Fermat Factorization using a Multi-Core System

Hazem M. Bahig[1*], Hatem M. Bahig[2], Yasser Kotb[3]
College of Computer Science and Engineering, University of Ha'il, Ha'il, Kingdom of Saudi Arabia[1]
Computer Science Division, Department of Mathematics, Faculty of Science, Ain Shams University, Egypt[1, 2, 3]
College of Computer and Information Sciences, Information Systems Department[3],
Imam Mohammad ibn Saud Islamic University, Kingdom of Saudi Arabia[3]

***Abstract*—Factoring a composite odd integer into its prime factors is one of the security problems for some public-key cryptosystems such as the Rivest-Shamir-Adleman cryptosystem. Many strategies have been proposed to solve factorization problem in a fast running time. However, the main drawback of the algorithms used in such strategies is the high computational time needed to find prime factors. Therefore, in this study, we focus on one of the factorization algorithms that is used when the two prime factors are of the same size, namely, the Fermat factorization (FF) algorithm. We investigate the performance of the FF method using three parameters: (1) the number of bits for the composite odd integer, (2) size of the difference between the two prime factors, and (3) number of threads used. The results of our experiments in which we used different parameters values indicate that the running time of the parallel FF algorithm is faster than that of the sequential FF algorithm. The maximum speed up achieved by the parallel FF algorithm is 6.7 times that of the sequential FF algorithm using 12 cores. Moreover, the parallel FF algorithm has near-linear scalability.**

Keywords*—Integer factorization; fermat factorization; parallel algorithm; multi-core

I. Introduction

The extensive use of digital systems has led to an increased need for information security. The main tool used to ensure the security of information is cryptography. In order to provide information security services, a set of cryptographic strategies is needed to convert plaintext into ciphertext. A set of such strategies is known as a cryptosystem. There are two main types of modern cryptosystems:- (1) public-key (asymmetric) cryptosystems such as the ElGamal digital signature scheme, Rivest-Shamir-Adleman (RSA) cryptosystem, Diffe-Hellman scheme and digital signature algorithm [1], and (2) private-key (symmetric) cryptosystems such as the advanced encryption standard algorithm [2].

The RSA cryptosystem is one of the important cryptosystems with security based on integer factorization problem, which is defined as follows: Given a positive integer n, the aim of the factorization of n is to find two positive integers (also known as factors) p_1 and p_2 such that n equals the product of p_1 and p_2, and $p_1, p_2 > 1$. In this case, n is called a composite integer. On the other hand, if n cannot be factored, then n is called a prime number. Thus, we can represent any positive integer as a unique product of prime factors.

In the RSA cryptosystem, the key is constructed by detecting two prime numbers p_1 and p_2 such that the size of each of them is large and approximately equal. The modulus for the key is defined as $n = p_1 p_2$. Then an encryption exponent e is chosen that is relatively prime to $\varphi(n) = (p_1 - 1)(p_2 - 1)$. Finally, the decryption exponent d is defined as $d \equiv e^{-1} \left(\bmod \varphi(n)\right)$.

The main challenge of factorization is the amount of time that is consumed to arrive at a solution, especially when the size of the prime factors is large. Also, there exists no deterministic polynomial algorithm to factor a composite number into two prime numbers.

A. High-Performance Computing

One of the strategies that can be utilized to reduce the high computational time needed by factorization methods is the high-performance computing (HPC). The main objective of using HPC is to design a parallel algorithm in running time, T_p, which is almost equal T_{seq}/p, where T_{seq} is the execution time of the problem using one processor and p is the number of processors used in the HPC. However, the achievement of this objective is not easy for several reasons such as the difficulty of dividing the problem into equal-sized, the communication between processors, and the dependences in some steps of the solution.

The effectiveness of the parallel algorithm can be measured using the speedup criteria. The speedup of a parallel algorithm is the ratio between the running time of the problem using one processor over the running time of the problem using p processors and is denoted by $S_p=T_{seq}/T_p$. The main goal of designing a parallel algorithm is to achieve linear speedup. Another important criteria for the parallel algorithm is scalability, which represents the parallel system's capacity to increase speedup in proportion to the number of processors.

Many hardware and software platforms have been introduced to measure parallel algorithms practically. Examples of parallel hardware are the cluster, multi-core, graphics processing unit (GPU) and cloud. There are also many different parallel programming languages or libraries such as open multi-processing (openMP), the message passing interface (MPI), and compute unified device architecture (CUDA).

B. State of the Art

Many integer factorization algorithms have been proposed based on a range of different strategies [1,3,4] such as trial division [5], Fermat [6], Brent, Pollard rho and p-1 [1,7], elliptic curves [8], Lehman's method [1,6], continued fraction [1,5], multiple polynomial quadratic sieve [9], and number

*Corresponding Author: h.bahig@uoh.edu.sa field sieve [9,10]. These algorithms can be categorized based on the properties of the numbers to be factorized into general-purpose and special-purpose algorithms [1,3].

The time complexity of the algorithms that belong to the general-purpose group is almost independent of the size of the factor found and depends on the size of n. Examples of some of the methods that belong to this group are Lehman's method, Shanks' square form factorization method, continued fraction, multiple polynomial quadratic sieves, and number field sieve. In the case of the algorithms that belong to the special-purpose group, the time complexity of the algorithms mainly depends on the size of the factor found. Examples of some of the methods belong to this group are the trial division, Fermat, Pollard rho, and Lenstra's elliptic curve methods.

In this study, we focus on the Fermat factorization (FF) algorithm, which is an efficient method when the difference between two factors is small. Many research studies have attempted to enhance this method from the sequential computation viewpoint [11,12,13,14]. However, from the parallel computation perspective, to our knowledge there is only one published paper on implementing the FF algorithm on a GPU, namely, the NVIDIA GeForce GT 630 [15]. Also, in this study, the experimental conducted to parallelize the FF algorithm on the GPU was based on a small input size of less than 60.

C. Study Outline

In this study, we show how to utilize HPC to speed up the computation of FF method. We use a multi-core platform that executes 12 threads concurrently to reduce the execution time of the FF algorithm. Also, we study the effect of using HPC when we increase the difference between the two primes, even of two primes of the same size. The results show that the proposed parallel FF algorithm improves execution time and that the maximum speed up achieved by parallelization is 6.7 times that of a sequential FF algorithm. Moreover, the parallelization of the proposed parallel FF algorithm shows near-linear scalability.

The rest of this paper is arranged as follows. In Section 2, we provide an overview of the FF algorithm, including the mathematical concept and pseudocode algorithm, as well as a complexity analysis and example. In Section 3, we introduce our proposed strategy for parallelizing the FF algorithm. Then, in Section 4 we present and discuss the results of our experimental evaluation according to execution time, speed-up and scalability. Finally, in Section 5, we present the conclusion of this work.

II. The FF Algorithm

In this section, first we introduce, briefly, the mathematical concept on which the FF algorithm is based. Second, we present the idea underpinning the FF algorithm as well as the pseudocode of the FF algorithm. Third, we provide a complexity analysis of the FF algorithm. Finally, we provide an illustrative example to show the effect of the difference between two primes on the performance of the FF algorithm.

A. Mathematical Concept

Assume that n is an odd integer of the form $n = p_1p_2$, where $p_1 > p_2 > 0$. Then the integer n can be formed as a subtraction of two squares q_1 and q_2, i.e., $n = {q_1}^2 - {q_2}^2$.

We can easily prove this statement by setting q_1 and q_2 as follows:

$$q_1 = \frac{p_1 + p_2}{2} \text{ and } q_2 = \frac{p_1 - p_2}{2}$$

Then,

$$n = {q_1}^2 - {q_2}^2$$
$$\Longrightarrow\ n = \left(\frac{p_1 + p_2}{2}\right)^2 - \left(\frac{p_1 - p_2}{2}\right)^2$$
$$\Longrightarrow\ n = p_1p_2$$

Also, $n = {q_1}^2 - {q_2}^2$ can be rewritten as follows:

$$n = {q_1}^2 - {q_2}^2 = (q_1 + q_2)(q_1 - q_2)$$

If the two values $(q_1 + q_2)$ and $(q_1 - q_2)$ are not equal to 1, then the two values are factors of n.

B. The Algorithm

The main idea of the algorithm is to search for two possible values q_1 and q_2 such that $n = {q_1}^2 - {q_2}^2$. We can rewrite the relation between n, q_1 and q_2 as ${q_2}^2 = {q_1}^2 - n$. So, if we know the value of q_1, we can find the value of q_2. Since the value of ${q_2}^2$ is a positive integer, this means that ${q_1}^2 > n$. So, the initial value of q_1 is $\lfloor\sqrt{n}\rfloor + 1$.

The idea of FF algorithm is to test iteratively, increasing by a value of 1, all values of q_1 beginning with $\lfloor\sqrt{n}\rfloor$ +1 until we detect a value of q_1 that satisfies the condition that ${q_1}^2 - n$ is a perfect square. In this case, the two factors are $(q_1 + q_2)$ and $(q_1 - q_2)$.

The complete pseudocode of the FF algorithm is as shown in Algorithm 1. The algorithm consists of three main steps. The first step is to compute the square root of n to determine the start value of q_1. The second step is an iterative step that increases the value of q_1 by 1 until the value ${q_1}^2 - n$ is a perfect square. At this point, the two factors are determined in the third step.

```
Algorithm 1: Fermat Factorization (FF)
Input: Composite odd integer $n$.
Output: two prime factors, $p_1, p_2 > 1$, such that $n = p_1\, p_2$.
1.    $q_1 \leftarrow \lfloor\sqrt{n}\rfloor$
2.    Do
          $q_1 \leftarrow q_1 + 1$
          $q_2 \leftarrow {q_1}^2 - n$
      While ($q_2$ is not a perfect square)
3.    $p_1 \leftarrow q_1 + \sqrt{q_2}$
      $p_2 \leftarrow q_1 - \sqrt{q_2}$
```

TABLE I. THE EFFECT OF THE DIFFERENCE BETWEEN TWO FACTORS ON FERMAT FACTORIZATION

n	p_1	p_2	α	$\lfloor\sqrt{n}\rfloor$	q_1	# of trials
16181393	4079 (12 bits)	3967 (12 bits)	6	4022	4023	1
15634807	4079 (12 bits)	3833 (12 bits)	7	3954	3955, 3956	2
14566109	4079 (12 bits)	3571 (12 bits)	8	3816	3817, … , 3825	9
12510293	4079 (12 bits)	3067 (12 bits)	9	3536	3537, …, 3573	37
8439451	4079 (12 bits)	2069 (12 bits)	10	2905	2906, … , 3074	169

C. Complexity Analysis

The best case of the FF algorithm occurs when the two factors are close together. This means that the value of $q_2 = \frac{p_1-p_2}{2}$ is small and the value of q_1 is slightly greater than $\sqrt{n}$. Therefore, the number of iterations in the second step is small.

The worst case of the FF algorithm can be calculated as follows. Assume that the minimum value of $q_1 - q_2$ is δ. This implies that:

$n = (q_1 + q_2)(q_1 - q_2) = (q_1 + (q_1 - \delta)) \times \delta = (2q_1 - \delta) \times \delta$. Therefore,

$n = 2q_1\delta - \delta^2 \implies q_1 = \frac{n-\delta^2}{2\delta}$.

If δ =3, for large primes, then $q_1 = \frac{n+9}{6}$.

In general, the performance of FF algroithm is based on the difference between the two prime factors, and can be given by the following rule [16]:-

$$\mathcal{O}\left(\frac{|p_1-p_2|^2}{4\sqrt{n}}\right)$$

In case of $|p_1-p_2| = \mathcal{O}(\sqrt[4]{n})$, the FF solution can be found easily [16].

D. Example

Table I shows that the main step of FF algorithm, i.e., Step 2, is affected by the difference between the prime factors even when the two factors are of the same size. The table consists of seven columns. The first three columns are related to the numbers to be factor and their factorization, n, p_1, and p_2. The two prime factors have sizes of 12 bits each, but they have different values. The fourth column, α, represents the number of bits in the difference between two factors, Δ. The relation between α and Δ is $2^{\alpha} \leq \Delta < 2^{\alpha+1}$. The fifth and sixth columns represent the square root of n and all the trail values of q_1, respectively. The last column represents the number of iterations in the second step of FF method.

For all the values of n, the number of bits is $l = 24$, and the number of bits for each factor is $\frac{l}{2} = 12$. In the first row, the number of bits in the difference between two factors is $\frac{l}{4} = 6$. The number of bits in the difference between two factors is increased by 1 in each next row. It is clear from Table I, that when the difference between two factors increases, the number of iterations in the main step (Step 2) of the FF algorithm also increases.

III. PARALLEL FF ALGORITHM

In this section, we present the mechanism that is used to parallelize FF method. The FF algorithm can be considered as a searching algorithm over the range from $\lfloor\sqrt{n}\rfloor + 1$ to $\frac{n+9}{6}$. Therefore, the proposed approach to parallelize the FF method is based on assigning the first t integers to t threads, such that each thread, t_i, takes one integer. This means that integers $\lfloor\sqrt{n}\rfloor + 1$, $\lfloor\sqrt{n}\rfloor + 2$, …, $\lfloor\sqrt{n}\rfloor + t$ are assigned to threads t_1, t_2, …, t_t, respectively. If the target goal is not found by any thread, then the second t integers, $\lfloor\sqrt{n}\rfloor + t + 1$, $\lfloor\sqrt{n}\rfloor + t + 2$, …, $\lfloor\sqrt{n}\rfloor + 2t$, are assigned to t threads t_1, t_2, …, t_t, respectively. This process continues dynamically until a thread finds a value of q_{2_i} and satisfies the condition that q_{2_i} is a perfect power.

In general, the assignment of integer, q_{1_i}, to thread t_i is given by the following formula:

$$q_{1_i} = \lfloor\sqrt{n}\rfloor + (j-1)\, t + i$$

where j represents the jth t integers, $j \geq 1$, and $1 \leq i \leq t$.

All the steps in this parallelization method are given by Algorithm 2. The first step of the algorithm is a sequential steps that are used to (1) determine the value of the square root that is used by all threads, and (2) assign the shared variable found with false. The second step is a parallel step that is executed by all threads, where each thread i, $1 \leq i \leq t$, has two local variables, q_{1_i} and q_{2_i}. This step consists of three substeps, 2.1, 2.2, and 2.3. Substep 2.1 is used to assign initial values for *j* (iteration number) and q_{1_i}. Substep 2.2 is used to update the value of q_{1_i} and q_{2_i} if the value of q_{2_i} is still not a perfect square or no other thread has found the solution. Finally, in Substep 2.3 the thread that has found the solution, i.e., q_{2_i} that is a perfect square, changes the value of *found* from false to true and then calculates the two factors p_1 and p_2.

In order to improve the performance of Algorithm 2, we applied the following modifications. First, in order to be able to read a shared value between all threads, for each shared value between threads, we used a local variable instead of the shared value, except at the beginning of executing each thread. Also, for the shared value *found*, we used a shared array *Ok* of t elements of Boolean type. We also changed the second condition in the While-loop in Substep 2.2, to *Ok*[*i*]. Second, we implemented a modification to enable writing on a shared variable. This occurs when thread *j* has found the solution. In this case, thread *j* is responsible to changing all values of *Ok* using the critical region command. The complete steps of the modified algorithm are shown in Algorithm 3.

```
Algorithm 2: Parallel Fermat Factorization (PFF)
Input: Composite odd integer $n$.
Output: Two prime factors $p_1, p_2 > 1$, such that $n = p_1 p_2$.
1.    $q_1 \leftarrow \lfloor\sqrt{n}\rfloor$
      found =false
2.    for $i \leftarrow 1$ to $t$ do parallel
2.1       $j \leftarrow 0$
          $q_{1_i} \leftarrow q_1 + i$
          $q_{2_i} \leftarrow {q_{1_i}}^2 - n$
2.2       while ($q_{2_i}$ is not a perfect square) and (not found) do
              $q_{1_i} \leftarrow q_{1_i} + t$
              $q_{2_i} \leftarrow {q_{1_i}}^2 - n$
2.3       if ($q_{2_i}$ is a perfect square) then
              found =True
              $p_1 \leftarrow q_{1_i} + \sqrt{q_{2_i}}$
              $p_2 \leftarrow q_{1_i} - \sqrt{q_{2_i}}$
```

Note: There is another approach that can be used to parallelize the range search, R for FF algorithm. This approach is based on dividing the search range into t, number of threads, subranges. Each thread t_i, $1 \leq i \leq t$, searches subrange, R_i, which is defined as follows.

$$\left[q_1 + 1 + (i-1)\frac{R}{t}, q_1 + i\frac{R}{t}\right]$$

```
Algorithm 3: Modified Parallel Fermat Factorization (MPFF)
Input: Composite odd integer $n$.
Output: Two prime factors $p_1, p_2 > 1$, such that $n = p_1 p_2$.
1.    $q_1 \leftarrow \lfloor\sqrt{n}\rfloor$
2.    for $i \leftarrow 1$ to $t$ do parallel
2.1       $j \leftarrow 0$
          $Ok_i \leftarrow false$
          $n_i \leftarrow n$
          $t_i \leftarrow t$
          $q_{1_i} \leftarrow q_1 + i$
          $q_{2_i} \leftarrow {q_{1_i}}^2 - n_i$
2.2       while ($q_{2_i}$ is not a perfect square) and (not $Ok_i$) do
              $q_{1_i} \leftarrow q_{1_i} + t_i$
              $q_{2_i} \leftarrow {q_{1_i}}^2 - n_i$
2.3       if ($q_{2_i}$ is not a perfect square) then
              for $i \leftarrow 1$ to $t$ do // critical region
                  $Ok_i \leftarrow true$
              $p_1 \leftarrow q_{1_i} + \sqrt{q_{2_i}}$
              $p_2 \leftarrow q_{1_i} - \sqrt{q_{2_i}}$
```

Thread t_i starts the search with $q_{1_i} = \lfloor\sqrt{n}\rfloor + 1 + (i-1)\frac{R}{t}$ and tries to find the value of q_{2_i} satisfying the condition that q_{2_i} is a perfect power. If thread t_i finds the target goal, q_{2_i} is a perfect power, then the shared variable, *found*, is changed from false to true. This means that all the other threads stop searching if one of the threads changes the variable *found* to true.

In general, this approach is not efficient for two factors of the same size. For example, referring to Table I, consider $n = 4079 \times 2069 = 8439451$, and let the number of threads $t = 8$. The range of the search is $[2905, 1406576]$ and therefore the range of the search for each thread is approximately 175822. The first thread will therefore find the solution after 169 iterations. In contrast, by using Algorithm 2, the solution can be found after just 22 iterations.

IV. Experimental Evaluations

In this section, we present the procedures and the results of our evaluations of the impact of the suggested parallel approach on the FF method according to the following three parameters: (1) the number of bits for the composite odd integer, (2) size of the difference between two prime factors, and (3) number of threads used. To achieve these goals, the section involves two subsections. The first subsection provides the configurations of the platform and data used in the experiments. The second provides the measurement and analysis of the running times and the scalability of the suggested parallel method.

A. Platform and Data Setting

The platform settings in the experiments are based on the configurations shown in Table II.

The experiments on all the studied algorithms are based on three parameters. The first two parameters are related to the generation of two prime numbers, p_1 and p_2, of the same size to construct a composite odd number $n = p_1 p_2$. The first parameter is the number of bits for the integer n, which is l. This means that the number of bits for each prime factor, p_1 and p_2, is $\frac{l}{2}$. The second parameter is the difference between the two prime factors, which is $\Delta = |p_1 - p_2|$, $2^\alpha \leq \Delta < 2^{\alpha+1}$, where $\alpha < \frac{l}{2} - 1$. This means that a prime factor p_1 of size $\frac{l}{2}$ is generated, the size of the second prime factor generated is $\frac{l}{2}$ such that the difference between them is Δ and $2^\alpha \leq \Delta < 2^{\alpha+1}$, for a certain value of α. The setting of these two parameters is shown in Table III. The maximum value of α is $\frac{l}{2} - 2$ in order to ensure that the two prime factors are the same size. The minimum value of α is $\frac{l}{2} - 15$, because this value is near to $\frac{l}{4}$, for the studied cases. Also, if α is less than $\frac{l}{2} - 15$, for the studied cases, the running time of the algorithms tends to be toward zero. The third parameter is the number of cores, t, used in the experiments and the values of t are 4, 8, and 12.

In the experiments, we initially fix the value of l, say $l = 80$, and then generate two prime numbers, each of size $\frac{l}{2}$ such that the difference between them is Δ, say $\Delta = \frac{l}{2} - 5$. We repeat the same process to generate 25 different data, $d_i, 1 \leq i \leq 25$, for the same values of l and Δ. After that, we run Algorithm A on d_i using a fixed number of cores, t_j. Therefore, the running time for Algorithm A using t_j cores is the average of the running times of Algorithm A on 25 instances. In the case of $l = 100$, we run the FF algorithm only one time, because the running time is very large (see Table IV). Also, in this case, we run the parallel FF algorithm using t threads for five instances only.

In general, the running time for Algorithm A is computed using the three parameters as follows: For each fixed value of l, Δ, and t, we measure the running time of Algorithm A by executing Algorithm A on 25 different instances and then compute the average of these running times in seconds.

In addition, for the fixed value of l, we have 12 values for the running time of Algorithm A, and each of them is the average time for 25 instances. These 12 values come from all the combinations of four values of Δ $\left(\frac{l}{2} - 15, \frac{l}{2} - 10, \frac{l}{2} - 5, \text{and } \frac{l}{2} - 2\right)$ and three values of t $(4, 8, \text{and } 12)$.

B. Discussion of the Results

Based on the platform and data settings described in the previous subsection, the running times of Algorithm 1 (sequential FF algorithm) and Algorithm 3 (parallel FF algorithm) are shown in Table IV. The table consists of six columns. The first column represents the number of bits for the composite odd integer n, while the second column represents the number of bits for the difference between the factors. The third column represents the running time for the sequential FF algorithm, Algorithm 1. The fourth to sixth columns represent the running time for the parallel FF algorithm, Algorithm 3, using 4, 8, and 12 threads, respectively.

TABLE II. HARDWARE AND SOFTWARE CONFIGURATIONS

Type of Platform	**Components**	**Description**
Hardware	Processor	2 hexa-core (12 cores)
	Speed	2.6 GHz
	Memory	16 GB
	Cash Memory	15 MB
Software	Operating System	Windows 10
	Language	C++
	Parallel Library	OpenMP (Open Multi-Processing)
	Big Integer Library	GMP (GNU Multiple Precision)

TABLE III. PARAMETER SETTINGS FOR l AND Δ

l	$\frac{l}{2}$	$2^{\alpha} \leq \Delta < 2^{\alpha+1}$			
		α			
		$\frac{l}{2} - 15$	$\frac{l}{2} - 10$	$\frac{l}{2} - 5$	$\frac{l}{2} - 2$
70	35	20	25	30	33
80	40	25	30	35	38
90	45	30	35	40	33
100	50	35	40	45	48

TABLE IV. RUNNING TIME FOR SEQUENTIAL AND PARALLEL FF ALGORITHMS

l	α	**Number of threads**			
		1	**4**	**8**	**12**
70	$(l/2) - 15 = 20$	0	0.0002	0.0004	0.0004
	$(l/2) - 10 = 25$	0.0022	0.0008	0.0006	0.0006
	$(l/2) - 5 = 30$	1.1628	0.4412	0.303	0.201
	$(l/2) - 2 = 33$	57.9	19.1	13.7	9.8
80	$(l/2) - 15 = 25$	0.0002	0.0004	0.0002	0.0002
	$(l/2) - 10 = 30$	0.0544	0.0206	0.017	0.0106
	$(l/2) - 5 = 35$	35.7	10.7	8.8	6.4
	$(l/2) - 2 = 38$	524.1	146.5	113.5	80.1
90	$(l/2) - 15 = 30$	0.002	0.0007	0.0006	0.0004
	$(l/2) - 10 = 35$	1.158	0.404	0.271	0.216
	$(l/2) - 5 = 40$	796.3	248.3	185.7	117.5
	$(l/2) - 2 = 43$	19348.7	6046.5	4398.5	3064.5
100	$(l/2) - 15 = 35$	0.0564	0.0224	0.014	0.01
	$(l/2) - 10 = 40$	42.7	13.7	10.1	6.9
	$(l/2) - 5 = 45$	18695.7	5665.4	3995.8	2948.2
	$(l/2) - 2 = 48$	873041.6	253055.5	174608.3	130956.2

From results of the analysis of the running times of the two algorithms, 1 and 3, using different factors shown in Table IV, several observations can be made. First, in respect of the sequential FF algorithm, Algorithm 1:

1) The running time of the sequential FF algorithm increases with increased difference between the two prime factors. This means that, for a fixed value of l, the running time of the FF algorithm when $\alpha = \alpha_1$ is less than when $\alpha = \alpha_2$, where $\alpha_1 < \alpha_2$. For example, when $l = 80$, $\alpha_1 = 35$, and $\alpha_2 = 40$, the running times of the sequential FF algorithm are 0.05 and 35.7 seconds, respectively.

2) For a fixed value of l and two different values of α, α_1 and α_2, the difference in the running time of the FF algorithm between α_1 and α_2 is significant.

3) The minimum and maximum running times of the sequential FF algorithm occur when the values of α are a minimum of $\frac{l}{4}$, and a maximum of $\frac{l}{2} - 2$, respectively.

Second, in respect of the running time of the parallel FF algorithm, Algorithm 3:

1) The running time of the parallel FF algorithm decreases with an increase in the number of threads. This means that for fixed values of l and α, the running time of the parallel FF algorithm using t threads is less than the running time for the same instance using t' threads, where $t > t'$. As an example, for $l = 80$ and $t =$4, 8, and 12, the running times of the parallel FF algorithm are 10.7, 8.8, and 6.4, respectively.

2) The running time of the parallel FF algorithm is faster than the running time of the sequential FF algorithm using any number of threads, $t \geq 4$, except when the running time for FF algorithm is near to zero. In this case, when $l = 70$ and

$\alpha = 20$, the parallelization approach is not efficient in terms of running time because the search range is very small.

3) For fixed values of l and α, the running time of the parallel FF algorithm is different from one instance to another. This is because the range of Δ is large for a large value of α. As an example, Fig. 1 shows the running time of the parallel FF algorithm on 25 different instances using four threads for the case of $l = 80$ and $\alpha = 35$.

4) The amount of improvement in the parallel FF algorithm, using t threads, with respect to the FF algorithm is greater than the improvement in the parallel FF algorithm using t' threads, $t > t'$, see Fig. 2. For example, in the case of $l = 90$ and $\alpha = 40$, the amount of improvement in the parallel FF algorithm using four threads is 68.8%, whereas the amount of improvement increases to 76.6% using eight threads.

Third, we also measured the speedup of the parallel FF algorithm based on two viewpoints: (1) fixed values of l and α, and (2) fixed values of l and t.

1) Fig. 3 shows the speedup values with fixed l and α, and varied values of t, from which it can be observed that the speedup of the parallel FF algorithm increases with increased t. This is true for every l and α studied except when $l = 70$ and $\alpha = 20$, because the running time of the FF algorithm at these values is zero. For example, when $l = 90$ and $\alpha = 40$, the speedup values of the parallel FF algorithm using $t = 4$, 8, and 12 are 3.2, 4.3, and 6.8, respectively. In addition, in general, the speedup value equals, approximately, half of the number of threads.

2) Fig. 4 shows the speedup values with fixed l and t, and varied values of α, from which it can be observed that the speedup of the parallel FF algorithm increases, slightly, with increased α. This means that for a fixed problem size and number of threads, the speedup value of the parallel FF algorithm increases, even slightly, with an increase in the difference between two prime factors. For example, when $l = 80$ and $t = 12$, the speedup values of the parallel FF algorithm are 1, 5.1, 5.6, and 6.5 for α =25, 30, 35, and 38, respectively.

In general, the maximum speedup achieved by the parallel FF algorithm was 6.7 times greater than that achieved by the FF algorithm. Moreover, the parallel FF algorithm had near-linear scalability.

Fourth, Fig. 5 shows the efficiency of the parallel FF algorithm in the case of $l = 100$ and different values of α. The maximum efficiency value achieved when the number of threads equals four.

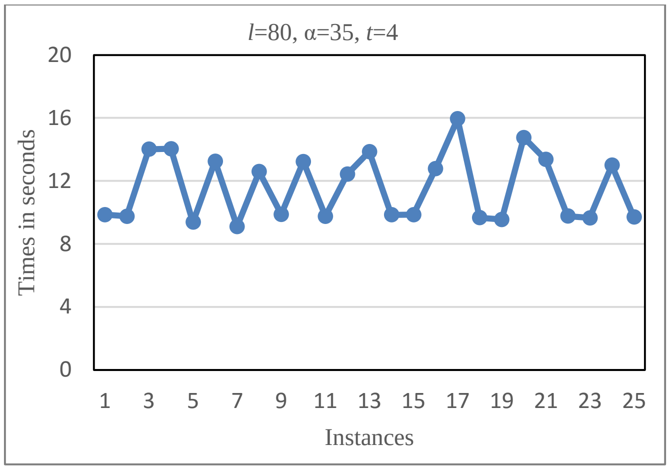

Fig. 1. Running Time of the Parallel FF Algorithm Over different instances.

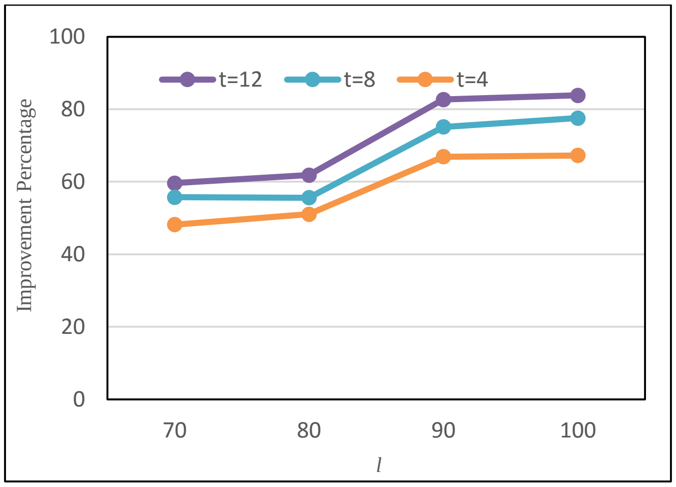

Fig. 2. Percentage of Improvements for the Parallel FF Algorithm.

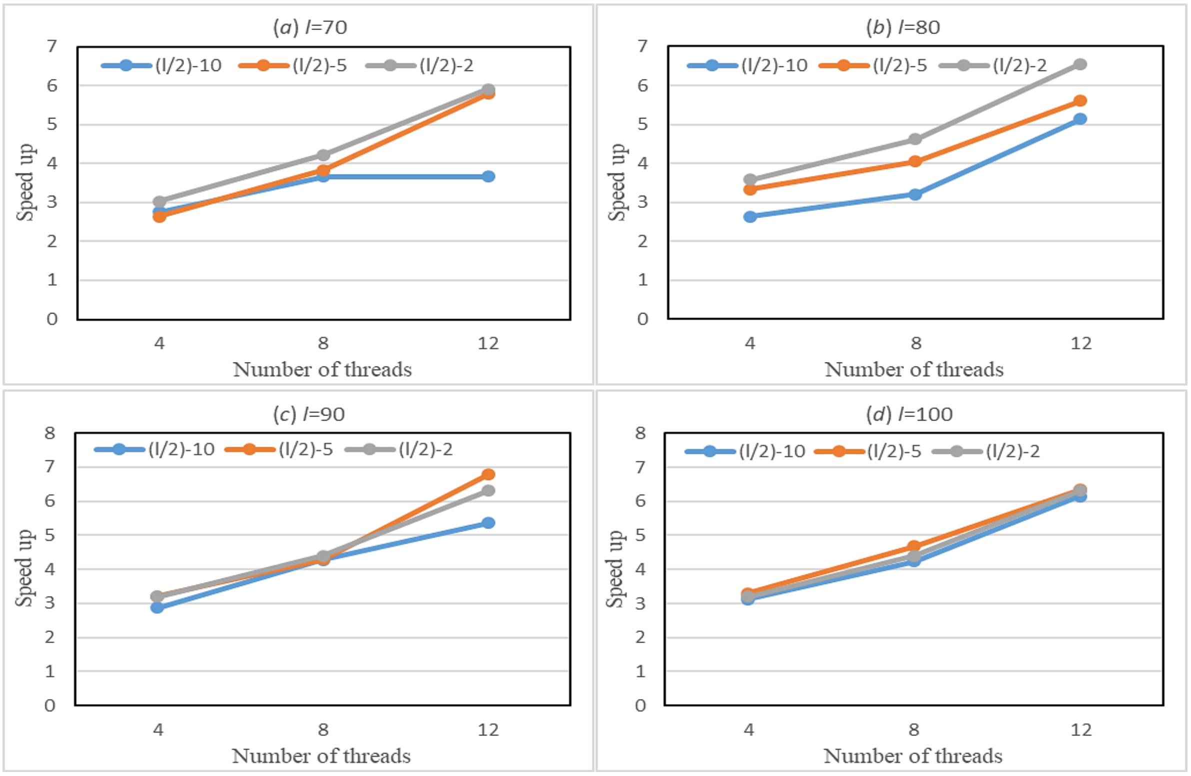

Fig. 3. Scalability of the Parallel FF Algorithm with Fixed l and α.

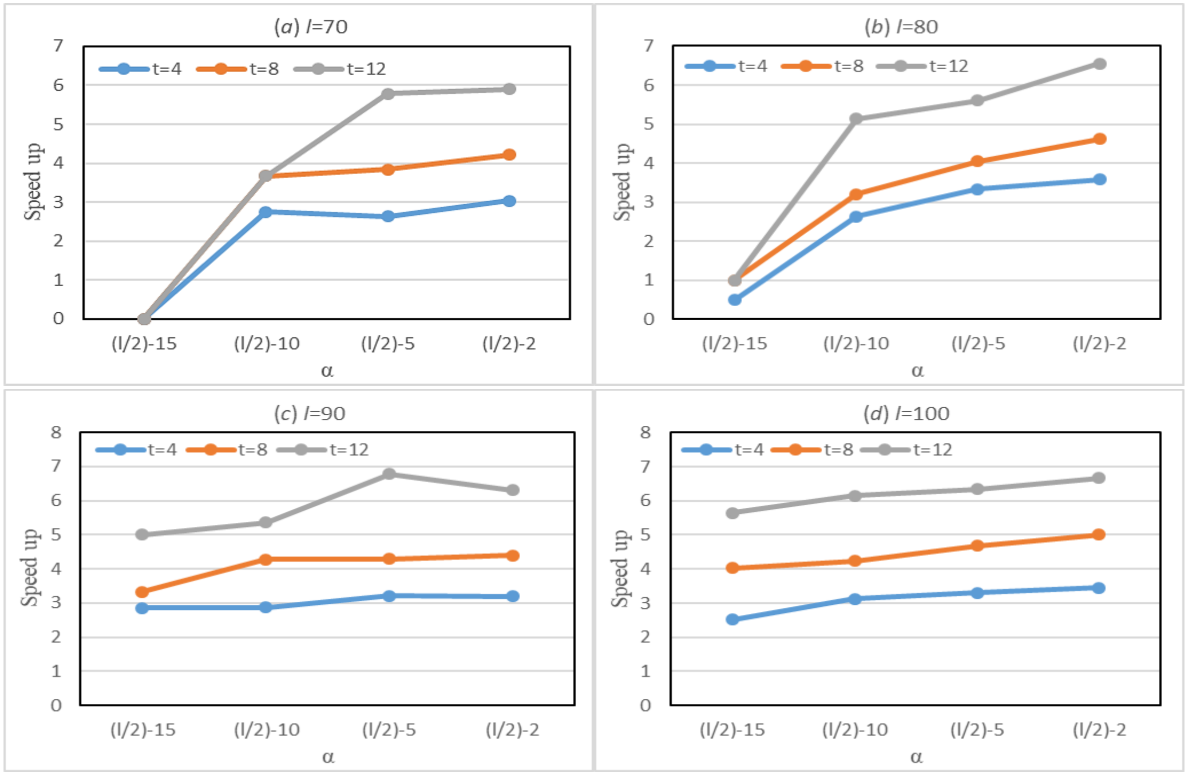

Fig. 4. Scalability of the Parallel FF Algorithm with Fixed l and t.

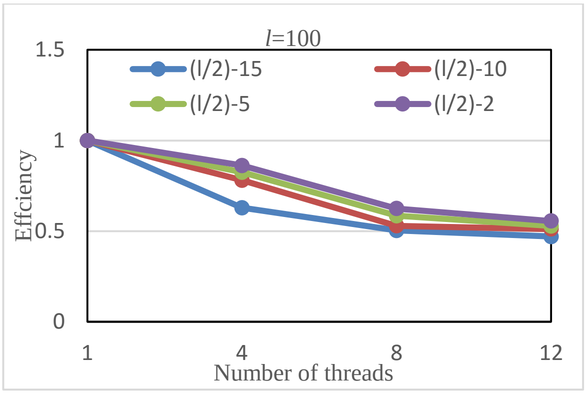

Fig. 5. Efficiency of the Parallel FF Algorithm.

V. Conclusion

In this study, we addressed one of the challenging problems related to cryptography, namely, integer factorization. The goal of integer factorization is to factor a composite number into two prime factors. The FF algorithm is one of the factorization algorithms that is used when the two factors are the same size. We investigated the use of a multi-core system on the performance of the FF method based on three parameters: (1) the number of bits for the composite positive integer, (2) size of the difference between two prime factors, and (3) number of threads used. The experimental results showed that the running time for the parallel FF algorithm was faster than that of the FF algorithm. The maximum speedup achieved by the parallel algorithm was 6.7 times that of the sequential FF algorithm. Moreover, the parallel FF algorithm had near-linear scalability.

There are still some interesting open questions related to FF algorithm such as (1) how to use GPUs to parallelize FF algorithm, (2) how to reduce the running time of FF algorithm when the difference between the two prime factors is large, and (3) how to use FF algorithm in internet of things [17].

Acknowledgment

This research has been funded by Scientific Research Deanship at University of Ha'il – Saudi Arabia through project number 0161034.

References

[1] S. Yan, Primality testing and integer factorization in public-key cryptography, 2009, Springer.

[2] J. Alfred ,Menezes , Paul C. van Oorschot , Scott A. Vanstone . Handbook of Applied Cryptography (Discrete Mathematics and Its Applications) 1st Edition CRC Press; 1996.

[3] A. Lenstra, Integer factoring. Designs, Codes and Cryptography, 19(2–3), 101–128 , 2000.

[4] J. Milan, Factoring small integers: an experimental comparison. INRIA report (2007), http://hal.inria.fr/inria-00188645/en/

[5] G. Hiary, A deterministic algorithm for integer factorization Mathematics of Computation 85, 2065-2069, 2016.

[6] P. Lehman, R. Sherman, Factoring large integers. Mathematics of Computation, 28, 637-646, 1974.

[7] J. Pollard, A Monte Carlo method for factorization. BIT Numerical Mathematics, 15, 331–334, 1975.

[8] H. Lenstra, Factoring integers with elliptic curves. Mathematische Annalen, 126, 649–673, 1987.

[9] A. Lenstra, H. Lenstra, The Development of the Number Field Sieve, in: Lecture Notes in Mathematics 1554, 1993.

[10] R. Silverman, Optimal parameterization of SNFS. Journal Mathematical Cryptology, 1, 105–124, 2007.

[11] J. McKee, Speeding Fermat's factoring method. Mathematics of Computation, 68, 1729-1737, 1999.

[12] K. Somsuk, MVFactorV2: An improved integer factorization algorithm to speed up computation time. 2014 International Computer Science and Engineering Conference, 30 July-1 Aug. 2014, pp 308-311.

[13] K. Somsuk, The improvement of initial value closer to the target for Fermat's factorization algorithm. Journal of Discrete Mathematics Science and Cryptography, 21(7), 1573-1580.

[14] M. Wu, R. Tso, H. Sun, On the improvement of Fermat factorization using a continued fraction technique. Future Generation Computer Systems, 30 (2014) 162–168.

[15] G. Kimsanova, R. Ismailova, R. Sultanov, Comparative analysis of integer factorization algorithms using CPU and GPU. MANAS Journal of Engineering, 5(1), 53-63, 2017.

[16] B. De Weger, Cryptanalysis of RSA with small prime difference. Applicable Algebra in Engineering, Communication and Computing, 13(1):17–28, 2002.

[17] S. Venkatraman, A. Overmars. New Method of prime factorisation-based attacks on RSA authentication in IoT. Cryptography 2019, 3(3), 20.