The Journal of Supercomputing (2022) 78:15730–15748
https://doi.org/10.1007/s11227-022-04470-y

Speeding up wheel factoring method

Hazem M. Bahig[1,2] · Dieaa I. Nassr[2] · Mohammed A. Mahdi[1] · Mohamed A. G. Hazber[1] · Khaled Al-Utaibi[3] · Hatem M. Bahig[2]

Accepted: 18 March 2022 / Published online: 23 April 2022

Abstract

The security of many public key cryptosystems that are used today depends on the difficulty of factoring an integer into its prime factors. Although there is a polynomial time quantum-based algorithm for integer factorization, there is no polynomial time algorithm on a classical computer. In this paper, we study how to improve the wheel factoring method using two approaches. The first approach is introducing two sequential modifications on the wheel factoring method. The second approach is parallelizing the modified algorithms on a parallel system. The experimental studies on composite integers n that are a product of two primes of equal size show the following results. (1) The percentages of improvements for the two modified sequential methods compared to the wheel factoring method are almost 47% and 90%. (2) The percentage of improvement for the two proposed parallel methods compared to the two modified sequential algorithms is 90% on the average. (3) The maximum speedup achieved by the best parallel proposed algorithm using 24 threads is almost 336 times the wheel factoring method.

Keywords Integer factoring · Wheel factorization · Parallel algorithm · RSA · Multicore

1 Introduction

The efficiency of many public key cryptosystems and protocols, such as [11, 20, 23, 25], is based on exponentiation problem [9, 10], while the security of such cryptosystems and protocols depends on the difficulty of integer factorization problem. In integer factorization problem: given a positive integer n, find its prime factorization $n = p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$, where p_i's are pairwise distinct primes and each $e_i \geq 1$. This problem is one of the most important problems in number theory, complexity theory, and cryptography for several reasons:

✉ Hazem M. Bahig
h.bahig@uoh.edu.sa

Extended author information available on the last page of the article

1. it is a fundamental problem in mathematics;
2. it offers a good example of a problem that has not yet a polynomial time algorithm in classical computers but has a polynomial time algorithm in quantum computers [22, 27, 32];
3. there are many public key cryptosystems and protocols [11, 20, 23, 25] with security based on the difficulty of integer factorization problem.

Algorithms for factoring a large odd composite integer n can be divided into two major types [5, 16–18, 26, 30, 31]: special-purpose and general-purpose. The special-purpose factoring algorithms find small prime factors quickly regardless of the size of n. The main problem with special-purpose factoring algorithms is if n has no small factor, then applying one of the special-purpose factoring algorithms will have virtually no chance. Examples of such factoring algorithms are trial division, wheel factoring method, Pollard's ρ-method, Pollard's $p-1$ method, elliptic curve method and Fermat method. The general purpose factoring algorithms factor n regardless of the size of its prime factors, but they take exponential or subexponential time. Examples of such algorithms are continued fraction method, quadratic sieve, and number field sieve. Recently, Raddum, and Varadharajan [14] and Wu et al. [29] proposed subexponential factoring algorithms using binary decision and Pisano period, respectively. Until now, the number field sieve method [28] is the best method for factoring large n with large prime factors.

There are other factoring methods but they work with additional information about cryptosystems or some conditions on the prime factors, see [1, 3, 4, 19]. One of the approaches to speed up factoring algorithms is using high-performance systems such as multicore [6, 7, 13, 15], cloud systems [28], and graphics processing units [2, 8].

In this paper, we concern with the problem of factoring an odd composite integer n, where n is a product of two primes of the same size. We study how to improve the wheel factoring method (WFM) [24] sequentially and parallel. The main advantage of WFM over the trail division method is reducing the number of divisions [30]. On another hand, WFM requires more divisions than the sieving method [30]. The main advantage of WFM over sieving is that it does not need to have a list of primes available, i.e., sieving requires large memory. We present two (sequential) modified versions of WFM. Then, we parallelize them on a shared memory model. The percentage of improvements of the two proposed sequential modifications are 47% and 90%, on the average case, While the percentages of improvements for the two proposed parallel methods compared to the two proposed sequential methods are 90% on the average. Also, the scalability of the proposed parallel algorithms is almost sublinear.

The paper is organized as follows. In Sect. 2, we give an overview of WFM and its pseudocode. In Sect. 3, we introduce two sequential improvements for WFM and two parallel strategies for the two (sequential) modified WFMs. An illustrative example to show the difference between the two proposed parallel algorithms is given in Sect. 4. The experimental studies are presented in Sect. 5. The analysis of the presented algorithms according to execution time, speedup, and scalability

is discussed in Sect. 6. Section 7 discusses the effect of wheel size on the performance of wheel factoring method. In Sect. 8, we present the conclusion of this work.

2 Wheel factorization method

In this section, we describe briefly the main idea of WFM and present the pseudocode of WFM to factorize a composite integer into two factors.

The WFM is based on selecting a list of the first k consecutive primes, called basis, say $B = \{b_1, b_2, \ldots, b_k\}$, where b_i is a prime number and k is a very small number, say 3 or 4. Then, WFM generates a list T, called the turn or wheel, of numbers that are coprime with all numbers in B. The circumference of the wheel (also called *primorial*) is equal to the product of the basis numbers $s = b_1 \times b_2 \times \cdots \times b_k$. Next, WFM uses the numbers in the wheel/turn to find the smallest divisor of the number to be factorized. Figure 1 shows WFM when $B = \{2, 3, 5\}$, and the circumference of a turn is $s = 30 = 2 \times 3 \times 5$. It shows to find a prime factor of n, we divide n by 2, 3, 5, and by all the numbers congruent to 1, 7, 11, 13, 17, 19, 23, and 29, modulo 30 until we find a prime factor or reach to the square root of n. In this case, n is prime.

As we can see from Fig. 1, the first turn $T = \{1, 7, 11, 13, 17, 19, 23, 29\}$, while the second turn $T = \{31, 37, 41, 43, 47, 49, 53, 59\}$. All elements of T are coprimes to the elements of B, i.e., $gcd(b_i, t_j) = 1, b_i \in B, t_j \in T$. Thus, WFM tries to find a factor of n by the elements of T.

It is not necessary to start WFM from 1. We can start, for example, from 11 and so the first turn set is $\{11, 13, 17, 19, 23, 29, 31, 37\}$. In this case, we should first divide n by the elements of B and the number 7, i.e., first divide n by 2, 3, 5, and 7. In the next turn, WFM tests the divisibility of n by the numbers in the next turn set $\{41, 43, 47, 49, 53, 59, 61, 67\}$. Table 1 displays the first ten turn sets when WFM starts

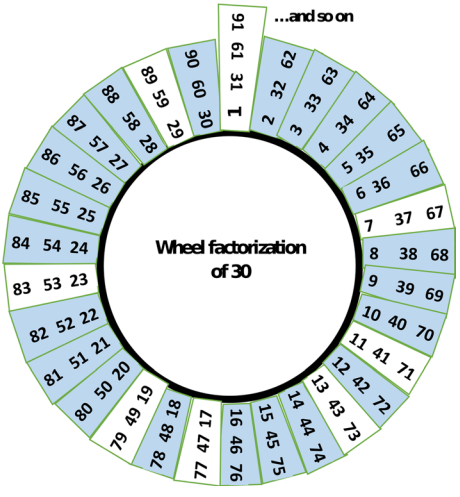

Fig. 1 Wheel of size 30

Table 1 The first ten turns of circumference 30 starting from 11

Turn no.	i	i+2	i+6	i+8	i+12	i+18	i+20	i+26
1	11	13	17	19	23	29	31	37
2	41	43	47	49	53	59	61	67
3	71	73	77	79	83	89	91	97
4	101	103	107	109	113	119	121	127
5	131	133	137	139	143	149	151	157
6	161	163	167	169	173	179	181	187
7	191	193	197	199	203	209	211	217
8	221	223	227	229	233	239	241	247
9	251	253	257	259	263	269	271	277
10	281	283	287	289	293	299	301	307

from 11. It is clear that the first integer of the jth turn is equal to $i = 11 + 30(j - 1)$ and the next elements of the jth turn are $i + 2, i + 6, i + 8, i + 12, i + 18, i + 20$, and $i + 26$. The differences between two successive integers in any turn are 2, 4, 2, 4, 6, 2, and 6, respectively, while the difference between the last integer in any turn and the first integer in the next turn is 4. These differences can be stored in an array, say *inc,* to handle all turns. The complete pseudocode for WFM is shown in Algorithm 1. The algorithm uses a subroutine called FactorBasis to test if n has a factor from the basis set B or not, as shown in Algorithm 2.

Algorithm 1: WFM

```
Input: A composite integer number n.
p =FactorBasis(n)
if p ≠ 1 then return p and n/p;
m = ⌊√n⌋
k = 11
i = 1
inc[8] = {2, 4, 2, 4, 6, 2, 6, 4}
while k ≤ m do
    if n mod k = 0 then return k and n/k ;
    else
        k = k + inc[i]
        if i < 8 then i = i + 1;
        else i = 1;
    end if
end while
Output: Two factors p and q of n.
```

3 The improvements

This section presents two sequential and parallel improvements for WFM. We present the main idea and the pseudocode for each improvement. We use *NTrd* to denote the number of threads used in parallel systems.

3.1 Forward WFM

In WFM, each iteration of the while-loop, Lines 8–15 of Algorithm 1, tests the divisibility of n by the number k. For each turn, the value of k varies based on the value of $inc[i], 1 \leq i \leq 7$, Lines 11–13. The value of $inc[8]$ is used to generate the start of the next turn.

Algorithm 2: FactorBasis

Input: A composite integer number n.
$p = 1$
if $n \mod 2 = 0$ **then** $p = 2$;
else if $n \mod 3 = 0$ **then** $p = 3$;
else if $n \mod 5 = 0$ **then** $p = 5$;
else if $n \mod 7 = 0$ **then** $p = 7$;
return p
Output: Either a prime number $p \in \{2, 3, 5, 7\}$ or 1

The idea of the first improvement is instead of making eight different iterations in the while-loop for one turn, we can do it in only one iteration. The idea can be done by making eight tests inside each iteration of the while-loop, and so there is no need to use the array *inc*. Also, there is no need to update the counter i and test the value of i in every iteration of the while-loop. Thus, we can remove the Lines 12–13, from the while-loop.

This improvement is presented in Algorithm 3, called Forward WFM (**FWFM**). FWFM uses the subroutine TurnTest, Algorithm 4, to test each turn j of length 30 starting from $11 + 30j, j \geq 1$. The subroutine TurnTest returns the factors of n if exist, otherwise returns 1. The subroutine starts with k, the first element of each turn and then updates k, respectively, by the following values: 2, 4, 2, 4, 6, 2, 6. The full pseudocodes for FWFM and TurnTest algorithms are shown in Algorithms 3 and 4, respectively.

Algorithm 3: Forward Wheel Factoring Method (**FWFM**)

Input: A composite integer number n.
$p =$FactorBasis(n)
if $p \neq 1$ **then** return p and n/p;
$m = \lfloor \sqrt{n} \rfloor$
$i = 11$
while $i \leq m$ **do**
  $p =$ TurnTest(n, i)
  if $p \neq 1$ **then** return p and n/p ;
  else $i = i + 30$;
end while
Output: Two factors p and q of n.

Algorithm 4: TurnTest

```
Input: A composite integer number n, and a positive integer k.
p = 1
if n mod k = 0 then p = k;
else
    k = k + 2
    if n mod k = 0 then p = k;
    else
        k = k + 4
        if n mod k = 0 then p = k;
        else
            k = k + 2
            if n mod k = 0 then p = k;
            else
                k = k + 4
                if n mod k = 0 then p = k;
                else
                    k = k + 6
                    if n mod k = 0 then p = k;
                    else
                        k = k + 2
                        if n mod k = 0 then p = k;
                        else
                            k = k + 6
                            if n mod k = 0 then p = k;
                        end if
                    end if
                end if
            end if
        end if
    end if
end if
return p
Output: Either a prime number p ∈ {2, 3, 5, 7} or 1
```

3.2 Backward WFM

In some public-key cryptosystems, such as RSA [25], the public modulus n is a product of two prime numbers of the same size, and so the smallest prime factor is near to $\lfloor\sqrt{n}\rfloor$. Therefore, it is better to search for the factor from $\lfloor\sqrt{n}\rfloor$ down to 2 (backward direction).

The number of turns in the search space, starting from 11, is equal to $nTurns = \lceil(\lfloor\sqrt{n}\rfloor - 10)/30\rceil$. The algorithm starts from the last turn started from $i = 30(nTurns - 1) + 11$. If the algorithm finds a factor of n, then the algorithm terminates. Otherwise, the algorithm searches for a factor of n in the next turn, in reverse order, by subtracting the circumference of the turn 30 from i, i.e., $i = i - 30$, and repeating the process for all turns. If no factor is found in the *nTurns*-turns, then

the algorithm searches for the factors 7, 5, 3, and 2. The complete pseudocode for backward wheel factoring method (BWFM) is given in Algorithm 5. For simplicity, Lines 9–10 can be done before Line 2 as in Algorithm 1.

Algorithm 5: Backward Wheel Factoring Method (**BWFM**)

1 **Input:** A composite integer number n.
2 $nTurns = \lceil \frac{\lfloor \sqrt{n} \rfloor - 10}{30} \rceil$
3 $i = 30 \times (nTurns - 1) + 11$
4 **while** $i \geq 11$ **do**
5 $p = \text{TurnTest}(n, i)$
6 **if** $p \neq 1$ **then** return p and n/p ;
7 **else** $i = i - 30$;
8 **end while**
9 $p = \text{FactorBasis}(n)$
10 return p and n/p
11 **Output:** Two factors p and q of n.

3.3 Parallel FWFM

Since the search space for finding the factors is from 2 to $\lfloor \sqrt{n} \rfloor$, the straight forward to parallelize FWFM (PFWFM) is dividing the search space into almost equal size between the *NTrd*-threads. The range of the search space is equal to $\lfloor \sqrt{n} \rfloor - 10$ since FWFM starts the search from 11.

In order to calculate the start and end of each thread, we first compute the number of turns in the search space, which is equal to $nTurns = \lceil (\lfloor \sqrt{n} \rfloor - 10)/30 \rceil$. Then, the turns are distributed, approximately, equally between the threads, where each thread either tests $\lceil nTurns/(NTrd) \rceil$ or $\lfloor nTurns/NTrd \rfloor$ turns. The turns are distributed on the threads based on the following rule. Let *nTurnsTrd* denote the number of turns per thread, i.e., $nTurnsTrd = \lfloor nTurns/(NTrd) \rfloor$. The first ($nTurns \mod NTrd$) threads take $nTurnsTrd + 1$ turns for each, while the other threads take *nTurnsTrd* turns for each. Based on this distribution, we can determine the start and end of each thread as in Lines 9–17, Algorithm 6.

The algorithm consists of three main steps. In the first step, the algorithm checks, sequentially, the divisibility of n by the elements of the basis set, see Lines 2–3. The second step is preprocessing to determine the number of turns for each thread, see Lines 4–7. In the third step, each thread determines its search interval, i.e., the first number in its first turn and the last number in its last turn, see Lines 9–17. For example, a thread number $i, 0 \leq i < (nTurns \mod NTrd)$ has the search interval

$$[11 + 30 \times i \times (nTurnsTrd + 1), 11 + 30 \times (i + 1)(nTurnsTrd + 1)].$$

Then, a thread tries to find a factor of n in its search interval and changes the value of the boolean variable found to true if one of the factors is found, Lines 20–21. If

no factor is found in the current turn, the algorithm updates *j* to the next turn as in Line 22. The variable *found* is shared between all threads. The complete pseudocode of PFWFM is given in Algorithm 6.

Algorithm 6: Parallel Forward Wheel Factoring Method (**PFWFM**)

```
1  Input: A composite integer number n.
2  p =FactorBasis(n)
3  if p ≠ 1 then return p and n/p ;
4  nTurns = ⌈(⌊√n⌋−10)/30⌉
5  nTurnsTrd = ⌊nTurns/NTrd⌋
6  rem = nTurns mod NTrd
7  found = False
8  for i ← 0 to NTrd − 1 parallel do
9      if i < rem then
10         iStart = 11 + 30 × i × (nTurnsTrd + 1)
11         iEnd = 11 + 30 × (i + 1) × (nTurnsTrd + 1) − 1
12     else
13         tmpStart = 11 + 30 × rem × (nTurnsTrd + 1)
14         iStart = tmpStart + 30 × (i − rem) × (nTurnsTrd)
15         if i < NTrd − 1 then
             iEnd = tmpStart + 30 × (i − rem + 1) × (nTurnsTrd) − 1;
16         else iEnd = ⌊√n⌋;
17     end if
18     j = iStart
19     while j ≤ iEnd and Not found do
20         p = TurnTest(n, j)
21         if p ≠ 1 then found = True;
22         else j = j + 30 ;
23     end while
24     return p and n/p
25 end for
26 Output: Two factors p and q of n.
```

3.4 Parallel BWFM

The idea of parallel BWFM (PBWFM) is based on that each thread will search for a factor of *n* in non-contiguous turns, not as in PFWFM, i.e., the first *NTrd*-turns, from the back, are assigned to the *NTrd* threads such that each thread will search in one turn. If no factor of *n* is found, then a new turn from the second *NTrd*-turns, in the back direction, will be assigned to a thread. The process will continue until one thread finds the factors or no factor is found in the search space. In this case, the algorithm tests the factors 7, 5, 3, and 2 in a sequential way.

Therefore, the algorithm first needs to determine the start of the first turn that will be tested for each thread *i*, which is given by the formula:

$$j = 30(nTurns - 1 - i) + 11,$$

where $nTurns = \lceil (\lfloor \sqrt{n} \rfloor - 10)/30 \rceil$ and $0 \leq i < NTrd$. If no factor is found in the current turn for a thread i, the start of the next turn is calculated by the following formula:

$$j = j - 30 \times NTrd,$$

where j on the right-hand side of the previous formula is the start value of the current turn. We subtract the value $30 \times NTrd$ from the value of j for the current turn since we have $NTrd$ threads, each thread works on a turn of size 30. The complete pseudocodes for PBWFM are given in Algorithm 7.

Algorithm 7: Parallel Backward Wheel Factoring Method (**PBWFM**)

Input: A composite integer number n.
$nTurns = \lceil \frac{\lfloor \sqrt{n} \rfloor - 10}{30} \rceil$
$found = False$
for $i \leftarrow 0$ *to* $NTrd - 1$ *parallel* **do**
  $j = 11 + 30 \times (nTurns - 1 - i)$
  while $j \geq 11$ *and Not Found* **do**
    $p = \text{TurnTest}(n, j)$
    if $p \neq 1$ **then** $found = True$;
    else $j = j - (30 \times NTrd)$;
  end while
  if $p \neq 1$ **then** return p and n/p;
end for
$p =$FactorBasis(n)
return p and n/p
Output: Two factors p and q of n.

4 An illustrative example

In this section, we give an illustrative example to show the significant of PBWFM over PFWFM for a composite integer n as in RSA modulus [25].

Let $n = 277 \times 409 = 113293$. Assume that the number of threads $NTrd = 4$. The execution of PFWFM is as follows. First, the algorithm starts by calling the subroutine Factorsbases on n, and the result of calling is $p = 1$, which means that the integers 2, 3, 5, and 7 are not factors of n. Second, the algorithm determines the values of *nTurns*, *nTurnsTrd*, *rem*, and *found* which are equal to 11, 2, 3, and false, respectively, where $\lfloor \sqrt{n} \rfloor = 336$. Third, each thread determines the start and end of its search interval as in Table 2 .

The third thread finds the factor after 24 mod-tests since each turn requires 8 mod-tests and the factor 277 exists at the last test of the third turn. For more details, the third thread tests the following: the integers 191, 193, 197, 199, 203, 209, 211, and 217 for the first turn; while the integers 221, 223, 227, 229, 233, 239, 241, and 247 for the second turn and then the integers 251, 253, 257, 259, 263, 269, 271, and 277 for the third turn; see Fig. 2a.

Now, we show how PBWFM improves the search for finding the factor of the number $n = 113293$. Initially, the algorithm computes *nTurns* and *found* which are equal to 11, and false, respectively. Next, each thread determines the start of the assigned turns from a backward direction. The starts of the four turns for the available four threads are 311, 281, 251, and 221, respectively. The third thread finds the factor 277 after eight tests since the thread tests the divisibility of n by the integers, 251, 253, 257, 259, 263, 269, 271, and 277, see Fig. 2b.

5 Results

This section presents the implementation of the proposed improvements (sequential and parallel) on WFM. It consists of two subsections. The first subsection contains the setting of the experimental studies including hardware, software, and data set. The second subsection includes the average execution times for the five algorithms, WFM, FWFM, BWFM, PFWFM, and PBWFM.

5.1 Experimental setup

The experimental studies have been conducted on a computer consisting of two hexa-core processors which can run 24 threads concurrently. The processor is of type Xeon E5-2630 and of speed 2.6 GHz. The used computer has global and cache memories of sizes 16 GB and 15 MB, respectively.

The C++ language and OpenMP library [21] are used to implement the algorithms under Microsoft Windows 10 operation system, where the OpenMP library is used to implement the parallel loops. The GMP library (GNU Multiple Precision) [12] is used to manipulate big integers, greater than 64 bits.

In the experimental study, we concentrate on the effect of the presented algorithms for factoring n when n is a product of two primes of the same size. For example, if n is of size $m = 50$ bits, then each prime factor has 25 bits. The sizes of the composite numbers conducted in the experimental study are $m = 40, 50, 60, 70$ and 80 bits. For each fixed number of bits, the running time of each implemented algorithm is the average time for 50 random numbers except for the size = 80, we used 20 random numbers since the running times of WFM and FWFM are large. The numbers of threads used in the implementation are 6, 12, 18, and 24.

5.2 Execution time

This subsection presents the execution times of the presented algorithms (WFM, FWFM, BWFM, PFWFM, PBWFM) when the integer n is of size m-bits and a product of two primes of equal size.

Table 3 shows the average execution times (in seconds) for WFM, FWFM, BWFM, PFWFM, and PBWFM.

Table 2 Starts and ends of each thread for PFWFM

i	iStart	iEnd
0	11	100
1	101	190
2	191	280
3	281	336

6 Discussion

In this section, we discuss and analyze the results of implementing the five algorithms on the dataset according to different criteria. In Sect. 6.1, we present the comparison between the five algorithms in terms of running time. Section 6.2 discusses the speedup and scalability of the proposed parallel algorithms.

6.1 Comparison

We compare between the implemented algorithms in terms of the following cases:

1. Sequential algorithms: We compare between each pair of sequential algorithms (WFM, FWFM), (WFM, BWFM), and (FWFM, BWFM).
2. Sequential and parallel algorithms: We compare between the sequential algorithm and its parallelization (FWFM, PFWFM) and (BWFM, PBWFM).
3. Parallel algorithms: We compare between the two parallel algorithms (PFWFM, PBWFM).

In case 1, Table 3 illustrates the execution times for WFM, FWFM, and BWFM on different size of n, 40-80 bits. The analysis of the results in Table 3 shows the following observations:

(1) The two modified sequential algorithms, FWFM and BWFM, have execution times less than WFM in all cases.

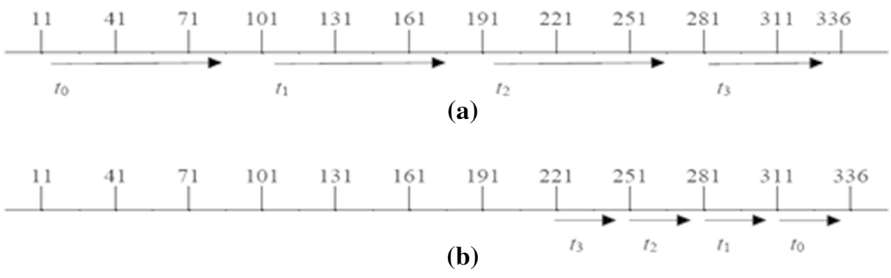

Fig. 2 Tracing of **a** PFWFM and **b** PBWFM on $n = 113293$

(2) The percentage of improvement for FWFM compared to WFM is 47.8% on the average. The minimum and maximum improvements for all cases are 45.2% and 48.8%, respectively, see Table 4. The improvement is based on reducing the number of loops and making the test of all numbers inside one turn in one loop instead of eight loops.
(3) The percentage of improvement for BWFM compared to WFM is 89.9% on the average. The minimum and maximum improvements for all cases are 82.05% and 95.68%, respectively, see Table 4.
(4) The execution time for BWFM is less than that for FWFM and the percentage of improvement is 80.8% on the average.

In case 2, the analysis of the results in Table 3 indicates the following:

(1) For $m \geq 50$, the execution time for PFWFM is less than that for FWFM using different number of threads, 6, 12, 18, and 24. For example, when $m = 70$, the execution time for FWFM is 753.7 seconds, while the execution times for PFWFM are 110.7, 80.2, 46.3, and 42.9 seconds using 6, 12, 18, and 24 threads, respectively. In the case of $m = 40$, the execution time for FWFM is less than PFWFM because the running time of FWFM is very small.
(2) The percentage of improvement for PFWFM compared to FWFM almost increases with increasing the number of threads. For example, when $m = 60$, the percentages of improvements for PFWFM are 87.6%, 89.2%, 93%, and 93.5% using 6, 12, 18, and 24 threads, respectively.
(3) On the average, the percentage of improvements for PFWFM compared to FWFM are 65.3%, 90.9%, 90.7%, and 94.1%, for $m = 50, 60, 70$, and 80, respectively. The overall average percentage of improvement for PFWFM compared to FWFM is 85.2%.

Table 3 Execution time (sec.) for WFM, FWFM, BWFM, PFWFM, and PBWFM

Algorithm	NTrd	m bits				
		40	50	60	70	80
WFM	1	0.031	0.741	35.8	1,471.4	67,848.1
FWFM	1	0.017	0.380	18.6	753.7	34,973.3
BWFM	1	0.005	0.133	2.5	72.8	2929.5
PFWFM	6	0.021	0.157	2.3	110.7	3277.6
	12	0.020	0.126	2.0	80.2	2346.5
	18	0.028	0.117	1.3	46.3	1344.9
	24	0.040	0.127	1.2	42.9	1290.2
PBWFM	6	0.016	0.032	0.4	12.8	400.8
	12	0.014	0.024	0.3	10.1	303.1
	18	0.023	0.027	0.3	7.1	205.9
	24	0.032	0.050	0.3	6.2	185.0

Similarly, we compare BWFM and PBWFM. In this case, the execution time for PBWFM is less than for BWFM using different number of threads, 6, 12, 18, and 24, for $m \geq 50$. Additionally, the percentages of improvements for PBWFM compared to BWFM are 75%, 87%, 87.6%, and 90.7% on the average case for $m = 50, 60, 70$, and 80, respectively. Therefore, the overall percentage of improvement for PBWFM compared to BWFM is 85.1% on the average.

In case 3, the analysis of the results in the Table 3 shows the following observations.

(1) For $m \geq 60$, the running time for PFWFM and PBWFM decreases with increasing the number of threads for a fixed problem size. For example, when $m = 60$ bits, the execution times for PFWFM using 6, 12, 18, and 24 threads are 2.3, 2, 1.3, and 1.2 seconds, respectively. Also, when $m = 70$ bits, the execution times for PBWFM using 6, 12, 18, and 24 threads are 12.8, 10.1, 7.1, and 6.2 seconds, respectively.
(2) The execution time for PBWFM using six threads is less than the execution time for PFWFM using 24 threads for a fixed size of n since n has prime factors near $\lfloor \sqrt{n} \rfloor$. For example, if $m = 80$ bits, then the execution times for PBWFM and PFWFM are 400.8 and 1,290.2 seconds, respectively.
(3) The percentage of improvements of PBWFM compared to PFWFM are given in Table 5.

Note: Since PFWFM, and PBWFM reduce the execution time for WFM, we execute PFWFM, and PBWFM on $m = 90$ using 24 threads only. The execution times for this case are 37, 242.3 and 2, 636.1 for PFWFM and PBWFM, respectively. In this case, the percentage of improvement for PBWFM is 92.9%.

6.2 The speedup and scalability

We measure the speedup of PFWFM and PBWFM from two viewpoints. The first viewpoint is when the problem size is fixed, while the second viewpoint is when the number of threads is fixed.

Case 1: The speedup of PFWFM.

Table 4 Percentage of improvements in time between sequential algorithms

m-bits	WFM-FWFM	WFM–BWFM	FWFM–BWFM
40	45.2%	83.87%	70.59%
50	48.7%	82.05%	65.0%
60	48.0%	93.02%	86.6%
70	48.8%	95.05%	90.34%
80	48.5%	95.68%	91.62%
Average	47.8%	89.9%	80.8%

Table 5 Improvements of PBWFM with respect to PFWFM

m-bit	Number of threads			
	6	12	18	24
40	23.8%	30.0%	17.9%	20.0%
50	79.6%	81.0%	76.9%	60.6%
60	82.6%	85.0%	76.9%	75.0%
70	88.4%	87.4%	84.7%	85.5%
80	87.8%	87.1%	84.7%	85.7%
Average	72.4%	74.1%	68.2%	65.4%

The speedup of PFWFM is measured based on FWFM. For the first viewpoint, Fig. 3 shows the speedup of PFWFM with a fixed problem size. The figure illustrates the following:

(1) The speedup of PFWFM increases with increasing the number of threads, except when $m = 40$, and 50, and the number of threads equals to 24.
(2) The speedup of PFWFM is small when n is small, $m = 40$ and 50 since the running time of PFWFM is less than one second.
(3) The speedup of PFWFM increases with increasing the size of inputs.
(4) The speedup of PFWFM is less than the number of threads used for $m \leq 70$. Therefore, the speedup of PFWFM is sublinear. On the other hand, the speedup of PFWFM is greater than the number of threads used for $m = 80$. This means that the speedup of PFWFM is superlinear in the case of $m = 80$.

For the second viewpoint, Figure 4 shows the speedup of PFWFM with fixed number of threads. The figure illustrates the following:

(1) The speedup for PFWFM increases with increasing the size of the problem, except when $m = 70$ and the number of threads equals to 6.
(2) The maximum speedup of PFWFM is 27 times of FWFM using 24 threads.

Case 2: The speedup of PBWFM.

Similarly, Figs. 5 and 6 show the speedup of PBWFM with fixed problem size and number of threads, respectively.

From the two cases related to the speedup, we found that PFWFM and PBWFM are scalable and the maximum speedups for PFWFM and PBWFM are 27 and 15.8 using 24 threads, see Figs. 4, and 6. Therefore, PFWFM and PBWFM have almost sublinear scalability, except when $m = 80$, the speedup of PFWFM algorithm has superlinear scalability.

7 BWFM with different wheel sizes

In this section, we study the effect of wheel size on the performance of BWFM. It is known that larger wheel removes more composites. For examples [24], if the wheel size equals $210 = 2 \times 3 \times 5 \times 7$, then WFM removes about 77% of the composites, while WFM removes about 90% of the composites if the wheel size equals the product of the primes up to 251. Table 6 shows that the execution time of BWFM decreases with increase in the wheel size. It shows that the execution time of BWFM decreases by about 13.5%, when we use wheel size equals to 210 instead of 30.

8 Conclusion

In this paper, we have addressed one of the challenging problems related to cryptanalysis of some public-key cryptosystems which is the integer factorization problem. The goal of integer factorization is to factor a composite number into prime factors. We have presented two approaches for improving the wheel factoring method, WFM. The first approach is based on introducing two modifications on WFM sequentially. The second approach is based on parallelizing the modified WFMs on a shared memory model. The implementation of the two approaches on composite integers of size less than or equal to 80-bits demonstrates significant improvements in the execution times for sequential and parallel proposed algorithms. The improvement percentage for BWFM over WFM is 90% on the average, while the improvement percentage for PBWFM over BWFM is 85% on the average. Finally, the scalability of the proposed parallel algorithm is almost sublinear.

Additionally, the performance of PBWFM increases with an increasing number of threads, so an important question raised from these results is how to use graphics

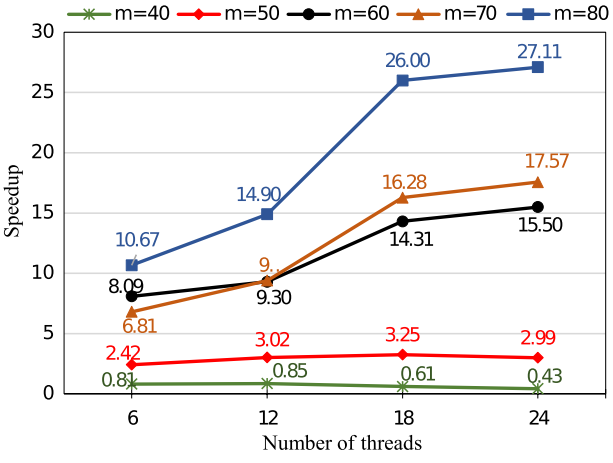

Fig. 3 Speedup of PFWFM with fixed problem size

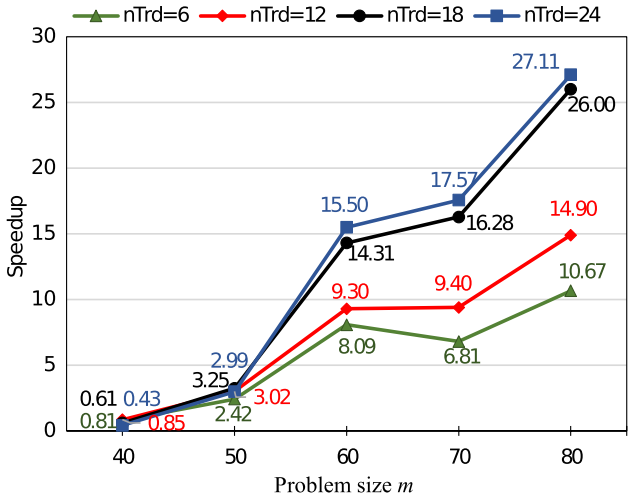

Fig. 4 Speedup of PFWFM with fixed number of threads

processing units (GPUs) for BWFM. We expect that the performance will increase dramatically, in particular when we use a large wheel and multi-GPUs. The main problem with using GPUs is that CUDA, which is the framework of NVIDIA for implementing techniques on GPUs, doesn't support large integers (say 128-1024 bits) until now.

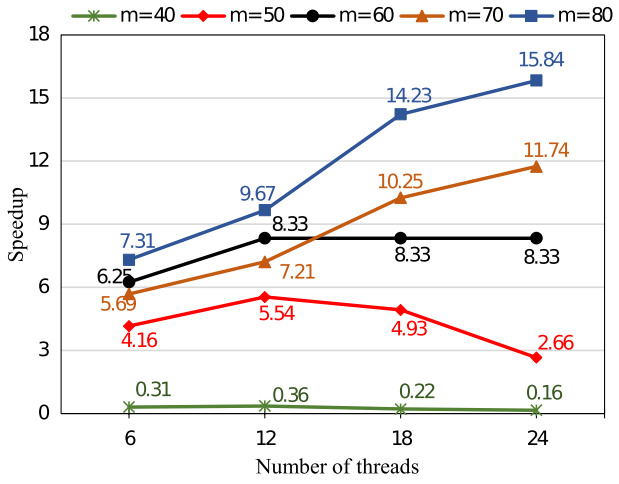

Fig. 5 Speedup of PBWFM with fixed problem size

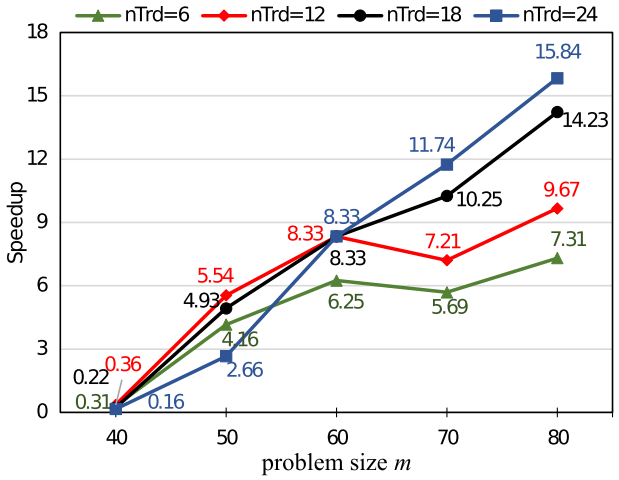

Fig. 6 Speedup of PBWFM with fixed number of threads

Table 6 Performance of BWFM with large wheel size

m-bits	BWFM with		Improvement
	Wheel size =30	Wheel size =210	
50	0.133	0.112	15.78%
60	2.5	2.2	12.0%
70	72.8	63.0	13.46%
80	2929.5	2518.76	14.02%

Acknowledgements This research has been funded by Scientific Research Deanship at University of Ha'il - Saudi Arabia through project number RG-21 124.

References

1. Akchiche O, Khadir O (2018) Factoring rsa moduli with primes sharing bits in the middle. Appl Algebra Eng. Commun. Comput. 29(3):245–259
2. Atanassov E, Georgiev D, Manev N (2014) Number theory algorithms on gpu clusters. 2:131–138
3. Bahig HM, Nassr DI, Bhery A (2017) Factoring rsa modulus with primes not necessarily sharing least significant bits. Appl Math Inform Sci 11:243–249
4. Bahig HM, Nassr DI, Bhery A, Nitaj A (2020) A unified method for private exponent attacks on rsa using lattices. Int J Found Comput Sci 31(2):207–231
5. Bahig HM, Mohammed A, Khaled A, AlGhadhban A, Bahig HM (2020) Performance analysis of fermat factorization algorithms. Int J Adv Comput Sci Appl 11(12):340–350
6. Bahig HM, Bahig HM, Kotb Y (2020) Fermat factorization using a multi-core system. Int J Adv Comput Sci Appl 11(4)
7. Brent RP (1999) Some parallel algorithms for integer factorisation. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), volume 1685 LNCS, pages 1–22
8. Durmuş O, Çabuk UC, Dalkiliç F(2020) A study on the performance of base-m polynomial selection algorithm using gpu

9. Fathy K, Bahig H, Farag M (2018) Speeding up multi- exponentiation algorithm on a multicore system. J Egypt Math Soc 26
10. Fathy KA, Bahig HM, Ragab AA (2018) A fast parallel modular exponentiation algorithm. Arab J Sci Eng 43
11. Fujioka A, Suzuki K, Xagawa K, Yoneyama K (2015) Strongly secure authenticated key exchange from factoring, codes, and lattices. Design Codes Cryptogr 76:469–504
12. GMP. library, gnu multiple precision arithmetic library. https://gmplib.org/
13. Gulida KR, Ultanov IR (2017) Comparative analysis of integer factorization algorithms using cpu and gpu. MANAS J Eng 5:53–63
14. Varadharajan S, Raddum H (2019) Factorization using binary decision diagrams. Cryptogr Commun 11:443–460
15. Koundinya AK, Harish G, Srinath NK, Raghavendra GE, Pramod YV, Sandeep R, Punith KG (2013) Performance analysis of parallel pollard's rho factoring algorithm. Int J Comput Sci Inform Technol 5
16. Lenstra AK (2000) Integer factoring. Designs Codes Cryptogr 19:101–128
17. Menezes AJ, Katz J, van Oorschot PC, Vanstone SA (1996) Handbook of Applied Cryptography. CRC Press
18. Montgomery PL (1994) A survey of modern integer factorization algorithms. CWI Quart 7:337–366
19. Nassr DI, Bahig HM, Bhery A, Daoud SS (2008) A new rsa vulnerability using continued fractions. In AICCSA 08 - 6th IEEE/ACS International Conference on Computer Systems and Applications, pages 694–701
20. Nimbalkar AB (2018) The digital signature schemes based on two hard problems: factorization and discrete logarithm. In: In: Bokhari M, Agrawal N, Saini D (eds) Cyber Security, volume 729 of Advances in Intelligent Systems and Computing, pages 493–498
21. OpenMP. https://www.openmp.org/
22. Peng WC, Wang BN, Hu F, Wang YJ, Fang XJ, Chen XY, Wang C (2019) Factoring larger integers with fewer qubits via quantum annealing with optimized parameters. Sci China Phys Mech Astron 62:60311
23. Poulakis D (2009) A public key encryption scheme based on factoring and discrete logarithm. J Discrete Math Sci Cryptogr 12:745–752
24. Pritchard P (1982) Explaining the wheel sieve. Acta Inform 17:477–485
25. Rivest RL, Shamir A, Adleman LM (1978) A method for obtaining digital signatures and public key cryptosystems. Commun ACM, pp 120–126
26. Rubinstein-Salzedo S (2018) Clever factorization algorithms and primality testing
27. Shor PW (1997) Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J Comput 26:1484–1509
28. Valenta L, Cohney S, Liao A, Fried J, Bodduluri S, Heninger N (2017) Factoring as a service. In: In: Grossklags J, Preneel B (eds) Financial cryptography and data security. FC 2016., volume LNCS 9603 of Lecture Notes in Computer Science, pp 321–338
29. Liangshun W, Cai HJ, Gong Z (2019) The integer factorization algorithm with pisano period. IEEE Access 7:167250–167259
30. Yan SY (2009) Primality testing and integer factorization in public-key cryptography, volume 11 of advances in information security. Springer
31. Yan SY (2019) Factoring Based Cryptography, pages 217–286
32. Yan SY, James G (2006) Can integer factorization be in p? In: 2006 International Conference on Computational Inteligence for Modelling Control and Automation and International Conference on Intelligent Agents Web Technologies and International Commerce (CIMCA'06), pp 266–266

Authors and Affiliations

Hazem M. Bahig[1,2] · Dieaa I. Nassr[2] · Mohammed A. Mahdi[1] · Mohamed A. G. Hazber[1] · Khaled Al-Utaibi[3] · Hatem M. Bahig[2]

Dieaa I. Nassr
dieaa.nassr@sci.asu.edu.eg

Mohammed A. Mahdi
m.mahdi@uoh.edu.sa

Mohamed A. G. Hazber
m.hazber@uoh.edu.sa

Khaled Al-Utaibi
alutaibi@uoh.edu.sa

Hatem M. Bahig
hmbahig@sci.asu.edu.eg

[1] Information and Computer Science Department, College of Computer Science and Engineering, University of Ha'il, Hail, Kingdom of Saudi Arabia

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

[3] Computer Engineering Department, College of Computer Science and Engineering, University of Ha'il, Hail, Kingdom of Saudi Arabia