Parallel Scalar Multiplication Based on Signed Binary Representation of CRT and DRM

Mohammad Anagreh\(^1\),
\(^1\)Institute of Computer Science, University of Tartu, Estonia
Supervisor Prof. Eero Vainikko
November 27, 2018

This is a scientific report for several issues in Parallelization elliptic curve cryptosystem based on using different signed binary representations. In this paper, we introduce the parallel implementation of elliptic curve scalar multiplication over the prime field using signed binary representations. Our implementation speeds up the calculation of scalar multiplication in comparison with the standard case. We introduce parallel algorithm for computing elliptic curve scalar multiplication based on Complementary Recoding Technique (CRT). Also we propose a parallel algorithm for computing elliptic curve scalar multiplication based on another representation which is a Direct Recording Method (DRM). Both implementations of the proposed algorithms show speed-ups reaching up to 60% in comparison with execution time for sequential cases of the algorithms. We find that ECC-DRM is faster than ECC-CRT in both parallel and sequential counterparts.

Index Terms—ECC, Parallel Computing, CRT, DRM.

I. INTRODUCTION

Elliptic curve cryptosystems (ECC) were independently proposed by two researchers, Koblitz [1] and Miller [2]. Elliptic curve cryptosystem (ECC) is widely used in many cryptographic protocols such as asymmetric encryption, digital signature and key exchange. One of the most important advantages with ECC is its suitability for using it in case of limited memory resources, such as portable devices, because it has a shorter key size. ECC shows high-level of security with shorter key sizes in comparison with other existing algorithms like RSA [3]. The minimum key size of the ECC is 160-bits having the same security level with a standard key size of RSA of 1024-bits [4].

Computing the scalar multiplication is an expensive operation in the elliptic curve cryptosystem. Elliptic curve scalar multiplication is the operation of successively adding a EC point along an elliptic curve to itself \(n\) times repeatedly based on the length of scalar \(d\). Scalar multiplication is given by the equation is \(Q = dP\), where \(d\) is an integer number converted to the binary - \(d = \sum_{i=0}^{n-1}2^i d_i \in \{0,1\} - \) or signed binary - \(d = \sum_{i=0}^{n-1}2^i d_i \in \{1,0,1\} - \), while a given \(P = (x,y)\) is a point in elliptic curve. Therefore, many researchers have focused to enhance the calculation of scalar multiplication by proposing a new related algorithm such as signed binary representation. As well as by enhancing the calculation method itself such as using a parallel calculation. Hamming Weight (HW) is the number of non-zero bits in the scalar representation \(d\) (scalar multiplication \(d\) is a ECC key). The number of adding and doubling operations on an elliptic curve scalar multiplication are based on length \(n\) of the scalar \(d\). It is an integer number which, for the purposes of the computation, is represented in binary or in signed binary representation. Reducing the number of non-zero bits \(1\) (or \(-1\)) in the scalar representation \(d\) will reduce the number of adding operations in the ECC scalar multiplication. Therefore, lower HW is preferred to be used in the ECC scalar multiplication. Several researchers have proposed new methods to convert the binary representation to the signed binary representation in order to reduce the Hamming Weight of the scalar \(d\). These methods are Mutual Opposite Form (MOF) [5], Joint Sparse Form (JSF) [6], Non-Adjacent Form (NAF) [7]. As well as, Complementary Recoding Method (CRM) is proposed [8] which enhanced to be Direct Recording method (DRM) [9] and other methods [10]. As well as, there are several methods proposed to accelerate the calculation of the ECC scalar multiplication by parallel computing [11] [12] [13].

In this paper, we proposed two algorithms to accelerate the performance of computing elliptic curve scalar multiplication by parallelizing scalar multiplication algorithm. The proposed algorithms are based on combining the Add-subtract scalar multiplication algorithm and transforming the scalar \(d\) from binary representation to the signed binary representation. The first algorithm based on combining Complementary Recoding Technique (CRT) with scalar multiplication algorithm. While the second algorithm is based on Direct Recording method (DRM).

Our implementation of the two algorithms show that the proposed methods are faster than sequential calculation of the ECC scalar multiplication. The parallelization in the both algorithms is done by using two processors. Finding the CRT or DRM conversion and performing the doubling operation on the first processor, calculation the addition operation on the second processor. We use a circular buffer to pass the data from processor 1 to processor 2. Circular buffer is considered as a shared memory among the processors.

This paper is organized as follows: Section 2 briefly presents the preliminaries. Section 3 shows some related work while section 4 is the proposed work and the algorithms. Section 5 shows the results and presents the experiments. The last section concludes the proposed method and discusses future work.
A. Prime Fields $F_p$ of Elliptic Curve

In this paper, we focus on the prime curves over $F_p$. The prime curves over $F_p$ make use of the cubic equation as identified in Equation (2) with Cartesian coordinate variables $(x, y)$ and coefficients $(a, b)$ as elements of $F_p$. All the values can be considered integers that are computed modulo the prime number $p$ [2]. The cubic equation with coefficients $(a, b)$ and variables $(x, y)$ for the elliptic curves over $F_p$ is the following:

$$
y^2 \quad \text{mod} = (x^3 + ax + b) \quad \text{mod} \ p \ (1)
$$

Let the point $P = (x_1, y_1)$ and point $Q = (x_2, y_2)$ be in the elliptic curve over $F_p$, defined by the coefficients $(a, b)$. In addition, let $O$ be the point at infinity. The rules for addition operation in the EC is as follows:

$$
P + O = P
$$

Given point $P$ and point $Q$, if $x_1 = x_2$ and $y_2 = -y_1$ then

$$
P + Q = 0
$$

In general, $R = Q + P$, where the result $R = (x_3, y_3)$ is defined as follows:

$$
x_2 = \lambda^2 - x_1 - x_2 \ \text{mod} \ p \ (4)
$$

$$
y_3 = \lambda(x_1 - x_3) - y_2 \ \text{mod} \ p \ (5)
$$

$$
\lambda = \begin{cases} 
\frac{y_2-y_1}{x_2-x_1} \mod p, & \text{if } P \neq Q \\
\frac{3x_1^2+a}{2y_1} \mod p, & \text{if } P = Q
\end{cases} \ (6)
$$

We can summarize, for any two points $P, Q$ on a given elliptic curve, there are two main operations. The addition $P + Q$ when $P \neq Q$ is called point addition and $Q = 2P$ when $P = Q$ is called point doubling. Addition operation has 5 sub-operatons, 2 squaring 2 multiplications and 1 inversion. Consequently, for non-negative integer number $d$, it is possible to define the scalar point multiplication $Q = dP$ on the elliptic curve, doubling and adding operation illustrated in Figure 1.

![Fig. 1: Dubling and adding point in EC](image)

B. Signed Binary Presentation

Signed Binary representation is a representation converted from binary representation $d = \sum_{i=0}^{n-1} 2^i d_i, d_i \in \{0,1\}$ to a new representation $d = \sum_{i=0}^{n-1} 2^i d_i, d_i \in \{I,0,1\}$ by applying some methods such as MOF [5], NAF [7], CRT [8], DRM [9] and others.

1) Complementary Recoding method (CRT)

CRT is one of techniques to convert the binary representation to canonical binary representation to reduce the hamming weight [CRT]. CRT representation is achieved by representing the integer $d = \sum_{i=0}^{n-1} 2^i d_i = (100...0)_{(n+1)\text{bits}} - \bar{d} - 1$. This conversion is very simple, efficient and low time complexity in compassion with another methods [DRM].

The procedure to get signed binary representation from binary representation by applying CRT method is given by $d = \sum_{i=0}^{n-1} 2^i d_i = (100...0)_{(n+1)\text{bits}} - \bar{d} - 1$, where $\bar{d} = \bar{d}_{n-1}, \bar{d}_{n-2}, ..., \bar{d}_0$ and $d_i = 0$ if $d_i = 1$, $\bar{d}_i = 1$ if $d_i = 0$ for $i = 0, 1, ..., d - 1$.

**Example 1:** $d = 7327$, converting $d$ to the binary representation is $(1110010011111)$. Converting the binary representation to signed binary representation by applying CRT is $d = \sum_{i=0}^{n-1} 2^i d_i = (100...0)_{(n+1)\text{bits}} - \bar{d} - 1 = (1000000000000000)_{2} - (1110010011111)_2 - 1 = (10001101010000)_{2}$.

To prove the solution, let us convert the signed binary representation $(1110010011111)_2$ we got from applying CRT to decimal, $d = 8192 - 512 - 256 - 64 - 32 - 1 = 7327$.

The hamming weight for binary representation of 7327 is 9, while the hamming weight for signed binary representation using CRT is 6. Less hamming weight will save the number of operations of calculating the EC scalar multiplication.

2) Direct Recodign Method (DRM)

DRM is a converting method from the binary representation to signed binary representation [DRM]. This method is based on the CRT but with time complexity less than CRT because it uses only single operation of bitwise subtraction with $0 - 1 = 1^'$. As well as, the hamming weight of DRM is less than the Hamming weight of CRT [DRM]

The procedure to convert the binary representation to the signed binary representation using DRM is as follows: $2^{p+1} > d > 2^p$ then $d = (2^{p+1})_2 - (2^{p+1} - k)_2$, where $p$ is the number of digits in binary representation (index = 0).

**Example 2:** $d = 248$. Converting $d$ to the binary representation is $(11111000001000)$. Converting the binary representation to the signed binary representation by applying DRM as follows: $2^8 = 100000000$ and $(2^8 - 248) = 1000$. Then $d = (100000000) - (1000) = 100000000$.

To prove the solution, let us convert the signed binary representation $(100000000)$ we got by applying the DRM to decimal, $d = 256 - 8 = 248$. The hamming weight for the binary representation of 248 is 5, while the hamming weight for signed binary representation using DRM is 2.

So, the converting will save the calculation of the EC scalar
multiplication. As well as, the signed binary representation of 248 using CRT is \((1000001111)_2\) while the hamming weight of signed binary representation using CRT is 5.

C. ECC Scalar Multiplication

The scalar multiplication is the main operation in the ECC. Scalar multiplication is built up from two main operations addition of points, and the doubling of a point. The scalar \(d\) is an integer has to be converted to a bitstring. The occurrence of bit 1 in the representation corresponds to the operation of adding two points. There are approximately \(n = 2\) such additions in a scalar multiplication. On the other hand, the number of doubling operations is \(n - 1\). In case of signed binary representation, the third digit which is \(T\) will be processed by the subtracting operations. Algorithm 1 is an Adding-Subtracting Scalar Multiplication Algorithm, which is used to compute the elliptic curve scalar multiplication based for a scalar \(-d = \sum_{i=1}^{n-1} 2^i \cdot d_i\), represented either in binary \(d_i \in \{0, 1\}\) or in signed binary \(d_i \in \{-1, 0, 1\}\).

**Algorithm 1 Adding-Subtracting Scalar Multiplication**

<table>
<thead>
<tr>
<th>Data:</th>
<th>Point on EC (P), a non-zero string ((d_0, \ldots, d_{n-1}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>(Q = dP)</td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>(Q \leftarrow 0), (R \leftarrow P)</td>
<td></td>
</tr>
<tr>
<td>for (i = 0) to (n-1) do</td>
<td></td>
</tr>
<tr>
<td>if ((d_i = 1)) then</td>
<td></td>
</tr>
<tr>
<td>(Q \leftarrow Q + R)</td>
<td></td>
</tr>
<tr>
<td>else if ((d_i = T)) then</td>
<td></td>
</tr>
<tr>
<td>(Q \leftarrow Q - R)</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>(R \leftarrow 2R)</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>return (Q)</td>
<td></td>
</tr>
</tbody>
</table>

The example below shows how to find the ECC scalar multiplication for a small scalar \(d\).

**Example 3: Finding the ECC scalar multiplication using \(d = 115\)**

First, we have to convert the integer \(d\) to the binary, so \(d = 1001101\).

Then, finding the ECC scalar multiplication based on scalar \(d\) from right to left as illustrated in Figure 2.

![Fig. 2: Finding ECC Scalar Multiplication](image)

III. RELATED WORK

Many researchers have been working to enhance the ECC by enhancing the calculation in the scalar multiplication. The improvement of the scalar multiplication can be achieved by improving or proposing some related algorithms in scalar multiplication. Applying the signed binary representation algorithms to find the scalar multiplication is an efficient way to reduce the number of non-zero bits in the key. Hamming Weight is a big player to reduce the number of adding operations in computing the scalar multiplication. In 1951 Booth proposed a new scalar representation called signed binary representation. There are many methods to represent integers in signed binary such as NAF, JSF, and MOF. Also in 2003 a new method to compute general multiplication was proposed by Change et al [14] which is the result of using NAF, MOF, and JSF. Different researchers proposed methods to calculate the scalar multiplication in parallel computing using the binary or signed binary representation.

Anagreh et al [15], proposed a parallel method to compute scalar multiplication based on the mutual opposite form (MOF). They extracted a new algorithm that combined Adding-Subtracting Scalar Multiplication Algorithm and Mutual Opposite Form (MOF). They used two processors to perform the parallel calculation, the Method calculates the doubling operation in a processor and adding operation in another processor at the same time. The proposed method computes the scalar multiplication without performing the MOF conversion. The proposed method is performing the comparison operation of the given bit-string \(d\) to decide where the second processor has to add or subtract the doubled point in case of non-zero bits \(\{T, 1\}\). The proposed method achieves the speed-up 90% faster than the sequential version of the ECC scalar multiplication with MOF.

Negre et al [16] proposed a new parallel approach for finding the scalar multiplication. They split the scalar multiplication based on NAF into two parts for the prime field \(F_p\) and three parts for the binary field \(F_{2^m}\). In their method, both operations doubling and (addition or subtraction) will be performed in a separate thread. We interest the prime field of their work, researcher split the operations of scalar multiplication into two section, the first section \(Q_1 = k_1P\) will be performed in a single thread. The second part \(Q_2 = 2^k k_2 P\) will be performed in another thread. Finding the scalar multiplication in their proposed job given by \(Q = Q_1 + Q_2\), the two points \(Q_1\) and \(Q_2\) are added to get the scalar multiplication \(Q\). The window size of Non-Adjacent Form (w-NAF) is used. The window size of NAF is can be reached to 4. The proposed method achieved an improvement by at least 10% the computation time of the scalar multiplication.

Software implementation proposed by Robert [17] for finding ECC scalar multiplication. In their proposed method, they used two thread to perform the parallel calculation. As well as, for various elliptic curves over the prime \(F_p\) used four threads. Two algorithms are used in their job Double-and-add and Half-and-add algorithms. In this work, putting the doubling operations into one thread (producer) while additions and subtractions operation into other another thread (consumer). One single mutex at the beginning of the computation is used to avoid using the mutex synchronization as much as possible. The goal of using the mutex is to keep the consumer inactive state at the beginning of the processing while the producer
processes the doubling operation. The method shows some violation of the read-after-write dependency. The memory violation might happen because of the size of the first batch of points which is before releasing the mutex was too small. As well as, in the case of the long sequence of zeros in the binary or NAF scalar representation. The results show that there is an error rate which is limited to less than 1% but is not acceptable. To eliminate this problem, a variable in a global memory as a loop counter is used. An extra operation is added to the scheme that will cause the reducing of the execution time in the parallel version. The NAF conversion is not a part of the parallel section. The result shows that the enhancement reached to the %15 in comparison with the sequential version.

Phalakarn et al [18] proposed a new representation for right-to-left parallel elliptic curve scalar multiplication. The mathematical model reduced the calculation time for finding ECC scalar multiplication. They proposed algorithms that will generate the representations which will reduce the execution time of the scheme. Three processors are used to perform the whole calculation in the scheme. Two processor is for performing the doubling $P$ and $Q$. The third processor is for performing the addition operation using two binary representation $m$ and $n$. The issue of the communication between the processors in the model is still opened and may it cause an increasing time complexity because it is an extra operation.

Accelerate Performance for the ECC scalar multiplication implemented by Anagreh et al [19]. A new algorithm is introduced to find the ECC scalar multiplication based on NAF representation. They used two processors to perform the whole calculation in Parallel computing. The first processor perform the doubling operations while the second processor perform the NAF conversion and (addition or subtraction) operations at the same time. A shared memory are used to transmit the doubled point from the first processor to the second processor. They performed the NAF conversion by the second Processor before starting to calculate the addition or subtraction operation. this method to works as a mutex in the first calculation to avoid perform the reading operation before writing in the shared memory. The result shows an enhancement is 60% faster than the standard version of the ECC calculation based on NAF.

IV. PARALLEL ALGORITHM

Reducing the execution time of the scalar multiplication by applying some an efficient method is desired.

In this work, we proposed two parallel algorithms to calculate the scalar multiplication based on signed binary representations. We extract both algorithms by combining the Add-Subtract Scalar Multiplication Algorithm and Converting Methods for finding the signed binary representation. The converting methods from binary representation to signed binary representation are CRT and DRM respectively. The first algorithm based on the CRT and the second algorithm based on DRM.

In our parallel algorithm, we use a circular buffer to transmit the processed data among the two processors in the scheme. Circular buffer is considered as a shared memory. The processors can access to the shared memory at any time with some organization to avoid read some an empty location in the memory. Processor-1 can write the doubled point $P$ and the scalar $d_i$ in a specific location in the circular buffer. Processor-2 can read the doubled point $P$ and the scalar $d_i$ from the circular buffer to perform the addition or subtraction operations. Circular buffer has two pointers front and rear to organize the reading and the writing operations. In each iteration in the scheme, writing should be in a location pointed by a front pointer $Push_{front}$. The reading in circular buffer should be in a location pointed by a rear pointer $Pull_{rear}$, where $front > rear$ for all writing and reading operations in the scheme. Such reading and writing operation is the most important issue to avoid any corruption in the calculation. As well as, we use two attributes for performing the reading and writing operations which are $is-full()$ and $is-empty()$. The main goal of using the attributes is to check the situation of the circular buffer before performing the reading or the writing operations. In case the circular buffer is full, then keep cycling without performing any operation until there is an empty location in the circular buffer, then Processor-1 write the point and scalar in the empty location in the circular buffer. The second attribute will be performed by Processor-2 before performing the addition or subtraction operations. The number of writing operations in the scheme that will be performed in the Processor-1 is based on the number of the bits $n$ in the scalar $d$. Moreover, the number of the reading operations that will be performed by the Processor-2 is based on number of non-zero bits $\{T, 1\}$ in the scalar $d$. We proposed two algorithms for finding scalar multiplication based on signed binary representation (CRT and DRM consequently) in the parallel calculation as follows:

A. Parallel ECC Scalar Multiplication based on CRT (PECC-CRT)

Task decomposition strategy is applied in our parallel implementation of the Algorithm (PECC-CRT). We use two Processors to perform the implementation of the PECC-CRT Algorithm. The Processor-1 is responsible about performing three sub tasks, see Processor-1 section in Algorithm 2.

The first task, is the conversion from binary representation $\{0, 1\}$ to the signed binary representation $\{T, 0, 1\}$ using CRT method. The integer $d$ is given, and should be converted to signed binary representation $\{T, 0, 1\}$. The second task is calculating the doubling operations in the elliptic curve based on the number of bits $n$ in the scalar $d$, where the point in elliptic curve $P = (x, y)$ is given. Performing the doubling operation by calling the function $n$ times, where $n$ is the number of bits in the signed binary representation. Regardless, is it a $T$, $0$ or $1$. The last task should be performed by Processor-1 is writing the doubled point $R$ and the scalar $d_i$ in an empty location in the circular buffer. As we explained above, the circular buffer is a shared memory and both Processors can access to the shared data for performing reading or writing operations.
Algorithm 2 Parallel Scalar Multiplication based on CRT (PECC-CRT)

Data: Integer \( d \), Point in EC \( P \)
Result: \( Q = dP \), based on (CRT)
Processor 1 CRT Conversion, Doubling Operations (DBL)
begin
\[ R \leftarrow P \]
\[ \text{Bin} = \text{Convert int to bin}(d) \]
\[ \text{CRT} = \text{Convert bin to CRT}((\text{Bin})) \]
for \( i = 0 \) to \( n - 1 \) do
repeat
if \( \text{buffer is full()} \) then
| continue
else if \( \text{CRT}_i \neq 0 \) then
| \( \text{Push}(R, \text{CRT}_i) \)
end
\[ R \leftarrow 2R \]
until \( i < n \)
end
Processor 2 Addition Operations (ADD)
begin
repeat
if \( \text{buffer is not_empty()} \) then
\[ \text{Pull}(R, \text{CRT}_i) \]
if \( \text{CRT}_i = 1 \) then
| \( Q \leftarrow Q + R \)
else
| \( Q \leftarrow Q \cdot R \)
end
until \( \text{buffer is not_empty()} \)
end

Processor-2 is responsible about performing three sub task as well, see Processor-2 section in Algorithm 2. The first task is reading the doubled point \( R \) and the scalar \( d_i \) from the circular buffer. Note, each doubled point has a specific scalar digit \( d_i \), that will be stored together in the circular buffer to keep the sequence of the doubling operations \( P, 2P, 4P, 8P, ..., 2^n P \). The second task is performing the addition or subtraction operations based on non-zero bits in the scalar \( d \), if the bit \( d_i \) is 1, Processor-2 has to perform the addition operation. If the bit \( d_i \) in the scalar \( d \) is \( T \), Processor-2 has to perform the subtraction operation which is the third task Processor-2 has to perform. Calculating the addition operation or/and subtracting operation will be saved in the accumulator \( Q \) which is the final result of finding EC scalar multiplication. Circular buffer is used to organize transmitting the data between two processors in the whole scheme. The data which has to transmit from Processor-1 to Processor-2 is located in the shared memory. Processor-1 writes in the circular buffer while Processor-2 reads the stored data from the circular buffer. Every time Processor-1 is going to write in the circular buffer, Processor-1 has to check that circular is not full and there is an empty location to the doubled point \( R \) and the scalar \( d_i \). In case circular buffer is full, Processor-1 has to keep looping until there is available location in the circular buffer. Processor-2 has to check every time that there is a new data stored in circular buffer by Processor-1. Then, Read the data and performing the addition or subtraction operation based on the scalar \( d \).

B. Parallel ECC Scalar Multiplication based on DRM (PECC-DRM)

In this section, we describe the PECC-DRM Algorithm exactly like the same PECC-CRT Algorithm. We extract the PECC-DRM Algorithm by combining Add-Subtract Scalar Multiplication Algorithm with Direct Recoding Method. In Processor-1 there are a three sub tasks have to be performed, see the Processor-1 section in Algorithm 3. The first task is, convert the given integer number \( d \) to singed binary representation \( \{T, 0, 1\} \) by applying DRM method, see example 2. The second task is computing the doubling operations \( n \) times based on the number of the bits in the singed representation (DRM). The third task Processor-1 has to perform, is writing the doubled point \( R \) and scalar \( d_i \) (same \( DRM_i \)) in an empty location in the circular buffer. It is important to note, that performing the writing operation in the circular buffer, should be after checking that buffer is not full by calling the function \( \text{is-full()} \). In case buffer is not full, and \( DRM_i \) is not zero, then push the data to the circular buffer. Pushing the data to the circular buffer by Processor-1 is only in case \( DRM_i \) is not zero.

It is a possible to push all the data to circular buffer even in case \( DRM_i \) is zero. But, that will increase the number of checking operations in Processor-2. It is not recommended to add any an extra operation in the Processor-2 which is responsible about performing the addition and subtraction operations. As known, the execution time of performing a one addition operation is more than doubling operation. Addition and subtraction are expensive operations in comparison with doubling operation. So, is highly recommended to keep the section that will be performed by Processor-2 in a state with less operation. As the same state with PECC-CRT, Processor-2 in the PECC-DRM Algorithm has to perform three sub tasks, see Processor-2 section in the Algorithm 3. The first task is reading the doubled point \( R \) and scalar \( d_i \) (same \( DRM_i \)). Processor-2 has to check that the circular buffer is not empty, then read the stored data which are \( R \) and \( DRM_i \) by calling the Pull function. The second task Processor-2 has to perform is the addition or subtraction operation as we mentioned before. In case the \( d_i \) is 1 perform the addition operation while if \( d_i \) is \( T \) perform the subtraction operation and this is the third sub task. After finding the whole calculation based on the length of the key (number of the \( n \) bits), Processor-2 return the \( Q \) which has the final result of the scalar multiplication.

The main difference between both proposed algorithms is the conversion method using that convert the binary representation to the signed binary representation. In both algorithms, finding the signed binary representation \( \{T, 0, 1\} \) should be by the Processor-1 in the scheme to determine the writing operations in the circular buffer. In case that \( d_i = 0 \), then there is no need to write the doubled point in the circular buffer. Processor-1 has to keep doubling operations without writing until \( d_i = 1 \).
Algorithm 3 Parallel Scalar Multiplication based on DRM (PECC-DRM)

Data: Integer \( d \), Point in EC \( P \)

Result: \( Q = dP \), based on (DRM)

Process 1 DRM Conversion, Doubling Operations (DBL)

\[
\begin{align*}
R & \leftarrow P \\
\text{Bin} & = \text{Convert int to bin}(d) \\
\text{DRM} & = \text{Convert bin to DRM}(``\text{Bin}'') \\
\text{for } i = 0 \text{ to } n-1 & \text{ do} \\
\text{repeat} & \text{ if buffer is full() then} \\
& \quad \text{ continue} \\
\text{else if DRM}_i \neq 0 & \text{ then} \\
& \quad \text{Push} (R, \text{DRM}_i) \\
& \quad R \leftarrow 2R \\
\text{ until } i < n & \text{ end} \\
\text{end} \\
\end{align*}
\]

Process 2 Addition Operations (ADD)

\[
\begin{align*}
\text{begin} & \text{ repeat} \\
& \text{ if buffer is not empty() then} \\
& \quad \text{Pull}(R, \text{DRM}_i) \\
& \quad \text{ if DRM}_i = 1 \text{ then} \\
& \quad \quad Q \leftarrow Q + R \\
& \quad \text{ else } \\
& \quad \quad Q \leftarrow Q - R \\
& \quad \text{ end} \\
& \text{ until buffer is not empty()} \\
& \text{ return } Q \\
\text{end} \\
\end{align*}
\]

because Processor-2 will perform the operation based on non-zero bits. The time complexity of the both proposed methods is different based on the conversion method. As known, the execution time of finding DRM is less than the execution time of finding CRT, in the next section has more details.

V. RESULT AND EXPERIMENTAL

We can summarize that the proposed method is extracting a new algorithm that combines two algorithms: Add-Subtract Scalar Multiplication, and Complementary Recoding Technique. Second Algorithm is combined Add-subtract Scalar Multiplication Algorithm, and Direct Recoding Method. It performs the parallel computing on the extracted algorithm (PECC-CRT), given in Algorithm 1. As well as, performs the parallel computing on extracted algorithm (PECC-DRM), given in Algorithm 2. We performed both algorithms in two versions of the code, Parallel and Sequential. The evaluation of the algorithms is based on the parallel and sequential versions for both PECC-DRM and PECC-CRT.

As with almost all parallel applications, it is important to produce the best sequential code before starting to parallelize the code. Task decomposition strategy is used to divide the work into two Processors to perform the overall scheme to get the best result. Both sequential and the parallel codes are written in Visual C++.Net. We use the Open MP library that is supported in the Visual C++.Net package in order to write the parallel section in the parallel version. As well as, we used a ttmath library under c++ to define a big integer number (bigger than or equal 1024-bits). It is important to note that we use Intel CORE i5 7th-Gen machine to test both versions (Parallel and Sequential) using Windows 10. We performed each key size 10 times and the average execution time is taken for all key sizes as shown in Figure 3 and 4.

In the implementation, we tested six different key sizes for both algorithms in both cases parallel and sequential: 160-bits, 192-bits, 224-bits, 256-bits, 384-bits and 521-bits. We generated a big integer number randomly for all key sizes we use in the implementation. Each number used in the both parallel and sequential versions to determine the number of the addition and subtraction operations.

![Fig. 3: Execution time for PECC-CRT](image-url)

The execution times for serial and parallel versions are taken as shown in the figures for the different key size of the ECC. In the case of PECC-CRT, The variances between serial time and parallel time are a big variance in case of key size 521-bits, 192-bits and 160-bits as shown in figure 3. The speed-up reach 60% in comparison with serial version of the same key size.

![Fig. 4: Execution time for PECC-DRM](image-url)

In case of PECC-DRM algorithm, the variance between serial time and the execution time in parallel version is too big in 192-bits and 160-bits key size. The speed up is 60% in comparison with the execution time of serial version of the same key size.

The testing is according to a random key generated to perform the scalar multiplication. The same key is used to perform the
calculation of the scalar multiplication in both version parallel and serial for each key size.

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Serial</th>
<th>Parallel</th>
<th>Speed-up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>521-Bits</td>
<td>236</td>
<td>160</td>
<td>1.5</td>
<td>74%</td>
</tr>
<tr>
<td>384-Bits</td>
<td>185</td>
<td>126</td>
<td>1.5</td>
<td>74%</td>
</tr>
<tr>
<td>256-Bits</td>
<td>125</td>
<td>86</td>
<td>1.4</td>
<td>72%</td>
</tr>
<tr>
<td>224-Bits</td>
<td>118</td>
<td>78</td>
<td>1.5</td>
<td>76%</td>
</tr>
<tr>
<td>192-Bits</td>
<td>105</td>
<td>65</td>
<td>1.6</td>
<td>81%</td>
</tr>
<tr>
<td>160-Bits</td>
<td>88</td>
<td>54</td>
<td>1.6</td>
<td>81%</td>
</tr>
</tbody>
</table>

Fig. 5: Speed up and Efficiency for CRT

The number of non-zero bits in the key effect in the calculation of the ECC scalar multiplication. The occurrence of the bit 1 or/and T means performing the adding or/and subtraction operations by the Processor-2. The average number of the non-zero bits in the key is around 50% or less because of using the signed binary representation.

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Serial</th>
<th>Parallel</th>
<th>Speed-up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>521-Bits</td>
<td>247</td>
<td>154</td>
<td>1.6</td>
<td>80%</td>
</tr>
<tr>
<td>384-Bits</td>
<td>187</td>
<td>123</td>
<td>1.5</td>
<td>76%</td>
</tr>
<tr>
<td>256-Bits</td>
<td>118</td>
<td>82</td>
<td>1.4</td>
<td>72%</td>
</tr>
<tr>
<td>224-Bits</td>
<td>114</td>
<td>75</td>
<td>1.5</td>
<td>76%</td>
</tr>
<tr>
<td>192-Bits</td>
<td>89</td>
<td>54</td>
<td>1.6</td>
<td>82%</td>
</tr>
<tr>
<td>160-Bits</td>
<td>73</td>
<td>46</td>
<td>1.6</td>
<td>79%</td>
</tr>
</tbody>
</table>

Fig. 6: Speed up and Efficiency for DRM

The execution time of one adding operation (or subtraction) is round two times and a half of execution time of the doubling operation. Adding operation is much coast in comparison with doubling operation. Therefore, the occurrence of non-zero bits in the key even its around 50% doesn’t mean that the Processor-1 will process the operation more than Processor-2. In this case it is important to note, that one adding operation has a 5 sub-operations which are 2 squaring, 2 multiplications and 1 inversion, that make an adding operation is an expensive operation in comparison with doubling operation. Therefore, performing the whole calculation of the scalar multiplication by this method insure some kind of balancing. We can see the efficiency of the whole calculation of the different key size is around 70% to 80%, see both figures 5 and 6.

DRM is low cost operation in comparison with CRT and others conversion method. In DRM, the time complexity of the conversion is less than the time complexity of conversion by applying the CRT. As well as, the number of non-zero bits in the signed binary converted by DRM is less than signed binary converted by CRT and other standard method. As mentioned above in the example 2. The hamming weight of DRM representation is 2, which is less than hamming weight of CRT representation. Less hamming weight will save the calculation time of finding ECC scalar multiplication in comparison of using another representation. In the figure 5, we can recognize the difference in the execution time of both DRM and CRT for both serial and Parallel version. The calculation of the ECC scalar multiplication usign DRM Representation is faster than using CRT representation. Overall...

Fig. 7: Execution time for CRT and DRM

Key sizes and in both serial and parallel versions, finding scalar multiplication based on DRM representation is faster than findign the scalar multiplication based on CRT.

VI. CONCLUSION

In this work, we proposed two algorithms to calculate the ECC scalar multiplication based on CRT and DRM representation. The first algorithm based on CRT representation and the second algorithm based on DRM representation. We proposed a parallel algorithm to perform both calculation of the two proposed algorithm separately using two processors. The result show speed-up reach to 60% in comparison with a serial version for both algorithms. As well as, we introduced the difference in execution time for both DRM and CRT. Future work includes using three threads to perform the calculation in case the number of non-zero bits in the key is more than usual, which will make the calculation of adding point more costly — the third thread will help to reduce the execution time in this case.

REFERENCES


