(M_{\text{int}}(d \log N) \log N)$, where $M_{\text{int}}(k)$ is the cost of multiplying two polynomials of degree less than k in $\mathbb{Z}_N^*$ [25]. This time is equal to $O(d \log^2 d \log N \log \log N)$ complexity bits. The memory complexity of the SF algorithm is $O(d \log d + 4d)$, where the term $d \log d$ is the memory required by the subproduct tree algorithm used to compute the function f . In contrast, the term $4d$ is the memory required by the four auxiliary arrays B, S, Y , and G , each of size d .

3. The Proposed Solutions

In this section, we provide a detailed description to propose an improved algorithm for integer factorization from various perspectives, including memory and time consumption. This section includes three subsections. In Section 3.1, we provide five comments on the computational time and memory usage of the SF algorithm. In Section 3.2, we propose solutions for the comments on the SF algorithm. In Section 3.3, we measure the complexity analysis of the proposed algorithms.

3.1. Comments on the SF

By analyzing the SF algorithm from the perspectives of memory consumption and computational time, we find the following five observations.

The first observation is that the memory consumption of the SF increases rapidly as n increases. The SF uses four auxiliary arrays B , S , Y , and G of size d . The memory consumption for each array is equal to $O(d)$. Figure 1 illustrates how memory consumption increases as the number of bits increases. In the proposed improved SF, we reduced the memory requirement from $O(d \log d + 4d)$ to a constant, $O(1)$.

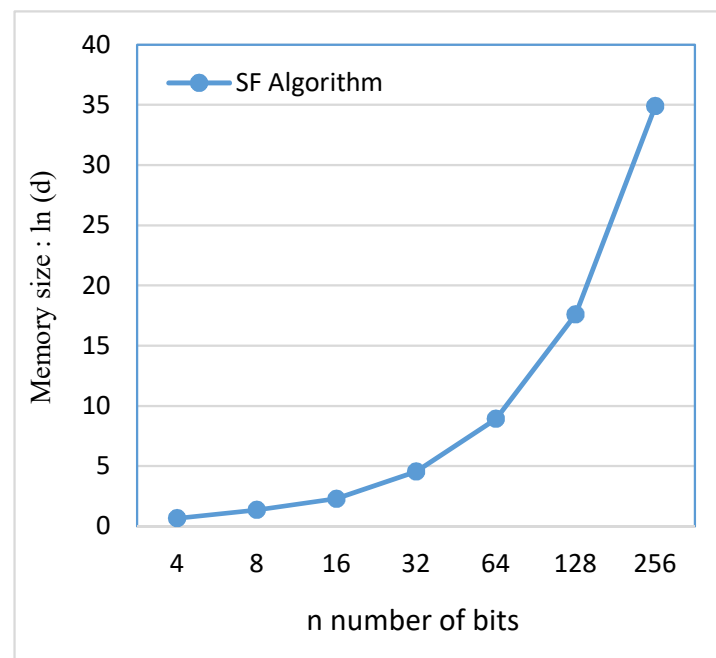


Figure 1. Expected memory consumption for the SF algorithm.

The second observation is that the elements of the auxiliary arrays G and Y are used only once, meaning they do not need to be retained before or after applying the GCD function. Additionally, we can generate the elements of the two arrays S and B without requiring precomputation or generation because the structure of these elements is simple and easy to use without being precomputed previously. We will use only two variables to store the results of applying the polynomial function and GCD. The values of the two variables will be updated at every iteration of the outer loop (line 8 in Algorithm 1).

The third observation is that, in some cases, the SF spends a significant amount of computational time in calculating the polynomial function for all elements of S , even though these calculations were not used in the testing. For example, if there exists an element s_i at the beginning of the array S that satisfies the condition $1 < \gcd(f(s_i), N) \leq N$, this scenario means that the time spent to calculate all the elements $s_j, i < j \leq d$, is useless.

Additionally, the running time for the first phase is significantly larger than the running time for the second phase. For example, assume that $N = 70,919,611,552,003$; the running time for the first phase is 2.44 s, while the running time for the second phase is 0.021 s to find the first factor, which is equal to 8,388,673. Therefore, instead of computing the polynomial function for all elements of S first and then testing one by one, we compute the polynomial function of s_i and then test the resultant. If there is a prime factor, then the algorithm terminates; otherwise, the algorithm takes another element of S and repeats the process.

The fourth observation is that, in the RSA cryptosystem, the modulus N is constructed as a product of two prime numbers p and q such that the size of each prime factor is $n/2$. This means that finding a small factor has no practical significance in real applications. Therefore, the probability of the existence of the solution starting from the back of the array S is higher than searching from the beginning. This results in a shift from testing the elements of array S in a forward manner to employing a backward strategy.

The fifth observation is that applying the polynomial function f on the elements of S and then computing the gcd of the resultant for each iteration can be done independently. This means that if we have a parallel model that consists of t threads, we can test t elements concurrently. This leads to a significant improvement in running time due to the use of parallel computing. If the parallel model includes a large number of threads, we can also use the parallelism when computing the two functions f and GCD. Moreover, we can parallelize the inner loop of Algorithm 1 at line 15.

3.2. The Algorithms

In this subsection, we explain how to address the five observations related to the SF algorithm for factoring N . For the first four observations, we propose a sequential improvement algorithm based on polynomial evaluation function and GCD, while the last observation represents the proposed parallel algorithm.

Algorithm 2 presents the pseudocode of an improved algorithm (ISF). The algorithm begins by calculating the value of d using η and Δ , as outlined in lines 1, 2, and 3. In line 4, the algorithm begins its calculations using a backwards approach, starting from a large value of j (equal to d) and proceeding to the smallest value ($j = 1$). The backwards loop is used to overcome the fourth observation. Additionally, there is no need to construct the two sets B and S , as was done in lines 4 and 5 of Algorithm 1.

In line 5, the polynomial function f is called with the value $d \cdot j$, and it does not utilize the auxiliary arrays S and B , nor does it save the results in the auxiliary arrays Y and G . Line 6 saves the result of applying the GCD function in an auxiliary variable of size 1, y , and this variable will be updated in every iteration. This overcomes the second observation. Additionally, this step overcomes the third observation by reducing the time spent on non-tested elements.

Lines 7–11 are similar to the SF algorithm. In line 13, the algorithm uses the value $d \cdot j + k$ instead of using the two auxiliary arrays S and B . Also, the result of applying the GCD is saved in the auxiliary variable y , without saving it in the auxiliary array G , to test the possibility of finding the prime factor.

The element dj belongs to the set S . Therefore, the first step is to compute $\text{GCD}(f(dj), N)$. If the result of GCD is greater than 1 or less than N , the algorithm returns $\text{GCD}(f(dj), N)$, where $f(dj) = \prod_{k=0}^{d-1} (dj + k) \bmod N$ from the definition of f and the set B .

Otherwise, $\text{GCD}(f(dj), N) = N$. This means that $n|f(dj)$. That means every prime factor of n divides the product:

$$(dj)(dj+1)(dj+2) \cdots (dj+d-1)$$

Therefore, the algorithm computes $GCD(s_j - b_i) = GCD(dj + k)$, $0 \leq k < d$.

Note that the element $dj + k$ does not belong to the set B (non-positive integer). Hence, directly computing dj and $dj + k$ is not arbitrary, but is guaranteed by number theory to expose factors of N .

Algorithm 2 Improved Integer Factorization (ISF)

Require: A composite positive integer N of size n bits.

Ensure: Two factors p and q of N , or no factor found.

```

1:  $\eta = \lceil N^{2/9} / \log^{1/3} N \rceil$ 
2:  $\Delta = \lceil (N/(\eta + 1))^{1/2} \rceil$ 
3:  $d = \lceil \Delta^{1/2} \rceil$ 
4: for  $j = d$  downto 1 do
5:    $y = f(d \cdot j)$ 
6:    $g = GCD(y, N)$ 
7:   if  $1 < g < N$  then
8:      $p = g$ 
9:      $q = N/p$ 
10:    return  $p, q$ 
11:   else if  $g = N$  then
12:     for  $k = 1$  to  $d$  do
13:        $g = GCD(d \cdot j + k, N)$ 
14:       if  $g > 1$  then
15:          $p = g$ 
16:          $q = N/p$ 
17:         return  $p, q$ 
18:       end if
19:     end for
20:   end if
21: end for
22: return "No factor found"

```

From the fifth observation, we propose two parallel algorithms for the ISF algorithm. The first proposed parallel algorithm is based on parallelizing the outer loop of the ISF only; i.e., one level of parallelization. The second proposed parallel algorithm is based on multi-level parallelism. The first level of parallelism is the outer loop, while the second level uses parallelism in the body of the outer loop, including parallelizing the two functions f and GCD , and the inner loop (line 13 in the ISF algorithm).

Algorithm 3 describes the first direction of parallelizing the ISF algorithm, denoted by PISF. In this algorithm, t threads work concurrently to parallelize the outer loop by assigning one iteration to each thread. Initially, thread i is assigned to the iteration $j = d - i + 1$, as in line 6. Then each thread works sequentially, as in the ISF algorithm, lines 8–23. If no prime factor is found at the current iteration, thread i updates the value of j to $j + t$, as in line 24, and repeats the same process, as in line 7, until one of the threads finds the prime factors ($\text{found} = \text{true}$) or the outer loop terminates and the integer N is prime.

The second direction of parallelizing the ISF algorithm is based on parallelizing the following additional parts:

- (1) Parallelizing the function f using parallel prefix-product and parallel multiplication algorithms [26–31].
- (2) Parallelizing the function GCD [32,33].
- (3) Parallelizing the inner loop at line 15 in Algorithm 3 using m threads.

Algorithm 3 Parallel Improved Integer Factorization (PISF)**Require:** A composite number N of size n bits, and t threads.**Ensure:** Two factors p and q of N , or no factor found.

```

1:  $\eta = \lceil N^{2/9} / \log^{1/3} N \rceil$ 
2:  $\Delta = \lceil (N/(\eta + 1))^{1/2} \rceil$ 
3:  $d = \lceil \Delta^{1/2} \rceil$ 
4: found = false
5: for  $i = 1$  to  $t$  do parallel
6:    $j = d - i + 1$ 
7:   while (found == false) and ( $j \geq 1$ ) do
8:      $y_i = f(d \cdot j)$ 
9:      $g_i = \text{GCD}(y_i, N)$ 
10:    if  $1 < g_i < N$  then
11:       $p = g_i$ 
12:       $q = N/p$ 
13:      found = true
14:    else if  $g_i = N$  then
15:      for  $k = 1$  to  $d$  do
16:         $g_i = \text{GCD}(d \cdot j + k, N)$ 
17:        if  $g_i > 1$  then
18:           $p = g_i$ 
19:           $q = N/p$ 
20:          found = true
21:        end if
22:      end for
23:    end if
24:     $j = j + t$ 
25:  end while
26: end for
27: if found == true then
28:   return  $p, q$ 
29: else
30:   return "No factor found"
31: end if

```

3.3. Complexity Analysis

In this subsection, we measure the complexity analysis of the proposed algorithms in terms of running time and memory consumption.

For Algorithm 2, the time complexity is bounded by $O(d(t_f + t_{\text{GCD}}))$, where d represents the total number of iterations for the outer loop (line 4) and t_f and t_{GCD} are the time complexities for the functions f and GCD, respectively. Assume that $M(\log N)$ is the cost of multiplying two integers, each of size $\log N$ bits. Therefore,

$$t_f = O(d M(\log N)), \quad t_{\text{GCD}} = O(M(\log N) \log \log N).$$

Note that in computing $f(s_i)$, we considered it as a multiplication of d integers, each of size $\log N$.

Since the time complexity of t_f is greater than t_{GCD} , the time complexity of the ISF algorithm is $O(d^2 M(\log N)) = O(d^2 \log N \log \log N)$.

In the best case, the ISF algorithm finds the prime factor after a few iterations from the outer loop, so the time complexity of the ISF algorithm is reduced to $\Omega(d M(\log N)) = \Omega(d \log N \log \log N)$. In contrast, the time complexity of SF in the best case is $\Omega(d M(d \log N)) = \Omega(d \log^2 d \log N \log \log N)$ complexity bit.

The memory consumption of Algorithm 2 is $O(1)$ because the algorithm uses a constant number of auxiliary variables during the computation.

For Algorithm 3, the time complexity is $O(\frac{d^2}{t} M(\log N))$, while the memory consumption is $O(t)$. The PISF algorithm reduces the time complexity and memory consumption of the SF and ISF algorithms.

4. Experimental Studies

In this section, we conduct an experimental investigation into the time and memory consumption of three algorithms: SF, ISF, and PISF. Additionally, we investigate the speedup and scalability of the proposed algorithm PISF as a function of increasing the number of threads. This section is divided into three subsections. The methodology for evaluating the experimental performance of the three algorithms—SF, ISF, and PISF—is covered in the first subsection. In the second subsection, we show how well the suggested sequential algorithm ISF performs in comparison to other algorithms. In the last subsection, we show how well the parallel algorithm PISF performs in comparison to the sequential algorithm.

4.1. Methodology

We have two directions for measuring the performance of the proposed algorithms.

The first direction is related to sequential computation, in which we compare the two algorithms SF and ISF based on running time and memory consumption. The percentage of improvement in the running time of the ISF algorithm is equal to $1 - (T_{ISF}/T_{SF})$.

The second direction is related to parallel computation, in which we evaluate the proposed parallel algorithm PISF using several criteria, including running time, speedup, efficiency, and scalability. The speedup of the parallel solution, S_p , is the ratio of the running time of the sequential algorithm, ISF, to the running time of the parallel solution, PISF. The speedup is linear if $S_p = p$, while the speedup is sublinear if $S_p < p$. The efficiency of the parallel algorithm is the ratio of its speedup to the number of processors.

For both directions, we use the C language to implement all algorithms as well as a library to manipulate large integers named GNU Multiple Precision (GMP) [34]. We also utilize an application programming interface to implement parallel regions in PISF, known as Open Multi-Processing (OpenMP) [35]. All programs run on a machine consisting of a processor with a speed of 2.5 GHz and work under the Linux OS. For a parallel program, the system can run 16 threads simultaneously. Additionally, the computer has 15 MB of cache memory and 32 GB of RAM.

For all experimental studies, we generate a test dataset that verifies the following properties:

1. The size of each prime factor is $n/2$ because in real applications, such as RSA, the modulus N of size n is a product of two primes of the same size. Additionally, if the size of a prime factor is small, the trial division and wheel methods are efficient strategies for finding it.
2. The difference between two prime factors is bounded by d .

Therefore, a random integer N of size n bits is constructed by generating a random prime number p of size $n/2$. Then we generate another prime number q of size $n/2$ such that the size of $p - q$ is ω , where $n/4 \leq \omega \leq d$. For fixed values of n and ω starting from $n/4$ to $n/3$, we generate 50 instances that verify the properties of the test dataset.

The primary metric employed to compare various algorithms is the running time. For fixed n and ω , the running time of each algorithm is measured by taking the average of the running time for 50 instances. For the first direction (sequential computation), we use only one thread for both algorithms: SF and ISF. For the second direction (parallelism), we run the PISF on the same data using different numbers of threads: 2, 4, 8, and 16. Additionally, in the second direction, we calculate other measurements such as (1) the speedup measurement, which is equal to the running time of ISF divided by the running

time of PISF, and (2) the efficiency measurement, which is equal to the running time of ISF divided by the product of the running time of PISF with the number of threads. The values of n are 32, 40, 48, 56, 64, 72, 80, and 88.

4.2. Performance of ISF

To measure the performance of the proposed algorithm ISF, we compare it with the SF algorithm. Table 1 shows the performance of the two algorithms SF and ISF in terms of the running time over different values of n and ω . The results indicate the following observations:

- There is a significant difference between the running time of SF and ISF algorithms for each n and ω ; see Columns 3 and 4. For example, when $n = 64$ bits and the difference between two prime factors is 16 bits, the running times for the SF and ISF algorithms are 0.521 and 11,358.32 s, respectively.
- The average percentage of improvement for the ISF algorithm compared to the SF algorithm is 99% for all studied cases, which is a significant improvement over the SF algorithm.
- For a fixed value of n less than 64 bits, the running time of the original algorithm SF is acceptable because it takes at most a few minutes. On the other hand, when $n \geq 64$ bits, the running time of SF increases rapidly, especially when ω increases.
- Figure 2 illustrates the execution time of the ISF algorithm for 50 instances when $\omega = 21$.
- For a fixed value of n less than 64 bits, the ISF running time is very fast.
- For $n > 64$ bits, the running time of SF is considerable (greater than one hour), so we neglected measuring it. The running time for ISF will be measured when measuring the performance of PISF for $n > 64$.

Table 1. Running time for the SF and ISF algorithms on different values of n and ω .

n	ω	T_{ISF} (s)	T_{SF} (s)	Improvement (%)
40	10	0.0012	0.046	97.48
	11	0.0013	0.047	97.20
	12	0.0015	0.047	96.72
	13	0.0019	0.072	97.36
48	12	0.003	2.29	99.87
	13	0.0039	2.311	99.82
	14	0.0093	2.323	99.60
	15	0.0095	2.363	99.59
	16	0.0135	2.510	99.46
56	14	0.0313	153.427	99.97
	15	0.0456	153.793	99.97
	16	0.0742	154.153	99.95
	17	0.1267	154.439	99.91
	18	0.1831	156.336	99.88
64	16	0.521	11,358.32	99.99
	17	0.749	11,398.05	99.99
	18	1.544	11,690.17	99.98
	19	2.254	18,391.55	99.98
	20	3.509	30,016.05	99.98
	21	6.522	57991.1	99.99

Note: All times are in seconds, averaged over 50 instances.

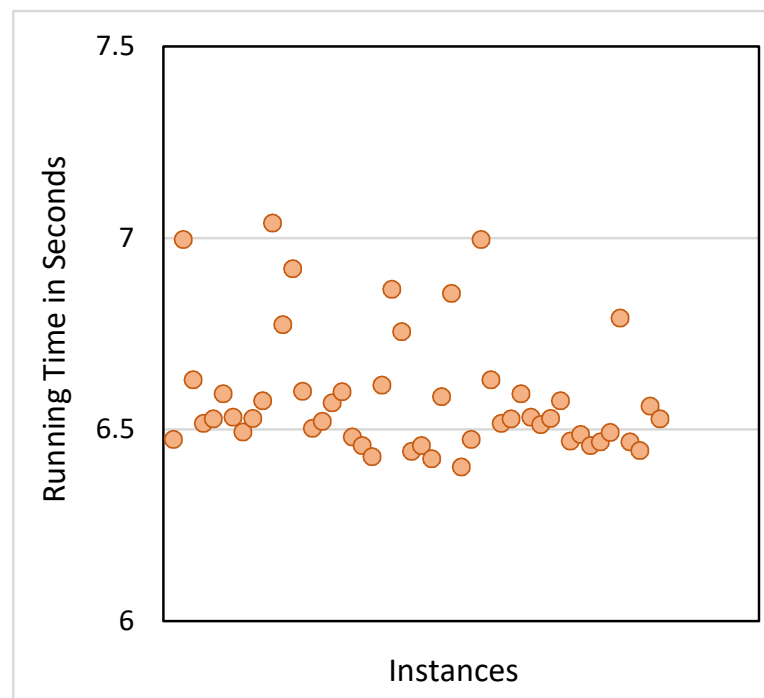


Figure 2. Running time of the ISF algorithm for 50 instances when $n = 64$ and $\omega = 21$.

4.3. Performance of PISF

To measure the performance of the PISF algorithm, we run it on the same data used in the previous section with different numbers of threads for values of n equal to 72, 80, and 88, since the running time of the ISF algorithm is small for values below this number of bits. Additionally, to measure the speedup of the PISF algorithm, we run the ISF algorithm on the same dataset using one thread. Figure 3 demonstrates the performance of the PISF algorithm compared to the ISF algorithm with varying numbers of threads. From Figure 3, we observe the following:

- Parallelism significantly reduces the running time of the ISF as the number of threads increases.
- The PISF is scalable, which means that increasing the number of threads reduces its running time. For example, when $n = 80$, the running time of ISF is 166.22 s, while the running times of PISF using 2, 4, 8, and 16 threads are 88.42, 48.89, 22.77, and 13.1 s, respectively.
- The average percentage of improvement in the running time for the ISF algorithm when we use two threads for PISF is approximately 44% for the studied cases. This percentage increases to 72% when we use four threads. Using eight and sixteen threads, the PISF improved the ISF with average percentages of 86% and 92%, respectively, for all studied cases.
- The average speedup of the PISF using 2, 4, 8, and 16 threads is 1.78, 3.64, 7.1, and 12.9, respectively. This means that the speedup of PISF is nearly linear.
- The average efficiency of the PISF using 2, 4, 8, and 16 threads is 0.89, 0.91, 0.88, and 0.81, respectively.

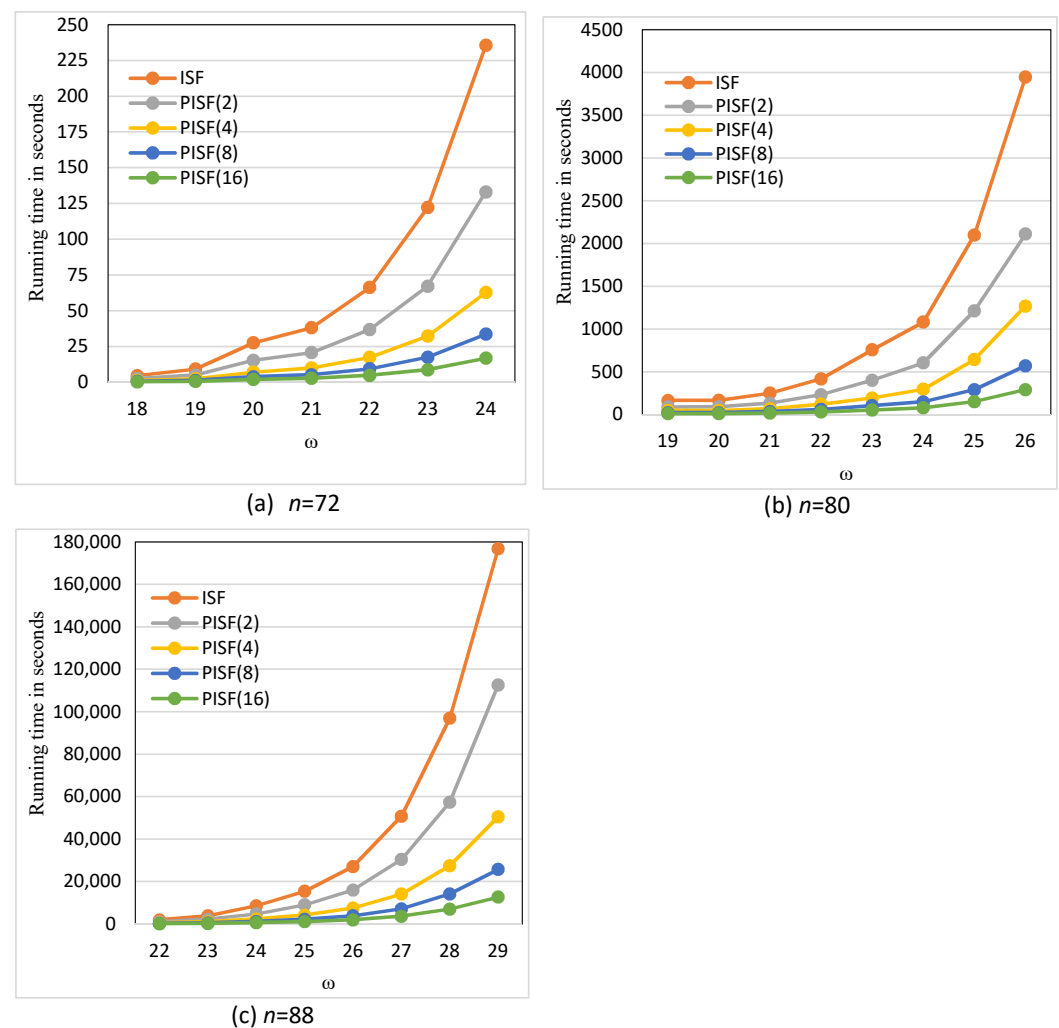


Figure 3. Running time for the ISF and PISF algorithms using different numbers of threads and ω .

5. Conclusions and Future Work

We have discussed how to factor a number into two prime factors using the polynomial evaluation function and the greatest common divisor. Also, the proposed algorithm uses backward strategy for finding the factors. In order to decrease memory usage and accelerate the running time, we proposed two algorithms, sequential and parallel. Furthermore, the experimental studies show that the suggested algorithms perform better than other algorithms in terms of memory usage and running time. Additionally, the parallel solution has sublinear speedup and is scalable.

The following are the method's future research directions: (1) How can sets S and B be made smaller? (2) How can the second level of parallelization of the proposed algorithm be implemented? (3) How can the proposed parallel algorithm be implemented using a graphics processing unit (GPU)?

Author Contributions: E.T.A.: Conceptualization, methodology, software, and writing—review and editing. H.M.B. (Hazem M. Bahig): Conceptualization, methodology, software, formal analysis, writing—original draft preparation, writing—review and editing, supervision, and project administration. H.M.B. (Hatem M. Bahig): Conceptualization, methodology, and writing—review and editing. R.E.: Conceptualization, methodology, and writing—review and editing. 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, grant number IRP-24 030.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Acknowledgments: The authors extend their thanks to the Scientific Research Deanship at the University of Ha'il—Saudi Arabia, through project number IRP-24 030. The authors would also like to thank the referees for their valuable comments that led to the enhancement of the paper.

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

Abbreviations

The following abbreviations are used in this manuscript:

EC	Elliptic curve
GCD	Greatest common divisor
GMP	GNU Multiple Precision
GNFS	General number field sieve
IF	Integer factorization
ISF	Improved Strassen-based factorization
OpenMP	Open Multi-Processing
PISF	Parallel improved Strassen-based factorization
QS	Quadratic sieve
RSA	Rivest, Shamir, and Adleman
SF	Strassen-based factorization

References

1. Roettger, E.L.F.; Williams, H.C. *The Enchantment of Numbers*; CMS/CAIMS Books in Mathematics; Springer: Berlin/Heidelberg, Germany, 2025. [\[CrossRef\]](#)
2. Rivest, R.; Shamir, A.; Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM.* **1978**, *21*, 120–126. [\[CrossRef\]](#)
3. Stallings, W. *Cryptography and Network Security: Principles and Practice*, 8th ed.; Pearson: London, UK, 2020.
4. 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. [\[CrossRef\]](#)
5. Lenstra, A. Integer factoring. *Des. Codes Cryptogr.* **2000**, *19*, 101–128. [\[CrossRef\]](#)
6. Bahig, H.M.; Nassr, D.I.; Mahdi, M.A.; Hazber, M.; Al-Utaibi, K.; Bahig, H.M. Speeding up wheel factoring method. *J. Supercomput.* **2022**, *78*, 15730–15748. [\[CrossRef\]](#)
7. Zaki, A.; Bakr, M.; Alsahangiti, A.; Khosa, S.; Fathy, K. Acceleration of wheel factoring techniques. *Mathematics* **2023**, *11*, 1203. [\[CrossRef\]](#)
8. Bahig, H.M.; Bahig, H.M.; Kotb, Y. Fermat factorization using a multi-core system. *Int. J. Adv. Comput. Sci. Appl.* **2020**, *11*, 323–330. [\[CrossRef\]](#)
9. Hittmeir, M. A time-space tradeoff for Lehman's deterministic integer factorization method. *Math. Comput.* **2021**, *90*, 1999–2010. [\[CrossRef\]](#)
10. Pollard, J. A Monte Carlo method for factorization. *BIT Numer Math.* **1975**, *15*, 331–334. [\[CrossRef\]](#)
11. Eric, B. Toward a theory of Pollard's rho method. *Inf. Comput.* **1991**, *90*, 139–155. [\[CrossRef\]](#)
12. Zralek, B. A deterministic version of Pollards $p - 1$ algorithm. *Math. Comput.* **2023**, *79*, 513–533. [\[CrossRef\]](#)
13. Zimmermann, P. Elliptic curve method for factoring. In *Encyclopedia of Cryptography and Security*; Van Tilborg, H.C.A., Jajodia, S., Eds.; Springer: Boston, MA, USA, 2011. [\[CrossRef\]](#)
14. Urroz, J.; Pomykala, J. Factoring numbers with elliptic curves. *Ramanujan J.* **2024**, *64*, 265–273. [\[CrossRef\]](#)
15. Guan, Z.; Zhuang, X.; Mastorakis, N. Optimization of quadratic sieve algorithm implementation for large integer factorization. *WSEAS Trans. Comput.* **2025**, *24*, 86–91. [\[CrossRef\]](#)
16. Carneiro, E.; Helfgott, H.; Mejía-Cordero, J. Optimality for the two-parameter quadratic sieve. *Acta Arith.* **2022**, *203*, 195–226. [\[CrossRef\]](#)
17. Wang, Q.; Fan, X.; Zang, H.; Wang, Y. The space complexity analysis in the general number field sieve integer factorization. *Theor. Comput. Sci.* **2016**, *630*, 76–94. [\[CrossRef\]](#)
18. Hales, J.; Hiary, G. A generalization of Lehman's method. *Ramanujan J.* **2024**, *65*, 1773–1790. [\[CrossRef\]](#)

19. Brown, D.; Skidmore, D. Representing the integer factorization problem using ordered binary decision diagrams. *Theory Comput. Syst.* **2023**, *67*, 1307–1332. [[CrossRef](#)]
20. Hittmeir, M. A Babystep-Giantstep method for faster deterministic integer factorization. *Math. Comput.* **2018**, *87*, 2915–2935. [[CrossRef](#)]
21. Pollard, J. Theorems on factorization and primality testing. *Proc. Camb. Philos. Soc.* **1974**, *76*, 521–528. [[CrossRef](#)]
22. Harvey, D. An exponent one-fifth algorithm for deterministic integer factorisation. *Math. Comput.* **2021**, *90*, 2937–2950. [[CrossRef](#)]
23. Harvey, D.; Hittmeir, M. A log-log speedup for exponent one-fifth deterministic integer factorisation. *Math. Comput.* **2022**, *91*, 1367–1379. [[CrossRef](#)]
24. Strassen, V. Einige Resultate über Berechnungskomplexität. *Jahresber. Der Dtsch. Math.-Ver.* **1977**, *78*, 1–8. Available online: <http://eudml.org/doc/146659> (accessed on 15 January 2026).
25. Gathen, J.; Gerhard, J. *Modern Computer Algebra*, 3rd ed.; Cambridge University Press: Cambridge, UK, 2013. [[CrossRef](#)]
26. Haque, S.; Mansouri, F.; Moreno Maza, M. On the parallelization of subproduct tree techniques targeting many-core architectures. In *Computer Algebra in Scientific Computing. CASC 2014*; Gerdt, V.P., Koepf, W., Seiler, W.M., Vorozhtsov, E.V., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2014; Volume 8660, pp. 171–185. [[CrossRef](#)]
27. Haque, A.; Li, S.; Mansouri, F.; Moreno Maza, M.; Mohajerani, D.; Pan, W. CUMODP: A CUDA library for modular polynomial computation. *ACM Commun. Comput. Algebra* **2018**, *51*, 89–91. [[CrossRef](#)]
28. Bahig, H.; Fathy, K. An efficient parallel strategy for high-cost prefix operation. *J. Supercomput.* **2021**, *77*, 5267–5288. [[CrossRef](#)]
29. Fan, H.; Hasan, M. A survey of some recent bit-parallel GF(2n) multipliers. *Finite Fields Their Appl.* **2015**, *32*, 5–43. [[CrossRef](#)]
30. Harvey, D.; Hoeven, J. Integer multiplication in time $O(n \log n)$. *Ann. Math.* **2021**, *193*, 563–617. [[CrossRef](#)]
31. Bahig, H.; Fathy, K. Sequential and parallel sliding window algorithms for multiplying large integers. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 131–140. [[CrossRef](#)] [[CrossRef](#)]
32. Sedjelmaci, S. Two fast parallel GCD algorithms of many integers. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*; ACM: New York, NY, USA, 2017; pp. 397–404. [[CrossRef](#)]
33. Pathirana, G.T.; Sotheeswaran, S.; Ratnarajah, N. Efficient parallel GCD algorithms for multicore shared memory architectures. In *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*, Colombo, Sri Lanka, 4–7 November 2020; IEEE: New York, NY, USA, 2020; pp. 272–273. [[CrossRef](#)]
34. GMP: The GNU Multiple Precision Arithmetic Library [Internet]. Available online: <https://gmplib.org/> (accessed on 15 January 2026).
35. OpenMP [Internet]. Available online: <https://www.openmp.org/> (accessed on 15 January 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.