The first efficient generic-point parallel scalar multiplication method is presented here. The novelty of the proposed method is that the precomputation overhead can be replaced by postcomputations that can be parallelised. This method will be very attractive for use in high-performance end servers that employ parallel elliptic curve cryptoprocessors.

*Introduction:* Scalar multiplication is an essential operation in elliptic curve cryptosystems [1]. A scalar multiplication operation is denoted as $kP$, where $k$ is an integer and $P$ is a point on the elliptic curve that represents the addition of $k$ copies of point $P$. $kP$ can be computed using the straightforward binary method [2] based on the binary expression of multiplier $k$. However, sequential scalar multiplication methods are too slow for high-performance end servers because of the demand resulting from increasing numbers of customers, and thus efficient scalar multiplication methods are now crucial for such servers. Scalar multiplication methods that can be parallelised are now being used to provide high-speed implementations.

Precomputations [2] have been used to speed up scalar multiplications, but they require sequential steps that cannot be parallelised, and they are primarily advantageous when the elliptic curve point is fixed. However, during secure communication sessions that use public keys, the elliptic curve point changes, as it depends on the public key of the communicating entity, i.e., it is session dependant. This is also the case when digital signatures are used. Hence, the computation of scalar multiplications is generally performed with a generic elliptic curve point. Because of the likelihood that the elliptic curve point will be different for each session, the overhead resulting from the necessary precomputations must be considered when estimating the total computational time required. To address these issues, this work presents the first efficient generic-point parallel scalar multiplication method. The novelty of the proposed method is that the precomputation overhead can be replaced by postcomputations that can be parallelised.
Proposed method: The essential concept underlying the proposed method is the replacement of the sequential precomputations with parallelisable postcomputations. Multiplier $k$ is partitioned into $u$ partitions that can be processed in parallel by $u$ processors using the binary method. Postcomputations are then distributed on $u-1$ processors to be performed in parallel. The points that result from processing these key partitions with the postcomputations are finally assimilated to produce $kP$. Let $k = (k_{m-1},\ldots,k_0)$, where $k_{m-1}$ is the most significant bit of $k$, be the binary representation of multiplier $k$. Then, after partitioning $k$ into $u$ partitions, multiplier $k$ can be written as

$$k = (k^{(u-1)}||k^{(u-2)}|\ldots||k^{(0)}).$$

Scalar multiplication product $kP$ can then be computed as

$$kP = \sum_{0\leq i \leq u} t_i,$$  \hspace{1cm} (2)

where $t_i$ is defined as

$$t_i = 2(\cdots 2(2k_{iv+\nu-1}(2^{iv}P) + k_{iv+\nu-2}(2^{iv}P)) + \cdots + k_{iv+1}(2^{iv}P)) + k_{iv+0}(2^{iv}P).$$  \hspace{1cm} (3)

A key observation is that Eq. (3) can be rewritten as

$$t_i = (2^{iv})[2(\cdots 2(2k_{iv+\nu-1}P + k_{iv+\nu-2}P) + \cdots + k_{iv+1}P) + k_{iv+0}P].$$  \hspace{1cm} (4)

Eq. (4) implies that the required precomputations of Eq. (3) can be replaced by postcomputations, which are point doublings. Each partition requires $iv$ point doublings to produce the correct partial product. To balance the number of point operations, we need to balance the total number of field multiplications, as field multiplication is the dominant type of operation in elliptic curve point operations in projective coordinates [2]. This implies that multiplier $k$ should be partitioned into $u$ partitions of different sizes, as shown in Eq. (5).

$$m = m_{(u-1)} + m_{(u-2)} + \cdots + m_{(1)} + m_{(0)}.$$  \hspace{1cm} (5)

Accordingly, the number of bits in partition $t_{(i)}$ must be greater than the number of those in $t_{(i+1)}$ and fewer than the number of those in $t_{(i-1)}$, as can be seen from Eq. (6).
Assume that the double and add point operations require \( r \) and \( s \) field multiplications, respectively. Then, let the total number of field multiplications in partition \( k^{(i)} \) equal \( M^{(i)} \).

Because partition \( k^{(0)} \) is the only one to require no postcomputations, a balanced number of point operations can be reached by solving Eqs. (5) and (6) together with the following equations (7-9).

\[
M_{(0)} = m_{(0)}(r) + \frac{m_{(0)}(s)}{2}. \tag{7}
\]

\[
M_{(i)} = m_{(j)}(r) + \frac{m_{(j)}(s)}{2} + (r) \sum_{0 \leq j < i} m_j. \tag{8}
\]

\[
M_{(0)} = M_{(1)} = \cdots = M_{(u-1)}. \tag{9}
\]

The computation of \( kP \) in parallel without precomputations can be performed efficiently using the following algorithm.

\textbf{Algorithm 1: Generic-Point Parallel Scalar Multiplication}

1. Inputs: \( P, k \)

2. By padding \( k \) with zeros if necessary, solve Eqs. (5)-(9) together, and write \( k = (k^{(u-1)}||k^{(u-2)}|| \cdots ||k^{(0)}) \), where \( k^{(i)} \) is a partition of length \( m_{(i)} \) bits.

3. Initialisation: \( Q \leftarrow P, R \leftarrow 0. \)

4. Parallel Scalar Multiplication:

   4.1. For \( i = 0 \) to \( u-1 \) do in parallel

      4.1.1. \( Q \leftarrow \text{Binary method } (k^{(i)}, P) \)

      4.1.2. If \( (i > 0) \), then

         4.1.2.1. for \( c = 0 \) to \( (\sum_{0 \leq j < i} m_j) - 1 \) do

            4.1.2.1.1. \( Q \leftarrow 2Q \)

            4.1.2.1.2. \( R \leftarrow R + Q \)

4.1.3. \( R \leftarrow R + Q \)

5. Output \( R \)
Example: Let $k = (1000\ 0101\ 1100\ 0011)_2 = (34243)_{10}$, $m = 16$, $u = 4$ and $r = \frac{x}{2}$. The sizes of the key partitions are $m_0 = 9$, $m_1 = 4$, $m_2 = 2$ and $m_3 = 1$. The key partitions are $k^{(0)} = 111000011\_2$, $k^{(1)} = 0010\_2$, $k^{(2)} = 00\_2$, and $k^{(3)} = 1\_2$. The scalar multiplication of these partitions is then computed in parallel according to the following.

$$t_0 = 2(2(2(2(2(2(1)P + (1)P) + (1)P) + (0)P) + (0)P) + (0)P) + (0)P) + (1)P + (1)P = 451P,$$
$$t_1 = (2^0)[2(2(0)P + (0)P) + (1)P] + (0)P] = 1024P,$$
$$t_2 = (2^{15})[2(0)P + (0)P] = 0$$
and $t_3 = (2^{15})[(1)P] = 32768P$.

Finally, $kP$ is computed as

$$kP = t_0 + t_1 + t_2 + t_3 = 451P + 1024P + O + 32768P = 34243P.$$

Performance Analysis: The time required to perform the scalar multiplication for partition $k^{(0)}$, which requires no postcomputations, is the same as that required for the remaining partitions. Thus, the time complexity of the proposed method is equal to $m^{(0)}$ point doubling and $(\frac{m^{(0)}}{2} + u - 1)$ point additions. However, the accumulation process can be further enhanced if more than one processor is involved in the accumulation process, which reduces the accumulation time to $log_2(u)$ point additions. NAF encoding [2] also reduces the average point additions of the binary method to $\frac{m}{3}$. Accordingly, when these enhancements are adopted, the time complexity becomes equal to $(m^{(0)})$ point doublings + $(\frac{m^{(0)}}{3} + log_2(u))$ point additions.

Conclusion: Sequential scalar multiplication methods are too slow for high-performance end servers because of the demand resulting from increasing numbers of customers. Existing parallel methods, however, require sequential precomputations for each new session. For these reasons, we have here proposed the first efficient generic-point parallel scalar multiplication method. The novelty of the proposed method is that precomputation overhead can be replaced by postcomputations that can be parallelised.

References


Authors’ affiliations:
1. Turki F. Al-Somani, Computer Engineering Department, Umm Al-Qura University, PO Box 715, Makkah 21955, Saudi Arabia, email: tfsomani@uqu.edu.sa.
2. M.K. Ibrahim, De Montfort University, School of Engineering & Technology, Leicester LE19BH, UK, email: ibrahim@dmu.ac.uk.