Software Implementation of Parallelized ECSM over Binary and Prime Fields

Jean-Marc Robert[1,2]

[1] Team DALI, Université de Perpignan, France
[2] LIRMM, UMR 5506, Université Montpellier 2 and CNRS, France

Abstract. Recent developments of multicore architectures over various platforms (desktop computers and servers as well as embedded systems) challenge the classical approaches of sequential computation algorithms, in particular elliptic curve cryptography protocols. In this work, we deploy different parallel software implementations of elliptic curve scalar multiplication of point, in order to improve the performances in comparison with the sequential counter parts, taking into account the multithreading synchronization, scalar recoding and memory management issues. Two thread and four thread algorithms are tested on various curves over prime and binary fields, they provide improvement ratio of around 15% in comparison with their sequential counterparts.

Keywords: Elliptic curve cryptography, parallel algorithm, efficient software implementation

1 Introduction

Elliptic curve cryptography (ECC) is widely used in a large number of protocols: secret key exchanges, asymmetric encryption-decryption, digital signatures... The main operation in these protocols is the scalar multiplication (ECSM) defined as $k \cdot P$ where P is a point of order r on an elliptic curve $E(\mathbb{F}_q)$ and $k \in [0, r[$ is an integer. The scalar multiplication is computed with *Double-and-add* approaches which consist of sequences of several hundreds of doublings and additions of curve points. It is thus a costly operation which might be implemented efficiently.

In this paper we consider parallel approaches for software implementation of scalar multiplication. There are two versions of the *Double-and-add* scalar multiplication: the left-to-right and the right-to-left depending on the way the bits of k are scanned. On the one hand, the left-to-right version cannot be parallelized due to the strong dependence of the consecutive doublings and additions. On the other hand, the right-to-left version is easier to parallelize: this was noticed by Moreno and Hasan in [15]. Indeed, in [15], the authors provide an algorithm consisting in one thread producing the points $2^i P$ through consecutive doublings, which are then consumed by a second thread performing all the necessary additions. They did not provide any implementation results of their approach. In

practice this can be challenging to implement efficiently the synchronizations between the two threads.

When the elliptic curve is defined over a binary field $\mathbb{F}_{2^m}$, a formula exists (cf. [12,5]) which computes efficiently the halving of a point, i.e., $\frac{1}{2}P$. This makes possible to perform the scalar multiplication through a sequence of halvings and additions of points. This can be used to parallelize the scalar multiplication into two totally independent threads: one thread performing a halve-and-add scalar multiplication and a second thread performing a double-and-add. This approach has been implemented by Taverne *et al.* in [20] showing a significant speed-up compared to non-parallelized versions.

In this paper we first explore the implementation of the two threads parallel approach of Moreno and Hasan [15]. Specifically, we analyze three different strategies to perform synchronization between both threads: using `signals`, `mutexes` or `busy-waiting` approaches, we propose a synchronization strategy based on this analysis. We also study the best approach for the coding of the integer k: this impacts the number of additions and post-computations, i.e., the work load of the thread performing the additions.

We then investigate a four thread parallelization of the scalar multiplication in $E(\mathbb{F}_{2^m})$. This approach combines the *Double/halve-and-add* algorithm of [20] with the approach of Moreno and Hasan.

We provide experimental results for two curves defined over a prime field $p = 2^{255} - 19$ and for the two binary elliptic curves B409 and B233 recommended by NIST in [18]. Our experimental results show that the parallelized scalar multiplication is up to 15 % faster than their non-parallelized counterparts (depending of the curve type and the field size).

The remaining of the paper is organized as follows: in Section 2 we review basic definitions of elliptic curve and scalar multiplication algorithms. In Section 3, we present our implementation approaches of scalar multiplication. We then provide in Section 4 the experimental results and comparisons with the state of the art. We end the paper in Section 5 with some concluding remarks.

2 Background on elliptic curve scalar multiplication

In this section, we briefly review basic results concerning elliptic curve and their use in cryptography. For further details on this matter we refer the reader to [9]. An elliptic curve over a finite field $E(\mathbb{F}_q)$ is the set of point $(x, y) \in \mathbb{F}_q^2$ satisfying a smooth curve equation of degree 3 in x and y plus a point at infinity $\mathcal{O}$. A group law can be defined using the so-called chord-and-tangent approach, providing formulas in terms of point coordinates which compute doubling $2P$ and addition $P+Q$ in the group. The element $\mathcal{O}$ is the neutral element of the group. Cryptographic protocols are based on the intractability of the discrete logarithm problem: given a generator of the group P and a point Q, compute k such that $Q = kP$. The most costly operation involved in most ECC protocols is the scalar multiplication: given $P \in E(\mathbb{F}_q)$ and an integer k, the scalar multiplication consists in computing $kP = P + P + \cdots + P$ (k times). The elliptic curves used

in practice are defined either over prime field $\mathbb{F}_p$ with p prime or over binary field $\mathbb{F}_{2^m}$. In the remainder of this section, we briefly review explicit formulas and algorithms for scalar multiplication over these two fields.

2.1 Scalar multiplication over prime field

Weierstrass Elliptic curve. An elliptic curve E over a prime field $\mathbb{F}_p$ is generally defined by a short Weierstrass equation:

$$E : y^2 = x^3 + ax + b, (a, b) \in \mathbb{F}_p^2.$$

Then, in this case, addition and doubling on $E(\mathbb{F}_p)$ works as follows: let $P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$, be three points of E such that $P_3 = P_1 + P_2$, then we have:

$$\begin{cases} x_3 = \lambda^2 - x_1 - x_2, \\ y_3 = \lambda(x_1 - x_3) - y_1, \end{cases} \text{ where } \begin{cases} \lambda = \frac{y_2 - y_1}{x_2 - x_1} \text{ if } P_1 \neq P_2, \\ \lambda = \frac{3x_1^2 + a}{2y_1} + x_1 \text{ if } P_1 = P_2. \end{cases}$$

Jacobi Quartic curves over prime field. This curve was suggested by Billet *et al.* in [4]. The curve equation of E is:

$$y^2 = x^4 - \frac{3}{2}\theta x^2 + 1, \theta \in \mathbb{F}_p.$$

For such curve, the addition and doubling formulas are unified. Let $P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$, be three points of E such that $P_3 = P_1 + P_2$, then we have:

$$\begin{cases} x_3 = (x_1 y_2 + y_1 x_2)/(1 - (x_1 x_2)^2), \\ y_3 = ((1 + (x_1 x_2)^2)(y_1 y_2 + 2a x_1 x_2) + 2x_1 x_2 (x_1^2 + x_2^2))/(1 - (x_1 x_2)^2)^2. \end{cases}$$

The Jacobi Quartic curve is isomorphic to the following Weierstrass elliptic curve:

$$y^2 = x^3 + ax + b \text{ where } a = (-16 - 3\theta^2)/4 \text{ and } b = -\theta^3 - a\theta.$$

Elliptic curve point operations. The most expensive field operation is the inversion which roughly requires several tens of field multiplications. In order to avoid such operation, additions and doublings utilize projective coordinate system. In our implementation, we consider two systems: the Jacobian coordinate where the point $(X : Y : Z)$ corresponds to the affine point $(X/Z^2, Y/Z^3)$ and the $XXYZZ$ coordinate system where the point $(X : XX : Y : Z : ZZ)$ corresponds to the affine point $(X/Z, Y/ZZ)$ with $XX = X^2$ and $ZZ = Z^2$. Explicit formulas for addition and doubling in these systems can be found in [1]

The resulting complexities are shown in Table 1, which shows that the complexities of the Jacobi Quartic curve operations are better than for the Weierstrass equation case. Moreover, based on the elliptic curve formula database in [1], the Jacobi Quartic curves provide the most efficient point operation among all known curves and formulas. This is the reason why we used such curve and these formulas in our implementations.

Complexity comparison for: point operations	Weierstrass with Jacobian coord.	Jacobi Quartic curve with $XXYZZ$ coord.
Doubling	$4M + 4S + 8R$	$3M + 4S + 7R$
mixed Addition	$9M + 3S + 12R$	$6M + 3S + 9R$
full projective Addition	$13M + 2S + 15R$	$7M + 4S + 11R$

Table 1. Weierstrass curve and Jacobi Quartic curve point operations, M = multiplications, S = squaring, R = field reduction.

Scalar multiplication algorithm. The basic method to compute a scalar multiplication consists in scanning the bits k_i of $k = \sum_{i=0}^{t-1} k_i \cdot 2^i$ and performing a sequence of doubling followed by an addition when $k_i = 1$. This approach is described in Algorithm 1.

In order to reduce the number of additions, the non adjacent form (NAF) and the window non adjacent form (W-NAF) recoding of the scalar are well-known methods, which reduce the number of non zero digit representing the scalar. In the binary scalar representation, half of the digits are either zero or one on average. In the NAF representation, one uses three digits instead of two: $k = \sum_{i=0}^{t} k_i \cdot 2^i$ with $k_i \in \{-1, 0, 1\}$ and there are only $t/3$ non zero digits k_i on average.

The W-NAF representation extends this concept by using more digits: $k = \sum_{i=0}^{t} k_i \cdot 2^i$ with $k_i \in \{-(2^{w-1} - 1), \ldots, -5, -3 - 1, 0, 1, 3, 5, \ldots, (2^{w-1} - 1)\}$. The number of non zero digits is now $t/(w + 1)$ on average. Algorithm 1 can be adapted to use k recoded as NAF or W-NAF. The complexities of the resulting scalar multiplication are given in Table 2.

	nb. of doublings	nb. of additions
Double-and-add	$t - 1$	$t/2$
NAF Double-and-add	t	$t/3$
W-NAF Double-and-add	t	$t/(w + 1) + 2^{w-2} - 1$

Table 2. Complexity comparison between binary, NAF and W-NAF scalar representation in terms of t the bit length of the scalar.

The reader may refer to [8] for further details and algorithms to compute NAF and W-NAF representation.

2.2 Elliptic curve scalar multiplication over binary field

An elliptic curve E over a binary field $\mathbb{F}_{2^m}$ is the set of points $P = (x, y) \in \mathbb{F}_{2^m}^2$ satisfying the following equation:

$$E : y^2 + xy = x^3 + ax^2 + b, (a, b) \in \mathbb{F}_{2^m}^2.$$

Algorithm 1 `Left-to-Right Double-and-add`

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_{2^m})$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from $t-1$ downto 0
3: $\quad Q \leftarrow 2 \cdot Q$
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **endif**
7: **endfor**
8: **return** (Q)

Algorithm 2 `Right-to-left Halve-and-add`

Require: $P \in E(\mathbb{F}_{2^m})$ of order r and k an integer in $[0, r[$
Ensure: $Q = k \cdot P$
1: Compute $k' = 2^t \cdot k \mod r = \sum_{i=0}^{t} k'_i 2^i$ with $t = \lfloor \log_2(r) \rfloor + 1$
2: $Q \leftarrow \mathcal{O}$
3: **for** i from t downto 0
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **endif**
7: $\quad P \leftarrow P/2$
8: **endfor**
9: **return** (Q)

Let $P_1 = (x_1, y_1), P_2 = (x_2, y_2), P_3 = (x_3, y_3)$, be three points of E such that $P_3 = P_1 + P_2$, then we have:

$$\begin{cases} x_3 = \lambda^2 + \lambda + x_1 + x_2 + a, \\ y_3 = (x_1 + x_3)\lambda + x_3 + y_1, \end{cases} \text{ where } \begin{cases} \lambda = \frac{y_1 + y_2}{x_1 + x_2} \text{ if } P_1 \neq P_2, \\ \\ \lambda = \frac{y_1}{x_1} + x_1 \text{ if } P_1 = P_2. \end{cases} \tag{1}$$

Elliptic Curve Scalar Multiplication with halving. It was noticed by Knudsen in [12] that over a binary field, halving of points is possible in case of points of odd order since 2 admits an inverse modulo the order of the point. In other words, point *halving* is the reciprocal operation of point doubling: given $Q = (u, v) \in E(\mathbb{F}_{2^m})$, one looks for $P = (x, y) \in E(\mathbb{F}_{2^m}), P \neq -P$ such as $Q = 2 \cdot P$. Based on equation (1), we know that x, y, u and v satisfy the following relations:

$$\lambda = x + y/x \tag{2}$$
$$u = \lambda^2 + \lambda + a \tag{3}$$
$$v = x^2 + u(\lambda + 1) \tag{4}$$

Consequently, in order to compute P, we first have to solve equation (3) to get λ (which means solve $\lambda^2 + \lambda = u + a$), then, equation (4) gives $x = \sqrt{v + u(\lambda + 1)}$, and finally, equation (2) gives $y = \lambda x + x^2$. The reader may refer to Knudsen in [12] and Fong *et al.* in [5] for further details. In practice, this can be implemented efficiently and has roughly the same cost as two field multiplications (see [20]).

The *Double-and-add* method can be modified into a *Halve-and-add* scalar multiplication. Preliminary, we need to change the scalar. Assuming the point P to be multiplied is of odd order r, we compute $k' = 2^t \cdot k \mod r = \sum_{i=0}^{t} k'_i 2^i$ with $t = \lfloor \log_2(r) \rfloor + 1$. Then, we have $k \equiv k'/2^t \equiv \sum_{i=0}^{t} k'_i 2^{i-t} \mod r$ and the scalar multiplication can be computed as follows:

$$k \cdot P = (k'_t + k'_{t-1} \cdot 2^{-1} + \ldots + k'_0 2^{-t}) \cdot P.$$

This can be computed as a sequence of halvings and additions as shown in Algorithm 2.

Cost of elliptic curve point operations. Over a field of characteristic 2, and in order to avoid the inversions during the computation, which is the most expensive field operation again, one may use projective coordinate systems. The most interesting systems are the Lopez-Dahab ($\mathcal{LD}$, as shown in [8]) and the Kim-Kim ($\mathcal{KK}$, see [11]) projective coordinate systems. With such point representation, the addition and doubling operations do not include any inversion as shown in Table 3, and the whole scalar multiplication is computed with a significant speed-up. Table 3 shows that the complexities of $\mathcal{KK}$ are slightly better and then, when possible, we give the preference to the $\mathcal{KK}$ coordinate system.

Point operation	Coord. System	Cost
Doubling	$\mathcal{LD}$	$4M + 4S + 8R$
Mixed Addition	$\mathcal{LD}$	$9M + 4S + 13R$
Projective Addition	$\mathcal{LD}$	$13M + 4S + 17R$
Doubling	$\mathcal{KK}$	$4M + 5S + 7R$
Mixed Addition	$\mathcal{KK}$	$8M + 4S + 9R$
Halving	Affine	$1M + 1SR + 1R + 1QS$

Table 3. Elliptic curve point operations, M = multiplications, S = squaring, SR = square root, QS = quadratic solver, R = reduction.

3 Strategies for parallel implementation of scalar multiplication

In this section, after a quick review of the implementation strategies used for the field operations, we expose how we elaborate the parallelized algorithm, taking into account all the constraints for such concurrent programming.

The platform used for the experimentations is an Optiplex 990 DELL®, with a Linux 12.04 operating system. The processor is an Intel core i7®-2600 Sandy Bridge 3.4GHz. This processor owns four physical cores, which corresponds to the maximum thread number of our implementations. The code is written in C language and compiled with `gcc` version `4.6.3`.

3.1 Field implementation strategies

Prime field implementation strategies. We considered the prime field $\mathbb{F}_p$, with $p = 2^{255} - 19$, which was introduced by Bernstein in [2]. To compute the field operations, we reused the publicly available code of Adam Langley in [13]. Based

on our experiments, the code of Langley is significantly more efficient compared to low level functions of the GMP library [6] for the considered field. In the code of Adam Langley a field element is stored in a table of five 64 bit words, each word containing only 51 bits. This allows a better management of carries in field addition and subtraction operations. The field multiplications and squarings are performed in two steps, which are multiprecision integer multiplication (respectively squaring) and a modular reduction of integer of 510 bit size (less than p^2) into field element of size 255 bits (reduction modulo p). The multiprecision integer multiplications and squarings are computed with the schoolbook method. The squaring operation is optimized with the usual trick which reduces the number of word multiplications. The reduction modulo $p = 2^{255} - 19$ of 510 bit size integer consists in multiplying by 19 the 255 most significant bits and adding the result to the 255 least significant bits. An inversion of a field element is computed using the Itoh-Tsujii method [10]: $a^{-1} \equiv a^{p-2} \mod p$, and the exponentiation to $p - 2$ is performed with a sequence of squarings and multiplications.

Binary field implementation strategies. Our implementations deal with NIST recommended fields $\mathbb{F}_{2^{233}} = \mathbb{F}[x]/(x^{233} + x^{74} + 1)$ and $\mathbb{F}_{2^{409}} = \mathbb{F}[x]/(x^{409} + x^{87} + 1)$. Concerning the binary polynomial multiplication, we apply a small number of recursions of the Karatsuba algorithm. The Karatsuba algorithm breaks the m bit polynomial multiplication into several 64 bit polynomial multiplications. Such 64 bit multiplications are computed with the `PCLMUL` instruction, available on Intel Core i7 processors. Due to the special form of the irreducible polynomials, the reduction is done with a small number of shifts and bit-wise `XORs`. We compute the field inversion with the Itoh-Tsujii algorithm, that is a sequence of field multiplications and multisquarings performed with look-up table. For field squaring, square root and quadratic solver (needed in halvings), we also use a look-up table method, which is the fastest way according to our tests.

Remark 1. The use of Karatsuba for binary fields and schoolbook method for the prime field is due to the relative cost of word addition compared to word multiplication and to carry managements on our platform. Indeed, integer word additions and multiplications have roughly the same cost (1 vs 2 cycles). The use of Karatsuba algorithm for $\mathbb{F}_p$ decreases the number of word multiplications, but, in counter part, it increases the number of additions and carry managements. For binary field the relative cost of addition (bit-wise XOR) and multiplications (PCLMUL instruction) is more important: 1 cycle vs 10 cycles. In this case Karatsuba is efficient to decrease the timing of a field multiplication.

3.2 Parallelization

The left-to-right *Double-and-add* algorithm (see Algorithm 1 page 5) does not allow any parallelization of the computations, due to the read-after-write dependency inside each loop iteration, between step 5 (addition) and step 3 (doubling). It is necessary to use the right-to-left variant of this algorithm (see Algorithm

3) which allows the parallelization. Indeed Algorithm 3 can be parallelized into two threads as follows:

- A producer-thread performing the sequence of doublings generating the points $2^i P$.
- An addition-thread accumulating the points generated by the producer-thread.

In the sequential case, the left-to-right *Double-and-add* algorithm (Algorithm 1) is better, because the point addition in step 5 can use a mixed coordinate addition. This is faster than the projective addition used in the right-to-left version in step 4 (Algorithm 3). We will see that this penalty is overcome in most of the cases, thanks to the parallelization.

The *Halve-and-add* algorithm (Algorithm 2 page 5) can also be parallelized with two threads. Indeed, since the computation in step 7 of Algorithm 2 only depends on the same step in the previous loop iteration (read-after-write dependency), the sequence of halvings (step 7) can be performed in a separate thread (the producer-thread) and the addition in an addition-thread which accumulates the points generated by the producer-thread.

Algorithm 3 Right-to-left Double-and-add

Require: $k = (k_{t-1}, \ldots, k_1, k_0), P \in E(\mathbb{F}_q)$
Ensure: $Q = k \cdot P$
1: $Q \leftarrow \mathcal{O}$
2: **for** i from 0 to $t-1$ **do**
3: **if** $k_i = 1$ **then**
4: $Q \leftarrow Q + P$
5: **end if**
6: $P \leftarrow 2 \cdot P$
7: **end for**
8: **return** (Q)

Synchronization between threads. Both parallelization (right-to-left *Double-and-add*, Algorithm 3 and *Halve-and-add*, Algorithm 2) are classical producer-consumer configurations.

The safest way to guarantee absolute correct computation is to use a strong synchronization device, processing the computation by small batches: the producer-thread computes and stores a small batch of point doublings/halvings, sends a signal in order to trigger the addition computation in the addition-thread only concerning the batch in shared memory. In parallel, the producer-thread goes on with the next batch and the addition-thread waits the end of each batch before processing the corresponding additions (in the way described by Mueller in [16] or by Tannenbaum in [19]). In our case, on the one hand, the batch size has to be small to compute the maximum of additions in parallel. But on the other hand, if the batches are too small, the synchronization cost would increase, due to the bigger number of synchronization signals to manage. This is especially true as

the granularity of doublings/halvings and additions (several hundreds of processor clock cycles) is too small in comparison with the cost of synchronization barriers and signals.

The three following methods can be used to synchronize the two threads:

- `mutex.` A `mutex` is a mutual exclusion lock provided by the `pthread` library used to synchronize threads. When a thread holds a `mutex`, another thread, trying to take it, is locked, waiting for the releasing of the `mutex` from the first thread. `Mutexes` are generally used to protect critical sections of code. The cost of a lock or an unlock is about 150-200 processor clock cycles, which is almost negligible.
- `signals`: they are used in the inter-thread and inter-process communication. A thread waiting for a signal is put in a sleeping state until another thread sends the corresponding signal. Then, the thread wakes up and goes on running. The sleeping state allows savings of resources which are then available for another process. In our experience and on our platform, the cost to send a signal is about 2000 clock cycles.
- `busy-waiting`: this method consists in using a shared flag (in the global memory) and use it to keep the addition-thread in a *busy-waiting* loop while waiting for the producer-thread to output the next point and modify the flag. The main drawback of this method is to waste processor resources.

According to our experiments, `signals` are too costly compared to the two other techniques. The `busy-waiting` and `mutex` techniques almost give the same results in terms of performance, although the `mutex` method is slightly better in some cases. Thus we decided to use exclusively `mutexes`.

Proposed synchronization method. Our strategy was to avoid the use of mutex synchronization as much as possible. We chose to use only one single `mutex`: at the very beginning of the computation the mutex keeps the addition-thread in an inactive state while a first batch of doublings or halvings is computed by the producer-thread. At the end of the computation of this batch, the producer-thread releases the mutex and pursues the whole sequence of doubling without performing any further locking on the mutex. This approach is depicted in Figure 1.

The correctness of the final result depends on the size of the first batch of points before the `mutex` releasing, which ensures that the writings of the point stored in shared memory by the doubling thread precedes the reading of the same point by the addition thread. If this batch is too small and in case of long sequence of zeros in the binary or NAF scalar representation, one can meet a violation of the read-after-write dependency, and the computation is not correct. To avoid this configuration, we carefully tuned this batch size in order to have the error rate as close as possible to zero. In our test results shown below, this error rate is limited to less than 1%. This is a compromise chosen in order to limit the first batch of doublings/halvings size, and to get the best performances. But at this step, such an error rate remains unacceptable.

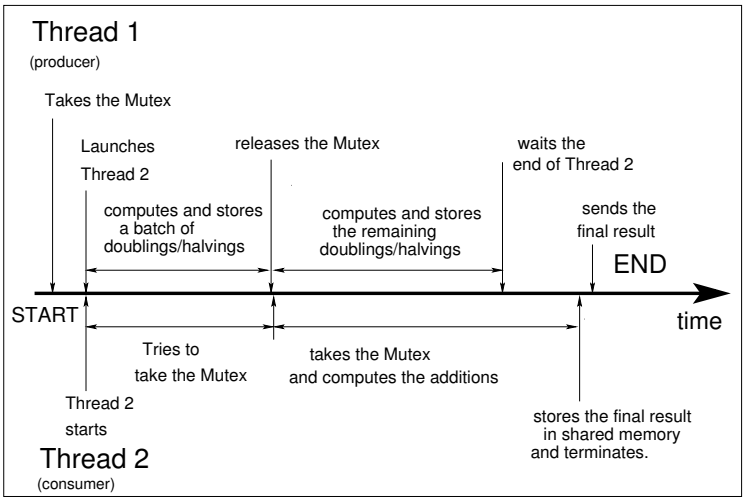


Fig. 1. Synchronization and thread processing for our ECSM implementation

In order to eliminate these errors, we added a test on the addition-thread. In the producer-thread, we used a variable which is stored in global memory as the loop counter. This allows to check if the addition processed uses a point which has already been computed by the producer-thread, i.e. the read-after-write dependency is ensured. The cost of this test is almost negligible, although the use of a global memory counter is not totally free. When an error is detected (that is to say a read-after-write dependency violation), we break the addition-thread loop, and launch a sequential computation of $k \cdot P$. Due to the small error rate, the cost of this rescue computation, which frequency is near zero, is negligible on average.

Algorithm 4 presents an algorithmic formulation in the case of Right-to-left *Double-and-add* scalar multiplication of this approach, including the elimination of the error computations due to a synchronization failure.

Impact of scalar recoding. In the sequential case, it is a useful technique to recode the scalar using NAF and W-NAF to speed-up the computation (as previously mentioned in Subsection 2.1 page 4). In the parallel algorithms, the situation is different. Indeed, the NAF and W-NAF recodings reduce the number of additions performed by the addition-thread. This fact can be seen when analyzing the amount of computations performed by the two threads. We can evaluate this amount using the results given Table 2 and Table 1 in the case of curves over $E(\mathbb{F}_p)$, and using the results given Table 2 and Table 3 in the case of curves over $E(\mathbb{F}_{2^m})$. For simplicity we assumed that $S = 0.8M$ in $\mathbb{F}_p$ and that a squaring and square root are negligible in $\mathbb{F}_{2^m}$ and that the cost of a quadratic solver is roughly $1M$. The resulting complexities are given in Table 4.

	Double-and-add over $\mathbb{F}_p$			*Double-and-add* over $\mathbb{F}_{2^m}$			*Halve-and-add* over $\mathbb{F}_{2^m}$		
Recoding	producer-thread	addition-thread	post-comp.	producer-thread	addition-thread	post-comp.	producer-thread	addition-thread	post-comp.
binary	$6.2tM$	$5.1tM$	0	$4tM$	$6.5tM$	0	$2tM$	$4tM$	0
NAF	$6.2tM$	$3.4tM$	0	$4tM$	$4.33tM$	0	$2tM$	$2.66tM$	0
W-NAF ($w = 4$)	$6.2tM$	$2.04tM$	$33M$	$4tM$	$2.6tM$	$39M$	$2tM$	$1.6tM$	$39M$

Table 4. Complexity of the two threads for a t-bit scalar coded in binary, NAF and W-NAF, in multiplication number.

Table 4, we remark that, generally, the amount of computation of the addition-thread is larger than the producer-thread for the binary coding. When using the NAF recoding the amount of computation of the two threads are roughly the same. Finally, the use of W-NAF makes the amount of computation of the addition-thread significantly smaller than the producer thread. This means that when using W-NAF recoding, the addition-thread progresses faster and even would have to wait for the producer-thread to output new points. But in any case, the addition-thread terminates after the producer-thread. Moreover in the W-NAF case, the post-computations delay the output of the results after the end of the producing process, since in the parallel algorithms, this final reconstruction cannot be done before the end of the parallelized additions.

These remarks are confirmed by the chronogram given in Figure 2 which shows the different timings required by each thread related to the recoding used for the execution of the parallelized halve-and-add for scalar multiplication in $E(\mathbb{F}_{2^{233}})$. This fact leads us to opt for the NAF recoding for our implementations.

In addition to this choice, and in order to improve the performances, we implement a variable initial batch size of doublings/halvings in the producer-thread. Indeed, the number of additions performed by the addition-thread depends on the Hamming weight of the NAF representation of the scalar (i.e. the non-zero digits). As stated previously, a read-after-write dependency violation can appear if this batch is too small. But if the Hamming weight of the scalar is higher, the risk of this dependency violation is lower, and the batch size can be reduced in this case. This improvement applies only when the addition-thread has roughly the same running time as the producer thread. This concerns the *Double-and-add* approach over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ but not the *Halve-and-add* approach (cf. Table 4).

When possible, we use the variable initial batch size in the producer-thread.

3.3 Four-thread parallel version over binary elliptic curve

Over binary field, the parallelization proposed by Taverne *et al.* in [20] splits the scalar multiplication into two independent threads. Specifically, they split the t-bit scalar $k = k_1 + k_2$ where k_1 and k_2 are as follows

$$k = \underbrace{(k'_t 2^{t-\ell} + \ldots + k'_\ell)}_{k_1} + \underbrace{(k'_{\ell-1} 2^{-1} + \ldots + k'_0 2^{-\ell})}_{k_2}. \tag{5}$$

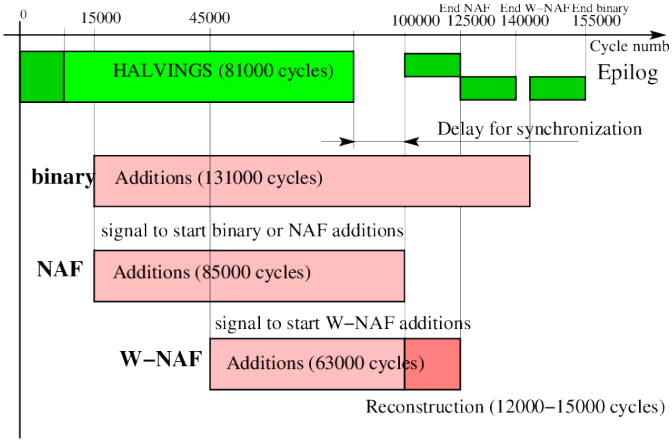


Fig. 2. Chronogram of the *halve-and-add* computation with binary, NAF and W-NAF scalar representation over B233

Algorithm 4 Parallel *Double-and-add* Elliptic Curve Scalar Multiplication

Require: scalar k, $P \in \mathbb{F}_{2^m}$.
Ensure: kP.
(Barrier)

Compute Doublings (producer-thread)

1: $D[0] \leftarrow P$
2: **for** $glblMmry.i = 1$ to $initBtchSze$ **do**
3: $//Doubling$ LD projective $D[glblMmry.i] \leftarrow D[glblMmry.i-1] \times 2$
4: **end for**
5: signal to thread addition
6: **for** $glblMmry.i = initBtchSze + 1$ to $M-1$ **do**
7: $//Doubling$ LD projective $D[glblMmry.i] \leftarrow D[glblMmry.i-1] \times 2$
8: **end for**

Compute Additions (addition-thread)

9: $Q \leftarrow \mathcal{O}$
10: Wait for signal from thread Doubling
11: **for** $i = 0$ to $M-1$ **do**
12: **if** $i > glblMmry.i - 1$ **then**
13: launch rescue computation ($Q \leftarrow kP$)
14: break
15: **end if**
16: **if** $k_i = 1$ **then**
17: //Full LD projective addition $Q \leftarrow Q + D[i]$
18: **end if**
19: **end for**

(Barrier)
20: **return** Q

In general ℓ is close to $t/2$ and represents the length of the *Halve-and-add* subkey. Then the computations can be parallelized into one thread computing k_1P with the *Double-and-add* algorithm and a second thread computing k_2P with the *Halve-and-add* algorithm.

We propose to combine the approach of Taverne *et al.* with the parallelization approach discussed in Subsection 3.2. This results in a four-thread algorithm: the partial scalar multiplication k_1P is computed with the parallel two-thread algorithm *// Double-and-add* and k_1P is computed with the parallel two-thread algorithm *// Halve-and-add*. This four-thread approach is shown in Figure 3. This approach increases the level of parallelization, but it also requires additional thread launching and management. Therefore, this algorithm works better on large fields, as it will be shown in the next section.

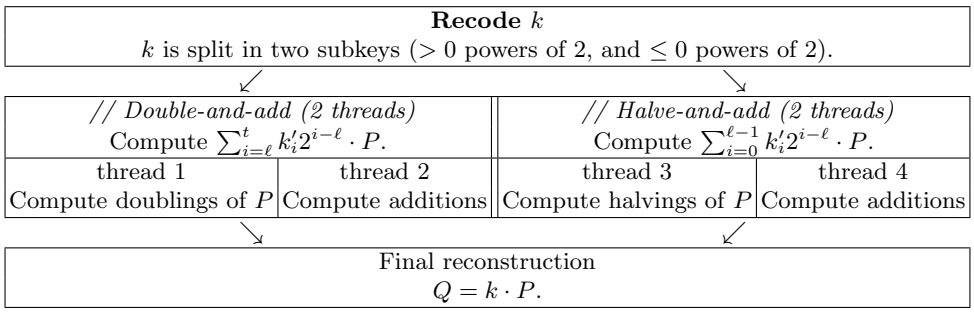


Fig. 3. Four-thread algorithm.

Our implementations of the four-thread of Figure 3 use the following strategies for thread launching and synchronization:

- The threads are launched in this order: 1) the *Halve-and-add* producer-thread, which launches 2) the *Double-and-add* producer-thread which launches 3) the *Halve-and-add* addition-thread which finally launches 4) the *Double-and-add* addition-thread.
- The recoding of the scalar is done by 3) the *Halve-and-add* addition-thread before launching 4) the *Double-and-add* addition-thread.
- Due to the delay of thread launching and key recoding computation, it is not necessary to use mutexes with initial batch size of halvings or doublings of points for each producer-thread.
- In order to eliminate computation errors due to synchronization failure, we use the same method as the one described Subsection 3.2 for. Thus, we have two global memory counters, one for the doubling producer-thread and one for the halving producer-thread. Each addition-thread compare its own counter with the global counter of its corresponding, and launches a partial rescue computation if a synchronization dependency violation is detected.

4 Timings

The platform used for the experimentations is an Optiplex 990 DELL®, with a Linux 12.04 operating system. The processor is an Intel Core i7®-2600 Sandy Bridge 3.4GHz, which owns four physical cores. The code is written in C language, compiled with `gcc version 4.6.3`. The *Hyperthreading*® BIOS and also the Turbo-boost® options have been deactivated on our platform in order to measure the performances as accurately as possible.

Since the operating system has the possibility to preempt the resources in order to launch another task, we avoid such difficulties by choosing to run our codes in a recovery mode shell. But we noticed that the codes generally run well in normal operating system conditions too, although perturbations may be observed in a few cases.

		Binary Field		**Prime Field $\mathbb{F}_p$**	
		B233	**B409**	**Weierstrass**	**Jacobi Quartic**
References					
Sequential 1 thread	*Double-and-add*	159000 (W-NAF, $w = 4$)	706000	256631 (NAF)	222558
	Halve-and-add	135000 (W-NAF, $w = 4$)	534000	-	-
2 threads	*Dbl/Hlv-and-add*	98000 (W-NAF, $w = 4$)	347000	-	-
NAF // *Double-and-add*					
2 threads	mean	**154621**	**598491**	**218606**	**184048**
	Doublings	114713	505662	168958	134398
	Additions-D	120748	522869	125990	87415
NAF // *Halve-and-add*					
2 threads	mean	**126639**	**430222**		
	Halvings	81630	300113		
	Additions-D	85107	373534		
NAF // *Halve-Double-and-add*					
4 threads	mean	**133273** ($\ell = 151$)	**324395** ($\ell = 246$)		
	Doublings	44672	202393		
	Additions-D	39333	200660		
	Halvings	67076	199625		
	Additions-H	55615	217534		

Table 5. Timings (in clock cycles)

We give the parameters of the curves used in the experimentation in Appendix:

- Appendix A.1 for the B233 and B409 binary field curves;
- Appendix A.2 for the Weierstrass prime field curve;
- Appendix A.3 for the Jacobi-Quartic prime field curve.

Table 5 shows the results of the proposed parallel strategies for scalar multiplication implementations. The above timings include the error detection and correction due to erroneous thread synchronization.

The performances are measured using one hundred batches of 2000 computations, each batch with a different random scalar. The minimum value of each batch is considered and the average value gives the performance. With this measurement process, we take into account the variations due to the different Hamming weights of the scalars.

For each case we provide the detailed duration of each thread. We notice that, generally, the overall computation finishes around several tens of thousands cycles after the producer-thread. These timings might correspond to the delayed start of the addition-thread (due to the initial batch size of points computed by the producer thread) and the synchronization and thread management time. For the four-thread versions, the given value ℓ corresponds to the scalar bit size of the *Halve-and-add* computation (cf. equation (5)). We have evaluated the overhead due to error management due to wrong error synchronization and it represents, in average, roughly 2-6% of the overall computation time.

Concerning the results over $\mathbb{F}_{2^{233}}$, we remark that the four-thread version is not competitive. This might be due to the synchronization and thread creation and management cost. Furthermore, the speed-ups with the two thread versions are not very important.

Over $\mathbb{F}_{2^{409}}$, the situation is different since the four-thread version is now better: it requires 324395 clock cycles whereas the two-thread parallel W-NAF *Double/halve-and-add* necessitates 347000 clock cycles (6.6% improvement). The speed-ups provided by the two-thread versions is also more important: between 15% (*Double-and-add* case) and 19.5% (*Halve-and-add* case).

Concerning the results over $\mathbb{F}_p$, we first notice that a scalar multiplication over a Jacobi Quartic is faster than over a Weierstrass curve. This corroborates the complexities of the curve operations shown in Table 1. We also notice that the tested two-thread parallelization provides performance improvements of around 15% to 17% compared to the NAF sequential *Double-and-add* approach.

Comparison. We give in Table 6 some published results in the literature. Over, $\mathbb{F}_p$, the work of Longa is on Intel Core 2 with $p = 2^{256} - 189$ and Hamburg is over a Sandy Bridge with $p = 2^{252} - 2^{232} - 1$ and has smaller key. The other works deal with the same processor and on the same fields as the one considered in this paper. We can see that, in the case of $E(\mathbb{F}_{2^{233}})$ our two-thread approach is not competitive with best know results. In the cases of $E(\mathbb{F}_{2^{409}})$, the proposed approach improves by 9.4% the previous best known timings reported in [20]. Finally, the timing provided in [7] is better than the timing obtain by our method, but Hamburg uses a slightly smaller field and key size. On the other hand, we improve the best known results for curve defined for a 128 bit security level.

5 Conclusion

In this work, we have considered parallelized software implementations of scalar multiplication kP over $E(\mathbb{F}_{2^m})$ and $E(\mathbb{F}_p)$. We first have considered the parallelization suggested by Moreno et Hasan in [15] which splits the right-to-left scalar multiplication into two threads: one producer-thread computing 2^iP or

	Scalar multiplication	Curve	Security	processor	Method	Cycles
$E(\mathbb{F}_p)$	Hamburg [7]	Montgomery	126	Intel core i7 SB	Montgomery ladder	153000
	Langley [13]	Curve25519	128	Intel core i7 SB	Montgomery ladder	229000[1]
	Bernstein [3,2]	Curve25519	128	Intel core i7 SB	Montgomery ladder	194000
	Longa *et al.* [14]	jac256189	128	Intel core 2 Duo	WNAF D&A	337000
	Longa *et al.* [14]	ted256189	128	Intel core 2 Duo	WNAF D&A	281000
	This work	jac25519	128	Intel core i7 SB	//NAF D&A	184048
$E(\mathbb{F}_{2^m})$	Nègre *et al.* [17]	B233	112	Intel core i7 SB	WNAF D-H&A	98000
	Taverne *et al.* [20]	B233	112	Intel core i7 SB	WNAF D-H&A	102000
	This work	B233	112	Intel core i7 SB	//NAF H&A 2 th.	126639
	Nègre *et al.* [17]	B409	192	Intel core i7 SB	WNAF D-H&A	347000
	Taverne *et al.* [20]	B409	192	Intel core i7 SB	WNAF D-H&A	358000
	This work	B409	192	Intel core i7 SB	//NAF D-H&A 4 th.	324395

[1] compiled and run on our platform.

Table 6. Performance comparison with the state of the art

$2^{-i}P$ for $i = 1, \ldots, t$ and one addition-thread which accumulates these points to compute kP. We have proposed a lightweight approach for thread synchronization. In addition, in order to avoid remaining computation error due to dependency violation, we proposed a low cost checking method of the synchronization between threads with a rescue computation. We have also evaluated the best approach for the scalar recoding in this context. In the special case of $E(\mathbb{F}_{2^m})$ we have combined this approach to the parallelized Double/halve-and-add approach of [20]. The experimental results show that these parallelization techniques provide some speed-up on elliptic curve scalar multiplication computations compared to previously best known implementations. Indeed, over prime field and binary fields, in most cases the parallelization provides an improvement of roughly 15% on the computation time.

Acknowledgement: We would like to thank Christophe Nègre for his valuable and helpful comments.

This work has been suported by a PHD grant from PAVOIS project (ANR 12 BS02 002 01).

References

1. Explicit formula database, 2014. http://www.hyperelliptic.org/EFD/index.html.
2. D.J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography*, pages 207–228, 2006.
3. D.J. Bernstein and Lange T. (eds). eBACS: ECRYPT Benchmarking of Cryptograhic Systems. http://bench.cr.yp.to/, 2012. accessed May 25th, 14.

4. O. Billet and M. Joye. The Jacobi Model of an Elliptic Curve and Side-Channel Analysis. In *AAECC*, pages 34–42, 2003.
5. K. Fong, D. Hankerson, J. López, and A. Menezes. Field Inversion and Point Halving Revisited. *IEEE Trans. Computers*, 53(8):1047–1059, 2004.
6. Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. http://gmplib.org/.
7. Mike Hamburg. Fast and compact elliptic-curve cryptography. Technical report, Cryptology ePrint Archive, Report 2012/309, 2012. http://eprint.iacr.org/.
8. D. Hankerson, J. López Hernandez, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *CHES 2000*, volume 1965 of *LNCS*, pages 1–24. Springer, 2000.
9. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
10. T. Itoh and S. Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.
11. K.H. Kim and S.I. Kim. A New Method for Speeding Up Arithmetic on Elliptic Curves over Binary Fields. Technical report, National Academy of Science, Pyongyang, D.P.R. of Korea, 2007.
12. E. W. Knudsen. Elliptic Scalar Multiplication Using Point Halving. In *ASIACRYPT*, pages 135–149, 1999.
13. A. Langley. C25519 code, 2008. http://code.google.com/p/curve25519-donna/.
14. P. Longa and C. H. Gebotys. Efficient Techniques for High-Speed Elliptic Curve Cryptography. In *CHES*, pages 80–94, 2010.
15. C. Moreno and M. A. Hasan. SPA-Resistant Binary Exponentiation with Optimal Execution Time. *J. Cryptographic Engineering*, 1(2):87–99, 2011.
16. F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *USENIX Winter*, pages 29–42, 1993.
17. C. Nègre and J.-M. Robert. Impact of Optimized Field Operations AB, AC and $AB + CD$ in Scalar Multiplication over Binary Elliptic Curve. Technical Report hal-00724785, HAL, July 2014.
18. Deputy Director P. Gallagher and C. Furlani. Digital Signature Standard (DSS). In *FIPS Publications*, volume FIPS 186-3, page 93. NIST, 2009.
19. A.S. Tannenbaum. Modern Operating Systems, 2009. http://www.freewebs.com/ictft/sisop/Tanenbaum_Chapter2.pdf.
20. J. Taverne, A. Faz-Hernández, D. F. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López. Speeding Scalar Multiplication over Binary Elliptic Curves using the New Carry-Less Multiplication Instruction. *J. Cryptographic Engineering*, 1(3):187–199, 2011.

A Appendix: Curve Parameters

A.1 Elliptic curves over binary field

The curve equation is:

$$y^2 + xy = x^3 + x^2 + b \text{ where } b \in \mathbb{F}_{2^m}.$$

The parameters are for B233:

$a = 1$,
$h = 2$,
$f(x) = x^{233} + x^{74} + 1$,
$b =$ `0x00000066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad`,
$r =$ `0x00000100 00000000 00000000 00000000 0013e974 e72f8a69 22031d26 03cfe0d7`.

where the order of the curve is $n \times h$. For B409 we have:

$a = 1$,
$h = 2$,
$f(x) = x^{409} + x^{87} + 1$,
$b =$ `0x0021a5c2 c8ee9feb 5c4b9a75 3b7b476b 7fd6422e f1f3dd67 4761fa99 d6ac27c8 a9a197b2 72822f6c d57a55aa 4f50ae31 7b13545f`,
$r =$ `0x01000000 00000000 00000000 00000000 00000000 00000000 000001e2 aad6a612 f33307be 5fa47c3c 9e052f83 8164cd37 d9a21173`.

A.2 Weierstrass curve over prime field

The curve equation is:

$$y^2 = x^3 - 3x + b \text{ where } b \in \mathbb{F}_p.$$

The parameters are:

$p = 2^{255} - 19$
$b =$ `0x1d09bac9ffe9e7f8284aed0442552779bcdef2e62b9cb1d568513fa798b94003`
$r =$ `0x800000000000000000000000000000012c18945a05ad7f2edf026258ea5288ef`

r is the prime order of P.

A.3 Jacobi Quartic curve over prime field

The curve equation is:

$$y^2 = x^4 - \frac{3}{2}\theta x^2 + 1, \theta \in \mathbb{F}_p.$$

The parameters are:

θ = `0x1731beeea2156180446f9e5ab64af78d4ed3e0eb68d5070c10ef2468b910d5f7`
number of points:
$h \times r =$ `0x8000000000000000000000000000002672bdbb41f31390c5527cab6e282744`
$= 4 \cdot$ `0x2000000000000000000000000000000099caf6ed07cc4e431549f2adb8a09d1`

The Jacobi Quartic curve is isomorphic to the following Weierstrass elliptic curve:

$$y^2 = x^3 + ax + b$$

where: $a = (-16 - 3\theta^2)/4$ and $b = -\theta^3 - a\theta$. Hence, in our case:

$a =$ `0xc500be2450246d16c114830a5d1aef9c2b80c567b4fd87562c69db659713ad2`,
$b =$ `0xa38f53e5d27462dcdada9a78b9eac482ef06e855af92ca704060c551a9a5854`.