International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 8, Number 2, pages 166–185, July 2018

Optimal Representation for Right-to-Left Parallel Scalar and Multi-Scalar Point Multiplication

Kittiphon Phalakarn
Department of Computer Engineering
Chulalongkorn University, Bangkok, Thailand

Kittiphop Phalakarn
Department of Computer Engineering
Chulalongkorn University, Bangkok, Thailand

Vorapong Suppakitpaisarn
Department of Computer Science
The University of Tokyo, Tokyo, Japan

Received: January 26, 2018
Revised: April 19, 2018
Accepted: May 31, 2018
Communicated by Yasuyuki Nogami

Abstract

This paper introduces an optimal representation for a right-to-left parallel elliptic curve scalar point multiplication. The right-to-left approach is easier to parallelize than the conventional left-to-right approach. However, unlike the left-to-right approach, there is still no work considering number representations for the right-to-left parallel calculation. By simplifying the implementation by Robert, we devise a mathematical model to capture the computation time of the calculation. Then, for any arbitrary amount of doubling time and addition time, we propose algorithms to generate representations which minimize the time in that model. As a result, we can show a negative result that a conventional representation like NAF is almost optimal. The parallel computation time obtained from any representation cannot be better than NAF by more than 1%. In addition to that, we devise a time model and propose an algorithm to generate optimal representations for multi-scalar point multiplication (under a condition). Similar to the result of scalar point multiplication, NAF is almost optimal also for multi-scalar point multiplication as the difference of parallel computation time obtained from optimal representation and NAF is less than 1% in all experimental settings.

Keywords: information and communication security, efficient implementation, elliptic curve cryptography, scalar point multiplication, binary representation, parallel algorithms

[0]This paper is an extended version of [18] presented at CANDAR 2017.

1 Introduction

Scalar point multiplication, where we want to compute nP for some integer n and elliptic curve point P, is one of the most important operations in elliptic curve cryptography. Many techniques were proposed to improve this operation. Since binary representation of n affects the speed of "double-and-add" scalar point multiplication, there are techniques, including sliding window [9, 25], non-adjacent form (NAF) [19], window NAF (wNAF) [13], and fractional wNAF (f-wNAF) [15], trying to improve the binary representations. Among them, some of the algorithms are vulnerable to side-channel attacks [10], which are attacks that allow attackers to use computational information such as time or power to obtain secrets of cryptosystems. However, those schemes can still be used as tools for cryptanalysis, which is an analysis to find weakness of a proposed cryptographic protocol [2]. Because of that, it is still useful to speed up scalar point multiplication without considering the side-channel attacks.

It is known that the elliptic curve scalar point multiplication is a special case of exponentiation, where we want to compute g^n for some integer n and a group member g. When the inverse of g can be calculated easily, many algorithms for the double-and-add method, including those proposed in this paper, can be also applied for reducing the computation of the square-and-multiply method for the exponentiation. Although all notations in this paper are for scalar point multiplication, our contribution also includes those general arithmetic operations.

To improve scalar point multiplication further, parallelism is used instead of sequentiality. Garcia and Garcia [8] proposed a scheme that parallelizes a part of computations for scalar point multiplication. However, the part that is not parallelized becomes a bottleneck of their algorithm. The algorithm can be significantly improved if that part is parallelized. Nöcker [17] proposed an algorithm to distribute the workload among all processors optimally. Nonetheless, the algorithm can be significantly improved when the number of processors we have is as small as two (See our proof in Section 4.1). The similar idea is proposed for multi-scalar point multiplication by Borges, Lara, and Portugal in [5], but this technique interests in optimizing computation time with a lower bound in the number of processors used. When using the technique, we cannot specify the maximum number of processors, and, in many cases, the number of processors required by the algorithm can be up to 200. A parallel algorithm for scalar point multiplication is also proposed by Izu and Takagi [12]. However, the authors aim to find an algorithm that can resist the side channel attacks in the paper, and, to resist the attacks, more steps have to be done. By that, the algorithm is slightly slower than those that do not resist the attacks.

Most of the work for scalar point multiplication, including all of those mentioned in the previous paragraph, are based on the left-to-right approach, because, when the operation is calculated sequentially, the left-to-right technique can be sped up using precomputation points [24]. However, when scalar point multiplication is calculated in parallel, the right-to-left technique is known to be an easier way [20]. The right-to-left technique is also useful when the integer n is produced in a serial fashion from right to left, e.g. as a result of a calculation. From our best knowledge, no work in literature can speed up the right-to-left technique using precomputation.

There is not many works considering number representations for the right-to-left technique. The only technique we know can be found in [6], which we will call as "parallel double-and-add" algorithm in this paper. Although the technique is optimal, the model assumes that point doubling and addition use the same amount of time. However, this assumption is not true for scalar point multiplication. For example, in the fastest recorded elliptic curve scalar point multiplication, twisted Edwards curve, addition takes 50% more time than doubling [3]. In other curves, it is also found that addition takes significantly more time than doubling [1]. Thus, this algorithm is not optimal for scalar point multiplication.

That motivates us to consider an optimal representation for the parallel right-to-left scalar point multiplication in this paper. Based on the right-to-left technique by Moreno and Hasan in [16] and its implementation by Robert in [20], in this paper, we propose a new time model and problem for parallel scalar point multiplication with arbitrary amount of doubling time D and addition time A. Then, we show a negative result that a conventional representation like NAF is almost optimal for any arbitrary amount of doubling time and addition time.

To show that NAF is almost optimal, we propose algorithms to generate optimal representations for all cases. Our algorithms have $O(\log_2 n)$ complexity and use $O(1)$ additional space. Then, we prove and perform a numerical experiment to show that the average computation time obtained from NAF is very close to the optimal computation time obtained from our algorithms. The difference between the average time from the algorithms is not more than 1% in all experimental settings.

Although techniques such as wNAF or f-wNAF provide much faster scalar point multiplications than NAF in the left-to-right setting, they do not perform better in our parallel setting. Our results indicate that there is no representation can improve the average computation time of NAF by more than 1%, while, in those schemes, we have to perform some precomputation tasks before the calculation for scalar point multiplication.

Another important operation in elliptic curve cryptography is multi-scalar point multiplication where we want to compute $nP+mQ$ for some integers n, m and elliptic curve points P, Q. Improving multi-scalar point multiplication can be done by many techniques, e.g. using joint sparse form (JSF) [21], and minimal joint Hamming weight representations [23]. In this paper, we also propose a time model for multi-scalar point multiplication and an algorithm to find optimal representations for the model, under a condition that three processors are used and only canonical binary representations are considered. Again, the difference between the average computation time obtained from the algorithms and NAF is not more than 1% in all experimental settings, which means that NAF is nearly optimal in the case of multi-scalar point multiplication.

This paper is organized as follows. In Section 2, we formally define binary representations, NAF technique, and parallel double-and-add scalar point multiplication. In Section 3, we define our calculation model, and in Section 4, we give properties of the model. Then, in Section 5 and 6, we use the properties to devise algorithms for the case when $0 < 2D \leq A$ and $0 < D \leq A < 2D$, respectively. We discuss in Section 5.2 and 6.2 why NAF is almost optimal for the case. We give our experimental results in Section 7. In Section 8, we define the time model for multi-scalar point multiplication, propose the algorithm to find the optimal representations (under the condition), and show experimental results in this case. Finally, we conclude the paper in Section 9.

2 Preliminaries

2.1 Binary Representation

We first introduce the notation of binary representation used in this paper. We assume that n is an input and all representations are of n if not state otherwise.

Definition 1 (Binary representation of n using digit set $\mathcal{S}$). *Let $n, \lambda \in \mathbb{Z}_{\geq 0}$ and $\{0,1\} \subseteq \mathcal{S} \subseteq \mathbb{Z}$. The set of all possible binary representations of n using digit set $\mathcal{S}$ is $\mathscr{N}_{\mathcal{S}} = \{N_{\mathcal{S}} = n_{\lambda}...n_0\}$ such that $\sum_{i=0}^{\lambda} n_i 2^i = n$, $n_i \in \mathcal{S}$ for all $0 \leq i \leq \lambda$. We use $\bar{s}$ instead of $-s$ to simplify the notation. If $\mathcal{S} = \mathcal{B} = \{0,1\}$, we call $\mathscr{N}_{\mathcal{B}} = \{N_{\mathcal{B}}\}$ set of all binary representations of n. If $\mathcal{S} = \mathcal{C} = \{\bar{1}, 0, 1\}$, we call $\mathscr{N}_{\mathcal{C}} = \{N_{\mathcal{C}}\}$ set of all canonical binary representations of n.*

Note that $N_{\mathcal{B}}$ is unique while $N_{\mathcal{C}}$ is not. In this paper, we use regular expression to represent sequences or patterns of digits, e.g. $10^3\bar{1}$ means $1000\bar{1}$, and 1^* means zero or more '1's.

2.2 Non-Adjacent Form (NAF) Technique

NAF technique [19] is one of the techniques used to improve scalar point multiplication. It changes binary representation to canonical binary representation with no consecutive non-zero digits which is proved to have minimal Hamming weight [19]. The algorithm can be described as in Algorithm 1. Note that NAF representation is unique, and we have a starting index i as input for further use in our proposed algorithms.

In short, Algorithm 1 changes 01^p to $10^{p-1}\bar{1}$ for $p \geq 2$ from n'_i to $n'_{\lambda+1}$. To get NAF representation, we set $i = 0$.

Algorithm 1: Transform binary representation to non-adjacent form (toNAF)

```
input  : $N_{\mathcal{B}} = n_\lambda ... n_0$, starting index $i$
output: $N'_{\mathcal{C}} = n'_{\lambda+1} ... n'_0$ with no consecutive non-zero digits in $n'_{\lambda+1} ... n'_i$
begin
    $N'_{\mathcal{C}} \leftarrow N_{\mathcal{B}}$
    $n'_{\lambda+1} \leftarrow 0$
    while $i < \lambda$ do
        if $n'_i = 1$ and $n'_{i+1} = 1$ then
            $n'_i \leftarrow \bar{1}$
            $i \leftarrow i + 1$
            while $n'_i = 1$ do
                $n'_i \leftarrow 0$
                $i \leftarrow i + 1$
            $n'_i \leftarrow 1$
        else $i \leftarrow i + 1$
    return $N'_{\mathcal{C}}$
```

Example 1. *Consider $n = 371$ with $N_{\mathcal{B}} = 101110011$. If $i = 0$, we have $N'_{\mathcal{C}} = 10\bar{1}00\bar{1}010\bar{1}$, and if $i = 5$, we have $N'_{\mathcal{C}} = 10\bar{1}0\bar{1}10011$.* □

2.3 "Parallel Double-and-Add" Scalar Point Multiplication

Borodin and Munro (cf. [6]) presented Theorem 1 with a technique we can apply to "parallel double-and-add" technique. This technique does scalar point multiplication in $\lceil \log_2 n \rceil$ steps. Notice that the time is measured in "steps" which means that point doubling and addition use same amount of time.

Theorem 1 (adapt from Lemma 6.1.1 in [6])**.** *Let $n \in \mathbb{Z}_+$ and P be elliptic curve point. nP can be computed from P using two processors in $\lceil \log_2 n \rceil$ steps.*

"Parallel double-and-add" technique uses two processors: doubling processor and addition processor. Doubling processor calculates $2P, 4P, ..., 2^iP$, and addition processor adds the result from doubling processor cumulatively according to $N_{\mathcal{B}} = n_\lambda ... n_0$. See Example 2 for more understanding. Note that for the least significant '1', e.g. n_0 in Example 2, addition processor can "copy" the result from doubling processor with no addition.

Example 2. *To calculate $87P$ using $N_{\mathcal{B}} = 1010111$, we need 7 steps as depicts in Figure 1.* □

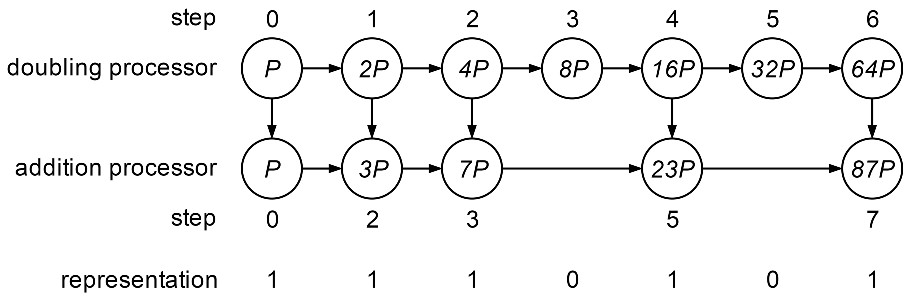

Figure 1: "Parallel double-and-add" scalar point multiplication for $N_{\mathcal{B}} = 1010111$

It is known that, in every elliptic curves, point addition takes more computation time than doubling. Hence, this applied model may not be practical for scalar point multiplication.

3 Our Calculation Model

We improve the model in Section 2.3 by defining the time used in "parallel double-and-add" scalar point multiplication as follows.

Let $D \in \mathbb{R}_{\geq 0}$ be the amount of time doubling processor uses for one doubling, and $A \in \mathbb{R}_{\geq 0}$ be the amount of time addition processor uses for one addition. To calculate nP using $N_{\mathcal{S}}$, addition processor considers each digit from n_0 to n_λ. Before we encounter the least significant non-zero digit, the time used is 0. At the least significant non-zero digit n_i, we wait for $2^i P$ to be finished at time iD, copy $2^i P$ or $-2^i P$ to addition processor using negligible additional time, and add/subtract $2^i P$ to/from addition processor $|n_i| - 1$ times, so the computation time is now $iD + (|n_i| - 1)A$. For later zero digits, the computation time is unchanged. And, for other non-zero digits n_j, we need to wait until doubling processor finishes $2^j P$ at time jD and until addition processor finishes its previous work, then we add/subtract $2^j P$ to/from the current result $|n_j|$ times which uses $|n_j|A$ time units. Consider the following example for more understanding.

Example 3. *The calculation of $87P$ using $\mathcal{S} = \{\bar{3}, \bar{2}, \bar{1}, 0, 1, 2, 3\}$ and $N_{\mathcal{S}} = 220\bar{1}\bar{3}1$ with $D = 2$ and $A = 3$ is shown in Figure 2. Doubling processor have the point P at time 0. Then, the processor keeps doubling the point to $2P, 4P, \ldots, 32P$, and send those points to addition processor. Since the point $2P$ is available after we perform one point doubling on P and one doubling takes two time units, we have $2P$ at time 2. By the same argument, we have $4P, 8P, 16P, 32P$ at time 4, 6, 8, 10. Consider addition processor, after the processor receives $2P$ from doubling processor at time 2, it adds $-2P$ to P. Since the addition takes 3 time units, we have $-P = P + (-2P)$ at time 5. Then, the processor adds $-2P$ to $-P$, and have $-3P$ at time 8. After that, the processor adds $-2P$ to $-3P$, and have $-5P$ at time 11. Next, addition processor calculates $-9P$ from $-4P$ and $-5P$. Although, the processor has $4P$ from doubling processor since time 4, it has to wait until $-5P$ is available at time 11. Because of that, we have $-9P$ at time 14. After we continue the process in the way shown in Figure 2, we have $87P$ at time 26.* □

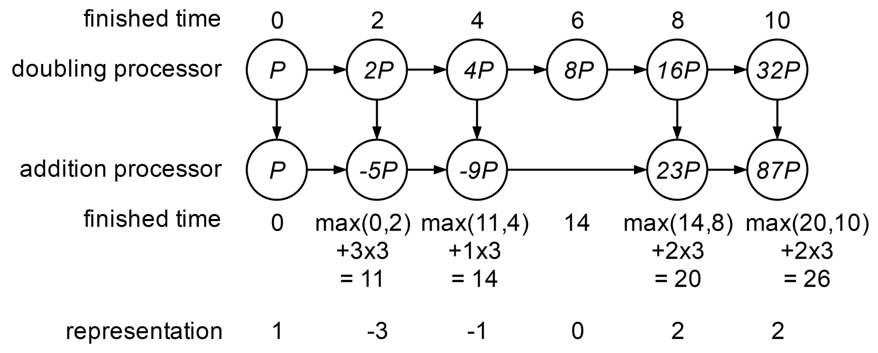

Figure 2: "Parallel double-and-add" scalar point multiplication for $D = 2$, $A = 3$, and $N_{\mathcal{S}} = 220\bar{1}\bar{3}1$

Hence, we define the time model as follows:

Definition 2 (Computation time of parallel scalar point multiplication)**.** *Let $D \in \mathbb{R}_{\geq 0}$ be the time processor used for one doubling, $A \in \mathbb{R}_{\geq 0}$ be the time used for one addition, and $n \in \mathbb{Z}_+$ with binary representation $N_{\mathcal{S}} = n_\lambda \ldots n_0$ using digit set $\mathcal{S}$. The computation time of parallel scalar point multiplication nP using $N_{\mathcal{S}}$ after calculating from n_0 up to n_i is $T(N_{\mathcal{S}}, i)$ which can be calculated by:*

$$T(N_{\mathcal{S}}, i) := \begin{cases} 0 & \text{if } i = 0 \text{ and } n_i = 0; \\ T(N_{\mathcal{S}}, i-1) & \text{if } i > 0 \text{ and } n_i = 0; \\ iD + (|n_i| - 1)A & \text{if } i \geq 0,\ n_i \neq 0, \text{ and } n_j = 0; \forall j, 0 \leq j < i; \\ \max(T(N_{\mathcal{S}}, i-1), iD) + |n_i|A & \text{otherwise.} \end{cases}$$

From Definition 2, the time used to calculate nP using $N_{\mathcal{S}}$ is $T(N_{\mathcal{S}}, \lambda)$. Our problem is to find a representation $N_{\mathcal{S}}$ which uses minimum time $T(N_{\mathcal{S}}, \lambda)$ for given inputs n, D and A. In this paper, we will consider only the case where $0 < D \leq A$ since when $D = 0$, minimal total Hamming weight representation, which is discussed in [11], is optimal; and $D \leq A$ for most elliptic curve implementations that have been proposed up to this state [1].

We do not consider the communication time between two processors in our model, and will consider them as future work. Because the communication time, denoted as S, is usually larger than $10D$, one might think that we should add that communication time into the communication time of double, i.e. we should use $D' = D + S$ to denote the calculation time for one double. However, doing that is clearly too pessimistic. By pipelining, we need only $D + D + S = 2D + S$ to obtain two double results, not $D' + D' = 2D + 2S$. It is straightforward to show that the time until the point $2^i P$ is no more than $i(D + S)$. Hence, by assigning D to $D + S$, one can calculate an upper bound of the computation time in our model. We strongly believe that the optimal representation will not significantly change by the additional communication time.

Unlike the implementation in [20], we assume that doubling processor does not wait for addition processor in any cases. It is proved in [16] that, even without waiting, the buffer size required is at most $O(\sqrt{\lambda})$.

Moreover, we can see that $T(N_{\mathcal{S}}, i)$ is always in the form $pA + qD$ for some integers p and q, so from now on we will normalize A and D by assuming that $D = 1$.

Example 4. *Consider the case where $D = 1$, $A = 3$, and $n = 29$. If we calculate $29P$ using $N_{\mathcal{B}} = 11101$, we have $T(N_{\mathcal{B}}, \lambda) = 11$. If we use $N_{\mathcal{C}} = 1000\bar{1}\bar{1}$, we have $T(N_{\mathcal{C}}, \lambda) = 8$. In Section 5.2, we will show that this is optimal among all representations. We note that the computation time for the sequential double-and-add method is 13 when we use $N_{\mathcal{B}}$, and 11 when we use $N_{\mathcal{C}}$.* □

4 Properties of Our Model

In this section, we will show properties of the model proposed in the previous section. In Section 4.1, we show that this computation model is faster than the model considered in previous works, when the number of processors is two. Then, we show in Section 4.2 that the digit set $\{\bar{1}, 0, 1\}$ is sufficient for having the optimal representation. By the property, when we devise an algorithm for finding the optimal representation in Section 5 and 6, we do not need to consider numbers which are not in the set. And, in Section 4.3, we introduce a concept about delay which is used to prove the optimality of our algorithms.

4.1 Comparing Computation Time with Nöcker's Algorithm

Nöcker [17] proposed an algorithm to distribute the workload to compute nP among all processors. The computation time used by his algorithm is as follows.

Theorem 2 (adapt from Theorem 1 in [17]). *Let $p \geq 2$ be the number of processors used to compute nP, $N_{\mathcal{B}} = n_\lambda ... n_0$, and $c = D/A$. For $0 < D \leq A$, the computation time used by Nöcker's algorithm is*

$$T \leq \lambda D + \left(\frac{c}{(1+c)^p - 1}(\lambda + 1) - 1 + \lceil \log_2 p \rceil \right) A.$$

Furthermore, this bound is tight since there is an integer n which achieves this computation time.

When we use $p = 2$ processors with normalized $D = 1$, we have

$$T \leq \lambda + \left(\frac{A}{2A+1}(\lambda + 1) \right) A.$$

We will show in Corollary 1 that, when $0 < 2D \leq A$, the computation time from our model is no more than $\frac{1}{2}\lambda A + A + 1$, and, in Corollary 2, we show that, when $0 < D \leq A < 2D$, the computation time from our model is no more than $A + \lambda + 1$. Using these facts, we will show in the next theorem that our representations give better computation time.

Theorem 3. *Algorithm 2 and 3 generate representations which have better worst case computation time than Nöcker's algorithm when $A \geq 1$ and $\lambda \geq \max\left(\frac{4}{3}A + \frac{4}{3}, 3 + \frac{3}{A}\right)$.*

Proof. Since $A \geq 1$ and $\lambda \geq \dfrac{4}{3}A + \dfrac{4}{3}$, we have

$$\begin{aligned}\lambda + \left(\frac{A}{2A+1}(\lambda+1)\right)A &\geq \lambda + \left(\frac{1}{2} - \frac{1}{4A+2}\right)\lambda A \\ &\geq \frac{1}{2}\lambda A + \lambda - \left(\frac{A}{4A+2}\right)\lambda \\ &\geq \frac{1}{2}\lambda A + \frac{3}{4}\lambda \\ &\geq \frac{1}{2}\lambda A + A + 1,\end{aligned}$$

and since $A \geq 1$ and $\lambda \geq 3 + \dfrac{3}{A}$, we have

$$\lambda + \left(\frac{A}{2A+1}(\lambda+1)\right)A \geq \lambda + \frac{1}{3}\lambda A \geq A + \lambda + 1.$$

Hence, it is proved that the upper bound of Nöcker's algorithm is larger than our algorithms. ■

We note that λ is usually larger than $\max\left(\dfrac{4}{3}A + \dfrac{4}{3}, 3 + \dfrac{3}{A}\right)$, since A is usually less than 10 and λ is usually more than 100.

4.2 Optimality Proof for Digit Set $\{\bar{1}, 0, 1\}$

Before analyzing the time model further, we have a proposition that using digit set $\mathcal{S} = \mathcal{C} = \{\bar{1}, 0, 1\}$ is sufficient.

Proposition 1. *If $0 < D \leq A$, binary representation of n using digit set $\mathcal{S} = \mathcal{C} = \{\bar{1}, 0, 1\}$ has no larger $T(N_{\mathcal{S}}, \lambda)$ than all other $\mathcal{S}' \supseteq \{\bar{1}, 0, 1\}$.*

Proof. We prove this proposition by contradiction. Suppose there is a binary representation $N_{\mathcal{S}} = n_\lambda \ldots n_0$ with some $|n_i| > 1$ that has smallest $T(N_{\mathcal{S}}, \lambda)$. Define $t := T(N_{\mathcal{S}}, i-1)$ (we define $t := 0$ if $i = 0$) and consider $n_{i+1}n_i$.

Case $n_i \equiv 1 \pmod 2$: We can construct new binary representation $N'_{\mathcal{S}} = n'_\lambda \ldots n'_0$ with $n'_k = n_k$ for all $0 \leq k \leq \lambda$ except $n'_i = \operatorname{sgn}(n_i)$ and

$$n'_{i+1} = n_{i+1} + \frac{n_i - \operatorname{sgn}(n_i)}{2} \leq |n_{i+1}| + \frac{|n_i| - 1}{2}.$$

In the case where n_i is not the least significant non-zero digit of $N_{\mathcal{S}}$, we have $T(N_{\mathcal{S}}, i) = \max(t, iD) + |n_i|A \geq (i+1)D$. By that,

$$\begin{aligned}T(N_{\mathcal{S}}, i+1) &= \max(T(N_{\mathcal{S}}, i), (i+1)D) + |n_{i+1}|A \\ &= \max(t, iD) + |n_i|A + |n_{i+1}|A.\end{aligned}$$

By the above equation and the fact that $T(N'_{\mathcal{S}}, i) = \max(t, iD) + A \geq (i+1)D$, the computation time of parallel scalar point multiplication of $N'_{\mathcal{S}}$ at n_{i+1} is

$$\begin{aligned}T(N'_{\mathcal{S}}, i+1) &\leq \max(T(N'_{\mathcal{S}}, i), (i+1)D) + \left(|n_{i+1}| + \frac{|n_i| - 1}{2}\right)A \\ &\leq \max(t, iD) + A + \left(|n_{i+1}| + \frac{|n_i| - 1}{2}\right)A \\ &\leq T(N_{\mathcal{S}}, i+1).\end{aligned}$$

We can use the similar argument to prove that $T(N'_{\mathcal{S}}, i+1) \leq T(N_{\mathcal{S}}, i+1)$ for the case where n_i is the least significant non-zero digit of $N_{\mathcal{S}}$. Because $n'_k = n_k$ for all $k > i+1$, we have $T(N'_{\mathcal{S}}, \lambda) \leq T(N_{\mathcal{S}}, \lambda)$.

Case $n_i \equiv 0$ (mod 2): We can construct new binary representation $N'_{\mathcal{S}} = n'_{\lambda}...n'_0$ with $n'_k = n_k$ for all $0 \leq k \leq \lambda$ except $n'_i = 0$ and

$$n'_{i+1} = n_{i+1} + \frac{n_i}{2} \leq |n_{i+1}| + \frac{|n_i|}{2}.$$

Then, we can use the similar argument as in the case when $n_i \equiv 1 \pmod 2$ to show that $T(N'_{\mathcal{S}}, \lambda) \leq T(N_{\mathcal{S}}, \lambda)$.

We can repeat changing n_i where $|n_i| > 1$ using the above method to get representation using only $\{\bar{1}, 0, 1\}$ with no more $T(N_{\mathcal{S}}, \lambda)$ than the starting representation. This means that using $\{\bar{1}, 0, 1\}$ is sufficient. ■

From Proposition 1, we can assume without loss of generality that the digit set used is $\mathcal{C} = \{\bar{1}, 0, 1\}$ and the representation we will consider for optimal representation is canonical binary representation.

4.3 Delay and Optimal Representation

Although Definition 2 is not difficult to understand, it is difficult to analyze, so we introduce a concept of delay when comparing time of doubling processor and addition processor (finished time of addition processor minus finished time of doubling processor at the same step) as follows:

Definition 3 (Delay of addition processor in parallel scalar point multiplication). *Let $D = 1$ be the time processor used for one doubling, $A \geq D$ be the time used for one addition, and $n \in \mathbb{Z}_+$ with canonical binary representation $N_{\mathcal{C}} = n_{\lambda}...n_0$. The delay of addition processor after calculating nP using $N_{\mathcal{C}}$ from n_0 up to n_i is $\delta(N_{\mathcal{C}}, i)$ which can be calculated by:*

$\delta(N_{\mathcal{C}}, i) := T(N_{\mathcal{C}}, i) - iD = T(N_{\mathcal{C}}, i) - i =$

$$\begin{cases} 0 & \text{if } i = 0; \\ \delta(N_{\mathcal{C}}, i-1) - 1 & \text{if } i > 0 \text{ and } n_i = 0; \\ 0 & \text{if } i > 0,\ n_i \neq 0, \text{ and } n_j = 0; \forall j, 0 \leq j < i; \\ \max(\delta(N_{\mathcal{C}}, i-1) + (A-1), A) & \text{otherwise.} \end{cases}$$

The delay of calculating nP using $N_{\mathcal{C}}$ is $\delta(N_{\mathcal{C}}, \lambda)$. To calculate the delay, we consider each digit from n_0 to n_{λ}. The delay at n_0 is 0. When $n_i = 0$, only doubling processor does its work, so the delay decreases by $D = 1$. When we consider the least significant non-zero digit, the delay is 0 as we wait until doubling processor finishes and copy the result. And, when we consider other non-zero digits n_i, both processors do their works, so the delay increases by $A - D = A - 1$. But, in the case where $\delta(N_{\mathcal{C}}, i-1) < D$, doubling processor will finish calculating $2^i P$ after addition processor finishes calculating up to n_{i-1}. This means addition processor needs to wait for $2^i P$, and after the addition, the delay is equal to A.

Example 5. *Consider the case where $D = 1$, $A = 3$, and $n = 29$. If we calculate $29P$ using $N_{\mathcal{B}} = 011101$ (we add leading '0' for easy comparison with $N_{\mathcal{C}}$), we have $\delta(N_{\mathcal{B}}, \lambda) = 6$. But, if we use $N_{\mathcal{C}} = 1000\bar{1}\bar{1}$, we have $\delta(N_{\mathcal{C}}, \lambda) = 3$. In Section 5.2, we will show that this is optimal among all representations.* □

From Definition 3, we have $\delta(N_{\mathcal{C}}, \lambda) + \lambda = T(N_{\mathcal{C}}, \lambda)$. Hence, a canonical binary representation $N^*_{\mathcal{C}}$ has smallest $T(N_{\mathcal{C}}, \lambda)$ among all $N_{\mathcal{C}} \in \mathscr{N}_{\mathcal{C}}$ if and only if it has smallest $\delta(N_{\mathcal{C}}, \lambda)$ among all $N_{\mathcal{C}} \in \mathscr{N}_{\mathcal{C}}$ (when compare using the same λ).

Definition 4 (Optimal canonical binary representation of n for parallel scalar point multiplication). *$N^*_{\mathcal{C}}$ is an optimal canonical binary representation of n for parallel scalar point multiplication with addition time A if for all $N_{\mathcal{C}} \in \mathscr{N}_{\mathcal{C}}$, $\delta(N^*_{\mathcal{C}}, \lambda) \leq \delta(N_{\mathcal{C}}, \lambda)$. We use $\mathscr{N}^*_{\mathcal{C}}$ to represent the set of all $N^*_{\mathcal{C}}$.*

5 Optimal Representation when $0 < 2D \leq A$

5.1 Algorithm

In the case where $A \geq 2$ (normalize $D = 1$), we can construct $N_{\mathcal{C}}^* \in \mathscr{N}_{\mathcal{C}}^*$ from $N_{\mathcal{B}}$ using Algorithm 2.

Algorithm 2: Changing binary representation to optimal representation when $0 < 2D \leq A$

```
input  : $N_{\mathcal{B}} = n_\lambda ... n_0$
output: $N_{\mathcal{C}}^* = n'_{\lambda+1} ... n'_0 \in \mathscr{N}_{\mathcal{C}}^*$
begin
    $\ell \leftarrow$ index of the least significant '1' of $N_{\mathcal{B}}$
    if $N_{\mathcal{B}}$ ends with $11(01)^*010^*$ then
        $n_\ell \leftarrow \bar{1}$
        $n_{\ell+1} \leftarrow 1$
        return toNAF($N_{\mathcal{B}}, \ell + 1$)
    else if $N_{\mathcal{B}}$ ends with $0(01)^*0110^*$ then
        return toNAF($N_{\mathcal{B}}, \ell + 1$)
    else return toNAF($N_{\mathcal{B}}, \ell$)
```

To increase understanding in Algorithm 2, consider an example below:

Example 6. *Consider $n = 29$ with $N_{\mathcal{B}} = 11101$. From Algorithm 2, we have $\ell = 0$ and $N_{\mathcal{B}}$ ends with $11(01)^*010^*$. We change the representation to $1111\bar{1}$ and then transform to NAF consider only $n_{\lambda+1} ... n_1$ using Algorithm 1. The new representation $N_{\mathcal{C}}^*$ is $1000\bar{1}\bar{1}$ which has smallest scalar point multiplication time and delay for any $A \geq 2D$.* □

We can see that Algorithm 2 has $O(\log_2 n)$ complexity and uses $O(1)$ additional space. Note that Algorithm 2 generates optimal representations not depends on the value A.

5.2 Optimality Proof for Algorithm 2

We begin our proof with three lemmas.

Lemma 1. *Let $D = 1$, $A \geq 1$, $N_{\mathcal{C}} = n_\lambda ... n_0$, and $N'_{\mathcal{C}} = n'_\lambda ... n'_0$. If $n_k = n'_k$ for all $0 < i \leq k \leq j$ with some $n_p \neq 0$, $n'_q \neq 0$ for some $0 \leq p, q < i$, and $\delta(N_{\mathcal{C}}, i-1) \geq \delta(N'_{\mathcal{C}}, i-1)$, then $\delta(N_{\mathcal{C}}, j) \geq \delta(N'_{\mathcal{C}}, j)$.*

Proof. We prove this lemma by induction on index k from i to j. Assume that $n_k = n'_k$ and $\delta(N_{\mathcal{C}}, k-1) \geq \delta(N'_{\mathcal{C}}, k-1)$. If $n_k = n'_k = 0$, then

$$\begin{aligned}\delta(N'_{\mathcal{C}}, k) &= \delta(N'_{\mathcal{C}}, k-1) - 1 \\ &\leq \delta(N_{\mathcal{C}}, k-1) - 1 = \delta(N_{\mathcal{C}}, k).\end{aligned}$$

If $n_k = n'_k = 1$ or $\bar{1}$, because both are not the least significant non-zero digits, then

$$\begin{aligned}\delta(N'_{\mathcal{C}}, k) &= \max(\delta(N'_{\mathcal{C}}, k-1) + (A-1), A) \\ &\leq \max(\delta(N_{\mathcal{C}}, k-1) + (A-1), A) = \delta(N_{\mathcal{C}}, k).\end{aligned}$$ ■

Lemma 2. *Let $D = 1$, $A \geq 2$, $k \geq 1$, $N_{\mathcal{C}} = n_\lambda ... n_0$, and $N'_{\mathcal{C}} = n'_\lambda ... n'_0$. If $n_{i+k} ... n_i = 01^k$, $n'_{i+k} ... n'_i = 10^k$, and $\delta(N_{\mathcal{C}}, i-1) \geq \delta(N'_{\mathcal{C}}, i-1) \geq 2$, then*

$$\delta(N_{\mathcal{C}}, i+k) \geq \delta(N'_{\mathcal{C}}, i+k) \geq 2.$$

Proof. Define $d_{i-1} := \delta(N_{\mathcal{C}}, i-1)$, $d'_{i-1} := \delta(N'_{\mathcal{C}}, i-1)$, $d_{i+k} := \delta(N_{\mathcal{C}}, i+k)$, and $d'_{i+k} := \delta(N'_{\mathcal{C}}, i+k)$. Because $d_{i-1} \geq d'_{i-1} \geq 2$, we know that n_i and n'_{i+k} are not the least significant non-zero digits. From Definition 3, we have

$$\begin{aligned}d_{i+k} &= d_{i-1} + k(A-1) - 1 \\ d'_{i+k} &= \max(d'_{i-1} - k + (A-1), A).\end{aligned}$$

Case $d'_{i-1} - k < 1$: Because $d_{i-1} \geq 2$, we have $d_{i-1} - 1 \geq 1$ and

$$\begin{aligned} 2 \leq d'_{i+k} &= A \leq (d_{i-1} - 1) + (A - 1) \\ &\leq d_{i-1} - 1 + k(A-1) = d_{i+k}. \end{aligned}$$

Case $d'_{i-1} - k \geq 1$: We have

$$\begin{aligned} 2 \leq d'_{i+k} &= d'_{i-1} - k + (A-1) \\ &\leq d_{i-1} - k + (A-1) \\ &\leq d_{i-1} - 1 + k(A-1) = d_{i+k}. \end{aligned}$$

■

Lemma 3. *If $A \geq 2$, considering from the second least significant non-zero digit, NAF representation has smallest delay among all canonical binary representations.*

Proof. We prove this lemma by contradiction. Suppose there is canonical binary representation $N_{\mathcal{C}} = n_\lambda ... n_0$ with consecutive non-zero digits (not consider the least significant non-zero digit) that has smallest delay. Consider the consecutive non-zero digits in four following cases with $k \geq 2$.

Case $n_{i+k}...n_i = 01^k$: Consider $N'_{\mathcal{C}}$ with $n'_j = n_j$ for all $0 \leq j \leq \lambda$ except $n'_{i+k}...n'_i = 10^{k-1}\bar{1}$. Because $\delta(N_{\mathcal{C}}, i-1) = \delta(N'_{\mathcal{C}}, i-1)$, then $\delta(N_{\mathcal{C}}, i) = \delta(N'_{\mathcal{C}}, i) \geq A \geq 2$ and by Lemma 2, we can conclude that $\delta(N_{\mathcal{C}}, i+k) \geq \delta(N'_{\mathcal{C}}, i+k) \geq 2$. Since $n'_\lambda ... n'_{i+k+1} = n_\lambda ... n_{i+k+1}$, by Lemma 1, we get $\delta(N_{\mathcal{C}}, \lambda) \geq \delta(N'_{\mathcal{C}}, \lambda)$. Hence, in this case, we have NAF representation with no higher delay.

Case $n_{i+k}...n_i = 0\bar{1}^k$: The proof is similar to case 01^k.

Case $n_{i+1}n_i = 1\bar{1}$: Consider $N'_{\mathcal{C}}$ with $n'_j = n_j$ for all $0 \leq j \leq \lambda$ except $n'_{i+1}n'_i = 01$. Because $\delta(N_{\mathcal{C}}, i-1) = \delta(N'_{\mathcal{C}}, i-1)$, then

$$\begin{aligned} \delta(N_{\mathcal{C}}, i+1) &= \max(\delta(N_{\mathcal{C}}, i-1) + (A-1), A) + (A-1) \\ \delta(N'_{\mathcal{C}}, i+1) &= \max(\delta(N'_{\mathcal{C}}, i-1) + (A-1), A) - 1 \\ &\leq \delta(N_{\mathcal{C}}, i+1). \end{aligned}$$

Since $n'_\lambda ... n'_{i+2} = n_\lambda ... n_{i+2}$, by Lemma 1, we get $\delta(N_{\mathcal{C}}, \lambda) \geq \delta(N'_{\mathcal{C}}, \lambda)$. Hence, in this case, we have NAF representation with no higher delay.

Case $n_{i+1}n_i = \bar{1}1$: The proof is similar to case $1\bar{1}$. ■

Lemmas 1-3 show that NAF representation, which is unique, has the smallest delay but only when we consider from the second least significant non-zero digit. However, we can choose where the second least significant non-zero digit will be from two options: if the ending is 010^*, it could be changed to $1\bar{1}0^*$, and if the ending is 01^p110^* for some $p \geq 0$, it could be changed to $10^p0\bar{1}0^*$. This change in the second least significant non-zero digit's position may decrease the delay. We prove this in Theorem 4.

Theorem 4 (Optimal representation when $0 < 2D \leq A$)**.** *Algorithm 2 produces an optimal canonical binary representation $N^*_{\mathcal{C}} \in \mathscr{N}^*_{\mathcal{C}}$ for $0 < 2D \leq A$. That is, if $D = 1$ and $A \geq 2$, the representation according to the following rules has the smallest delay.*

- *If $N_{\mathcal{B}}$ ends with $11(01)^*010^*$, change the ending 01 to $1\bar{1}$ and change this representation to NAF starting at '1' in this $1\bar{1}$ (the second least significant non-zero digit's position is changed).*
- *If $N_{\mathcal{B}}$ ends with $0(01)^*0110^*$, change this representation to NAF starting at the second least significant '1' (the second least significant non-zero digit's position is not changed).*
- *Otherwise, change the representation to NAF starting at the least significant '1' (the second least significant non-zero digit's position is changed if the ending is $11(01)^*0110^*$ or 1110^*, and is not changed if the ending is $0(01)^*010^*$).*

Proof. Let $\mathscr{N}$ and $\mathscr{N}'$ be some consecutive digits in canonical binary representation and the least significant '1' of $N_{\mathcal{B}}$ is at index ℓ, we will consider each case as follows:

Case $11(01)^*010^*$ *ending:* Let $N_{\mathcal{B}} = \mathcal{N}11(01)^p010^*$ for some $p \geq 0$. Following Theorem 4, we change $N_{\mathcal{B}}$ to $\mathcal{N}11(01)^p1\bar{1}0^*$, and after changing to NAF, we have $N^*_{\mathcal{C}} = \mathcal{N}'00(\bar{1}0)^p\bar{1}\bar{1}0^*$. Consider the case where we do not change the position, after changing $N_{\mathcal{B}}$ to NAF, we have $N'_{\mathcal{C}} = \mathcal{N}'0\bar{1}(01)^p010^*$. Hence,

$$\begin{aligned}\delta(N'_{\mathcal{C}}, \ell+2p+3) &= (A-1) + p(A-2)\\ \delta(N^*_{\mathcal{C}}, \ell+2p+3) &= A + p(A-2) - 2\\ &\leq \delta(N'_{\mathcal{C}}, \ell+2p+3).\end{aligned}$$

Because both prefixes are $\mathcal{N}'$, by Lemma 1, we can conclude that Theorem 4 gives representation $N^*_{\mathcal{C}}$ with smaller delay in this case.

Case $0(01)^*0110^*$ *ending:* Let $N_{\mathcal{B}} = \mathcal{N}0(01)^p0110^*$ for some $p \geq 0$. Following Theorem 4, we do not change the position of the second least significant '1' and after changing $N_{\mathcal{B}}$ to NAF, we have $N^*_{\mathcal{C}} = \mathcal{N}'0(01)^p0110^*$. Consider the case where we change the position to $\mathcal{N}0(01)^p10\bar{1}0^*$, after changing to NAF, we have $N'_{\mathcal{C}} = \mathcal{N}'01(0\bar{1})^p0\bar{1}0^*$. Hence,

$$\begin{aligned}\delta(N'_{\mathcal{C}}, \ell+2p+3) &= (A-1) + p(A-2)\\ \delta(N^*_{\mathcal{C}}, \ell+2p+3) &= (A-1) + p(A-2) - 1\\ &\leq \delta(N'_{\mathcal{C}}, \ell+2p+3).\end{aligned}$$

Because both prefixes are $\mathcal{N}'$, by Lemma 1, we can conclude that Theorem 4 gives representation $N^*_{\mathcal{C}}$ with smaller delay in this case.

Case $11(01)^*0110^*$ *ending:* Let $N_{\mathcal{B}} = \mathcal{N}11(01)^p0110^*$ for some $p \geq 0$. Following Theorem 4, we change $N_{\mathcal{B}}$ to NAF and have $N^*_{\mathcal{C}} = \mathcal{N}'00(\bar{1}0)^p\bar{1}0\bar{1}0^*$. Consider the case where we do not change the position, after changing $N_{\mathcal{B}}$ to NAF, we have $N'_{\mathcal{C}} = \mathcal{N}'0\bar{1}(01)^p0110^*$. Hence,

$$\begin{aligned}\delta(N'_{\mathcal{C}}, \ell+2p+4) &= (A-1) + (p+1)(A-2)\\ \delta(N^*_{\mathcal{C}}, \ell+2p+4) &= A + p(A-2) - 2\\ &\leq \delta(N'_{\mathcal{C}}, \ell+2p+4).\end{aligned}$$

Because both prefixes are $\mathcal{N}'$, by Lemma 1, we can conclude that Theorem 4 gives representation $N^*_{\mathcal{C}}$ with smaller delay in this case.

Case 1110^* *ending:* Let $N_{\mathcal{B}} = \mathcal{N}1110^*$. Following Theorem 4, we change $N_{\mathcal{B}}$ to NAF and have $N^*_{\mathcal{C}} = \mathcal{N}'00\bar{1}0^*$. Consider the case where we do not change the position, after changing $N_{\mathcal{B}}$ to NAF, we have $N'_{\mathcal{C}} = \mathcal{N}'0\bar{1}10^*$. Hence,

$$\begin{aligned}\delta(N'_{\mathcal{C}}, \ell+2) &= A-1\\ \delta(N^*_{\mathcal{C}}, \ell+2) &= -2 \qquad \leq \delta(N'_{\mathcal{C}}, \ell+2).\end{aligned}$$

Because both prefixes are $\mathcal{N}'$, by Lemma 1, we can conclude that Theorem 4 gives representation $N^*_{\mathcal{C}}$ with smaller delay in this case.

Case $0(01)^*010^*$ *ending:* Let $N_{\mathcal{B}} = \mathcal{N}0(01)^p010^*$ for some $p \geq 0$. Following Theorem 4, we do not change the position of the second least significant '1' and after changing $N_{\mathcal{B}}$ to NAF, we have $N^*_{\mathcal{C}} = \mathcal{N}'0(01)^p010^*$. Consider the case where we change the position to $\mathcal{N}0(01)^p1\bar{1}0^*$, after changing to NAF, we have $N'_{\mathcal{C}} = \mathcal{N}'01(0\bar{1})^p\bar{1}0^*$. Hence,

$$\begin{aligned}\delta(N'_{\mathcal{C}}, \ell+2p+2) &= (A-1) + p(A-2)\\ \delta(N^*_{\mathcal{C}}, \ell+2p+2) &= \begin{cases}-2 & \text{if } p=0;\\ p(A-2) & \text{otherwise}\end{cases}\\ &\leq \delta(N'_{\mathcal{C}}, \ell+2p+2).\end{aligned}$$

Because both prefixes are $\mathcal{N}'$, by Lemma 1, we can conclude that Theorem 4 gives representation $N^*_{\mathcal{C}}$ with smaller delay in this case which concludes the proof. ■

From Theorem 4, since our $N^*_{\mathcal{C}}$ is in NAF (except the least significant non-zero digit), we have an upper bound of computation time used by our optimal representation when $0 < 2D \leq A$ as follows.

Corollary 1. *Let $D = 1$, $A \geq 2$, and $N_{\mathcal{B}} = n_\lambda ... n_0$. The upper bound of the parallel scalar point multiplication time using $N^*_{\mathcal{C}} = n'_{\lambda+1} ... n'_0$ from Algorithm 2 is*

$$T(N^*_{\mathcal{C}}, \lambda + 1) \leq \left(\frac{1}{2}\lambda + 1\right) A + D.$$

Proof. Since our $N^*_{\mathcal{C}}$ is in NAF except the least significant non-zero digit, the worst optimal representation could be in the form $(10)^{\lambda/2}11$ which has $T(N^*_{\mathcal{C}}, \lambda + 1) = \left(\frac{1}{2}\lambda + 1\right) A + D$. ■

Note that $T(N_{\mathcal{B}}, \lambda) \leq \lambda A + D$. This bound is tight since it can be achieved from $1^{\lambda+1}$. This means Algorithm 2 generates representations with lower upper bound of computation time.

Moreover, we can see that if we change $N_{\mathcal{B}}$ to NAF regardless of the position of the second least significant non-zero digit, the delay of NAF is different from the optimal no more than 1 (consider case $11(01)^*010^*$ and $0(01)^*0110^*$). Hence, NAF is almost optimal in this case.

6 Optimal Representation when $0 < D \leq A < 2D$

6.1 Algorithm

In the case where $1 \leq A < 2$ (normalize $D = 1$), we cannot use Algorithm 2 to have optimal canonical binary representation because Lemmas 2 and 3 do not hold in this case. Consider an example below to see that NAF and minimal Hamming weight representations may not be optimal when $1 \leq A < 2$.

Example 7. *Consider the case where $D = 1$, $A = 1.2$, and $n = 29$. If we calculate $29P$ using $N_{\mathcal{B}} = 011101$, we have $\delta(N_{\mathcal{B}}, \lambda) = 0.6$. If we use $N_{\mathcal{C}} = 1000\bar{1}\bar{1}$, which is optimal when $A \geq 2$, we have $\delta(N_{\mathcal{C}}, \lambda) = 1.2$ which is not optimal in this case. Using $N_{\mathcal{C}} = 100\bar{1}01$ also gives $\delta(N_{\mathcal{C}}, \lambda) = 1.2$.* □

Fortunately, we can construct $N^*_{\mathcal{C}} \in \mathscr{N}^*_{\mathcal{C}}$ from $N_{\mathcal{B}}$ in the case where $1 \leq A < 2$ using Algorithm 3. In Algorithm 3, we consider each digit from the least significant '1' to $n'_{\lambda+1}$ and calculate the delay at each digit using Definition 3. We use ℓ to keep the index of the least significant digit still in consideration. When we encounter $n_i = 0$, if delay $d > A$, we flip $n'_i ... n'_\ell$ from $01...11$ to $10...0\bar{1}$, set $d \leftarrow A$, and set $\ell \leftarrow i$ (start new considering sequence at n_i). If delay $d \leq 1$, we set $\ell \leftarrow i + 1$ (start new sequence at n_{i+1}). In the case where $1 < d \leq A$, we keep going until one of the previous cases occurs. See the following as an example:

Example 8. *Consider the case where $D = 1$, $A = 1.7$, and $n = 13911$. $N_{\mathcal{B}} = 011011001010111$ with $n_{\lambda+1} = 0$. From Algorithm 3, consider when $n_i = 0$, we have*

$i = 3;$	$d = 1.4 \in (1, A]$	$\ell = 0$
$i = 5;$	$d = 1.1 \in (1, A]$	$\ell = 0$
$i = 7;$	$d = 0.8 \leq 1$	$\ell = 8$
$i = 8;$	$d = -0.2 \leq 1$	$\ell = 9$
$i = 11;$	$d = 1.4 \in (1, A]$	$\ell = 9$
$i = 14;$	$d = 1.8 > A.$	

We have $d = 1.8 > A$, so we flip $n_{14} ... n_9$ from 011011 to $100\bar{1}0\bar{1}$ and get $N^*_{\mathcal{C}} = 100\bar{1}0\bar{1}001010111$. We also set $\ell = 14$ and then algorithm terminates. □

We can see that Algorithm 3 has $O(\log_2 n)$ complexity and uses $O(1)$ additional space.

Algorithm 3: Changing binary representation to optimal representation when $0 < D \leq A < 2D$

```
input  : $N_\mathcal{B} = n_\lambda ... n_0$
output : $N^*_\mathcal{C} = n'_{\lambda+1} ... n'_0 \in \mathscr{N}^*_\mathcal{C}$
begin
    $N^*_\mathcal{C} \leftarrow N_\mathcal{B}$
    $\ell \leftarrow$ index of the least significant '1' of $N^*_\mathcal{C}$
    $n'_{\lambda+1} \leftarrow 0$
    $d \leftarrow 0$
    for $i \leftarrow \ell + 1$ to $\lambda + 1$ do
        if $n'_i = 1$ then $d \leftarrow \max(d + (A - 1), A)$
        else
            $d \leftarrow d - 1$
            if $d > A$ then
                // "flipping" $n'_i ... n'_\ell$
                $n'_\ell \leftarrow \bar{1}$
                for $j \leftarrow \ell + 1$ to $i - 1$ do
                    $n'_j \leftarrow n'_j - 1$
                $n'_i \leftarrow 1$
                $d \leftarrow A$
                $\ell \leftarrow i$
            else if $d \leq 1$ then $\ell \leftarrow i + 1$
    return $N^*_\mathcal{C}$
```

6.2 Optimality Proof for Algorithm 3

We prove the optimality of Algorithm 3 using three lemmas and Theorem 5.

Lemma 4. *Let $D = 1$, $1 \leq A < 2$, $N_\mathcal{B} = n_\lambda ... n_0$ and delay d at $\ell - 1$ is no more than 1. Following Algorithm 3, if $n_i = 0$, $d > A$, and we flip $n_i ... n_\ell$, we have delay d at i equals to A.*

Proof. We first define $d^{(0)} \leq 1$ as the delay at $\ell - 1$. We can see that the sequence $n_i ... n_\ell$ must be in the form $01^{p_k} ... 01^{p_2} 01^{p_1}$ for some $k > 0$ and $p_j > 0$ for all $1 \leq j \leq k$. The sequence has no consecutive zeros because if $1 < d \leq A$ at the first zero, d is then less than 1 after the second zero and '00' is not in the sequence.

We prove this lemma by induction on j from 1 to k. We define $d^{(j)}$ as the delay after considering up to 01^{p_j}. We first consider 01^{p_1} where the least significant '1' of 01^{p_1} is not the least significant '1' of $N_\mathcal{B}$. If $p_1 = 1$, when consider 01^{p_1}, we have $d^{(1)} \leq 1$, which means this case cannot happen. For $p_1 > 1$, changing 01^{p_1} to $10^{p_1 - 1}\bar{1}$ makes

$$d^{(1)} = \max(A - (p_1 - 1) + (A - 1), A) = A$$

since $A - (p_1 - 1) < 1$. In the case where the least significant '1' of 01^{p_1} is also the least significant '1' of $N_\mathcal{B}$, we also have $d^{(1)} = A$ for all $p_1 > 0$.

For 01^{p_j} where $j > 1$, by induction, we have that flipping $01^{p_{j-1}} ... 01^{p_1}$ gives $10^{p_{j-1}} ... \bar{1}0^{p_1 - 1}\bar{1}$ with $d^{(j-1)} = A$. When we change $01^{p_j}1$ to $10^{p_j}\bar{1}$ with '1' from $10^{p_{j-1}}$, since $d^{(j-1)} - p_j = A - p_j < 1$, we also have

$$d^{(j)} = \max(d^{(j-1)} - p_j + (A - 1), A) = A. \qquad \blacksquare$$

Lemma 5. *Let $D = 1$, $1 \leq A < 2$, and $N_\mathcal{B} = n_\lambda ... n_0$. Following Algorithm 3, all sequences $n_i ... n_\ell$ have delay d at $\ell - 1$ no more than 1.*

Proof. We prove this lemma by induction on each sequence $n_{i_j}...n_{\ell_j}$ from the least to the most significant digit. We first consider the least significant sequence $n_{i_0}...n_{\ell_0}$. Since n_{ℓ_0} is the least significant non-zero digit of $N_{\mathcal{B}}$, the delay d at $\ell_0 - 1$ is definitely no more than 1. The base case is proved.

By induction, we assume that the considering sequence $n_{i_j}...n_{\ell_j}$ has delay d at $\ell_j - 1$ no more than 1. If we do not flip this sequence, we have d at i_j no more than 1. Because $n_k = 0$ for all $i_j + 1 \leq k \leq \ell_{j+1} - 1$ (since the next sequence starts at $n_{\ell_{j+1}}$), we have d at $\ell_{j+1} - 1$ no more than 1.

If we flip $n_{i_j}...n_{\ell_j}$, by Lemma 4, we have d at n_{i_j} equals to A and from Algorithm 3, the next sequence starts at n_{i_j} (we have $\ell_{j+1} = i_j$). Since $n_{i_j} = 1$ and $n_{i_j - 1} = 0$ with d at i_j equals to A, by Definition 3, we have that d at $i_j - 1 = \ell_{j+1} - 1$ must be no more than 1. The induction step is completed. ■

Lemma 6. *Let $D = 1$, $1 \leq A < 2$, and $N_{\mathcal{B}} = n_\lambda...n_0$. Following Algorithm 3, if $n_i = 0$ and $d > A$, there is no representation of $n_i...n_\ell$ which gives d at i smaller than A.*

Proof. We prove this lemma by contradiction. We consider the sequence $n_i...n_\ell$ in the form $01^{p_k}...01^{p_2}01^{p_1}$ for some $k > 0$ and $p_j > 0$ for all $1 \leq j \leq k$. To have $d^{(k)} < A$, '0' in 01^{p_k} must be left as '0' because if it is changed to '1', we have $d^{(k)} \geq A$. So, we consider '0' in $01^{p_{k-1}}$ and it must be left as '0' because if it is changed to '1', we have $d^{(k-1)} \geq A$ which makes $d^{(k)} \geq A$ (since before changing, we have $1 < d^{(k-1)} \leq A$ but $d^{(k)} > A$ already). Using the same reason, we have that '0' in 01^{p_1} must not be changed to have $d^{(k)} < A$. This contradicts the assumption since $01^{p_k}...01^{p_2}01^{p_1}$ has $d^{(k)} > A$. ■

Theorem 5 (Optimal representation when $0 < D \leq A < 2D$). *Algorithm 3 produces an optimal canonical binary representation $N^*_{\mathcal{C}} \in \mathscr{N}^*_{\mathcal{C}}$ for $0 < D \leq A < 2D$. That is, if $D = 1$ and $1 \leq A < 2$, when consider from n_0 to $n_{\lambda+1}$, flipping $n_i...n_\ell$ when $d > A$ gives the representation with the smallest delay.*

Proof. We consider the delay d after considering $n_i...n_\ell$ in two cases. If $d \leq 1$, changing the sequence will give us more delay which equals to or more than A, so our algorithm does not change it. This representation is optimal since there is no other representation with smaller delay than d. Otherwise, we have $d > A$, and by Lemmas 4-6, changing the sequence will make $d = A$ which is the smallest delay we can achieve. The sequence is always classified in one of these two cases since there are always two consecutive zeros in front of the representation which make $d \leq 1$. ■

From Theorem 5, we have an upper bound of computation time used by our optimal representation when $0 < D \leq A < 2D$ as follows.

Corollary 2. *Let $D = 1$, $1 \leq A < 2$, and $N_{\mathcal{B}} = n_\lambda...n_0$. The upper bounds of the delay and parallel scalar point multiplication time using $N^*_{\mathcal{C}} = n'_{\lambda+1}...n'_0$ from Algorithm 3 are*

$$\delta(N^*_{\mathcal{C}}, \lambda+1) \leq A$$
$$T(N^*_{\mathcal{C}}, \lambda+1) \leq A + \lambda + 1.$$

Proof. Consider $n'_{\lambda+1}$, if $n'_{\lambda+1} = 0$, it is obvious that $\delta(N^*_{\mathcal{C}}, \lambda+1) \leq A$. If $n'_{\lambda+1} = 1$, this happens from flipping and we have $\delta(N^*_{\mathcal{C}}, \lambda+1) = A$. Because the delay is no more than A at $n'_{\lambda+1}$, hence $T(N^*_{\mathcal{C}}, \lambda+1) = \delta(N^*_{\mathcal{C}}, \lambda+1) + \lambda + 1 \leq A + \lambda + 1$. ■

Note that $T(N_{\mathcal{B}}, \lambda) \leq \lambda A + D$. This bound is tight since it can be achieved from $1^{\lambda+1}$. This means Algorithm 3 generates representations with lower upper bound of computation time.

Moreover, if we change $N_{\mathcal{B}}$ to NAF, its delay after considering $n_{\lambda+1}$ is also no more than A but may not be optimal (as shown in Example 7). We prove this in the following proposition.

Proposition 2. *Let $N_{\mathcal{C}}$ be a NAF representation of n. When $D = 1$ and $1 \leq A < 2$, we have $\delta(N_{\mathcal{C}}, i) \leq A$ for all $i \geq 0$. Furthermore, if $n_i = 0$, then $\delta(N_{\mathcal{C}}, i) \leq A - 1$.*

Proof. We prove this proposition by induction on i. When $i = 0$, we know that $\delta(N_{\mathcal{C}}, i) = 0$ by the definition of the function δ. For $i \geq 1$, assume that $\delta(N_{\mathcal{C}}, i-1) \leq A$ when $n_{i-1} \neq 0$, and $\delta(N_{\mathcal{C}}, i-1) \leq A - 1$ when $n_{i-1} = 0$.

If $n_i = 0$, then $\delta(N_{\mathcal{C}}, i) = \delta(N_{\mathcal{C}}, i-1) - 1 \leq A - 1$.

If $n_i \neq 0$, then, because there is not a consecutive non-zeros in NAF representation, we have $n_{i-1} = 0$. Hence, $\delta(N_{\mathcal{C}}, i) = 0$, $\delta(N_{\mathcal{C}}, i) = A$, or $\delta(N_{\mathcal{C}}, i) = \delta(N_{\mathcal{C}}, i-1) + (A-1) \leq \delta(N_{\mathcal{C}}, i-1) + 1 \leq (A-1) + 1 = A$. ■

Since the delay from the optimal representation is no more than A, the delay from NAF is not larger than the delay from the optimal representation by more than $A < 2$. Hence, NAF is also almost optimal in this case.

7 Experimental Results

We compare the parallel computation time and the buffer size required for scalar point multiplication when we use binary representations $N_{\mathcal{B}}$, our optimal representations $N_{\mathcal{C}}^*$, and NAFs in Table 1. The idea of this experiment is similar to the experiments in [4]. The buffer is used when doubling processor finishes its work before addition processor uses it. For example, from Figure 2, addition processor finishes calculating $-5P$ at time 11, but $4P$, $8P$, $16P$ and $32P$ are finished before that time. These results are waiting in addition processor's buffer. We keep only what is going to be used, i.e. keep 2^iP only when $n_i \neq 0$. The experimental results are obtained from 100,000 random integers from $[1, 2^{256} - 1]$.

Table 1: Computation time of scalar point multiplication and buffer space used with 100,000 random integers from $[1, 2^{256} - 1]$

$\frac{A}{D}$	Computation Time (unit)						Buffer Space (number of elliptic curve points)					
	using $N_{\mathcal{B}}$		using $N_{\mathcal{C}}^*$		using NAF		using $N_{\mathcal{B}}$		using $N_{\mathcal{C}}^*$		using NAF	
	avg.	max	avg.	max	avg.	max	avg.	max	avg.	max	avg.	max
1.00	**255.0**	**256.0**	**255.0**	**256.0**	255.7	257.0	**1.000**	**1**	**1.000**	**1**	**1.000**	**1**
1.25	**255.5**	261.0	**255.5**	**257.3**	255.9	**257.3**	2.682	6	2.000	2	**1.000**	**1**
1.50	256.3	268.5	**255.9**	**257.5**	256.2	**257.5**	3.854	10	2.000	2	**1.000**	**1**
1.75	258.4	291.3	**256.3**	**257.7**	256.4	**257.7**	5.991	22	2.000	2	**1.000**	**1**
2.00	268.2	326.0	**256.7**	**258.0**	**256.7**	**258.0**	10.238	37	**1.000**	**1**	**1.000**	**1**
2.25	292.2	366.5	**257.2**	**264.3**	**257.2**	**264.3**	18.817	50	**2.043**	**4**	2.044	**4**
2.50	322.1	407.0	**258.0**	**274.5**	**258.0**	**274.5**	28.245	61	**2.742**	**7**	2.745	**7**
2.75	353.3	447.5	**260.0**	**289.8**	**260.0**	**289.8**	36.824	69	**3.974**	**13**	3.979	**13**

We can see from the results that NAFs are not always the optimal representation as the average computation times from the representation are slightly larger than the optimal one. However, as discussed previously, the results from NAFs are almost as good as those from our optimal representation. The difference between two representations is always less than 1%, and the parallel computation time when $A/D \geq 2$ is nearly same. Also, the buffer size in scalar point multiplication method obtained from NAF is almost equal to that obtained from our optimal representation. Hence, NAF is almost an optimal choice for our model. On the other hand, the results from NAF and $N_{\mathcal{C}}^*$ are much better than those obtained from $N_{\mathcal{B}}$, especially when $A/D \geq 2$. The improvement from $N_{\mathcal{B}}$ in the parallel computation time is as large as 4.4% when $A/D = 2$, and the improvement is as large as 35.9% when $A/D = 2.75$.

8 Optimal Representation for Multi-Scalar Point Multiplication (under a condition)

In this section, we focus on multi-scalar point multiplication where we want to calculate $nP + mQ$ for some integers n, m and elliptic curve points P, Q. We use similar parallel model to scalar point multiplication with three processors, two for doubling P, Q and another one for adding doubles cumulatively. We use the same settings as scalar point multiplication (using canonical binary representation). See following example for more understanding.

Example 9. *To calculate $9P + 13Q$ using $N_{\mathcal{B}} = 1001$ and $M_{\mathcal{B}} = 1101$, we use three processors as depicts in Figure 3.* □

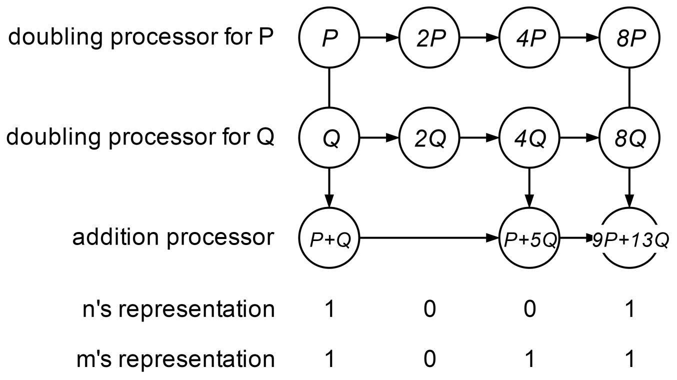

Figure 3: "Parallel double-and-add" multi-scalar point multiplication for $N_{\mathcal{B}} = 1001$ and $M_{\mathcal{B}} = 1101$

We define the time used in parallel multi-scalar point multiplication as follows:

Definition 5 (Computation time of parallel multi-scalar point multiplication)**.** *Let $D \geq 0$ be the time processor used for one doubling, $A \geq 0$ be the time used for one addition, and $n, m \in \mathbb{Z}_+$ with canonical binary representation $N_{\mathcal{C}} = n_\lambda ... n_0$ and $M_{\mathcal{C}} = m_\lambda ... m_0$. The computation time of parallel multi-scalar point multiplication $nP + mQ$ using $N_{\mathcal{C}}$ and $M_{\mathcal{C}}$ after calculating from n_0, m_0 up to n_i, m_i is $T(N_{\mathcal{C}}, M_{\mathcal{C}}, i)$ which can be calculated by:*

$$T(N_{\mathcal{C}}, M_{\mathcal{C}}, i) := \begin{cases} |n_i m_i| A & \text{if } i = 0; \\ T(N_{\mathcal{C}}, M_{\mathcal{C}}, i-1) & \text{if } i > 0 \text{ and } n_i = m_i = 0; \\ iD + |n_i m_i| A & \text{if } i > 0,\ |n_i| + |m_i| \neq 0, \\ & \text{and } n_j = m_j = 0; \forall j, 0 \leq j < i; \\ \max(T(N_{\mathcal{C}}, M_{\mathcal{C}}, i-1), iD) + (|n_i| + |m_i|)A & \text{otherwise.} \end{cases}$$

From Definition 5, the time used to calculate $nP + mQ$ using $N_{\mathcal{C}}, M_{\mathcal{C}}$ is $T(N_{\mathcal{C}}, M_{\mathcal{C}}, \lambda)$. If we consider representation X with $x_j = |n_j| + |m_j|$ for all $0 \leq j \leq \lambda$, we have $T(N_{\mathcal{C}}, M_{\mathcal{C}}, \lambda) = T(X, \lambda)$. However, finding optimal representations for parallel multi-scalar point multiplication is more complicated than parallel scalar point multiplication since we have to consider both $N_{\mathcal{C}}$ and $M_{\mathcal{C}}$ and changing one may affect other. In this paper, we propose a dynamic programming algorithm in Algorithm 4 to find optimal time *under a condition that three processors are used as described above, and only canonical binary representations using the digit set $\{\bar{1}, 0, 1\}$ are considered.*

Before we begin the explanation of Algorithm 4, we first define the word "carry-in" and "carry-out". When we consider digit n_i, "carry-in" is a number that is transferred from the less significant digit n_{i-1} and is added to n_i, and "carry-out" is a number that is transferred from n_i and is added to the more significant digit n_{i+1} after the digit n_i is changed. For example, if $n_i = 1$ with "carry-in" equals to 0, n_i may not be changed and have "carry-out" 0, or n_i may be changed to $\bar{1}$ and have "carry-out" 1 since $1 = 1\bar{1}$. Another example is when $n_i = 1$ with "carry-in" equals to 1. After

Algorithm 4: Calculating the smallest parallel multi-scalar point multiplication time of $nP + mQ$ using dynamic programming

```
input  : $N_{\mathcal{B}} = n_\lambda ... n_0$, $M_{\mathcal{B}} = m_\lambda ... m_0$
output: the smallest time $t$ to calculate $nP + mQ$
begin
    $\ell \leftarrow$ the smallest index where $n_\ell = 1$ or $m_\ell = 1$
    $t_{0,0}, t_{0,1}, t_{1,0}, t_{1,1} \leftarrow \infty$
    if $n_\ell = 0$ and $m_\ell = 1$ then $t_{0,0}, t_{0,1} \leftarrow \ell D$
    else if $n_\ell = 1$ and $m_\ell = 0$ then $t_{0,0}, t_{1,0} \leftarrow \ell D$
    else $t_{0,0}, t_{0,1}, t_{1,0}, t_{1,1} \leftarrow \ell D + A$
    for $i \leftarrow \ell + 1$ to $\lambda$ do
        $t'_{0,0}, t'_{0,1}, t'_{1,0}, t'_{1,1} \leftarrow \infty$
        for all $CI := (ci_n, ci_m) \in \{0, 1\}^2$ do
            $n' \leftarrow n_i + ci_n$; $m' \leftarrow m_i + ci_m$
            $n'' \leftarrow n' \mod 2$; $m'' \leftarrow m' \mod 2$
            $S_n \leftarrow \{\lfloor n'/2 \rfloor, \lceil n'/2 \rceil\}$; $S_m \leftarrow \{\lfloor m'/2 \rfloor, \lceil m'/2 \rceil\}$
            for all $CO := (co_n, co_m) \in S_n \times S_m$ do
                if $n'' = m'' = 0$ then $t'' \leftarrow t_{CI}$
                else $t'' \leftarrow \max(t_{CI}, iD) + (n'' + m'')A$
                $t'_{CO} \leftarrow \min(t'_{CO}, t'')$
        $t_{0,0} \leftarrow t'_{0,0}$; $t_{0,1} \leftarrow t'_{0,1}$; $t_{1,0} \leftarrow t'_{1,0}$; $t_{1,1} \leftarrow t'_{1,1}$
    $t_{0,1} \leftarrow \max(t_{0,1}, (\lambda + 1)D) + A$
    $t_{1,0} \leftarrow \max(t_{1,0}, (\lambda + 1)D) + A$
    $t_{1,1} \leftarrow \max(t_{1,1}, (\lambda + 1)D) + 2A$
    return $t \leftarrow \min(t_{0,0}, t_{0,1}, t_{1,0}, t_{1,1})$
```

adding the "carry-in", we have $n_i = 2$ which is then changed to $n_i = 0$ with "carry-out" 1 since $2 = 10$. We can say that "carry-out" of n_i is "carry-in" of n_{i+1}.

The idea of Algorithm 4 is as follows: we consider each digit from n_ℓ, m_ℓ to n_λ, m_λ where ℓ is the smallest index that $n_\ell = 1$ or $m_\ell = 1$. At n_ℓ, m_ℓ, we set the time according to Definition 5. If $n_\ell = 1$, we can change n_ℓ to $\bar{1}$ with "carry-out" 1 or leave n_ℓ as 1 with "carry-out" 0. This is similar for m_ℓ. We keep all possible results in $t_{x,y}$ which stores the time used up to the current digits with "carry-out" x for next digit of $N_{\mathcal{B}}$ and "carry-out" y for next digit of $M_{\mathcal{B}}$.

For other digits n_i, m_i, we consider every possibilities of "carry-in" from the previous digits. We use n' and m' to store the result after adding n_i, m_i with the "carry-in". If $n' = 0$, this digit must be 0 with "carry-out" 0. If $n' = 1$, there are two ways: this digit is unchanged with "carry-out" 0, or this digit is changed to $\bar{1}$ with "carry-out" 1. If $n' = 2$, this digit must be 0 with "carry-out" 1. This is similar for m'. We use n'', m'' to store the new current digits and S_n, S_m to store all possible "carry-out" as a set. This can be summarized as in Table 2 (Algorithm 4 treats $n'', m'' = \bar{1}$ as 1 since there is no difference in time calculation). After that, we compute the time and keep the minimum time possible for each t_{CO} to use at next digit (we use t'_{CO} as a temporary variable to calculate t_{CO} for next digit). Because minimum time of every possibilities are considered at all digits, it is straightforward to see that Algorithm 4 computes the optimal time under the condition.

After calculating up to n_λ, m_λ, there is still a carry for $n_{\lambda+1} = 0, m_{\lambda+1} = 0$ for $t_{0,1}, t_{1,0}$ and $t_{1,1}$. We calculate the time for the remaining carry and return the minimum time among all possibilities.

Example 10. *Consider the case where $n_i = 0, m_i = 1$ with "carry-in" $ci_n = 1$ and $ci_m = 1$. After adding the "carry-in", we have $n' = 1$ and $m' = 2$. From Table 2, the new n_i can be 1 or $\bar{1}$ which can be treated as 1 when calculating time ($n'' = 1$), and the new m_i must be 0 ($m'' = 0$). All possible "carry-out" of n_i is $S_n = \{0, 1\}$ and m_i is $S_m = \{1\}$. In this case, we update the value of $t'_{0,1}$ and $t'_{1,1}$. We need to consider other three cases of "carry-in" before considering n_{i+1}, m_{i+1}.* □

Table 2: All possibilities of new current digit and "carry-out" for each value of n', m'

n', m'	**new current digit** (n'', m'')	**carry-out**	S_n, S_m
0	0	0	$\{0\}$
1	1	0	$\{0, 1\}$
	$\bar{1}$	1	
2	0	1	$\{1\}$

Without loss of generality, we assume that $n \geq m$. Algorithm 4 has $O(\log_2 n)$ complexity and uses $O(1)$ additional space, but to do backtracking, $O(\log_2 n)$ additional space is required.

We compare the average parallel computation time and the buffer size required for multi-scalar point multiplication when we use binary representations $N_{\mathcal{B}}$, our optimal representations (under the condition) $N^*_{\mathcal{C}}$, NAFs, and joint sparse forms (JSFs) in Table 3. The experimental results are obtained from 100,000 random pairs of integers from $[1, 2^{256} - 1] \times [1, 2^{256} - 1]$.

Table 3: Computation time of multi-scalar point multiplication and buffer space used with 100,000 random pairs of integers from $[1, 2^{256} - 1] \times [1, 2^{256} - 1]$

$\frac{A}{D}$	Average Computation Time (unit)				Average Buffer Space (number of elliptic curve points)			
	$N_{\mathcal{B}}$	$N^*_{\mathcal{C}}$	NAF	JSF	$N_{\mathcal{B}}$	$N^*_{\mathcal{C}}$	NAF	JSF
1.00	264.5	**256.3**	256.7	256.7	14.789	3.000	**2.000**	**2.000**
1.25	320.7	**257.2**	257.4	257.7	54.366	4.299	**4.151**	5.234
1.50	384.0	**264.2**	264.4	269.9	87.163	9.329	**9.194**	12.856
1.75	447.5	**300.9**	301.0	310.2	111.089	27.463	**27.332**	32.766
2.00	511.2	**342.9**	343.0	353.6	128.910	44.661	**44.517**	50.039
2.25	574.9	**385.4**	385.5	397.4	143.193	58.762	**58.649**	64.157
2.50	638.6	**427.9**	428.0	441.3	154.324	69.847	**69.729**	75.242
2.75	702.3	**470.5**	470.6	485.2	163.507	79.047	**78.942**	84.442

The results show that NAFs are nearly optimal in our model. The difference between optimal representations and NAFs is always less than 1%, and the buffer size obtained from NAF is slightly better than that obtained from our optimal representation. Hence, NAF is almost an optimal choice for our model for multi-scalar point multiplication. On the other hand, the results from NAF and $N^*_{\mathcal{C}}$ are much better than those obtained from JSF, especially when $A/D \geq 2$. Nevertheless, it is not proved that using the digit set $\mathcal{C} = \{\bar{1}, 0, 1\}$ is optimal. We let this be an open problem.

9 Conclusion and Future Work

This paper presents that NAF is almost optimal for our proposed time model for "parallel double-and-add" scalar and multi-scalar point multiplication (under the condition). This is because NAF uses a little more time in the model, nearly the same buffer space, and the same time to generate the representation as our optimal representation. However, there are still some issues left unsolved. For multi-scalar point multiplication, the optimality of the digit set used is not proved, we still have not found an algorithm using $O(1)$ space, and the number of processors might affect the optimal representation, e.g. five processors with one addition and four doublings (start with P, Q, $P + Q$, and $P - Q$) will treat 10, 01, 11, and $1\bar{1}$ pairs the same. We aim to solve these problems in our future works. In addition, we might be able to improve the computation time by utilizing the Strauss-Shamir trick [7, 22] or radix-r NAF representation [24]. Also, we plan to consider the communication time between processors, and other parallel settings such as SIMD and SIMT paradigm [14].

Acknowledgment

This research was mainly done while Kittiphon Phalakarn and Kittiphop Phalakarn did an internship at The University of Tokyo. The authors would like to thank The University of Tokyo, Asst. Prof. Athasit Surarerks, and Prof. Hiroshi Imai for facilitating the internship. We would also like to thank the editor and reviewers for all corrections and useful comments. This work was supported by JST ERATO Grant Number JPMJER1201, Japan.

References

[1] Explicit-formulas database. https://hyperelliptic.org/EFD/index.html. Accessed: 2016-07-05.

[2] Daniel Bernstein and Tanja Lange. Two grumpy giants and a baby. In *Proceedings of the Tenth Algorithmic Number Theory Symposium (ANTS X)*, pages 87–111. Mathematical Sciences Publishers, 2013.

[3] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In *Proceedings of the 2008 International Conference on Cryptology in Africa (AfricaCrypt 2008)*, pages 389–405. Springer, 2008.

[4] Daniel J Bernstein, Peter Birkner, Tanja Lange, and Christiane Peters. Optimizing double-base elliptic-curve single-scalar multiplication. In *International Conference on Cryptology in India*, pages 167–182. Springer, 2007.

[5] Fábio Borges, Pedro Lara, and Renato Portugal. Parallel algorithms for modular multi-exponentiation. *Applied Mathematics and Computation*, 292:406–416, 2017.

[6] Allan B Borodin and Ian Munro. Notes on efficient and optimal algorithms. *U. of Toronto, Toronto, Canada, and U. of Waterloo, Waterloo, Canada*, 1972.

[7] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[8] Juan Manuel Garcia Garcia and Rolando Menchaca Garcia. Parallel algorithm for multiplication on elliptic curves. In *Proceedings of the 2001 Mexican International Conference on Computer Science (ENC 2001)*. Springer, 2001.

[9] Daniel M Gordon. A survey of fast exponentiation methods. *Journal of algorithms*, 27(1):129–146, 1998.

[10] Louis Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In *Proceedings of the 2003 International Workshop on Public Key Cryptography (PKC 2003)*, pages 199–211. Springer, 2003.

[11] Clemens Heuberger and Helmut Prodinger. On minimal expansions in redundant number systems: Algorithms and quantitative analysis. *Computing*, 66(4):377–393, 2001.

[12] Tetsuya Izu and Tsuyoshi Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *Proceedings of the 2002 International Workshop on Public Key Cryptography*, pages 280–296. Springer, 2002.

[13] Kenji Koyama and Yukio Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. In *Proceedings of the 1992 Annual International Cryptology Conference (Crypto 1992)*, pages 345–357. Springer, 1992.

[14] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.

[15] Bodo Möller. Improved techniques for fast exponentiation. In *Proceedings of the 2002 International Conference on Information Security and Cryptology (ICISC 2002)*, pages 298–312. Springer, 2002.

[16] Carlos Moreno and M Anwar Hasan. SPA-resistant binary exponentiation with optimal execution time. *Journal of Cryptographic Engineering*, 1(2):87–99, 2011.

[17] Michael Nöcker. Some remarks on parallel exponentiation. In *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation (ISSAC 2000)*, pages 250–257. ACM, 2000.

[18] Kittiphon Phalakarn, Kittiphop Phalakarn, and Vorapong Suppakitpaisarn. Optimal representation for right-to-left parallel scalar point multiplication. In *Computing and Networking (CANDAR), 2017 Fifth International Symposium on*, pages 482–488. IEEE, 2017.

[19] George W Reitwiesner. Binary arithmetic. *Advances in computers*, 1:231–308, 1960.

[20] J. Robert. Software implementation of parallelized ECSM over binary and prime fields. In *Proceedings of the 2014 International Conference on Information Security and Cryptography (Inscrypt 2014)*, pages 445–462. Springer, 2014.

[21] Jerome A Solinas. Low-weight binary representation for pairs of integers. *Centre for Applied Cryptographic Research, University of Waterloo, Combinatorics and Optimization Research Report CORR 2001-41*, 2001.

[22] Ernst Gabor Straus. Problems and solutions: Addition chains of vectors. *American Mathematical Monthly*, 71(806-808), 1964.

[23] Vorapong Suppakitpaisarn and Hiroshi Imai. Worst case computation time for minimal joint hamming weight numeral system. In *Proceedings of the 2014 International Symposium on Information Theory and its Applications (ISITA 2014)*, pages 138–142. IEEE, 2014.

[24] Tsuyoshi Takagi, David Jr Reis, Sung-Ming Yen, and Bo-Ching Wu. Radix-r non-adjacent form and its application to pairing-based cryptosystem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 89(1):115–123, 2006.

[25] Edward G Thurber. On addition chains $\ell(mn) \leq \ell(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40(4):907–913, 1973.