


Research Article

Optimized Euler-Based Fermat Factorization Method Using Parallel Computing

Hazem M. Bahig^{1,2*}, Ibrahim M. Alseadoon¹, Mohamed A.G. Hazber¹, Hatem M. Bahig³, Dieaa I. Nassr²

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

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

³College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, 13318, Saudi Arabia
E-mail: h.bahig@uoh.edu.sa

Received: 3 July 2025; **Revised:** 31 August 2025; **Accepted:** 2 September 2025

Abstract: The Euler-based Fermat Factorization (EFF) method is a technique used to factor a positive odd integer n into two prime factors p and q . The method is based on representing n as the difference between two squares and replacing the perfect square operation with modular multiplication. The computational time for the EFF method increases significantly with large values of n and $d = |p - q| > \sqrt[4]{n}$. This paper presents a new mathematical formula for computing the EFF method in a parallel shared-memory model. Then, it proposes a new parallel algorithm to speed up the computation of factorization when n is large and d exceeds $\sqrt[4]{n}$. Experiments were conducted on a multicore system to evaluate the performance of the proposed method across various parameters, including the size of n , the difference d , and the number of threads in the parallel system. The results show that the proposed parallel EFF algorithm achieves performance enhancements of 77.3% compared to the original EFF algorithm and 47.7% relative to the best-known parallel algorithm. Furthermore, the parallel EFF algorithm demonstrates superior scalability compared to the best-known parallel algorithm.

Keywords: integer factorization, Fermat method, Euler theorem, number theory, parallel computing, cryptography

MSC: 11A51, 11T71, 68W10, 94A60

Abbreviation

FF	Fermat Factorization
GMP	GNU Multiple Precision
GPU	Graphics Processing Unit
IF	Integer Factorization
OpenMP	Open Multi-Processing
PFF	Parallel Fermat Factorization
PEFF	Parallel Euler-based Fermat Factorization

PS	Perfect Square
RDC	Rotating Disk Cryptosystem
RSA	Rivest-Shamir-Adleman

1. Introduction

Given a composite odd number n , the objective of Integer Factorization (IF) is to decompose n into two prime factors, p and q , such that $n = pq$. The problem is important in number theory and cryptography such as Rivest-Shamir-Adleman (RSA) and Rotating Disk Cryptosystem (RDC) [1, 2]. Many strategies have been proposed to address the IF problem on computer systems, such as trial division [3], wheel factorization [4, 5], Fermat factorization [6–8], Pollard’s algorithm [9, 10], number field sieving algorithm [11, 12], quantum annealing factorization [13], factorization using ordered binary decision diagrams [14], and factorization using lattices [15]. These methods differ based on various criteria: (1) whether the method is general-purpose or specialized, (2) whether it is deterministic or randomized, and (3) the type of computational model used, such as sequential, quantum, or parallel.

Fermat’s Factorization (FF) method is one of the proposed approaches for solving the IF problem and has several advantages: (1) it is deterministic, (2) it is exact, (3) it runs in polynomial time when the size of $p-q$ is less than or equal to the size of $\sqrt[4]{n}$, and (4) it can be implemented in both sequential and parallel models.

The FF method is based on the principle that any composite positive odd number, $n = pq$, can be expressed as the difference of the squares of two numbers, u and v , i.e., $n = u^2 - v^2 = (u + v)(u - v)$. If the values of u and v can be determined, the values of p and q are obtained using the following equation.

$$p = u + v \quad \text{and} \quad q = u - v \quad (1)$$

The FF method starts with $u = \lceil \sqrt{n} \rceil$, and then tests if $v^2 = u^2 - n$ is a Perfect Square (PS), then the two factors are given by Equation 1. Otherwise, the method increments u by 1. For example, assume that $n = 32,929$. The FF method starts with $u = \lceil \sqrt{32,929} \rceil = 182$. Then the algorithm computes $v^2 = u^2 - n = 182^2 - 32,929 = 195$ which is not a PS. In the next step, $u = 183$ and $v^2 = 183^2 - 32,929 = 560$, which is not a PS. In the next step, $u = 184$ and $v^2 = 184^2 - 32,929 = 927$, which is not a PS. In the next step, $u = 185$ and $v^2 = 185^2 - 32,929 = 1,296$, which is a PS and $v = 36$. Therefore, $p = 185 + 36 = 221$ and $q = 185 - 36 = 149$.

The FF method is very fast when the size of search space is small and satisfies that $d = |p - q| \leq n^{0.25}$. However, the FF algorithm’s execution time increases as the difference between the two prime factors grows larger. Finding the values of u and v using the FF algorithm was accelerated through five main approaches. The first approach involves eliminating certain elements from the search space to reduce the number of squaring operations for u and PS operation for v [6, 7, 16–18]. The primary advantage of this approach is the reduction in the number of PS operations, while its main drawback lies in the cost associated with each PS operation. The second approach involves either rewriting the Fermat formula or optimizing the code of the FF algorithm [19, 20]. Generally, the algorithms within this approach show limited improvements. The third approach entails replacing the principal operation of Fermat’s method, which involves perfect squares, with a lower-cost operation: modular multiplication [6]. The key advantage of this method is the decreased cost of the main operation; however, it still presents a substantial search space. The fourth approach leverages the concepts of parallel and distributed computing [8]. The principal benefit of this strategy is the reduction in the running time required to find the prime factors. Finally, the fifth approach depends on heuristic solutions for integer factorization [21, 22]. This method does not guarantee that the prime factors will be successfully identified.

The author in [6] proposed an efficient fermat method to factorize n using Euler’s theorem, where the euler theorem is defined by the following equation:

$$a^{\Phi(n)} \equiv 1 \mod n \quad (2)$$

where $\gcd(a, n) = 1$, and $\Phi(n) = n - (p + q) + 1$.

The Euler-based Fermat Factorization (EFF) algorithm, represented in Figure 1, begins by searching for the two prime factors with initial values $u = 2 \lceil \sqrt{n} \rceil$ and $t = a^{n-u+1} \bmod n$. The algorithm iteratively updates t until $t = 1$. The value of t is updated using modular multiplication, $s \bmod n$, and given by the equation:

$$t = t * s \bmod n \quad (3)$$

where $s = c^2 \bmod n$ and c is a positive integer such that $\gcd(c, n) = 1$.

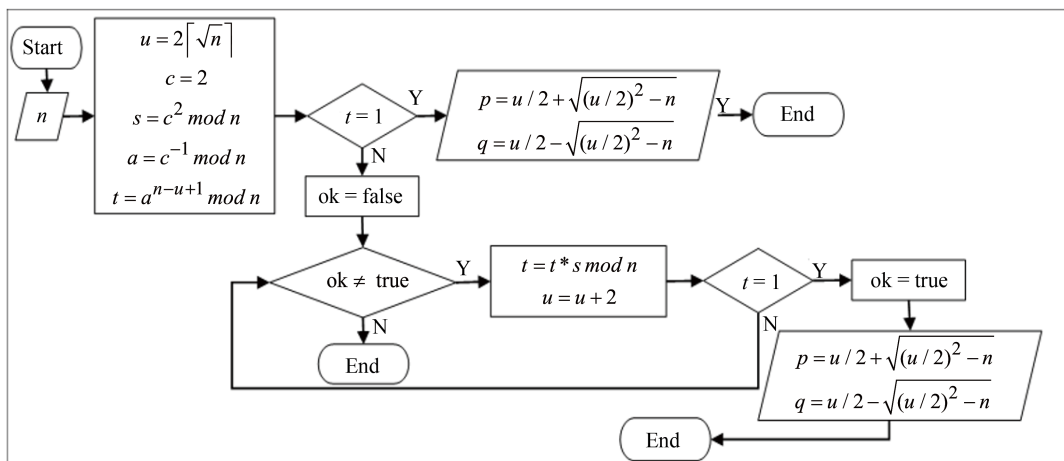


Figure 1. Flowchart of the EFF algorithm

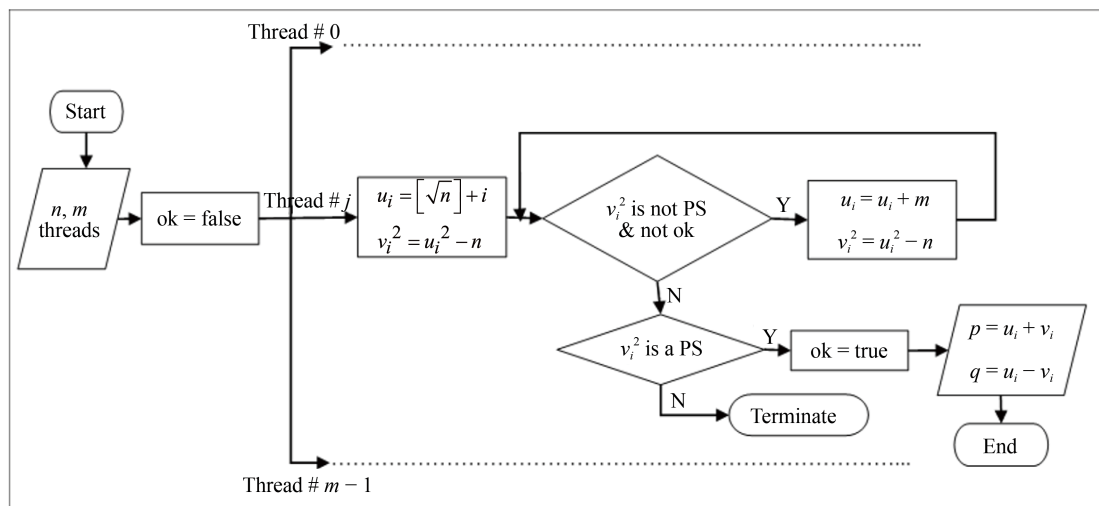


Figure 2. Flowchart of the PFF algorithm

The goal of this study is to propose a new mathematical formula for EFF method on a parallel shared model as well as design a new parallel algorithm to accelerate the execution time of factoring large n . Moreover, the proposed method is able to factor n such that $d > n^{0.25}$. Additionally, the performance of the developed parallel algorithm will be evaluated

by implementing it on a multicore system with 20 cores. Various metrics will be used to measure the effectiveness of the proposed parallel algorithm in comparison to previous works. The results show that the proposed parallel EFF algorithm achieves performance enhancements of 77.3% compared to the original EFF algorithm, represented in Figure 1, and 47.7% relative to the best-known Parallel Fermat Factorization (PFF) algorithm, represented in Figure 2. Furthermore, the parallel EFF algorithm demonstrates superior scalability compared to the best-known parallel algorithm.

The paper consists of an introduction and three sections, as follows: Section 2 presents the developed parallel algorithm, including the parallel formula for EFF and the complete pseudocode of the newly proposed algorithm. Section 3 demonstrates the performance of the proposed algorithm by comparing it to two existing algorithms: the sequential EFF algorithm and the parallel FF algorithm. Section 4 presents the conclusions of this study and discusses potential directions for future work.

2. The proposed solution

In this section, we discuss how to parallelize the EFF algorithm and provide full details of the proposed algorithm.

In the EFF algorithm, the new value of t is computed by multiplying the current value of t with s , i.e, Equation (2). This relation can be rewritten as a recurrence relation, as follows.

$$t_i = \begin{cases} a^{n-u+1} \mod n & i = 0 \\ t_{i-1}s \mod n & i \geq 1 \end{cases} \quad (4)$$

The challenge is how to compute m different values of t simultaneously, where m is the number of threads. To address this, the proposed algorithm computes the values of t in parallel way using two phases.

The first phase: Each thread computes the initial value of t that will be used to generate the next values of t . The equation for computing the initial value of t for each thread independently is described by the following theorem.

Theorem 1 Given the initial value of t , $t_0 = a^{n-u+1} \mod n$, the first m initial values of t for m threads are given by

$$t_i = t_0 s^i, \quad 1 \leq i \leq m. \quad (5)$$

Proof. We prove Equation (5) by induction on i .

At $i = 1$, we have $t_1 = t_0 s$ by Equation (4).

Assume that Equation (5) is true at $i = k$, i.e., $t_k = t_0 s^k$.

We will prove that Equation (5) is true at $i = k+1$, i.e., $t_{k+1} = t_0 s^{k+1}$.

From Equation (4), we have $t_{k+1} = t_k s = t_0 s^k s = t_0 s^{k+1}$.

Thus, Equation (5) holds.

Using this theorem, each thread, from the m threads, can independently compute the initial value of t and therefore, we have m values, t_1, t_2, \dots, t_m . Similarly, each thread can compute the initial value of u as follows.

$$u_i = u_0 + 2i \quad (6)$$

The second phase: This phase computes the next values of t_i for each thread i repeatedly, $1 \leq i \leq m$, until one of them finds $t_\alpha = 1$. Also, in this phase, we compute the next value of u_i . The following theorem shows how to compute the new value of t_i .

Theorem 2 Given the first m initial values of t , i.e., t_i , $1 \leq i \leq m$. The next value of t for each thread i , $1 \leq i \leq m$, is given by

$$t_{m+i} = t_i s^m, \quad 1 \leq i \leq m. \quad (7)$$

Proof. We prove Equation (7) by induction on i .

At $i = 1$, we have $t_{m+1} = t_m s$ by Equation (4).

$= t_0 s^m s$ by Equation (5).

$t_{m+1} = t_1 s^m$ which is true.

Assume that Equation (7) is true at $i = k$, i.e., $t_{m+k} = t_k s^m$.

We will prove that Equation 7 is true at $i = k+1$, i.e., $t_{m+k+1} = t_{k+1} s^m$.

$t_{m+k+1} = t_{m+k} s$ by Equation (4).

$= t_k s^m s$ from assumption ($i = k$).

$t_{m+k+1} = t_{k+1} s^m$ which is true.

Thus, Equation (7) holds.

From Theorem 2, the first iteration of the second phase generates m values of t : $t_{m+1}, t_{m+2}, \dots, t_{2m}$. This means that the thread number i generates two values t_i and t_{m+i} . By a similar way, the second iteration for the second phase generates m values, $t_{2m+1}, t_{2m+2}, \dots, t_{3m}$. The third iteration for the second phase generates m values, $t_{3m+1}, t_{3m+2}, \dots, t_{4m}$. In general, the k iteration for the second phase generates m values, $t_{km+1}, t_{km+2}, \dots, t_{(k+1)m}$, where each value of t_{km+i} is computed by multiplying the previous value, $t_{(k-1)m+i}$, with s^m , $1 \leq i \leq m$, i.e.,

$$t_{k \ m+i} = t_{(k-1)m+i} s^m. \quad (8)$$

In general, the thread number i generates $t_i, t_{m+i}, t_{2m+i}, \dots, t_{km+i}$, for the first k iterations. Each thread needs only a single value of t to generate a new value during the process, eliminating the need to keep all previous values $t_i: t_{m+i}, t_{2m+i}, \dots$ for the thread number i . This operation can be done by assigning the right-hand side of computing new value to t_i . Therefore, the Equation (7): $t_{m+i} = t_i s^m$ becomes $t_i = t_i s^m$.

Therefore, the general formula for computing the new value of t for the thread number i is given by the following corollary and it can be easily proved by induction.

Corollary 1 Given the initial value of t , $t_0 = a^{n-u+1} \bmod n$, and t_i as the current value of the term t for the thread number i , $1 \leq i \leq m$, then the new value for t_i is given by:

$$t_i = \begin{cases} t_0 \cdot s^i & \text{Initial step} \\ t_i \cdot s^m & \text{Otherwise} \end{cases} \quad (9)$$

By a similar way, the proposed method computes m values of u in parallel way, u_1, u_2, \dots, u_m . The initial values of u_i , $1 \leq i \leq m$, and any updating value of u_i are given by the following corollary.

Corollary 2 Given the initial value of u , $u_0 = 2 \lceil \sqrt{n} \rceil$, and u_i as the current value of the term u for the thread number i , $1 \leq i \leq m$, then the new value for u_i is given by:

$$u_i = \begin{cases} u_0 + 2i & \text{Initial step} \\ u_i + 2 \cdot m & \text{Otherwise} \end{cases} \quad (10)$$

Phase 2 of the proposed algorithm can be summarized as follows: all threads work concurrently to generate their own t_i and u_i values using Equation (9) and (10), respectively. For each value of t_i , the thread number i tests whether t_i equals 1. If t_i equals 1, it means the thread has found the solution; otherwise, the thread continues to compute new values of t_i and u_i .

Algorithm 1 represents the pseudocode of the Parallel Euler-based Fermat Factorization (PEFF) algorithm. The algorithm starts by computing the values of s and t_0 as described in lines 1-5. If t_0 equals 1, the algorithm finds the solution, as shown in lines 6-8. Otherwise, it computes s^i , in the array sp of length m as in lines 9-14, $1 \leq i \leq m$.

The steps in lines 11-14 can be computed in parallel using a prefix-sum algorithm on a parallel shared-memory system [23, 24] if the value of m is large. Based on t_0 and sp , the algorithm computes the initial values of t_i and u_i for each thread, and then tests the possibility of finding a solution when $t_i = 1$. The lines 15-24 represent the first phase of the proposed algorithm. All steps from line 1 to 24 represent Phase 1. If no thread finds the solution in the initial values of t_i , the algorithm executes Phase 2 by having the m threads work concurrently to find new values of t_i until one of them equals 1, as shown in lines 25-36.

Both algorithms, EFF and PEFF, can be divided into two parts. The first part represents a preprocessing calculation. Let α_0 and α'_0 are constant values and represent the cost of the preprocessing operations, first part, for the EFF and PEFF algorithms, respectively. In the PEFF algorithm, the first part starts from line 1 to 24. The value of α'_0 is greater than α_0 because the PEFF algorithm will generate the initial value of the term t for each thread and computes the array sp . The second part represents the core operations of each algorithm which include a repetition of the two operations, one modular multiplication and one addition. Assume that these operations are repeated α times in the EFF algorithm, and the value of α is not a constant and depends on d . Then the total number of repetitions for the two operations in the PEFF algorithm is α/m . Therefore, the total number of modular multiplication and addition operations in the EFF and PEFF algorithms are $O(\alpha_0 + \alpha) = O(\alpha)$ and $O(\alpha'_0 + \frac{\alpha}{m}) = O(\alpha/m)$, respectively. It is clear that for large values of m , the PEFF runs faster than the EFF algorithm.

Algorithm 1 Parallel Euler-based Fermat Factorization (PEFF)

Input: n is an odd number and m threads.

Output: p and q are two primes such that $n = pq$.

Begin

1. $u_0 = 2 \lceil \sqrt{n} \rceil$
2. $c = 2$
3. $a = c^{-1} \bmod n$
4. $s = c^2 \bmod n$
5. $t_0 = a^{n-u_0+1} \bmod n$
6. If ($t_0 = 1$) then
7. $p = u_0/2 + \sqrt{(u_0/2)^2 - n}$
8. $q = u_0/2 - \sqrt{(u_0/2)^2 - n}$
9. Else
10. found = False, $m' = 2 * m$
11. $sp[1] = s$
12. For $i = 2$ to m do
13. $sp[i] = sp[i-1] \times s$
14. End for
15. For $i = 1$ to m do parallel
16. $t_i = t_0 * sp[i]$
17. If ($t_i > n$) then $t_i = t_i \bmod n$
18. $u_i = u_0 + 2 * i$
19. If ($t_i = 1$) then
20. found = True
21. $p = u_i/2 + \sqrt{(u_i/2)^2 - n}$

```

22.       $q = u_i/2 - \sqrt{(u_i/2)^2 - n}$ 
23.    End if
24.  End for parallel
25.  For  $i = 1$  to  $m$  do parallel
26.    While (found  $\neq$  True) do
27.       $t_i = t_i * sp[m]$ 
28.      If ( $t_i > n$ ) then  $t_i = t_i \bmod n$ 
29.       $u_i = u_i + m'$ 
30.      If ( $t_i = 1$ ) then
31.        found = True
32.         $p = u_i/2 + \sqrt{(u_i/2)^2 - n}$ 
33.         $q = u_i/2 - \sqrt{(u_i/2)^2 - n}$ 
34.      End if
35.    End while
36.  End for parallel
End

```

3. Results and discussions

This section presents the experimental evaluation of the proposed algorithm, PEFF, by comparing its performance with the best-known parallel algorithm, PFF, and the sequential algorithm, EFF, using various criteria. The first part of this section describes the platform used to execute the algorithms. The second part explains the parameters used in the experimental study and the measurements used to evaluate the algorithms. Finally, the results of the experiments are discussed from multiple perspectives.

3.1 Hardware and software platform

All algorithms, both sequential and parallel, were programmed in C++ using GMP [25], a GNU Multiple Precision library for performing arithmetic operations on large integers and other data types. Additionally, Open Multi-Processing (OpenMP), an application programming interface [26], was used to implement parallel regions for the parallel algorithms on a shared memory system. The algorithms are implemented on a multicore system running the Windows operating system. The system includes 10 cores and supports the execution of up to 20 threads. It also includes 64 GB of RAM and 15 MB of cache memory. The system operates at a speed of 2.3 GHz.

3.2 Experiment parameters and measurements

The experimental comparison for all algorithms is based on three parameters. The first parameter is the number of threads, m , used in the experiments. The experiments evaluate the parallel algorithms using different values of m , specifically 4, 8, and 16, while the sequential algorithm, EFF, uses $m = 1$. The second parameter is the size of the composite odd number n , which is l . The values of l used in the experiments are 1k, 2k, and 4k. The third parameter is the number of bits for the difference between two prime numbers, $|d|$ bits. In the experiment, the value of $|d|$ equals $(l/4) + d'$, where $d' = 12, 16$, and 20 . The reason for starting the value of $|d|$ with $(l/4) + 12$ is the running time of the sequential algorithm, EFF, which is less than 0.1 second. Therefore, parallelization does not result in significant improvement.

Note that when we generate the value of n as input for Algorithm 1, n is constructed as a product of two prime numbers such that the number of bits for the difference between them is $|d|$.

The criteria used to measure the performance of the algorithms are running time, speedup, and scalability. The running time is measured experimentally in seconds. For each fixed combination of m , l , and $|d|$, the running time of an algorithm represents the average over 25 instances. The speedup of a parallel algorithm is calculated as the ratio of

the running time of the EFF algorithm to the running time of the PEFF (or PFF) algorithm. The scalability of a parallel algorithm, PEFF or PFF, measures the effect of increasing the number of threads on the speedup.

3.3 Results analysis

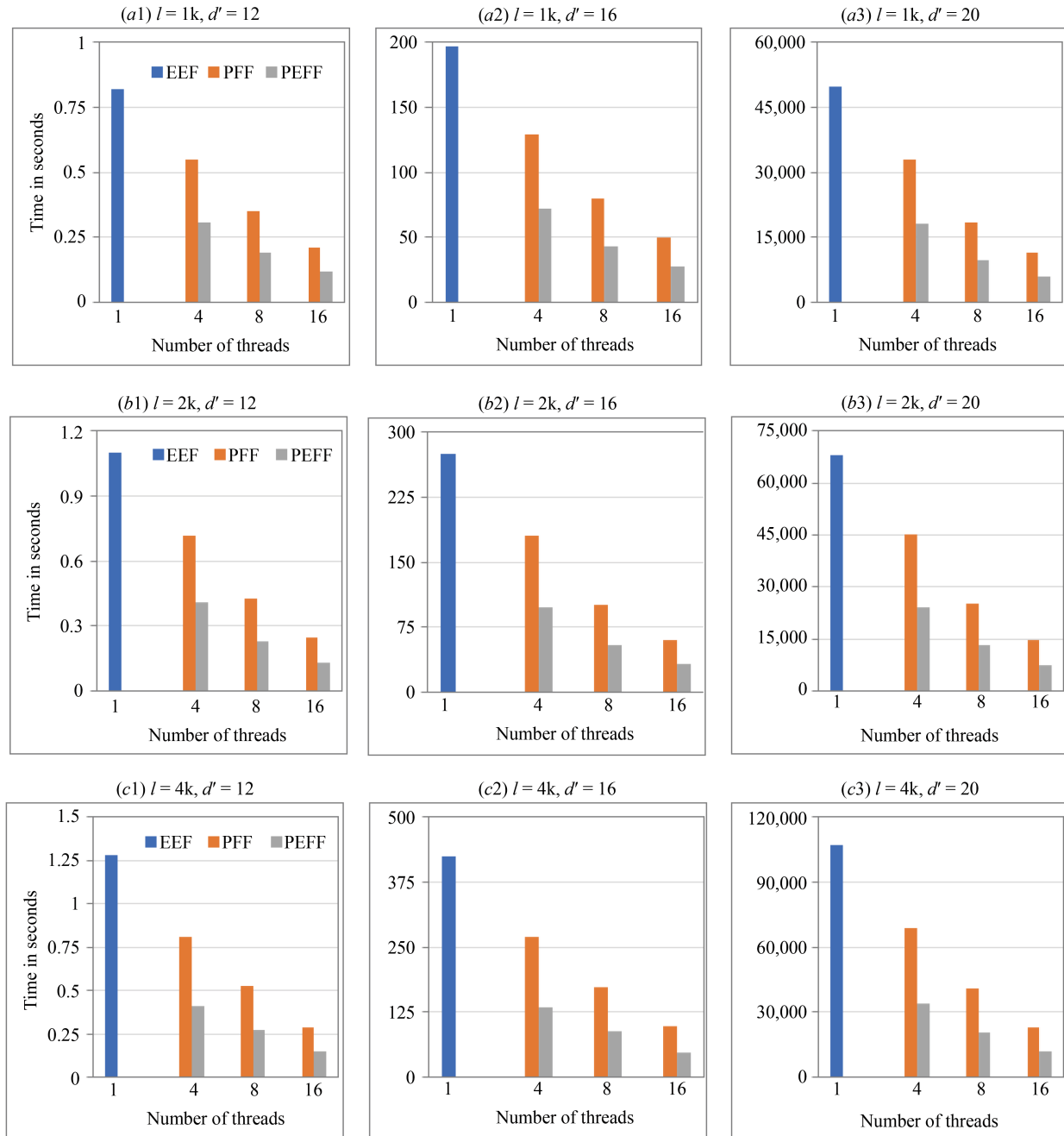


Figure 3. Running time for the three algorithms, EFF, PFF, and PEFF

Figure 3 shows the execution times for three algorithms-EFF, PFF, and PEFF-based on the experimental parameters and platforms discussed in Sections 3.1 and 3.2. The figure consists of nine subfigures. Subfigures 3-a1, 3-a2, and 3-a3, represent the execution times of the three algorithms for the parameter sets (1) $l = 1k = 1,024$ and $d' = 12$, (2) $l = 1k$ and

$d' = 16$, and (3) $l = 1k$ and $d' = 20$, respectively. Similarly, subfigures 3-b1, 3-b2, and 3-b3, represent the execution times for the parameter sets (1) $l = 2k = 2,048$ and $d' = 12$, (2) $l = 2k$ and $d' = 16$, and (3) $l = 2k$ and $d' = 20$. Lastly, subfigures 3-c1, 3-c2, and 3-c3, represent the execution times for the parameter sets (1) $l = 4k = 4,096$ and $d' = 12$, (2) $l = 4k$ and $d' = 16$, and (3) $l = 4k$ and $d' = 20$.

Analyzing the data presented in Figure 3 leads to the following recommendations:

First, the running time of each algorithm increases as the values of n and d' (consequently $|d|$) increase. For instance, with a fixed value of $l = 4k$ and varying values of $d' = 12, 16$, and 20 , the running times of the EFF algorithm are 1.28, 423.6, and 107,103 seconds, respectively. Additionally, with a fixed value of $d' = 16$ and different values of $l = 1k, 2k$, and $4k$, the EFF running times are 197.2, 277, and 423.6 seconds, respectively. It is evident that an increase in the difference between the two prime factors, d , results in a significant rise in the running time of the EFF algorithm.

Second, the execution times of the two parallel algorithms, PFF and PEFF, are faster than those of the EFF algorithm in all cases. This means that utilizing the high-performance system effectively reduces the running time of the sequential algorithm even when using the low-cost operation, modular multiplication, in the sequential computation.

Third, the average improvements in execution times for the PEFF algorithm compared to the EFF algorithm are 65%, 79%, and 87.8% when using 4, 8, and 16 threads, respectively, across all cases of l and $|d|$. It is clear that increasing the number of threads in parallel computation will increase the improvement in the running time.

Fourth, the average improvements in execution times of the PFF algorithm compared to the EFF algorithm are 34.7%, 60.6%, and 76.9% when using 4, 8, and 16 threads, respectively, across all cases of l and $|d|$.

Fifth, the PEFF algorithm consistently outperforms the PFF algorithm, with average improvements of 47.6%, 47.3%, and 48.2% when using 4, 8, and 16 threads, respectively, across all cases of l and $|d|$. In general, the PEFF algorithm demonstrates superior execution times compared to both the EFF and PFF algorithms.

Sixth, the speedup of the two parallel algorithms, PFF and PEFF, compared to the sequential FE algorithm is shown in Table 1. It is clear that the speedup of the PEFF algorithm surpasses that of the PFF algorithm for two main reasons: First, the performance of the PEFF algorithm is superior to the PFF algorithm due to the replacement of the PS operation with modulo multiplication. Second, the speedup of the PFF algorithm was measured relative to the EFF algorithm rather than the original FF algorithm, where the running time of the EFF algorithm is shorter than that of the FF algorithm.

Table 1. Speedup of the PEF and PEFF algorithms

Algorithm	Number of threads		
	4	8	16
PFF	1.9	2.5	4.3
PEFF	2.9	4.8	8.3

Seventh, Figure 4 shows the scalability of the two parallel algorithms, PFF and PEFF, when $d' = 20$ and varying values of $l = 1k, 2k$, and $4k$. Using different values of thread numbers, the PEFF algorithm is more scalable than the PFF algorithm. The results in Figure 4 demonstrate that the scalability of the PEFF algorithm is approximately twice that of the PFF algorithm.

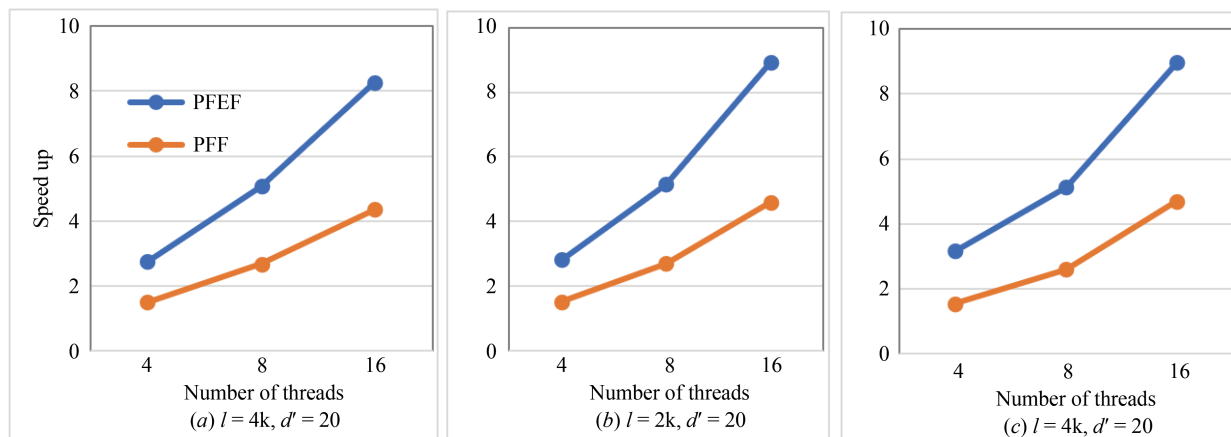


Figure 4. Scalability of the two parallel algorithms, PFF and PEF

4. Conclusion

In this study, we have addressed the problem of factoring an odd number, n , into two prime factors based on Fermat's method and Euler's theorem. The computational running time of this method increases with increasing difference between the two prime factors. To mitigate this issue, we used high-performance computing to reduce computation time. First, we have derived a parallel formula for the Euler-based Fermat factoring method. Then, we have provided a complete pseudocode for factoring n .

Many practical studies have demonstrated the effectiveness of the proposed parallel algorithm compared to two other algorithms: one sequential and one parallel. The experiments are conducted using three parameters and measurements. The proposed parallel algorithm achieves a 77.3% improvement in execution time compared to the sequential algorithm; furthermore, it outperforms the best-known parallel algorithm with a 47.7% improvement. Additionally, the proposed parallel algorithm demonstrates greater speedup and scalability than the best-known parallel algorithm.

One potential direction for future work is implementing the proposed algorithm on a Graphics Processing Unit (GPU). Additionally, an important question to explore is determining the maximum value of d that the parallel algorithm can solve within an efficient runtime.

Acknowledgement

The authors extend their thanks to Scientific Research Deanship at University of Ha'il-Saudi Arabia through project number «RG-23 120».

Funding

This research has been funded by Scientific Research Deanship at University of Ha'il-Saudi Arabia through project number «RG-23 120».

Conflict of interest

The authors declare no competing financial interest.

References

- [1] Hoffstein J, Pipher J, Silverman JH. *An Introduction to Mathematical Cryptography*. New York: Springer; 2014.
- [2] Bouroubi S, Rezkallah L. Rotating Disk Cryptosystem (RDC). *Journal of Information and Optimization Sciences*. 2020; 41(5): 1193-1205. Availavle from: <https://doi.org/10.1080/02522667.2019.1572977>.
- [3] Lenstra AK. Integer factoring. *Designs, Codes and Cryptography*. 2000; 19: 101-128. Availavle from: <https://doi.org/10.1023/A:1008397921377>.
- [4] Zaki AM, Bakr ME, Alsahangiti AM, Khosa SK, Fathy A. Acceleration of wheel factoring techniques. *Mathematics*. 2023; 11: 1203. Availavle from: <https://doi.org/10.3390/math11051203>.
- [5] Bahig HM, Nassr DI, Mahdi MA, Hazber MA, Al-Utaibi K, Bahig HM. Speeding up wheel factoring method. *Journal of Supercomputing*. 2022; 78: 15730-15748. Availavle from: <https://doi.org/10.1007/s11227-022-04470-y>.
- [6] Somsuk K. The new integer factorization algorithm based on Fermat's factorization algorithm and Euler's theorem. *International Journal of Electrical and Computer Engineering*. 2020; 10(2): 1469-1476. Availavle from: <http://doi.org/10.11591/ijece.v10i2.pp1469-1476>.
- [7] Shatnawi AS, Almazari MM, AlShara Z, Taqieddin E, Mustafa D. RSA cryptanalysis-Fermat factorization exact bound and the role of integer sequences in factorization problem. *Journal of Information Security and Applications*. 2023; 78: 103614. Availavle from: <https://doi.org/10.1016/j.jisa.2023.103614>.
- [8] Bahig HM, Bahig HM, Kotb Y. Fermat factorization using a multi-core system. *International Journal of Advanced Computer Science and Applications*. 2020; 11(4): 323-330. Availavle from: <http://dx.doi.org/10.14569/IJACSA.2020.0110444>.
- [9] Zralek B. A deterministic version of Pollard's p-1 algorithm. *Mathematics of Computation*. 2010; 79(269): 513-533. Availavle from: <https://www.jstor.org/stable/40590414>.
- [10] 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. Availavle from: <https://doi.org/10.3390/sym14020312>.
- [11] Kruppa A, Leyland P. Number field sieve for factoring. In: van Tilborg HCA, Jajodia S. (eds.) *Encyclopedia of Cryptography and Security*. Boston, MA: Springer; 2021. p.861-887.
- [12] Yang LT, Huang G, Feng J, Xu I. Parallel GNFS algorithm integrated with parallel block Wiedemann algorithm for RSA security in cloud computing. *Information Sciences*. 2017; 387: 254-265. Availavle from: <https://doi.org/10.1016/j.ins.2016.10.017>.
- [13] Ding J, Spallitta G, Sebastiani R. Effective prime factorization via quantum annealing by modular locally-structured embedding. *Scientific Reports*. 2024; 14: 3518. Availavle from: <https://doi.org/10.1038/s41598-024-53708-7>.
- [14] Brown DE, Skidmore D. Representing the integer factorization problem using ordered binary decision diagrams. *Theory of Computing Systems*. 2023; 67: 1307-1332. Availavle from: <https://doi.org/10.1007/s00224-023-10147-7>.
- [15] Sato A, Auzemery A, Katayama A, Yasuda M. Experimental analysis of integer factorization methods using lattices. In: Minematsu K, Mimura M. (eds.) *Advances in Information and Computer Security*. Singapore: Springer; 2024. p.142-157.
- [16] Bahig HM, Mahdi MA, Alutaibi KA, AlGhadhban A, Bahig HM. Performance analysis of fermat factorization algorithms. *International Journal of Advanced Computer Science and Applications*. 2020; 11(12): 340-352. Availavle from: <http://dx.doi.org/10.14569/IJACSA.2020.0111242>.
- [17] Somsuk K, Tientanopajai K. An improvement of Fermat's factorization by considering the last m digits of modulus to decrease computation time. *International Journal of Network Security*. 2017; 19: 99-111. Availavle from: [https://doi.org/10.6633/IJNS.201701.19\(1\).11](https://doi.org/10.6633/IJNS.201701.19(1).11).
- [18] Bahig HM. Speeding up Fermat's factoring method using precomputation. *Annals of Emerging Technologies in Computing*. 2022; 6(2): 50-60. Availavle from: <https://doi.org/10.33166/aetic.2022.02.004>.
- [19] Shiu P. Fermat's method of factorization. *The Mathematical Gazette*. 2015; 99(544): 97-103. Availavle from: <http://www.jstor.org/stable/24496908>.
- [20] Hart W. A one line factorization algorithm. *Journal of the Australian Mathematical Society*. 2012; 94: 61-69. Availavle from: <http://dx.doi.org/10.1017/S1446788712000146>.
- [21] Hittmeir M. Deterministic factorization of sums and differences of powers. *Mathematics of Computation*. 2023; 86(308): 2947-2954. Availavle from: <https://doi.org/10.1007/s00220-017-3017-5>.
- [22] RRM T, Asbullah M, Ariffin M, Mahad Z. Determination of a good indicator for estimated prime factor and its modification in Fermat's factoring algorithm. *Symmetry*. 2023; 13(5): 735. Availavle from: <https://doi.org/10.3390/sym13050735>.

- [23] Zhang W, Wang Y, Ross K. Parallel prefix sum with SIMD. *arXiv:2312.14874*. 2023. Available from: <https://arxiv.org/abs/2312.14874>.
- [24] Bahig HM, Fathy KA. An improved parallel prefix sums algorithm. *Parallel Processing Letters*. 2022; 32(03n03): 2250008. Available from: <https://doi.org/10.1142/S0129626422500086>.
- [25] GNU Multiple Precision Arithmetic Library. Available from: <https://gmplib.org/> [Accessed 15th July 2025].
- [26] OpenMP API Specification for Parallel Programming. Available from: <https://www.openmp.org/> [Accessed 15th July 2025].