Speeding up the Multiplication Algorithm for Large Integers

Hazem M. Bahig
Computer Science and Information Dept.
College of Computer Science and Engineering, University of Ha'il
Ha'il, Saudi Arabia
h.bahig@uoh.edu.sa

Amer Alghadhban
Electrical Engineering Dept.
College of Engineering
University of Ha'il
Ha'il, Saudi Arabia
a.alghadban@uoh.edu.sa

Mohammed A. Mahdi
Computer Science and Information Dept.
College of Computer Science and Engineering, University of Ha'il
Ha'il, Saudi Arabia
m.mahdi@uoh.edu.sa

Khaled A. Alutaibi
Computer Engineering Dept.
College of Computer Science and Engineering
University of Ha'il
Ha'il, Saudi Arabia
alutaibi@uoh.edu.sa

Hatem M. Bahig
Mathematics Department
Faculty of Science
Ain Shams University
Cairo, Egypt
h.m.bahig@gmail.com

***Abstract*-Multiplication is one of the basic operations that influence the performance of many computer applications such as cryptography. The main challenge of the multiplication operation is the cost of the operation as compared to other basic operations such as addition and subtraction, especially when the size of the numbers is large. In this work, we investigate the use of the window strategy for multiplying a sequence of large integers to design an efficient sequential algorithm in order to reduce the number of bit-multiplication operations involved in multiplying a sequence of large integers. In our implementation, several parameters are considered and measured for their effect on the proposed algorithm and the best-known sequential algorithm in the literature. These parameters are the size of the sequence of integers, the size of the integers, the size of the window, and the distribution of the data. The experimental results prove the effectiveness of the proposed algorithm are compared to the ones of the best-known sequential algorithm, and the proposed algorithm is able to achieve a reduction in computing time greater than 50% in most cases.**

Keywords-multiplication; big data; cryptography; algorithm performance; computer arithmetic

I. INTRODUCTION

Computer arithmetic plays an essential role in every layer of computing and it is an important consideration when developing computer solutions for many problems such as cryptography, image processing, and numerical computations. In computer arithmetic, we use different operations such as addition, subtraction, multiplication, and division to achieve the goal of computation. Among these, the operation that has particular significance for many applications is the multiplication operation [1]. The multiplication operation is important, mainly for three reasons. First, the time cost of performing the multiplication operation is greater than that of other operations such as addition and subtraction. For example, given two integer numbers of n-bits each, the addition and multiplication of the two integer numbers require $O(n)$ and $O(n^2)$ bit operations, respectively, using the Naïve method [1]. This means that there is a significant difference between the costs of the two operations. Second, many primitive and essential arithmetic operations, such as division, squaring, inverse multiplication, and modulo operations, are based on the multiplication operation. Therefore, the running time of the multiplication operation affects these operations. Third, several complex applications in computer science, such as cryptography and digital signal processing, are based on a huge number of multiplication operations [2-6]. For example, in RSA and El-Gamal public-key cryptosystem, the multiplication operation is necessary. So, a more efficient multiplication method would lead to the speeding up of the computation process in complex applications.

For the above reasons, different strategies have been suggested to reduce the total number of operations required for multiplication. Two main research directions have been followed to improve the efficiency of the multiplication operation on a data set that consists of m integer numbers each of size n. The first direction has been to reduce the cost of multiplying two numbers, x and y, of size n each and thereby decrease the total cost of the multiplication operations for a data set. The second direction has been to reduce the cost of multiplying the data set by proposing an efficient strategy to multiply the m numbers. Regarding the first research direction, many methods have been proposed to reduce the time complexity of multiplying two integers in both sequential [7-23] and parallel computation [24-29]. In the case of sequential computation, several techniques have been proposed such as the Naïve multiplication algorithm [1], Karatsuba's algorithm

Corresponding authors: Hazem M. Bahig

[1,15], the Toom–Cook multiplication algorithm [20], and a fast Fourier transform-based algorithm [18]. The time complexity of the Naïve multiplication method is $O(n^2)$, whereas Karatsuba's algorithm uses a divide and conquer strategy to multiply the two integers in $O(n^k)$, where $k = \log_2 3 \approx 1.585$. The Toom–Cook multiplication method is a generalization of Karatsuba's algorithm using *r*-way multiplication and has a cost of $O\left(n^{1+O(1/\sqrt{\log n})}\right)$. In contrast, Schonhage and Strassen utilize the fast Fourier transform to reduce the time complexity of multiplication. They propose two algorithms, where the best one runs on $O(n \log n \log\log n)$ and uses an arithmetical modulo operation. Another two algorithms are proposed to reduce the running time of the Schonhage-Strassen algorithm to achieve $O\left(n \log n \; 2^{O(\log^* n)}\right)$, where $\log^* n = minimum\{i: log^{(i)} n \leq 2\}$ and $log^{(0)} n = n$. The first algorithm is based on arithmetic over complex numbers [12], while the other is based on modular arithmetic [10].

On the other hand, many different attempts have been made to parallelize the multiplication problem using different parallel models [24-29]. Most of these attempts have been based on the shared memory model, where the processors in this model communicate through shared memory. Also, some research studies have focused on implementing some parallel algorithms on real machines, such as FPGAs, GPUs, and multicore [25-27]. In the case of the second research direction, a few algorithms have been proposed to reduce the time complexity of the multiplication of m integers in both sequential and parallel computation. In the case of sequential computation, the best-known algorithm is the Naïve method, which scans the sequence of integer numbers and multiplies one number in each iteration. In the case of parallel computation, a few strategies [30] have been proposed that use a shared memory model and are implemented on specific real machines such as the multicore system [31].

In this paper, we are interested in contributing to the second research direction that focuses on using sequential computation because, despite the progress that has been made toward developing an effective strategy, there is still room for improvement. Here, we present an efficient improvement sequential algorithm to multiply a large number of integers, each of large size. A comparison of the proposed algorithm and the best-known sequential algorithm indicates that, from a practical perspective, the proposed algorithm performs better. Our improved algorithm speeds up the best-known sequential algorithm when m and n are large.

II. THE OPTIMAL ALGORITHM

In this section, first we describe the optimal sequential algorithm that is used to multiply m numbers, $X = (x_0 x_1 \dots x_{m-1})$. Then we give an example to illustrate the number of multiplication operations required to multiply m numbers.

A. The Optimal Multiplication Algorithm

The optimal multiplication algorithm, denoted as OM, multiplies m numbers by sequentially scanning the input array of $m-1$ iterations. In each iteration i, the algorithm multiplies the number x_i with M, where $i \geq 1$ and M is initially equal to x_0 and finally $M = \prod_{j=0}^{m-1} x_j$. In the light of the above, the algorithm requires $O(m\, t_{mul})$ time. The term $m$is is derived from $m-1$ iterations, where $O(m) = O(m-1)$, while the term t_{mul} is the total number of operations required to multiply two numbers. The value of t_{mul} depends on the size of the two numbers. If we have two numbers of size *n* each:

$$x_i = \left(x_{i(n-1)}, x_{i(n-2)}, \dots, x_{i1}, x_{i0}\right) \text{ and}$$
$$x_j = \left(x_{j(n-1)}, x_{j(n-2)}, \dots, x_{j1}, x_{j0}\right),$$

then multiplying x_i and x_j requires $O(n^2)$ multiplication operations using the Naïve method, while the best-known time complexity for t_{mul} is $O\left(n \log n \; 2^{O(\log^* n)}\right)$ [10, 12]. To differentiate between the multiplication operation for the two integers x_i and x_j, and the multiplication operation for the two digits/bits x_{il} and x_{ik}, we name the second type of operation as the digit-multiplication or bit-multiplication operation. So, the multiplication of x_i and x_j requires $O(n^2)$ digit-multiplication or bit-multiplication operations. The main problem when multiplying m numbers, each of size n bits, is the size of the result of the multiplication, which increases with the increase in the value of m. For example, when $m = 4$, initially $M = x_0$ and in the first iteration, the algorithm computes $M = x_0 \times x_1$, of size $2n$ using n^2 bit-multiplication operations in the worst case. The second iteration of the algorithm involves multiplying $M = x_0 \times x_1$ of size $2n$ with x_2 of size n to produce $M = x_0 \times x_1 \times x_2$ of size $3n$ using $2n^2$ bit-multiplication operations. In the last iteration, the algorithm computes $M = x_0 \times x_1 \times x_2 \times x_3$ of size $4n$ using $3n^2$ bit-multiplication operations. Hence, the overall number of bit-multiplication operations to compute $M = x_0 \times x_1 \times x_2 \times x_3$ is $n^2 + 2n^2 + 3n^2 = 6n^2$ in the worst case. Furthermore, in general, the multiplication of m integer numbers, of size n bits each, requires $n^2 + 2n^2 + 3n^2 + \dots + (m-1)n^2$ bit-multiplication operations, which is equal to $n\, m\, (m-1)/2$ in the worst case.

B. Illustrative Example

An illustrative example is given to explain the total number of digit-multiplication operations required to multiply m numbers using the sequential OM algorithm. The notation t_* represents the total number of digit-multiplication operations. For simplicity, let us assume that we have an array of $m = 20$ integer numbers and that the value of each element in the array is 5, i.e.:

$$X = (5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5)$$

The execution of the optimal sequential algorithm on X is described below.

Initially, $M = x_0 = 5$ and $t_* = 0$, and the algorithm executes 19 iterations to compute $M = 5^{20}$. These iterations are represented in Figures 1–4. Each figure consists of many subfigures, where each subfigure represents an iteration in the execution of the OM algorithm. Three pieces of information are associated with each iteration. The first is the number of digit-multiplications for multiplying the two numbers and is denoted by $*\,(k)$, where k is the number of digit-multiplication

operations. For example, in Figure 1(c), the algorithm multiplies 5 with the digits 5, 2, and 1, so the number of digit-multiplication operations is 3, and is denoted by $*(3)$. The second piece of information is the total number of digit-multiplication operations from the start to the current iteration and is denoted by t_*. For example, in Figure 1(c), the total number of digit-multiplication operations is equal to the number of digit-multiplication operations for the current iteration, $i = 3$, plus the total number of digit-multiplication operations for the previous iterations. Therefore, $t_* = t_* + *(3) = 3 + 3 = 6$. The third and last piece of information in each subfigure consists of the iteration number, i, and the value of M that is equal to $\prod_{j=0}^{i} x_j$.

Figures 1-4 display the complete execution of the OM algorithm on X for iterations 1–5, 6–10, 11–14, and 15–19, respectively.

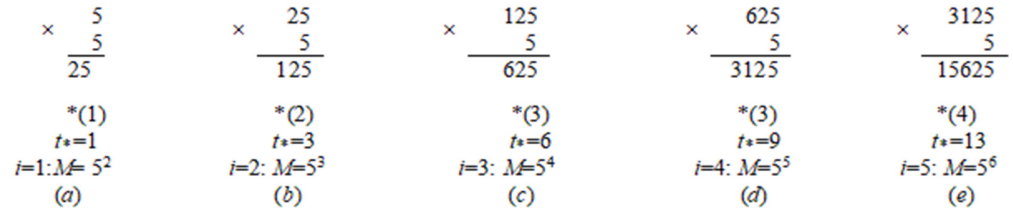

Fig. 1. Number of operations of the iterations 1-5 for the OM algorithm.

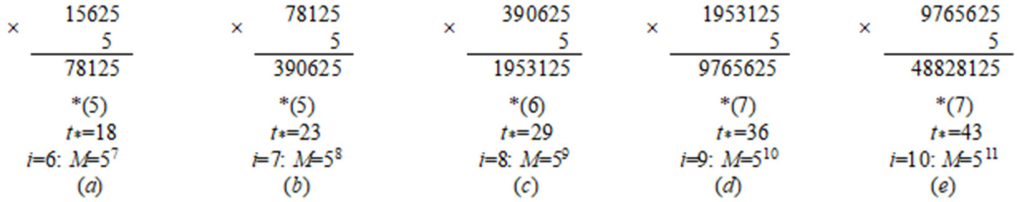

Fig. 2. Number of operations of the iterations 6-10 for the OM algorithm.

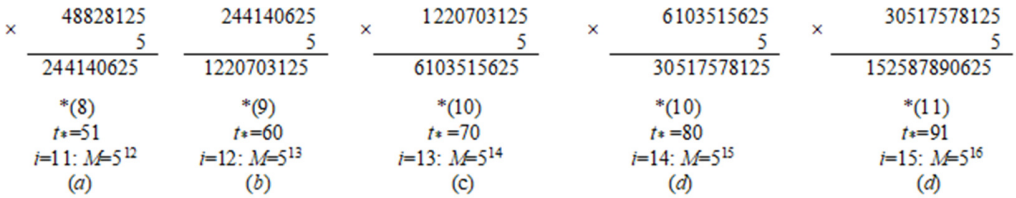

Fig. 3. Number of operations of the iterations 11-14 for the OM algorithm.

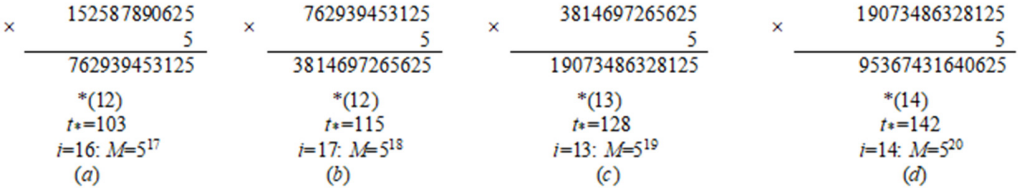

Fig. 4. Number of operations of the iterations 15-18 for the OM algorithm.

III. THE MODIFIED ALGORITHM

In this section, at first the idea and the steps of the suggested strategy for speeding up the execution time of the multiplication of m numbers, each of size n, where m is a large value are introduced. Then two examples are given to illustrate the proposed approach by applying it to the array discussed in Section II.

A. The Algorithm

The proposed algorithm is based on dividing the array into blocks, each of size b, where $b < m$. Then the algorithm computes the multiplication of the elements in each block independently. The result of the multiplication for each block, say $temp$, is accumulated to produce the final result of the multiplication, M. So, after computing the multiplication of each block the value of M is updated by calculating $M = M \times temp$, see Algorithm BM0.

Initially, the algorithm computes the total number of blocks, nB, by calculating $\lfloor m/b \rfloor$. Then the algorithm computes the multiplication of b elements in an auxiliary

variable $temp$, where the initial value of $temp$ is the first element in the block. After that, the algorithm updates the value of the final result by multiplying it with the value of $temp$. After it has finished multiplying all the elements in all the blocks, the algorithm updates the final result by multiplying it with the remaining elements, $m - \lfloor m/b \rfloor \times b$. The time complexity of the proposed algorithm is equal to $\left(\lfloor m/b \rfloor \times b + (m - \lfloor m/b \rfloor \times b)\right) \times t_{mult}$, which is $O(m\, t_{mul})$. The proposed algorithm executes m multiplication operations, while the OM algorithm executes $m - 1$ multiplication operations.

Algorithm BM0
Input: An array of m numbers, $X = (x_0, x_1, \ldots, x_{m-1})$ and b.
Output: $M = \prod_{i=0}^{m-1} x_i$.
Begin
1. $nB = \lfloor m/b \rfloor$
2. $M = 1$
3. For $i = 0$ to $nB - 1$ do
4. $\quad j = i \times b$
5. $\quad temp = x_j$
6. $\quad$ While $(j < (i+1)b)$ do
7. $\quad\quad j = j + 1$
8. $\quad\quad temp = temp \times x_j$
9. $\quad$ End while
10. $\quad M = M \times temp$
11. End for
12. If $m \neq nB \times b$ then
13. $\quad$ For $j = nB \times b$ to $m - 1$ do
14. $\quad\quad M = M \times x_j$
15. $\quad$ End for
16. End if

End.

In order to reduce the number of multiplications in the proposed algorithm, BM0, from m to $m - 1$, we apply the following modifications. First, we compute the multiplication of the first block using the final result M. Hence, the total number of multiplications is $b - 1$, whereas the first block of the initially proposed algorithm required b multiplications: $b - 1$ to compute $temp$ and one to update M. Second, the remaining elements of the array are divided into blocks, $\lfloor (m - b)/b \rfloor$, and then the same steps are performed as in the initial version of the proposed algorithm. The modification of the proposed algorithm is denoted as BM. Now, we compute the number of multiplications, to compute M for the BM algorithm. Thus, the first block requires $b - 1$ multiplication operations, see lines 1–3. Each block, in the other blocks, requires $b - 1$ multiplication operations, see lines 7–12, and the updating of the value of M, for each block, requires only one multiplication. So, each iteration in the do-while loop requires b multiplication operations. Therefore, the total number of multiplications for the do-while loop is $\lfloor m/b - 1 \rfloor \times b$ where the term m/b-1 represents the number of windows in the do-while loop. Additionally, the total number of the remaining elements is $m - \lfloor m/b \rfloor \times b$. Hence, the total number of multiplications for the BM algorithm is $m - 1$. Hence, the running time and memory consumption of the BM algorithm are $O(m)$ and $O(1)$, respectively.

Algorithm BM
Input: An array of m numbers, $X = (x_0, x_1, \ldots, x_{m-1})$ and b.
Output: $M = \prod_{i=0}^{m-1} x_i$.
Begin
1. $M = x_0$
2. For $j = 1$ to $b - 1$ do
3. $\quad M = M \times x_j$
4. End for
5. $nB = \lfloor (m - b)/b \rfloor$
6. For $i = 1$ to $nB - 1$ do
7. $\quad j = i \times b$
8. $\quad temp = x_j$
9. $\quad$ While $(j < (i+1)b)$ do
10. $\quad\quad j = j + 1$
11. $\quad\quad temp = temp \times x_j$
12. $\quad$ End while
13. $\quad M = M \times temp$
14. End for
15. If $m \neq (nB + 1) \times b$ then
16. $\quad$ For $j = nB \times b$ to $m - 1$ do
17. $\quad\quad M = M \times x_j$
18. $\quad$ End for
19. End if

End.

B. Illustrative Example

In this section, we illustrate how the proposed BM algorithm reduces the total number of digit-multiplication operations by applying it on the same array as in Section II.B and using two values of b: 5 and 10. We also show how different block size values affect the total number of digit-multiplication operations.

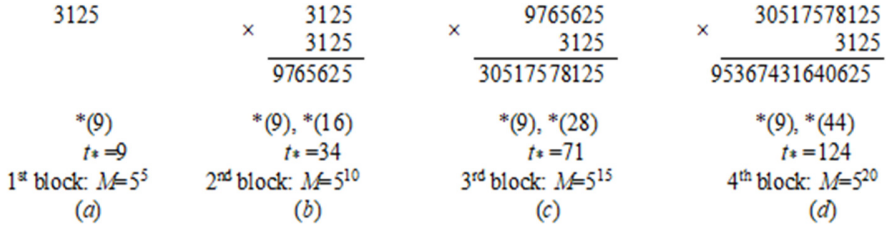

Fig. 5. Execution of the BM algorithm when b=5.

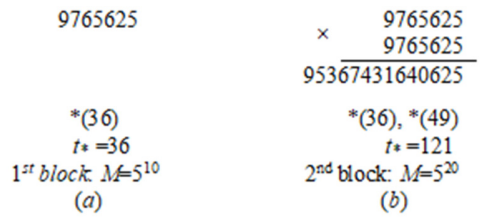

Fig. 6. Execution of BM algorithm when b=10.

First, let us assume that the size of the block is $b = 5$. Initially, the algorithm assigns 1 to M and executes lines 2–3. In the first iteration, the algorithm computes the multiplication of the first five numbers as $M = 3125$. The total number of digit-multiplication operations to compute M is nine, $t_* = 9$, similar to Figure 1(a)–(d). This step is shown in Figure 5(a), where the information in the Figure refers to the block number instead of the iteration number. The algorithm computes the multiplication of the second block in $temp = 3125$ using nine digit-multiplication operations. Then, the algorithm updates the value of M by multiplying it with $temp$, using 16 digit-multiplication operations. Therefore, the total number of digit-multiplication operations until the end of this step is $t_* = 9 + 16 + 9 = 34$, see Figure 5(b). Figures 5(c)–(d) represent the result of executing the third and fourth blocks, respectively. The total number of digit-multiplication operations for the final result of M is $t_* = 124$.

In the second example illustrated in Figure 6, when $b = 10$, the BM algorithm computes the first block in $M = 5^{10}$ using 36 digit-multiplication operations, which is similar to Figures 1(a)–(e) and 2(a)–(d). The second block is calculated by performing two steps. The first step involves computing the auxiliary variable $temp = 5^{10}$ using 36 digit-multiplication operations. The second step involves multiplying M with $temp$ using 49 digit-multiplication operations to produce $M = 5^{20}$. So, the total number of digit-multiplication operations is $t_* = 36 + 49 + 36 = 121$. Hence the BM algorithm requires fewer digit-multiplication operations when $b = 10$. In other words, the BM algorithm performs better when $b = 10$ than when $b = 5$ and $b = 1$. Also, the BM algorithm performs better when $b = 5$ than when $b = 1$.

IV. Experimental Study

In this section, we study the BM algorithm experimentally to find the answers to the following research questions:

1. Does the value of b have an effect on the running time of the BM algorithm experimentally, and if so, is this effect significant?
2. Do certain parameters (the size of the sequence of integers, the size of the integers, the size of the window, and the distribution of the data) have an effect on the performance of the BM algorithm?
3. Does the BM algorithm have a faster running time as compared to the OM algorithm?

The above questions are addressed in the following subsections. Subsection A contains a brief description of the platform (hardware and software) and the methodology used to test and measure the running time of both the BM and OM algorithms. Subsection B contains the answers to the first and second questions. Subsection C answers the third question.

A. Methodology

The experimental studies were conducted on a machine consisting of a 2.4 GHz processor with 32 GB of memory and a Windows operating system. Both algorithms, OM and BM, are implemented using C++ language and the GMP (GNU Multiple Precision Arithmetic) package. The GMP package is used to achieve two objectives. First, when we multiply many integer numbers of size 16 bits or more, the result is a number that is greater in size than 64 bits, and numbers of such size cannot be manipulated by the integer range in C++ language. Second, some of the applications that use the OM algorithm, such as cryptography, require a data of sizes greater than 64 bits. The experimental studies are based on the following four parameters:

- The first parameter is the size of the integer number, n, and is measured in bits. In the experiment, we used $n = 32$, 64, 128, 256, and 512.
- The second parameter is the size of the array, m. In the experiment, we use $m = 1/4k$, $1/2k$, $1k$, $2k$, $4k$, $8k$, $16k$, and $32k$, where $k = 1024$.
- The third parameter is the size of the block, b, used in the computation. In the experiment, we use different values of block size: $25, 50, \ldots, 500$, except when $m \leq 512$, the last value of b is 250.
- The fourth parameter is the data range, R, that is used to generate the elements of the array. In the experiment, we use two types of data range. The first data range is $R_1 = [2^{n-1}, 2^n - 1]$, which means that all data in the array of size n bits exactly, i.e., $2^{n-1} \leq x_i < 2^n$. The second data range is $R_2 = [0, 2^n - 1]$, which means that all data in the array of size less than or equal n bits, i.e., $0 \leq x_i < 2^n$.

The running time for each of the algorithms (BM and OM) is measured by taking the average time for 50 instances for fixed parameter values. The running time of the algorithms is measured in seconds.

B. Behavior of the BM Algorithm

In this subsection, we study the effect of changing the value of b on the running time of the BM algorithm using different values of m, n and R. To verify this goal, we first generate a data set by determining the values of m, n and R. Then we generate a data for one instance, $D = (m, n, R)$. For example, we set $m = 4k$, $n = 512$, and $R = R_1$. After that, we execute the BM algorithm on different values of b as described in Section IV.A and the running time for BM algorithm is measured for each value of b. Next, we repeat the previous steps using different values of data $D = (m, n, R)$. Following that, the same procedure is applied on BM algorithm using R_2 and different values of m, and n. Figure 7 shows the running time behavior of the BM algorithm using different values of m, n, R, and b. The results of our experiments lead to four observations for fixed values of n and m.

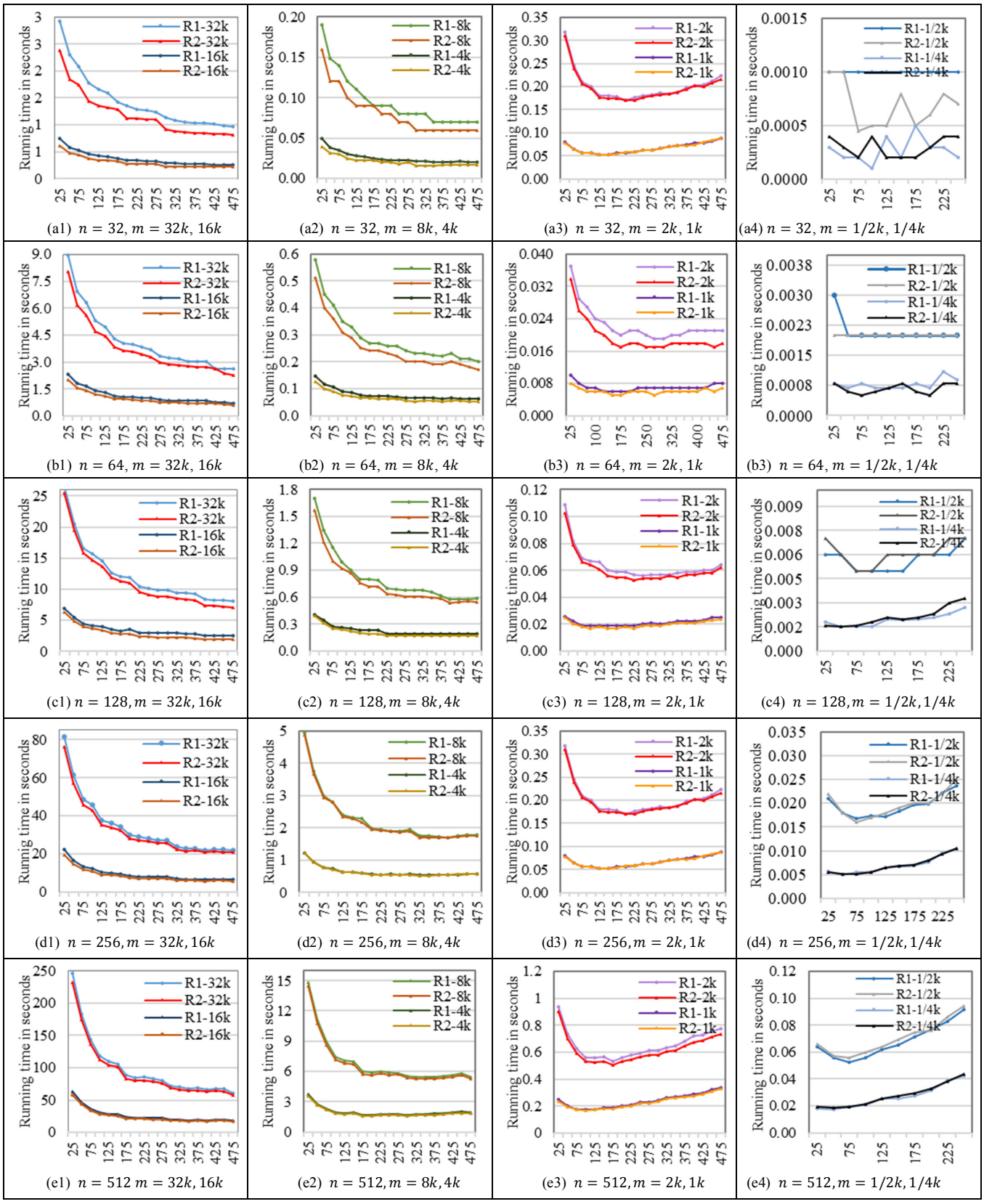

Fig. 7. Running time behavior of BM algorithm using different values of D, where the X-axis represents the different values of b.

First, the running time of the BM algorithm decreases with different values of b. For example, when $n = 32$ and $m = 32768$, the running time of the BM algorithm when $b = 25, 50, 75, 100, 125, 150, 175, 200, 225$, and 250 is 2.93, 2.3, $2.08, 1.77, 1.66, 1.60, 1.42, 1.36, 1.29$, and 1.27, respectively.

Second, it is not always the case that increasing the value of b leads to a decrease in the running time of the BM algorithm. For example, when $n = 256$, $m = 1/2k$ and $b = 25$, 50, 75, 100, 125, 150, 175, 200, 225, and 250, the running time of the BM algorithm is 0.021, 0.018, 0.017, 0.017, 0.017, 0.018, 0.020, 0.020, 0.022, and 0.024, respectively. Usually, this case occurs when n and m are small or when b is close in value to m. This means that not all values of b lead to an improvement in the running time of the BM algorithm, especially when m and n are small.

Third, it is clear that between two successive values of b, the improvement is small. However, when the difference between the two values of b is large, with a certain limitation, the improvement is significant. For example, when $m = 16k$, $n = 128$, and $b = 75$ and 100, the running time of the BM algorithm is 3.99 and 3.68, respectively, while the running time for the BM algorithm when $b = 75$ and 250 is 3.99 and 2.35, respectively.

Fourth, the running time of the BM algorithm on R_2 is a little faster than on R_1. The reason for this is that all the numbers in R_1 have n bits exactly, while the numbers in R_2 have a maximum length n.

C. Comparison of OM and BM Algorithms

In this subsection, the running times of the OM algorithm and BM algorithm are compared based on two values. The first value is the minimum running time obtained from the experimental results for the BM algorithm using different values of b, this value is denoted as BM_{min}. The second value is the average running time calculated from all running times of the BM algorithm using different values of b, denoted as BM_{Avg}. Therefore, two running time values for the BM algorithm are considered: BM_{min} and BM_{Avg}. Table I lists the running times of both algorithms, OM and BM, when the data range for the elements of the array taken from R_1. Several observations arise from the analysis of the data results in Table I.

TABLE I. RUNNING TIME (IN SECONDS) OF OM AND BM ALGORITHMS IN THE CASE OF R_1

n	Algorithm	m							
		$1/4k$	$1/2k$	$1k$	$2k$	$4k$	$8k$	$16k$	$32k$
32	OM	0.0003	0.001	0.003	0.012	0.046	0.18	0.74	2.92
	BM_{min}	0.0003	0.001	0.002	0.006	0.02	0.70	0.26	0.99
	BM_{Avg}	0.0003	0.001	0.002	0.007	0.026	0.096	0.370	1.433
64	OM	0.0009	0.003	0.014	0.055	0.219	0.87	3.50	13.37
	BM_{min}	0.0007	0.002	0.006	0.019	0.061	0.200	0.720	2.600
	BM_{Avg}	0.0008	0.002	0.007	0.022	0.078	0.288	1.107	4.163
128	OM	0.003	0.011	0.044	0.173	0.709	2.87	11.49	45.62
	BM_{min}	0.0015	0.005	0.017	0.052	0.168	0.56	1.94	7.11
	BM_{Avg}	0.0019	0.007	0.02	0.061	0.208	0.757	2.882	11.197
256	OM	0.0107	0.042	0.171	0.694	2.761	10.8	49.59	191.13
	BM_{min}	0.0049	0.017	0.052	0.17	0.535	1.694	6.628	21.943
	BM_{Avg}	0.0069	0.024	0.067	0.201	0.646	2.285	9.59	33.864
512	OM	0.0435	0.171	0.666	2.579	10.302	42.084	172.42	750.41
	BM_{min}	0.0175	0.053	0.168	0.503	1.63	5.256	16.581	60.094
	BM_{Avg}	0.0266	0.09	0.23	0.62	1.917	6.652	24.262	98.396

TABLE II. PERCENTAGE OF IMPROVEMENT BY THE BM ALGORITHM AS COMPARED TO THE OM ALGORITHM IN THE CASE OF R_1.

m	n bits									
	32		64		128		256		512	
	BM_{min}	BM_{Avg}	BM_{min}	BM_{Avg}	BM_{min}	BM_{Avg}	BM_{min}	BM_{Avg}	BM_{min}	BM_{Avg}
$1/4k$	0.00%	0.00%	22.22%	11.11%	50.00%	36.67%	54.21%	35.51%	59.72%	38.78%
$1/2k$	0.00%	0.00%	33.33%	33.33%	54.55%	36.36%	59.95%	43.46%	69.00%	47.35%
$1k$	33.33%	0.00%	57.14%	50.00%	61.36%	54.55%	69.63%	60.86%	74.80%	65.49%
$2k$	50.00%	14.29%	65.45%	60.00%	69.94%	64.74%	75.52%	71.05%	80.50%	75.97%
$4k$	56.52%	23.08%	72.15%	64.38%	76.30%	70.66%	80.62%	76.60%	84.18%	81.39%
$8k$	61.11%	27.08%	77.01%	66.90%	80.49%	73.62%	84.32%	78.85%	87.51%	84.19%
$16k$	64.86%	29.73%	79.43%	68.37%	83.12%	74.92%	86.63%	80.66%	90.38%	85.93%
$32k$	66.10%	30.91%	80.55%	68.86%	84.41%	75.46%	88.52%	82.28%	91.99%	86.89%

First, the running time of the BM algorithm is faster than that of the OM algorithm for all data sets, except for two cases: $D = (256,32,R_1)$ and $D = (512,32,R_1)$ where the running time of the two algorithms are equal.

Second, two factors have an effect on the difference in the running times of the two algorithms. The first factor is the size of array, m, while the second factor is the size of the integer number, n. For small values of m and n, the difference in

running time between the two algorithms is small. For example, when $n = 64$ and $m = 512$ the running times of OM, BM_{min} and BM_{Avg} are 0.003, 0.002, and 0.002, respectively. On the other hand, when $n = 128$ and $m = 16k$, the running times of OM, BM_{min} and BM_{Avg} are 11.49, 1.94, and 2.88 seconds, respectively.

Third, based on the second observation, the percentage of improvement achieved by the BM algorithm, in terms of BM_{min} and BM_{Avg}, as compared to OM algorithm increases with an increase in the values of m and n. This means that the improvement occurs in two cases: when m is fixed and n is varied and when n is fixed and m is varied. For example, if $n = 128$, the percentage of improvement achieved by the BM algorithm when $m = 1/4k$, $1/2k$, $1k$, $2k$, $4k$ $8k$, $16k$ and $32k$ is 50%, 54.55%, 61.36%, 69.94%, 76.30%, 80.49%, 83.12%, 84.41% respectively in the case of BM_{min}, while the percentage of improvement in the case of BM_{Avg} is 36.67%, 36.36%, 54.55%, 64.74%, 70.66%, 73.62%, 74.92%, and 75.46%, respectively. Also, if $m = 1k$, the percentage of improvement achieved by the BM algorithm when $n = 32$, 64, 128, and 256 is 33.33%, 57.14%, 61.36%, and 69.59%, respectively in the case of BM_{min}, while the percentage of improvement in the case of BM_{Avg} is 0.0%, 50%, 54.55%, and 60.82% respectively. The full details of the percentage improvement achieved by the BM algorithm as compared to the OM algorithm are shown in Table II.

Fourth, referring to Table II, it is clear that the percentage of improvement achieved by the BM algorithm in the case of BM_{min} is better than in the case of BM_{Avg}, for fixed n and m, because the running time for BM algorithm in case of BM_{min} is less than the running time for BM algorithm in case of BM_{Avg}.

Note that, with increasing m and n, the running times for BM in case of R_1 is near to the case of R_2, so we neglect the comparison between BM and OM algorithms in case of R_2.

V. CONCLUSION

In this paper, we studied the multiplication operation because it has a higher time cost as compared to other basic arithmetic operations and therefore has a significant influence on the performance of many applications such as cryptography and digital signal processing. To reduce the cost of this operation, we developed an efficient algorithm for sequential computation tasks that require the multiplication of a sequence of big integers. The algorithm is based on using the window strategy to reduce the cost of multiplying the sequence of big integers. The algorithm has the same time complexity as the best-known sequential algorithm but performs fewer digit-multiplication operations. In our experimental studies to test the performance of the proposed BM algorithm, we considered four parameters of different values: sequence size ($m = 1/4k$, $1/2k$, $1k$, $2k$, $4k$, $8k$, $16k$, and $32k$), integer size ($n = 32$, 64, 128, 256, and 512), block size ($b = 25, 50, \ldots, 500$), and data range (fixed and varied sizes). The results showed the effectiveness of the proposed algorithm as compared to the best-known sequential algorithm, where the percentage of improvement achieved by the proposed algorithm was 90% when m and n were large.

There are different future directions related to this study. Firstly, the study of the behavior of the BM algorithm when $n \geq 1k$. Secondly, the study of the effect of increased value of b on the BM algorithm. Third, the question of the existence of a relation between the values of b and m. Fourth, how to use the GPU to speedup the running time for the BM algorithm. Fifth, how to speedup modular multi-exponentiation using the BM algorithm and modular exponentiation.

ACKNOWLEDGEMENT

This work has been funded by the Scientific Research Deanship at the University of Ha'il – Saudi Arabia through project number RG-191309.

REFERENCES

[1] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge ; New York: Cambridge University Press, 2010.

[2] E. S. I. Harba, "Secure Data Encryption Through a Combination of AES, RSA and HMAC," *Engineering, Technology & Applied Science Research*, vol. 7, no. 4, pp. 1781–1785, Aug. 2017, https://doi.org/10.48084/etasr.1272.

[3] M. B. Apsara, P. Dayananda, and C. N. Sowmyarani, "A Review on Secure Group Key Management Schemes for Data Gathering in Wireless Sensor Networks," *Engineering, Technology & Applied Science Research*, vol. 10, no. 1, pp. 5108–5112, Feb. 2020, https://doi.org/10.48084/etasr.3213.

[4] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, https://doi.org/10.1145/359340.359342.

[5] K. A. Fathy, H. M. Bahig, and A. A. Ragab, "A Fast Parallel Modular Exponentiation Algorithm," *Arabian Journal for Science and Engineering*, vol. 43, no. 2, pp. 903–911, Feb. 2018, https://doi.org/10.1007/s13369-017-2797-3.

[6] H. M. Bahig and K. A. Fathy, "An efficient parallel strategy for high-cost prefix operation," *The Journal of Supercomputing*, Nov. 2020, https://doi.org/10.1007/s11227-020-03473-x.

[7] G. Bilardi and L. De Stefani, "The I/O complexity of toom-cook integer multiplication," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, USA, Jan. 2019, pp. 2034–2052.

[8] R. Brumnik, V. Kovtun, A. Okhrimenko, and S. Kavun, "Techniques for Performance Improvement of Integer Multiplication in Cryptographic Applications," *Mathematical Problems in Engineering*, vol. 2014, Feb. 2014, Art. no. 863617, https://doi.org/10.1155/2014/863617.

[9] S. A. Cook and S. O. Aanderaa, "On the Minimum Computation Time of Functions on JSTOR," *Transactions of the American Mathematical Society*, vol. 142, pp. 291–314, Aug. 1969, https://doi.org/10.2307/1995359.

[10] A. De, P. P. Kurur, C. Saha, and R. Saptharishi, "Fast Integer Multiplication Using Modular Arithmetic," *SIAM Journal on Computing*, vol. 42, no. 2, pp. 685–699, Jan. 2013, https://doi.org/10.1137/100811167.

[11] M. J. Fischer and L. J. Stockmeyer, "Fast on-line integer multiplication," in *Proceedings of the fifth annual ACM symposium on Theory of computing*, New York, NY, USA, Apr. 1973, pp. 67–72, https://doi.org/10.1145/800125.804037.

[12] M. Fürer, "Faster integer multiplication," in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, New York, NY, USA, Jun. 2007, pp. 57–66, https://doi.org/10.1145/1250790.1250800.

[13] P. Gaudry, A. Kruppa, and P. Zimmermann, "A gmp-based implementation of schönhage-strassen's large integer multiplication algorithm," in *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, New York, NY, USA, Jul. 2007, pp. 167–174, https://doi.org/10.1145/1277548.1277572.

[14] D. Harvey, J. van der Hoeven, and G. Lecerf, "Even faster integer multiplication," *Journal of Complexity*, vol. 36, pp. 1–30, Oct. 2016, https://doi.org/10.1016/j.jco.2016.03.001.

[15] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akademii Nauk.*, vol. 145, no. 2, pp. 293–294, 1962.

[16] C. Lüders, "Implementation of the DKSS Algorithm for Multiplication of Large Numbers," in *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, New York, NY, USA, Jun. 2015, pp. 267–274, https://doi.org/10.1145/2755996.2756643.

[17] S. R. S. Rao, "Interesting Results Arising from Karatsuba Multiplication - Montgomery family of formulae," in *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015*, New York, NY, USA, Sep. 2015, pp. 317–322, https://doi.org/10.1145/2818567.2818666.

[18] A. Schönhage and V. Strassen, "Schnelle multiplikation grosser zahlen," *Computing*, vol. 7, no. 3–4, pp. 281–292, 1971.

[19] A. L. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," *Soviet Mathematics Doklady*, vol. 3, no. 4, pp. 714–716, 1963.

[20] Y. Wu, "Strength reduction of multiplications by integer constants," *ACM SIGPLAN Notices*, vol. 30, no. 2, pp. 42–48, Feb. 1995, https://doi.org/10.1145/199873.199880.

[21] A. Zanoni, "Iterative Toom-Cook methods for very unbalanced long integer multiplication," in *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, New York, NY, USA, Jul. 2010, pp. 319–323, https://doi.org/10.1145/1837934.1837995.

[22] L. De Stefani, "On the I/O complexity of hybrid algorithms for Integer Multiplication," *arXiv:1912.08045 [cs]*, Jul. 2020, Accessed: Nov. 28, 2020. [Online]. Available: http://arxiv.org/abs/1912.08045.

[23] A. Rezai and P. Keshavarzi, "Compact SD: a new encoding algorithm and its application in multiplication," *International Journal of Computer Mathematics*, vol. 94, no. 3, pp. 554–569, Mar. 2017, https://doi.org/10.1080/00207160.2015.1119269.

[24] V. Bunimov and M. Schimmler, "Efficient Parallel Multiplication Algorithm for Large Integers," in *Euro-Par 2003 Parallel Processing*, Berlin, Heidelberg, 2003, pp. 923–928, https://doi.org/10.1007/978-3-540-45209-6_127.

[25] B. P. Sinha and P. K. Srimani, "A new parallel multiplication algorithm and its VLSI implementation," in *Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, New York, NY, USA, Feb. 1988, pp. 366–372, https://doi.org/10.1145/322609.322777.

[26] J. V. Tembhurne, "Parallel Multiplication of Big Integer on GPU," in *Smart and Innovative Trends in Next Generation Computing Technologies*, Singapore, 2018, pp. 276–285, https://doi.org/10.1007/978-981-10-8657-1_21.

[27] J. V. Tembhurne and S. R. Sathe, "Performance evaluation of long integer multiplication using OpenMP and MPI on shared memory architecture," in *2014 Seventh International Conference on Contemporary Computing (IC3)*, Aug. 2014, pp. 283–288, https://doi.org/10.1109/IC3.2014.6897187.

[28] C. K. Yuen and M. D. Feng, "Parallel multiplication: a case study in parallel programming," *ACM SIGPLAN Notices*, vol. 29, no. 3, pp. 12–17, Mar. 1994, https://doi.org/10.1145/181587.181589.

[29] L. De Stefani, "Communication-Optimal Parallel Standard and Karatsuba Integer Multiplication in the Distributed Memory Model," *arXiv:2009.14590 [cs]*, Sep. 2020, Accessed: Nov. 28, 2020. [Online]. Available: http://arxiv.org/abs/2009.14590.

[30] S. G. Akl, *Parallel Computation: Models and Methods*, 1st edition. Upper Saddle River, NJ, USA: Prentice Hall, 1996.

[31] H. M. Bahig, H. M. Bahig, and K. A. Fathy, "Fast and scalable algorithm for product large data on multicore system," *Concurrency and Computation: Practice and Experience*, Apr. 2019, Art. no. e5259, https://doi.org/10.1002/cpe.5259.