2004－AL－98　(6)
2004／11／5

モンゴメリのトリックを用いた 2^kP 倍点の改良計算法

安達 大亮　　平田 富夫
名古屋大学大学院工学研究科
〒 464-8603 名古屋市千種区不老町
Tel: 052-789-3440,　Fax: 052-789-3089
E-mail: {adachi,hirata}@hirata.nuee.nagoya-u.ac.jp

概 要

この論文では素体上で定義された楕円曲線に注目する。我々は、アフィン座標系において、楕円曲線上の点 P からその 2^k 倍の点 2^kP を求めるアルゴリズムを 2 つ提案する。1 つは $k = 2$ の場合に有効であり、もう 1 つは任意の自然数 k に対して有効である。これらのアルゴリズムは、素体上の逆元計算 1 回を素体上の乗算数回と置き換えることに基礎を置いている。我々は、逆元計算の置き換えにモンゴメリのトリックと呼ばれる技法を用いた。妥当な仮定の下で、我々のアルゴリズムは既存のアルゴリズムよりも効率良く動作する。

Refined Computations for Points of the Form 2^kP Using Montgomery Trick

Daisuke Adachi　　Tomio Hirata
Graduate School of Engineering, Nagoya University
Furou, Chikusa, Nagoya, Aichi, 464-8603, Japan
Tel: 052-789-3440,　Fax: 052-789-3089
E-mail: {adachi,hirata}@hirata.nuee.nagoya-u.ac.jp

Abstract

This paper focus on an elliptic curve defined over a prime field. We propose two algorithms for computing points of the form 2^kP in affine coordinates. The one only works for $k = 2$, and the other works for arbitrary natural number k. These algorithms are based on a trade-off between a field inversion and some field multiplications. We apply Montgomery trick to do this trade-off. Under a reasonable assumption, our algorithms are efficient in comparison with existing algorithms.

1 Introduction

In recent years, commercial use of elliptic curve cryptographic schemes has increased. The execution time of elliptic curve cryptographic schemes heavily depends on that of scalar multiplications. This multiplication takes a point P on an elliptic curve over a finite field and computes a scalar multiple dP for some scalar d.

The 2^w-ary method [2, 6] and the sliding window method [4] are useful for a scalar multiplication. These methods usually use the signed binary representation of the scalar [7, 8, 12] and repeatedly compute points of the form 2^kP and $2P + Q$ from points P and Q on an elliptic curve. This computation uses two arithmetics on the elliptic curve; an addition and a doubling. The computation time of additions and doublings varies depending on the coordinate system for representing the elliptic curve.

Here we focus on an elliptic curve defined over GF(p). In affine coordinates, additions and doublings include inversions over the finite field. However, a field inversion is much expensive than a field squaring or a field multiplication. In fact, Sakai-Sakurai [11] reported that the ratio of computation

time of a field inversion to a field multiplication became 25.0 for 160-bit p in their implementation. Therefore, reducing the number of field inversions is important for an efficient scalar multiplication.

One method for reducing the number of field inversions is *direct computation* for points on an elliptic curve. For example, the direct computation for 2^kP computes 2^kP directly from P, computing no intermediate points $2P, 4P, \ldots, 2^{k-1}P$. The concept of direct computation was firstly proposed by Guajardo and Paar [5]. They gave algorithms for direct computation of $4P$, $8P$ and $16P$ on an elliptic curve defined over a binary field in affine coordinates. In recent years, several algorithms of direct computation for 2^kP or $2P + Q$ in affine coordinates have been proposed [1, 3, 5, 10, 11]. Sakai and Sakurai [11] proposed an efficient algorithm for 2^kP on an elliptic curve defined over GF(p). This algorithm works for arbitrary natural number k. Also, Eisenträger, Lauter and Montgomery [3] proposed an efficient algorithm for $2P + Q$ on an elliptic curve defined over GF(p). Moreover, Ciet, Joye, Lauter and Montgomery [1] extended and refined this algorithm for an elliptic curve defined over both GF(p) and a binary field. In Ciet-Joye-Lauter-Montgomery's algorithm, one technique is used for reducing inversions, which is called "Montgomery trick" [9].

This paper is organized as follows. Section 2 describes assumptions and preliminaries. In Sect.3, we propose two algorithms for computing points of the form 2^kP which use Montgomery trick. The one only works for $k = 2$, and the other works for arbitrary natural number k. Section 4 applies our algorithms to scalar multiplication and estimates their savings.

2 Preliminaries

This chapter describes assumptions and preliminaries used in this paper.

2.1 Arithmetics over GF(p)

The set of points on an elliptic curve defined over GF(p) is used for implementation of elliptic curve cryptographic scheme. The set (plus the point at infinity) becomes additive group with addition $P + Q$ for points P, Q on the elliptic curve. If P and Q are different, $P + Q$ is called "addition", and if P and Q are identical, it is called "doubling" and denoted by $2P$.

Additions and doublings are both implemented by several kinds of field arithmetics. Among these arithmetics, a field squaring, a field multiplication and a field inversion are more expensive than other field arithmetics, such as a field addition and a field subtraction. We intend to estimate the efficiency of algorithms for computing points on an elliptic curve by the number of the former three field arithmetics[1]. Moreover, as in [1, 11], we will assume that the cost of a field squaring is 80% as expensive as that of a field multiplication. By dropping "field", we call these field arithmetics just as squaring, multiplication and inversion.

2.2 Addition Formula on Affine Coordinates

Let p denote a prime. $E_p : y^2 \equiv x^3 + ax + b \pmod p$ $(4a^3 + 27b^2 \not\equiv 0)$ is an elliptic curve defined over GF(p). We also focus on the case that the coordinate system for representing points is affine coordinates. Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be points on E_p.

The point $P + Q = (x_{P+Q}, y_{P+Q})$, the result of adding P to Q, is derived from the following formulae:

$$\begin{aligned} \lambda &= \frac{y_Q - y_P}{x_Q - x_P} , \\ x_{P+Q} &= \lambda^2 - x_P - x_Q , \\ y_{P+Q} &= \lambda(x_P - x_{P+Q}) - y_P . \end{aligned}$$

Thus, an addition requires 1 squaring, 2 multiplications and 1 inversion.

[1] We ignore field multiplication by small constant because it is much cheaper than generic field multiplication.

Furthermore, the point $2P = (x_{2P}, y_{2P})$, the result of doubling P, is derived from the following formulae:

$$\begin{aligned} \lambda &= \frac{3x_P^2 + a}{2y_P} \ , \\ x_{2P} &= \lambda^2 - 2x_P \ , \\ y_{2P} &= \lambda(x_P - x_{2P}) - y_P \ . \end{aligned}$$

Thus, a doubling requires 2 squarings, 2 multiplications and 1 inversion.

2.3 Montgomery Trick

Montgomery trick [9] is a technique for simultaneous inversions. As a simple example, the inverses modulo p of two numbers x, y can be calculated by the following way:

$$\begin{aligned} M &= xy, \\ I &= M^{-1}, \\ x^{-1} &= Iy, \\ y^{-1} &= Ix \ . \end{aligned}$$

These formulae indicate that 2 inversions can be replaced by 1 inversion and 3 multiplications. Similarly, the inverses modulo p of m numbers x_1^{-1}, x_2^{-1}, $\ldots, x_m^{-1} \in \mathrm{GF}(p)$ are calculated as follows.

(i) Calculate $\prod_{j=1}^{i} x_j$ for each $i = 2, 3, \ldots, m$ and store them.

(ii) Calculate $(\prod_{j=1}^{m} x_j)^{-1}$.

(iii) Calculate $(\prod_{j=1}^{i} x_j)^{-1} \cdot \prod_{j=1}^{i-1} x_j = x_i^{-1}$ and $(\prod_{j=1}^{i} x_j)^{-1} \cdot x_i = (\prod_{j=1}^{i-1} x_j)^{-1}$ for each $i = m, \ldots, 3, 2$.

In (i), $(m-1)$ multiplications are required, and $(2m-2)$ multiplications are required in (iii). Thus, x_1^{-1}, x_2^{-1}, $\ldots$, $x_m^{-1} \in \mathrm{GF}(p)$ can be calculated by 1 inversion and $(3m-3)$ multiplications. The above replacements are effective if 1 inversion costs more than 3 multiplications.

3 New Efficient Algorithms by Montgomery Trick

In this chapter, we will propose a new algorithm for computing the quadruple point $4P$ from P which uses Montgomery trick. The proposed algorithm saves 2 squarings in comparison with the Sakai-Sakurai's algorithm for $k = 2$. Moreover, we will modify the Sakai-Sakurai's algorithm using Montgomery trick. This modification saves 2 squarings and requires 1 additional multiplication comparing to the original version of the Sakai-Sakurai's algorithm.

3.1 Formulae for Quadrupling

This section handles quadrupling of a point P. The 2^w-ary method [2, 6] is known as an efficient algorithm for scalar multiplication dP. This method reads every w bits of a (signed) binary representation of d from left to right.

The 2^w-ary method

```
INPUT:  d = sum_{j=0}^{t-1} d[j]2^j, P
OUTPUT: a scalar multiple dP of P
Step 1. Precomputation
    Compute vP for all v ∈ {1, ..., 2^w − 1}
Step 2. Evaluation
    Y ← O (point at infinity)
    for j = ⌊(t − 1)/w⌋w down to 0 step w do
      Y ← 2^w Y
      if v = (d[j + w − 1] ... d[j])_2 ≠ 0 then
        Y ← Y + vP
    return Y
```

We focus on the situation in which the 2^w-ary method with small w is adopted to compute dP. This situation frequently appears in case of implementing an elliptic curve cryptographic scheme on a device whose computation resources are limited, such as a smart card. Quadrupling of a point obviously appears for $w = 2$. Also, Quadrupling appears in case of $w = 3$ if this method computes 2^2P first and then computes $2(2^2Y) + vP$. For this computation, direct computation of $2P + Q$ can be used [1, 3].

A straightforward computation of a quadruple point $4P$ is to perform two successive doublings.

$4P = (x_{4P}, y_{4P})$ is computed by

$$\begin{aligned} \lambda_1 &= \frac{3x_P^2 + a}{2y_P} \\ x_{2P} &= \lambda_1^2 - 2x_P \\ y_{2P} &= \lambda_1(x_P - x_{2P}) - y_P \end{aligned}$$

and

$$\begin{aligned} \lambda_2 &= \frac{3x_{2P}^2 + a}{2y_{2P}} \\ x_{4P} &= \lambda_2^2 - 2x_{2P} \\ y_{4P} &= \lambda_2(x_{2P} - x_{4P}) - y_{2P} \ . \end{aligned}$$

Thus, 4 squarings, 4 multiplications and 2 inversions are required. Müller proposed an algorithm for direct computation of $4P$ [10]. His algorithm requires 7 squarings, 14 multiplications and 1 inversion. The Sakai-Sakurai's algorithm for 2^kP can also compute $4P$ if $k = 2$ [11]. This algorithm requires 9 squarings, 9 multiplications and 1 inversion, and thus it is efficient in comparison with the two successive doublings if 1 inversion costs more than 9 multiplications under our assumptions.

We present a new algorithm for computing $4P$ which uses Montgomery trick. Our algorithm is mainly based on the two successive doublings. As we have seen, the straightforward way requires two inverses $(2y_P)^{-1}$ and $(2y_{2P})^{-1}$. To reduce the number of inversions, we focus on the formula for y_{2P}. Letting $e = 2(3x_P^2+a)\{12x_Py_P^2-(3x_P^2+a)^2\}-16y_P^4$, we can see that $e = 16y_P^3y_{2P}$. Defining $E = 2y_Pe$ and $I = E^{-1}$, we can obtain $(2y_P)^{-1}$ and $(2y_{2P})^{-1}$ by

$$(2y_{2P})^{-1} = 16y_P^4 I, \quad (2y_P)^{-1} = eI \ .$$

The remaining part of our quadrupling algorithm is identical to the two successive doublings. Namely, we first calculate $(2y_P)^{-1}$ and $(2y_{2P})^{-1}$, and then calculate λ_1, x_{2P}, y_{2P}, λ_2, x_{4P}, y_{4P} in this order. We show the detailed version of our quadrupling algorithm in the following.

Algorithm A: Computation of $4P$ in affine coordinates.

`INPUT:` $P = (x_P, y_P)$
`OUTPUT:` $4P = (x_{4P}, y_{4P})$
`Step 1.` Precomputation

$$\begin{aligned} m &= 3x_P^2 + a \\ s &= x_P(2y_P)^2 \\ t &= (2y_P)^4 \end{aligned}$$

`Step 2.` Computation of the inverses

$$\begin{aligned} e &= 2m(3s - m^2) - t \\ E &= (2y_P)e \\ I &= E^{-1} \\ (2y_P)^{-1} &= eI \\ (2y_{2P})^{-1} &= tI \end{aligned}$$

`Step 3.` Computation of $4P$

$$\begin{aligned} \lambda_1 &= m(2y_P)^{-1} \\ x_{2P} &= \lambda_1^2 - 2x_P \\ y_{2P} &= \lambda_1(x_P - x_{2P}) - y_P \\ \lambda_2 &= (3x_{2P}^2 + a)(2y_{2P})^{-1} \\ x_{4P} &= \lambda_2^2 - 2x_{2P} \\ y_{4P} &= \lambda_2(x_{2P} - x_{4P}) - y_{2P} \end{aligned}$$

We estimate the efficiency of our quadrupling algorithm. Step 1 requires 3 squarings and 1 multiplication. Step 2 requires 1 squaring, 4 multiplications and 1 inversion. Finally, Step 3 requires 3 squarings and 4 multiplications. Therefore, our quadrupling algorithm requires 7 squarings, 9 multiplications and 1 inversion.

We show a summary of the efficiency for quadrupling in Table 1. In comparison with the Müller's algorithm, our quadrupling algorithm saves 5 multiplications. Moreover, in comparison with the Sakai-Sakurai's algorithm of $k = 2$, our algorithm saves 2 squarings. Thus, we conclude that our quadrupling algorithm is more efficient than the existing algorithms.

3.2 Modification to the Sakai-Sakurai's Algorithm

This section handles computation of a point 2^kP from P for arbitrary natural number k. The sliding window method [4] is an extension of the 2^w-ary method, in which the window size is at most w.

Table 1: The efficiency for quadrupling

two doublings	4 squarings, 4 multiplications and 2 inversions
Müller [10]	7 squarings, 14 multiplications and 1 inversion
Sakai-Sakurai ($k = 2$) [11]	9 squarings, 9 multiplications and 1 inversion
proposed quadrupling	7 squarings, 9 multiplications and 1 inversion

This method performs $Y \leftarrow 2^i Y + vP$, where v is the value of the current window, and i is the size of the current window plus the interval between the current window and its neighbor to the left. Therefore, the sliding window method requires computation of $2^k P$ for various k.

3.2.1 The Sakai-Sakurai's algorithm.

Sakai and Sakurai [11] proposed an efficient algorithm for direct computation of $2^k P$. We show their algorithm in the following.

The Sakai-Sakurai's algorithm

`INPUT:` $P = (x_P, y_P)$
`OUTPUT:` $2^k P = (x_{2^k P}, y_{2^k P})$
`Step 1.` Computation of A_1, B_1, C_1

$$\begin{aligned} A_1 &= x_P \\ B_1 &= 3{x_P}^2 + a \\ C_1 &= -y_P \end{aligned}$$

`Step 2.` Computation of A_i, B_i, C_i $(i = 2, 3, \ldots, k)$

$$\begin{aligned} A_i &= B_{i-1}^2 - 8A_{i-1}C_{i-1}^2 \\ B_i &= 3A_i^2 + 16^{i-1} a (\prod_{j=1}^{i-1} C_j)^4 \\ C_i &= -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2) \end{aligned}$$

`Step 3.` Computation of $2^k P$

$$\begin{aligned} D_k &= 12A_k C_k^2 - B_k^2 \\ x_{2^k P} &= \frac{B_k^2 - 8A_k C_k^2}{\left(2^k \prod_{j=1}^{k} C_j\right)^2} \\ y_{2^k P} &= \frac{8C_k^4 - B_k D_k}{\left(2^k \prod_{j=1}^{k} C_j\right)^3} \end{aligned}$$

As we have noted, this algorithm does not compute intermediate points $2P, 4P, \ldots, 2^{k-1}P$ in explicit form. These intermediate points are stored as three terms, A_{i+1}, C_{i+1}, $2^i \prod_{j=1}^{i} C_j$. We can obtain $2^i P = (x_{2^i P}, y_{2^i P})$ $(1 \le i \le k-1)$ by

$$\begin{aligned} x_{2^i P} &= A_{i+1} \Big/ \left\{ 2^i \prod_{j=1}^{i} C_j \right\}^2 \\ y_{2^i P} &= -C_{i+1} \Big/ \left\{ 2^i \prod_{j=1}^{i} C_j \right\}^3 . \end{aligned}$$

This algorithm requires only 1 inversion at Step 3 since intermediate points are not computed explicitly. In addition, $4k + 1$ squarings and $4k + 1$ multiplications are required.

3.2.2 A Variant of the Sakai-Sakurai's Algorithm.

The Sakai-Sakurai's algorithm avoids the computation of $(x_{2^i P}, y_{2^i P})$ by storing intermediate points in implicit form. This algorithm repeatedly calculates three parameters A_{i+1}, B_{i+1}, C_{i+1} using A_i, B_i, C_i $(1 \le i \le k-1)$. $2^k P = (x_{2^k P}, y_{2^k P})$ is computed using the parameters which have been already calculated.

We first consider a variant of the Sakai-Sakurai's algorithm. This variant first computes $2^{k-1}P = (x_{2^{k-1}P}, y_{2^{k-1}P})$ by the Sakai-Sakurai's algorithm, and then computes $2^k P = (x_{2^k P}, y_{2^k P})$ by one doubling. Computing $(x_{2^{k-1}P}, y_{2^{k-1}P})$ requires $4k -$

3 squarings, $4k-3$ multiplications and 1 inversion. Moreover, one doubling in affine coordinates requires 2 squarings, 2 multiplications and 1 inversion. Therefore, This variant requires $4k-1$ squarings, $4k-1$ multiplications and 2 inversions. Comparing to the Sakai-Sakurai's algorithm, the variant of the Sakai-Sakurai's algorithm saves 2 squarings and 2 multiplications, and requires additional 1 inversion. Under our assumptions, this variant is inefficient if 1 inversion costs more than 3.6 multiplications. In the following part, we modify this variant using Montgomery trick.

3.2.3 A Variant with Montgomery Trick.

As we described above, the variant of the Sakai-Sakurai's algorithm requires two inverses. Computing $2^{k-1}P = (x_{2^{k-1}P}, y_{2^{k-1}P})$ requires

$$(2^{k-1}\prod_{j=1}^{k-1} C_j)^{-1} \ .$$

In addition, one doubling in affine coordinates requires $(y_{2^{k-1}P})^{-1}$.

Here we set

$$\begin{aligned} X_{2^{k-1}P} &= B_{k-1}^2 - 8A_{k-1}C_{k-1}^2 \ , \\ Y_{2^{k-1}P} &= 8C_{k-1}^4 - B_{k-1}D_{k-1} \ , \\ Z_{2^{k-1}P} &= 2^{k-1}\prod_{j=1}^{k-1} C_j \ . \end{aligned} \quad (1)$$

From (1), $x_{2^{k-1}P}$ and $y_{2^{k-1}P}$ satisfy the following equations[2];

$$x_{2^{k-1}P} = \frac{X_{2^{k-1}P}}{Z_{2^{k-1}P}^2}, \quad y_{2^{k-1}P} = \frac{Y_{2^{k-1}P}}{Z_{2^{k-1}P}^3} \ . \quad (2)$$

Therefore, $(y_{2^{k-1}P})^{-1}$ is equal to $Z_{2^{k-1}P}^3/2Y_{2^{k-1}P}$. We apply Montgomery trick to calculate $Z_{2^{k-1}P}^{-1}$ and $(2Y_{2^{k-1}P})^{-1}$ simultaneously.

We first calculate $X_{2^{k-1}P}$, $Y_{2^{k-1}P}$, $Z_{2^{k-1}P}$ using the Sakai-Sakurai's algorithm. Let d and I be $d = 2Y_{2^{k-1}P}Z_{2^{k-1}P}$ and $I = d^{-1}$. We can obtain $Z_{2^{k-1}p}^{-1} = 2Y_{2^{k-1}p}I$. Then, we calculate $x_{2^{k-1}P}$ and $y_{2^{k-1}P}$ by (2).

Next, we compute $2^kP = (x_{2^kP}, y_{2^kP})$ from $2^{k-1}P = (x_{2^{k-1}P}, y_{2^{k-1}P})$. We induce the formula for λ as follows[3]:

$$\begin{aligned} \lambda &= (3x_{2^{k-1}P}^2 + a)/2y_{2^{k-1}P} \\ &= (3x_{2^{k-1}P}^2 + a) \cdot Z_{2^{k-1}P}^3 \cdot (2Y_{2^{k-1}P})^{-1} \\ &= (3x_{2^{k-1}P}^2 + a) \cdot Z_{2^{k-1}P}^4 \cdot I \\ &= (3X_{2^{k-1}P}^2 + aZ_{2^{k-1}P}^4)I \\ &= (3X_{2^{k-1}P}^2 + 2 \cdot aZ_{2^{k-2}P}^4 \cdot 8C_{k-1}^4)I \ . \end{aligned} \quad (3)$$

The induced formula (3) contains two terms; $8C_{k-1}^4$ and $aZ_{2^{k-2}P}^4$. At a glance, these terms seem not to be calculated. However, from (1), $8C_{k-1}^4$ has been already calculated in the formula for $Y_{2^{k-1}P}$. Also, $aZ_{2^{k-2}P}^4$ has been calculated in the formula for B_{k-1}[4]. Therefore, if the values of $8C_{k-1}^4$ and $aZ_{2^{k-2}P}^4$ are stored, we can calculate λ from (3). Finally, using $x_{2^{k-1}P}$, $y_{2^{k-1}P}$ and λ, we compute (x_{2^kP}, y_{2^kP}).

We summarize the variant of the Sakai-Sakurai's algorithm with Montgomery trick in the following.

Algorithm B: A variant of the Sakai-Sakurai's algorithm

`INPUT:` $P = (x_P, y_P)$
`OUTPUT:` $2^kP = (x_{2^kP}, y_{2^kP})$
`Step 1.` Computation of A_{k-1}, B_{k-1}, C_{k-1} using the Sakai-Sakurai's algorithm
`Step 2.` Computation of $X_{2^{k-1}P}$, $Y_{2^{k-1}P}$, $Z_{2^{k-1}P}$

$$\begin{aligned} X_{2^{k-1}P} &= B_{k-1}^2 - 8A_{k-1}C_{k-1}^2 \\ Y_{2^{k-1}P} &= 8C_{k-1}^4 - B_{k-1}(4A_{k-1}C_{k-1}^2 \\ &\quad - X_{2^{k-1}P}) \\ Z_{2^{k-1}P} &= 2^{k-1}\prod_{j=1}^{k-1} C_j \end{aligned}$$

(Also storing $aZ_{2^{k-2}P}^4 = 16^{k-2}a(\prod_{j=1}^{k-2} C_j)^4$ and $8C_{k-1}^4$.)

---

[2] These equations are equivalent to the definition of Jacobian coordinates.

[3] From (1), $aZ_{2^{k-1}P}^4 = a(2^{k-1}\prod_{j=1}^{k-1} C_j)^4$ $= a(2^{k-2}\prod_{j=1}^{k-2} C_j)^4 \cdot 16C_{k-1}^4 = 2 \cdot aZ_{2^{k-1}P}^4 \cdot 8C_{k-1}^4$.

[4] From (1), $16^{k-2}a(\prod_{j=1}^{k-2} C_j)^4 = a(2^{k-2}\prod_{j=1}^{k-2} C_j)^4 = aZ_{2^{k-2}P}^4$.

Table 2: The efficiency for computing $2^k P$

Sakai-Sakurai [11]	$4k+1$ squarings, $4k+1$ multiplications and 1 inversion
proposed algorithm	$4k-1$ squarings, $4k+2$ multiplications and 1 inversion

`Step 3.` Computation of $(x_{2^{k-1}P}, y_{2^{k-1}P})$

$$\begin{aligned} d &= 2Y_{2^{k-1}P}Z_{2^{k-1}P} \\ I &= d^{-1} \\ Z^{-1}_{2^{k-1}P} &= 2Y_{2^{k-1}P}I \\ x_{2^{k-1}P} &= X_{2^{k-1}P}Z^{-2}_{2^{k-1}P} \\ y_{2^{k-1}P} &= Y_{2^{k-1}P}Z^{-3}_{2^{k-1}P} \end{aligned}$$

`Step 4.` Computation of (x_{2^kP}, y_{2^kP})

$$\begin{aligned} \lambda &= (3X^2_{2^{k-1}P} + 2 \cdot aZ^4_{2^{k-2}P} \\ &\quad \cdot 8C^4_{k-1})I \\ x_{2^kP} &= \lambda^2 - 2x_{2^{k-1}P} \\ y_{2^kP} &= \lambda(x_{2^{k-1}P} - x_{2^kP}) - y_{2^{k-1}P} \end{aligned}$$

Here we estimate the efficiency of the variant of the Sakai-Sakurai's algorithm. Step 1 requires $4k-7$ squarings and $3k-6$ multiplications. Step 2 requires 3 squarings and k multiplications. Step 3 requires 1 squaring, 5 multiplications and 1 inversion. Finally, step 4 requires 2 squarings and 3 multiplications. Therefore, our algorithm for 2^kP requires $4k-1$ squarings, $4k+2$ multiplications and 1 inversion.

We summarize the efficiency for computing 2^kP in Table 2. In comparison with the Sakai-Sakurai's algorithm, our variant algorithm saves 2 squarings and requires 1 additional multiplication. Namely, our variant algorithm saves 0.6 multiplications comparing to the Sakai-Sakurai's algorithm under our assumptions.

4 Application to Scalar Multiplication

As an instance, we compute dP using the 2^w-ary method with $w = 3$, where

$$\begin{aligned} d &= 49719768 \\ &= (10\bar{1}0\bar{1}00100100\bar{1}00\bar{1}0\bar{1}00\bar{1}0\bar{1}000)_2 \ . \end{aligned}$$

We use a symbol '$\bar{1}$' to represent -1. We compute the point of the form $2^3P + Q$ as follows; compute 2^2P first, and then compute $2(2^2P) + Q$ using the Ciet-Joye-Lauter-Montgomery algorithm [1]. We assume that $2P$, $3P$, $4P$ have been already precomputed.

We consider two cases for computing $49719768P$ between two cases(see Table 3). Case I computes a point 2^iP by the Sakai-Sakurai's algorithm, and Case II computes 2^iP by Algorithm A(for $k = 2$) or Algorithm B(for $k \geq 3$). We estimate the number of field arithmetics for each such case. Case I requires 90 squarings, 139 multiplications and 15 inversions, and Case II requires 74 squarings, 140 multiplications and 15 inversions. Namely, our algorithms save 16 squarings and require 1 additional multiplication. In case that the cost of a inversion is 10 times as expensive as that of a multiplication, the savings of the proposed algorithms translates to 3.3%.

5 Conclusion

We have presented two algorithms for computing points of the form 2^kP which use Montgomery trick. The one only works for $k = 2$, and the other works for arbitrary natural number k. We have shown that our proposed algorithms are more efficient in comparison with existing algorithms.

References

[1] M. Ciet, M. Joye, K. Lauter and P.L. Montgomery, "Trading inversions for multiplications in elliptic curve cryptography", IACR Cryptology ePrint Archive, 2003. available at http://eprint.iacr.org/2003/257.ps.gz

Table 3: An instance of computing $46719768P$

			(Case I)	(Case II)
$22P$	=	$2^3 \cdot 3P - 2P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$178P$	=	$2^3 \cdot 22P + 2P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$1426P$	=	$2^3 \cdot 178P + 2P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$11406P$	=	$2^3 \cdot 1426P - 2P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$91250P$	=	$2^3 \cdot 11406P + 2P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$729996P$	=	$2^3 \cdot 91250P - 4P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$5839971P$	=	$2^3 \cdot 729996P + 3P$	$[2^2P], [2P+Q]$	[Alg-A], $[2P+Q]$
$46719768P$	=	$2^3 \cdot 5839971P$	$[2^3P]$	[Alg-B($k=3$)]

[2] H. Cohen, A course in Computational Algebraic Number Theory, Graduate Texts in Math. **138**, Springer-Verlag, Berlin, 1993.

[3] K. Eisenträger, K. Lauter, P.L. Montgomery, "An efficient procedure to double and add points on an elliptic curve", IACR Cryptology ePrint Archive, 2002. available at http://eprint.iacr.org/2002/112.ps.gz

[4] D.M. Gordon, "A survey of fast exponentiation methods", *J. Algorithms*, **27** (1998), 129–146.

[5] J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems", *Advances in Cryptology—Crypto'97*, LNCS **1294** (1997), Springer-Verlag, 342–356.

[6] D.E. Knuth, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, 3rd ed., Addison-Wesley, Reading, MA, 1997.

[7] K. Koyama, and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method," *Advances in Cryptology—Crypto'92*, LNCS **740**, Springer-Verlag, 345–357.

[8] F. Morain, and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," Theoretical Informatics and Applications, **24** (1990), No.6, 531–544.

[9] P.L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization", Math. Comp., v. **48** (1987), 243–264.

[10] V. Müller, "Efficient algorithms for multiplication on elliptic curves", Proc. GI-Arbeitskonferenz Chipkarten 1998, TU München, 1998.

[11] Y. Sakai, K. Sakurai, "Efficient scalar multiplications on elliptic curves with direct computations of several doublings", IEICE Trans. Fundamentals, Vol.E84-A, No.1, 120–129, Jan, 2001.

[12] J.A. Solinas, "Efficient arithmetic on Koblitz curves", *Designs, Codes and Cryptography*, **19** (2000), 195–249.