Hindawi Publishing Corporation
Journal of Biomedicine and Biotechnology
Volume 2008, Article ID 518093, 10 pages
doi:10.1155/2008/518093

Research Article

Fast Parallel Molecular Algorithms for DNA-Based Computation: Solving the Elliptic Curve Discrete Logarithm Problem over $GF(2^n)$

Kenli Li,[1, 2] Shuting Zou,[1] and Jin Xv[2]

[1] *Embedded System and Networking Laboratory, College of Computer and Communication, Hunan University, Changsha 410082, China*
[2] *Department of Control Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China*

Correspondence should be addressed to Shuting Zou, zst991221@163.com

Received 16 July 2007; Accepted 25 January 2008

Recommended by Daniel Howard

Elliptic curve cryptographic algorithms convert input data to unrecognizable encryption and the unrecognizable data back again into its original decrypted form. The security of this form of encryption hinges on the enormous difficulty that is required to solve the elliptic curve discrete logarithm problem (ECDLP), especially over $GF(2^n)$, $n \in Z^+$. This paper describes an effective method to find solutions to the ECDLP by means of a molecular computer. We propose that this research accomplishment would represent a breakthrough for applied biological computation and this paper demonstrates that in principle this is possible. Three DNA-based algorithms: a parallel adder, a parallel multiplier, and a parallel inverse over $GF(2^n)$ are described. The biological operation time of all of these algorithms is polynomial with respect to n. Considering this analysis, cryptography using a public key might be less secure. In this respect, a principal contribution of this paper is to provide enhanced evidence of the potential of molecular computing to tackle such ambitious computations.

1. INTRODUCTION

This paper proposes theoretical work that introduces powerful algorithms of molecular computation that could potentially compromise the security that is afforded by certain cryptography algorithms. Molecular computation [1] involves biochemistry and DNA rather than silicon chips to tackle formidable computations. Theoretical aspects of this interdisciplinary field are important to develop the potential and interest in this form of computation [2].

Elliptic curve cryptography (ECC) is a mathematical approach to public key cryptography using elliptic curves that are typically defined over finite fields [3]. Elliptic curves [4, 5] constitute a major area of current research that is particularly important to number theory, for example, elliptic curves had a role in the recent proof of Fermats last theorem. As applied to cryptography, not only has ECC become applied in Diffie-Hellman key exchange but also in the digital signature algorithm (DSA), a US federal government standard for digital signatures. It is known as the elliptic curve DSA (ECDSA) or that variant of the DSA operating on elliptic curve groups.

The security of these cryptosystems relies on the difficulty of solving the elliptic curve discrete logarithm problem [6, 7]. If P is a point with order m on an elliptic curve, and Q is some other point on the same curve, then the elliptic curve discrete logarithm problem is to determine an l such that $Q = lP$, where l is an integer and $0 \le l \le m - 1$. If this problem can be solved efficiently, then elliptic curve-based cryptosystems can be broken efficiently.

In order to tackle such a problem, Feynman proposed molecular computation in 1961 [8]. However, his idea was not implemented experimentally for some decades. In 1994, Adleman succeeded in solving an instance of the Hamiltonian path problem in a test tube, simply by the manipulation of DNA strands [1]. Following this, Lipton demonstrated that the Adleman techniques offered a solution to the satisfiability problem (the first considered NP-complete problem) [9].

Recent advances in molecular biology [10, 11] have made it possible to produce roughly 10^{18} DNA strands in a test tube. Those 10^{18} DNA strands can be made to represent 10^{18} bits of information. In a distant future, if biological operations may be run error free using a test tube with 10^{18} DNA strands, then it would be possible to process 10^{18} bits of information simultaneously. More details about test tube distributed systems are given in [2]. The objective for biological computing technology is to provide this enormous amount of parallelism for dealing with computationally intensive real world problems [12–14].

Advancement in DNA computing has already been made in many areas. In the field of cryptology, Boneh et al. have cracked DES using identical principles to those of Adleman's solution of the travelling salesman problem. Also, Chang et al. have developed a way to factor integers. They proposed three DNA-based algorithms: parallel subtractor; parallel comparator; and parallel modular arithmetic unit [15, 16].

In this paper, we take a step further with respect to Chang's work [16] in order to solve the elliptic curve discrete logarithm problem. We develop DNA-based algorithms for a parallel adder; a parallel multiplier; a parallel divider over $GF(2^n)$ (i.e., a Galois field of characteristic 2); and a parallel adder for adding points on elliptic curves. We accomplish all of these by means of basic biological operations. We also showed that cryptosystems based on elliptic curves can be broken. Our work presents clear evidence of molecular computing abilities to accomplish complex mathematical operations.

The paper is organized as follows. Section 2 gives a brief background on DNA computing. Section 3 introduces the DNA computing that solves the elliptic curve discrete logarithm problem, for solution spaces of DNA strands. Conclusions are drawn in the final section.

2. BACKGROUND

DNA (*DeoxyriboNucleic Acid*) is the *molecule* that plays the main role in DNA-based computing. DNA is a polymer, which is strung together from monomers called *deoxyriboNucleotides*. Distinct nucleotides are detected only with their bases, which come in two sorts: *purines* and *pyrimidines*. Purines include *adenine* and *guanine*, abbreviated A and G. Pyrimidines contain *cytosine* and *thymine*, abbreviated C and T. A DNA strand is essentially a sequence (polymer) of four types of nucleotides detected by one of four bases they contain. Two strands of DNA can form (under appropriate conditions) a double strand, if the respective bases are the Watson-Crick complements of each other—A matches T and C matches G. Hybridization is a special technology term for the pairing of two single DNA strands to make a double helix and also takes advantages of the specificity of DNA base pairing for the detection of specific DNA strands (for more discussions of the relevant biological background, refer to [10, 11]).

In the past decade, there have been revolutionary advances in the field of biomedical engineering, particularly in recombinant DNA and RNA manipulating. Due to the industrialization of the biotechnology field, laboratory techniques for recombinant DNA and RNA manipulation are becoming highly standardized. Basic principles about recombinant DNA can be found in [17–20]. In the following, we describe five biological operations that are useful for solving the elliptic curve discrete logarithm problem.

A (test) tube is a set of molecules of DNA (a multiset of finite strings over the alphabet $\{A, C, G, T\}$). Given a tube, one can perform the following operations.

(1) *Extract*. Given a tube P and a short single strand of DNA, S, the operation produces two tubes $+(P, S)$ and $-(P, S)$, where $+(P, S)$ is all of the molecules of DNA in P which contain S as a substrand and $-(P, S)$ is all of the molecules of DNA in P which do not contain S.

(2) *Merge*. Given tubes P_1 and P_2, yield $\cup(P_1, P_2)$, where $\cup(P_1, P_2) = P_1 \cup P_2$. This operation is to pour two tubes into one, without any change in the individual strands.

(3) *Amplify*. Given a tube P, the operation Amplify (P, P_1, P_2), will produce two new tubes P_1 and P_2 so that P_1 and P_2 are totally a copy of P (P_1 and P_2 are now identical) and P becomes an empty tube.

(4) *Append*. Given a tube P containing a short strand of DNA Z, the operation will append Z onto the end of every strand in P.

(5) *Append-head*. Given a tube P containing a short strand of DNA, Z, the operation will append Z onto the head of every strand in P.

3. FINDING THE DISCRETE LOGARITHM ON ELLIPTIC CURVE OVER $GF(2^n)$

3.1. *Elliptic curve public key cryptosystem over* $GF(2^n)$

An elliptic curve is defined to be the set of solutions $(x, y) \in GF(2^n) \times GF(2^n)$ to the equation

$$y^2 + xy = x^3 + ax^2 + b, \tag{1}$$

where $a, b \in GF(2^n)$ and $b \neq 0$, together with the point on the curve at infinity O, (with homogeneous coordinates $(0, 0)$).

The points on an elliptic curve form an Abelian group under a well-defined group operation. The identity of the group operation is the point O. For $P = (x_1, y_1)$ a point on the curve, we define $-P$ to be $(x_1, y_1 + x_1)$, so $P + (-P) = (-P) + P = O$. Now suppose P and Q are not O, and $P \neq -Q$. Let P be as above and $Q = (x_2, y_2)$, then $P + Q = (x_3, y_3)$, where

$$\begin{aligned} x^3 &= \mu^2 + \mu + x_1 + x_2 + a, \\ y_3 &= \mu(x_1 + x_3) + x_3 + y_{1,} \\ \mu &= \begin{cases} \dfrac{y_2 + y_1}{x_2 + x_1} & \text{if } P \neq Q, \\[2ex] \dfrac{x_1^2 + y}{x_1} & \text{if } P = Q, \end{cases} \end{aligned} \tag{2}$$

(refer to [21]).

In this paper, for convenience, use $b_{n-1}b_{n-2} \cdots b_1 b_0$ to denote the value of $b_{n-1}\omega^{n-1} + b_{n-2}\omega^{n-2} + \cdots + b_1\omega + b_0$ over $GF(2^n)$.

```
(1) For $j = 0$ to $n - 1$
    (1a) $T_1 = +(T_0, x^1_{p+j})$ and $T_2 = -(T_0, x^1_{p+j})$
    (1b) $T_3 = +(T_1, x^1_{q+j})$ and $T_4 = -(T_1, x^1_{q+j})$
    (1c) $T_5 = +(T_2, x^1_{q+j})$ and $T_6 = -(T_2, x^1_{q+j})$
    (1d) $T_7 = \cup(T_4, T_5)$ and $T_8 = \cup(T_3, T_6)$
    (1e) Append $(T_7, x^1_{r+j})$ and Append $(T_8, x^0_{r+j})$
    (1f) $T_0 = \cup(T_7, T_8)$
    EndFor
EndProcedure
```

Algorithm 1: Procedure ParallelAdder (T_0, n, p, q, r).

Let E be an elliptic curve defined over $GF(2^n)$, and let $G \in E$ be a fixed and publicly known point. The receiver B chooses a randomly and publishes the key aG, while keeping a itself secret. To transmit a message m to B, user A chooses a random integer k and sends the pair of points $(kG, P_m + k(aG))$. To read the message, B multiplies the first point in the pair by his secret a, and then subtracts the result from the second point in the pair. So, if a breaker can compute a from public key G and aG, he can decrypt any encryption sent to B (refer to [3]).

3.2. The construction of a parallel adder over $GF(2^n)$

Over $GF(2^n)$, the additive operation on two numbers is just doing *XOR* on each bit, respectively, without any carry. For instance, $(1101) + (1001) = (0100)$. For every bit x_j, two distinct 15 base value sequences are designed. One represents the value zero for x_j and the other represents the value one for x_j. For convenience, we assume that x_j^1 denotes the value of x_j to be one and x_j^0 denotes the value of x_j to be zero. The following algorithm is used for parallel adding two n bits binary number in a strand with one starts from the pth bit and the other one starts from the qth bit and appending the result from rth bit (see Algorithm 1).

Consider that $n = 3$, $p = 1$, $q = 4$, and $r = 7$. That is, two binary numbers to be added in parallel are both 3 bits, while one from the 1st bit and the other one from the 4th bit in a strand, and "append" operation starts from the 7th bit. We then suppose, tube $T_0 = \{110001, 010101, 001111, 100011\}$ which is regarded as an input tube for the algorithm ParallelAdder (T_0, n, p, q, r). Because the value of n is 3, step (1a) to step (1f) will be run 3 times. After the first execution of step (1a) is finished, $T_1 = \{110001, 100011\}$ and $T_2 = \{010101, 001111\}$. Next, after the first execution of step (1b) and step (1c) is performed, $T3 = \Phi$, $T_4 = \{110001, 100011\}$, $T_5 = \{010101, 001111\}$, and $T_6 = \Phi$, while $T_0 = T_1 = T_2 = \Phi$. After first execution of step (1d) is run, tube $T_7 = \{110001, 010101, 001111, 100011\}$ and $T8 = \Phi$. After first execution of step (1e) and step (1f) is run, $T_0 = \{1100011, 0101011, 0011111, 1000111\}$. Then, after the rest operations are performed, the result of tube T_0 is shown in Table 1.

Lemma 1. *The algorithm ParallelAdder (T_0, n, p, q, r) is applied to finish the function of a parallel adder.*

Table 1: Result of tube T_0.

Tube	The result is generated by ParallelAdder
T_0	110001111, 010101111, 001111110, 100011111

Proof. The algorithm ParallelAdder (T_0, n, p, q, r) is implemented by means of the extract, merge, and append operations. Each execution of step (1a) is used to produce two tubes T_1 and T_2, where all of the molecules of DNA in T_1 contain x^1_{p+j} and all of the molecules of DNA in T_2 contain x^0_{p+j}. Each execution of step (1b) and step (1c) is used to produce four tubes T_3, T_4, T_5, T_6, where all DNA strands in T_3 contain x^1_{p+j} and x^1_{q+j}, all DNA strands in T_4 contain x^1_{p+j} and x^0_{q+j}, all DNA strands in T_5 contain x^0_{p+j} and x^1_{q+j}, and all DNA strands in T_6 contain x^0_{p+j} and x^0_{q+j}. According to the additive theorem over $GF(2^n)$, the jth bit of the sum in T_4 and T_5 is 1 and the jth bit of the sum in T_3 and T_6 is 0.

From ParallelAdder (T_0, n, p, q, r), it takes $3n$ extract operations, $3n$ merge operations, $2n$ append operations, and 9 test tubes to finish parallel addition. A value sequence for every bit contains 15 bases. Therefore, the algorithm will add $15n$ bases to all DNA strands in tube T_0. □

3.3. The construction of a parallel multiplier over $GF(2^n)$

Over $GF(2^n)$, the multiplicative operation runs as follows:

$$\begin{aligned}
&(b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0)(b'_{n-1}\omega^{n-1} + \cdots + b'_1\omega + b'_0) \\
&\quad = b_{n-1}\omega^{n-1}(b'_{n-1}\omega^{n-1} + \cdots + b'_1\omega + b'_0) \\
&\qquad + \cdots + b_0(b'_{n-1}\omega^{n-1} + \cdots + b'_1\omega + b'_0) \\
&\quad = h_{2n-2}\omega^{2n-2} + h_{2n-3}\omega^{2n-3} + \cdots + h_2\omega^2 + h_1\omega + h_0, \\
&\qquad\qquad h_{2n-2} = b_{n-1}b'_{n-1}, \\
&\qquad\qquad h_{2n-3} = b_{n-1}b'_{n-2} + b_{n-2}b'_{n-1}, \ldots, \\
&\qquad\qquad h_{n-1} = b_{n-1}b'_0 + b_{n-2}b'_1 + \cdots + b_0b'_{n-1}, \ldots, \\
&\qquad\qquad h_1 = b_1b'_0 + b_0b'_1, \\
&\qquad\qquad h_0 = b_0b'_0.
\end{aligned} \tag{3}$$

The algorithm ParallelMultiplier (T_0, n, p, q) is used to multiply two $n$bit binary numbers on every strand in parallel with one starts from the pth bit and the other one starts from the qth bit. It runs as follows: at first, employ extract operation to form two tubes: T_1 and T_2. The first tube T_1 includes all of the strands on which $x_p = 1$ and the second tube T_2 includes all of the strands on which $x_p = 0$. Then, we copy the bits from qth to $(q + n - 1)$th to the end of every strand in tube T_1 and append n bits 0 to the end of every strand in tube T_2. After these operations, the $(q + n)$th bit to $(q + n + n - 1)$th bit show the coefficients of ω^{2n-2} to ω^{n-1}. Using the same principle, we get the coefficients of ω^{2n-3} to ω^{n-2}, ω^{2n-4} to $\omega^{n-3}, \ldots, \omega^{n-1}$ to ω^0. At last, call algorithm ParallelAdder (T_0, n, p, q, r) to compute the sum of coefficients of ω^{2n-2}, the sum of coefficient of $\omega^{2n-3}, \ldots$, and so on. As a result, the length of every strand will increase $n \times n + 2n - 1$ bits.

From ParallelMultiplier (T_0, n, p, q), it takes $O(n^2)$ extract operations, $O(n^2)$ merge operations, $O(n^2)$ append operations, and $O(1)$ test tubes to finish the function of a parallel multiplier.

3.4. The construction of a parallel shifter for multiplicative result

Because over $GF(2^n)$, the exponent of ω cannot be beyond $n - 1$, the result of parallel multiplying should be shifted by a certain irreducible polynomial, called primitive polynomial: $\omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0 = 0 \Leftrightarrow \omega^n = b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0$. The algorithm ParallelShifter (T_0, n, p), p representing that the multiplicative result starts from the pth bit, is used to parallel shift the multiplicative result $b_{2n-2}\omega^{2n-2} + b_{2n-3}\omega^{2n-3} + \cdots + b_1\omega + b_0$ to legal form $b_{n-1}\omega^{n-1} + b_{n-2}\omega^{n-2} + \cdots + b_1\omega + b_0$ over $GF(2^n)$ which can be designed as follows: appends the primitive polynomial's coefficients from ω^{n-1} to ω^0 to the end of every strand at first. Employ the extract operation to form two test tubes T_1 and T_2. Tube T_1 includes all of the strands on which $x_p = 1$ and tube T_2 includes all of the strands on which $x_p = 0$. Then, we add the coefficients, from item ω^{2n-3} to item ω^{n-2}, to the coefficients of irreducible polynomial in parallel in tube T_1. This forms the new coefficients from ω^{2n-3} to ω^{n-2} and has deleted the ω^{2n-2} item. The coefficients from ω^{n-3} to ω^0 are without any change. For the T_2 includes all of the strands that have $x_p = 0$, so just copy the coefficients from ω^{2n-3} to ω^0 without any change. After all the executions before are run, the highest exponent of ω is reduced to $2n - 3$. Then, merge T_1 and T_2 and begin new reduction. The principle of rest reducing turn is all like this above. When this algorithm is run out, the highest exponent of ω is reduced to $n - 1$. This algorithm will totally append $n \times n + (n - 1)(n - 2)/2$ bits more to every strand.

From ParallelShifter (T_0, n, p), it takes $O(n^2)$ extract operations, $O(n^2)$ merge operations, $O(n^2)$ append operations, and $O(1)$ test tubes to finish the function of a parallel shifter.

3.5. The mathematical principle of division on $GF(2^n)$

Over $GF(2^n)$, to do a division operation for a dividend and a divisor, one should get the divisor's inverse first and then multiply the dividend. For the primitive polynomial $\omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0$ is irreducible, there exists a polynomial $g(\omega)$ and a polynomial $f(\omega)$ that fit the equation (according to Euclid algorithm):

$$g(\omega) \times \text{divisor} + f(\omega) \times (\omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0) = 1. \tag{4}$$

Because $\omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0 = 0$, $g(\omega) \times \text{divisor} = 1$, which is to say $g(\omega)$ is the inverse of the divisor. To find $g(\omega)$ and $f(\omega)$, one can do as follows, which is called Euclid algorithm, also called division algorithm; first, $\omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0$ is divided by the divisor. If the value of the remainder is 1, that is to say, $(\omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0) + g(\omega) \times \text{divisor} = 1$ with $g(\omega)$ is the division result and the inverse of divisor has found which is $g(\omega)$; else let remainder be $r(\omega)$, let the divisor be the dividend and $r(\omega)$ be the divisor and do division operation again. Repeat the process until the remainder is 1. Because the highest exponent of ω of remainder reduces by 1 in each repeat, it is at most repeating $n - 1$ times. So in the first time division, the dividend is $n + 1$ bits and the divisor is n bits and the remainder is at most $n - 1$ bits, and in the second time division the dividend is n bits and the divisor is $n - 1$ bits and the remainder is at most $n - 2$ bits, $\ldots$, and in last time division the dividend is 3 bits and the divisor is 2 bits and the remainder is 1 bit. Then, trace back to get the $g(\omega)$.

```
(1) If $(q - p > k - j)$ then
  (1a) For $m = 1$ to $q - p - (k - j)$
    (1a1) $T_3 = +(T_0, x^1_{p+m-1})$
    (1a2) $T_1 = \cup(T_1, T_3)$
  EndFor
EndIf
(2) For $m = q - (k - j) - p$ to $q - p$
  (2a) $T_3 = +(T_0, x^1_{p+m})$
  (2b) $T_4 = -(T_3, x^1_{j+m-(q-p-k+j)})$
  (2c) $T_5 = -(T_0, x^1_{p+m})$
  (2d) $T_6 = +(T_5, x^1_{j+m-(q-p-k+j)})$
  (2e) $T_1 = \cup(T_1, T_4)$ and $T_2 = \cup(T_2, T_3, T_6)$ and
       $T_0 = \cup(T_0, T_5)$
EndFor
EndProcedure
```

ALGORITHM 2: Procedure ParallelComparator $(T_0, p, q, j, k, T_1, T_2)$.

3.6. The construction of a parallel comparator

Prior to each step of long division, comparison should be done first. Suppose the divisor is n bits at that time. At first compare the first two bits of dividend with divisor to determine that addition operation between first two bits of dividend and divisor should be done or not, then compare the first three bits of result with divisor, $\ldots$, compare the first n bits of result with divisor, the last time (finally), compare the last n bits of result with divisor for this time the first bit of result is 0. The following algorithm is used to compare the divisor which is from pth bit to qth bit with the bits from jth bit to kth bit in parallel, and forms tube T_1 and T_2. T_2 contains all strands on which add execution will be done and T_1 contains all strands on which add execution will not be done (see Algorithm 2).

Lemma 2. *The algorithm Parallel Comparator $(T_0, p, q, j, k, T_1, T_2)$ is applied to finish the function of parallel comparator.*

Proof. If $q - p > k - j$, that means the bits of divisor is more than the bits of the result which are intended to compare this

```
(1) For $j = 1$ to $n - 1$
    (1a) ParallelComparator $(T_0, p, p+n-1, q+(j-1)(n+2), q+(j-1)(n+2)+j, T_1, T_2)$
    (1b) Append $(T_1, x^0_{q+n+1+(j-1)(n+2)})$ and Append $(T_2, x^1_{q+n+1+(j-1)(n+2)})$
    (1c) For $k = 0$ to $n$
        (1c1) $T_3 = +(T_1, x^1_{q+(n+2)(j-1)+k})$ and $T_4 = -(T_1, x^1_{q+(n+2)(j-1)+k})$
        (1c2) Append $(T_3, x^1_{q+j(n+2)+k})$ and Append $(T_4, x^0_{q+j(n+2)+k})$
        (1c3) $T_1 = \cup(T_3, T_4)$
    EndFor
    (1d) ParallelAdder $(T_2, j+1, q+(j-1)(n+2), p+n-j-1, q+(n+2)j)$
    (1e) For $k = n - j$ down to 1
        (1e1) $T_3 = +(T_2, x^1_{q+(n+2)j-k-1})$ and $T_4 = -(T_2, x^1_{q+(n+2)j-k-1})$
        (1e2) Append $(T_3, x^1_{q+(j+1)(n+2)-k-1})$ and Append $(T_4, x^0_{q+(j+1)(n+2)-k-1})$
        (1e3) $T_2 = \cup(T_3, T_4)$
    EndFor
    (1f) $T_0 = \cup(T_1, T_2)$
EndFor
(2) ParallelComparator $(T_0, p, p+n-1, q+(n+2)(n-1)+1, q+(n+2)(n-1)+n, T_1, T_2)$
(3) Append $(T_1, x^0_{q+(n+2)n-1})$ and Append $(T_2, x^1_{q+(n+2)n-1})$
(4) For $k = 0$ to $n$
    (4a) $T_3 = +(T_1, x^1_{q+(n+2)(n-1)+k})$ and $T_4 = -(T_1, x^1_{q+(n+2)(n-1)+k})$
    (4b) Append $(T_3, x^1_{q+(n+2)n+k})$ and Append $(T_4, x^0_{q+(n+2)n+k})$
    (4c) $T_1 = \cup(T_3, T_4)$
EndFor
(5) Append $(T_2, x^0_{q+(n+2)n})$
(6) ParallelAdder $(T_2, n, q+(n+2)(n-1)+1, p, q+(n+2)n+1)$
(7) $T_0 = \cup(T_1, T_2)$
EndProcedure
```

Algorithm 3: Procedure SimilarDiv (T_0, n, p, q).

time. Step (1) considers the excessive bits of divisor and if any one bit is 1, which means the divisor is “bigger” than the bits of the result which are intended to compare this time and pour the strands to T_1. Step (2) considers the rest of the bits of divisor and the bits of result which are intended to compare this time. T_3 contains all strands on which certain bit of divisor is 1 and corresponding bit of the result is 1; T_4 contains all strands on which certain bit of divisor is 1 and corresponding bit of the result is 0; T_5 contains all strands on which certain bit of divisor is 0 and corresponding bit of the result is 0; T_6 contains all strands on which certain bit of divisor is 0 and corresponding bit of the result is 1. So add execution can be done over strands in T_3 and T_6, which can not be done over strands in T_4, and strands in T_5 need more consideration.

From ParallelComparator $(T_0, p, q, j, k, T_1, T_2)$, it takes $O(n)$ extract operations, $O(n)$ merge operations, and $O(1)$ test tubes to finish the function of a parallel comparator. □

3.7. The construction of a parallel long division

Suppose the divisor is n bits and is from pth bit and the dividend is $n+1$ bits and is from the qth bit in each strand. To do the long division, first compare the divisor with first two bits of dividend using ParallelComparator to get the first bit of the result of division and note down the result of first time addition result. Then, compare the divisor with the first three bits of the addition result last time to get the second bit of the result of the long division, and note down the addition result. Finally, compare the divisor with the last n bits of addition result last time to get the last bit of division result and the remainder (see Algorithm 3).

Lemma 3. *The algorithm SimilarDiv (T_0, n, p, q) is applied to finish the function of parallel long division.*

Proof. Each execution of step (1) is to get each bit of the long division result. The rest part is to get the last bit of the long division result. Consider the first cycle of step (1), step (1a) compares the divisor with the first two bits of the dividend, and form two tubes T_1 and T_2 that T_2 contains all strands on which add execution can be done, contrarily the T_1. Step (1b) appends 0 to all strands in T_1 and appends 1 to all strands in T_2. This bit is the first bit of the division result. Step (1c) just finishes to append the dividend in tube T_1. Step (1d) adds the divisor and the first two bits of the dividend in T_2. Step (1e) copies the rest of the bits of the dividend in T_2. Step (1f) pours T_1 and T_2 together and finishes the first execution of step (1) to get the first bit of the division result and the first time addition result. The second execution of step (1) is to compare the divisor with the first three bits of the addition result last time and get the second bit of the division result. The principle of other cycles and the rest of the steps are similar to the principle above. The length of each strand will reach to $q+n+(n+2)n$ bits when this algorithm is run out.

```
(1) ParallelMultiplier ($T_0, n, d, p$)
(2) ParallelShifter ($T_0, n, p + n + n^2$)
(3) ParallelAdder ($T_0, n, t, p + M, p + n + M$)
EndProcedure
```

Algorithm 4: Procedure Traceback (T_0, t, d, p, n).

From SimilarDiv (T_0, n, p, q), it takes $O(n^2)$ extract operations, $O(n^2)$ append operations, $O(n^2)$ merge operations, and $O(1)$ test tubes to finish the function of a parallel long division. □

3.8. *The construction of parallel traceback*

Suppose the irreducible polynomial $A(\omega) = \omega^n + b_{n-1}\omega^{n-1} + \cdots + b_1\omega + b_0$ and suppose $g(\omega)$ and $f(\omega)$ satisfy that $g(\omega) \times \text{divisor} + f(\omega)A(\omega) = 1$, for the purpose of finding the divisor's inverse, $g(\omega)$, we need to do sometimes long division introduced in Section 3.7; let $A(\omega)$ be the dividend and divisor mentioned above be the divisor in first time and suppose the result is $g_1(\omega)$, and if the remainder is 1, then the division result $g_1(\omega)$ is $g(\omega)$. Else, let the divisor last time be the dividend and let the remainder last time be the divisor and do the long division. Suppose the result is $p(\omega)$ and $g_2(\omega) = p(\omega) \times g_1(\omega) + 1$, if the remainder is 1, the $g_2(\omega)$ is $g(\omega)$. Else, let the divisor last time be the dividend and let the remainder last time be the divisor and do the long division. Suppose the result is $p(\omega)$ and $g_3(\omega) = p(\omega) \times g_2(\omega) + g_1(\omega)$, if the remainder is 1, the $g_3(\omega)$ is $g(\omega)$. Generally speaking, we need to trace back after long division each time: first time, the tracing result is the division's result; the second time, the tracing's result is the sum of 1 and the product of the division's result and the tracing's result last time; from the third time, the tracing's result is the sum of the tracing's result last second time and the product of the division's result and the tracing's result last time. The following algorithm is used to do tracing operation from the third time in which t, d, and p mean that the tracing's result last second time is from the tth bit and the last tracing's result is from the dth bit and the division result is from the pth bit (see Algorithm 4).

Lemma 4. *The algorithm TraceBack (T_0, t, d, p, n) is used to trace back after long division from the third time in order to get the divisor's inverse.*

Proof. In this and following procedures, $M = n^2 + 2n - 1 + n^2 + (n-1)(n-2)/2$, which represent the total number of increased bits when ParallelMultiplier (T_0, n, p, q) and ParallelShifter (T_0, n, p) are called. Step (1) is used to multiply the last tracing's result from dth bit to the long division result from pth bit in parallel. The result will be from the $(p + n + n \times n)$th bit to $(p + n + n \times n + 2n - 1)$th bit. Step (2) is to shift the result of multiplication to legal form which will append $((n-1)(n-2)/2 + n^2)$ bits to every strand. And its result is from $(p + M)$th bit to $(p + n + M - 1)$th bit. Step (3) is used to add the result to the last second time's tracing's result in parallel.

From TraceBack (T_0, t, d, p, n), it takes $O(n^2)$ extract operations, $O(n^2)$ append operations, $O(n^2)$ merge operations, and $O(1)$ test tubes to finish the function of tracing back. □

3.9. *The construction of a parallel inverse*

From the algorithms introduced above, we can find divisors' inverses in parallel as follows: first pick out the strands on which divisor equals to 1. Then, let the primitive polynomial be the dividend and the divisor be the divisor and do long division. Trace back to get the tracing's result first time and pick up the strands on which the remainder equals to 1 and store them in tube T_1. Then, let the divisor last time be the dividend and the remainder last time be the divisor and do long division. Collect the quotient and trace back. Pick up the strands on which the remainder equals to 1 and store them in tube $T_2, \ldots,$ these executions, including long division, collecting every bit of the quotient and tracing back, are run $n - 1$ times at most. In the following algorithm, the parameters n and p mean that the divisor is n bits and it starts from pth bit in every strand. The last parameter r is used to represent that each strand in tube is $r - 1$ bit long and we begin append operation from rth bit of every strand.

Among the algorithm, the procedure Picking (T_0, n, p, T_s) is used to pick out the strands on which the pth bit to the $(p + n - 2)$th bit are all 0 and the $(p + n - 1)$th bit is 1 and store them in T_s. It is easy to program, so just omitted here (see Algorithm 5).

Lemma 5. *The algorithm ParallelInverse (T_0, n, p, r) is applied to find inverses over GF(2^n) in parallel.*

Proof. Step (2) is used to append $n + 1$ bits irreducible polynomial, which is the dividend in first long division, to every strand. The execution of step (3) calls the algorithm SimilarDiv (T_0, n, p, r) to finish long division. Now the length of strands is added up to $r + n + (n + 2)n$ bits. Step (4) finishes the function of collecting every bit of quotient and will add n bits to every strand. This is the first time tracing result. Step (5) calls the procedure Picking (T_0, n, p, T_s) to pick out the strands on which the remainder is 1 and store them in tube T_1. Step (6) finishes the operation of appending the dividend, which is divisor last time, to the strands. Note that $s_1 = n+1+(n+2)n+n$. Step (7) calls the algorithm SimilarDiv (T_0, n, p, q) to accomplish the second time long division. Step (8) employs the append operation to append a bit 0 in order to make the quotient to be n bits. Step (9) finishes the function of collecting every bit of quotient and will add $n - 1$ bits to every strand. Steps (10) and (11) call the algorithm ParallelMultiplier (T_0, n, p, q) and ParallelShifter (T_0, n, p). Step (12) is used to add 1 to the product. These three steps accomplish the function of tracing back of the second time. Now the length of every strand is $r+s_1+n+(n+1)(n-1)+n+M+n-1$. Step (13) calls the algorithm Picking (T_0, n, p, T_s) to pick out the strands on which the remainder is 1 and store them in tube T_2. One execution of step (14) finishes the function of long division and tracing back and picking out the strands on which the remainder is 1. The step will be looped $n - 3$ times, because the long division should be done $n - 1$ times to make

```
(1) Picking ($T_0, n, p, T_z$)
(2) Append $A(x)$ to the end of all strands in $T_0$
(3) SimilarDiv ($T_0, n, p, r$)
(4) For $j = 0$ to $n - 1$
    (4a) $T_m = +(T_0, x^1_{r+n+1+(n+2)j})$ and $T_s = -(T_0, x^1_{r+n+1+(n+2)j})$
    (4b) Append ($T_m, x^1_{r+n+1+(n+2)n+j}$) and Append ($T_s, x^0_{r+n+1+(n+2)n+j}$)
    (4c) $T_0 = \cup(T_m, T_s)$
EndFor
(5) Picking ($T_0, n - 1, r + n + 1 + (n + 2)(n - 1) + 3, T_1$)
(6) For $j = 0$ to $n - 1$
    (6a) $T_m = +(T_0, x^1_{p+j})$ and $T_s = -(T_0, x^1_{p+j})$
    (6b) Append ($T_m, x^1_{r+s_1+j}$) and Append ($T_s, x^0_{r+s_1+j}$)
    (6c) $T_0 = \cup(T_m, T_s)$
EndFor
(7) SimilarDiv ($T_0, n - 1, r + n + 1 + (n + 2)(n - 1) + 3, r + s_1$)
(8) Append ($T_0, x^0_{r+s_1+n+(n+1)(n-1)}$)
(9) For $j = 0$ to $n - 2$
    (9a) $T_m = +(T_0, x^1_{r+s_1+n+(n+1)j})$ and $T_s = -(T_0, x^1_{r+s_1+n+(n+1)j})$
    (9b) Append ($T_m, x^1_{r+s_1+n+(n+1)(n-1)+1+j}$) and Append ($T_s, x^0_{r+s_1+n+(n+1)(n-1)+1+j}$)
    (9c) $T_0 = \cup(T_m, T_s)$
EndFor
(10) ParallelMultiplier ($T_0, n, r + n + 1 + (n + 2)n, r + s_1 + n + (n + 1)(n - 1)$)
(11) ParallelShifter ($T_0, n, r + s_1 + n + (n + 1)(n - 1) + n + n^2$)
(12) Add 1 to the product above which will result in $n$ bits more to each strand
(13) Picking ($T_0, n - 2, r + s_1 + n + (n + 1)(n - 2) + 3, T_2$)
(14) For $j = 2$ to $n - 2$
    (14a) Copy dividend (divisor last time) to the end
    (14b) SimilarDiv ($T_0, n - j, r + s_1 + \cdots + s_{j-1} + n + 2 - j + (n + 3 - j)(n - j) + 3, r + s_1 + \cdots + s_j$)
    (14c) Append $j$ bits 0 to the end
    (14d) Collect $n - j$ bits quotient of this division to the end
    (14e) Traceback ($T_0, r + s_1 + \cdots + s_{j-1} - n, r + s_1 + \cdots + s_j - n, r + s_1 + \cdots + s_j + (n + 1 - j) + (n + 2 - j)(n - j), n$)
    (14f) Picking ($T_0, n - 1 - j, r + s_1 + \cdots + s_j + n + 1 - j + (n + 2 - j)(n - 1 - j) + 3, T_{j+1}$)
EndFor
(15) For $j = 1$ to $n - 1$
    (15a) For $k = 0$ to $s_{j+1} + \cdots + s_{n-1} - 1$
        (15a1) Append ($T_j, x^0_{r+s_1+\cdots+s_j+k}$)
    EndFor
    (15b) For $k = 0$ to $n - 1$
        (15b1) $T_m = +(T_j, x^1_{r+s_1+\cdots+s_j-n+k})$ and $T_s = -(T_j, x^1_{r+s_1+\cdots+s_j-n+k})$
        (15b2) Append ($T_m, x^1_{r+s_1+\cdots+s_{n-1}+k}$) and Append ($T_s, x^0_{r+s_1+\cdots+s_{n-1}+k}$)
        (15b3) $T_j = \cup(T_m, T_s)$
    EndFor
EndFor
(16) For $j = 0$ to $s_1 + \cdots + s_{n-1} + n - 2$
    Append ($T_z, x^0_{r+j}$)
EndFor
(17) Append ($T_z, x^1_{r+s_1+\cdots+s_{n-1}+n-1}$)
(18) $T_0 = \cup(T_z, T_1, T_2, \ldots, T_{n-1})$
EndProcedure
```

Algorithm 5: Procedure ParallelInverse (T_0, n, p, r).

sure that the remainder of each strand equals to 1 and long division has been done twice before. Note that while $j \geq 2$, $s_j = n+2-j+(n+3-j)(n+1-j)+n+M+n$. Step (15a) is used to append certain bits 0 to each tube T_j to make sure that strands in T_1 to T_{n-1} are all $r+s_1+\cdots+s_{n-1}-1$ bits long. Step (15b) is used to append the inverse gotten before to the last of every strand in T_1 to T_{n-1}. Step (18) pours strands in all tubes together to one tube T_0. From all above, getting inverse one time needs increasing $s_1+s_2+\cdots+s_{n-1}+n$ bits to every strand.

From ParallelInverse (T_0, n, p, r), it takes $O(n^3)$ extract operations, $O(n^3)$ append operations, $O(n^3)$ merge operations, and $O(n)$ test tubes to finish the function of finding inverse in parallel. □

```
(1) ParallelInverse ($T_0, n, p, r$)
(2) ParallelMultiplier ($T_0, n, q, r + s_1 + \cdots + s_{n-1}$)
(3) ParallelShifter ($T_0, n, r + s_1 + \cdots + s_{n-1} + n + n \times n$)
EndProcedure
```

Algorithm 6: Procedure ParallelDivision (T_0, n, p, q, r).

3.10. The construction of a parallel divider

To do a division operation on $GF(2^n)$, first one should calculate divisor's inverse using above mentioned algorithm then multiply the dividend. The following algorithm is used to finish the function of parallel division over $GF(2^n)$, where the dividend begins with the qth bit and the divisor begins with the pth bit, and current bit is the rth bit (see Algorithm 6).

Lemma 6. *The algorithm ParallelDivision (T_0, n, p, q, r) is used to finish the function of parallel division over $GF(2^n)$.*

Proof. Step (1) calls the algorithm ParallelInverse (T_0, n, p, r) to get the divisor's inverse of every strand. The length of every strand is $r+s_1+\cdots+s_{n-1}+n-1$ bits now with the inverse from $(r+s_1+\cdots+s_{n-1})$th bit to $(r+s_1+\cdots+s_{n-1}+n-1)$th bit. Step (2) calls ParallelMultiplier (T_0, n, p, q) to finish the function of the inverse being multiplied by the dividend every strand with the dividend starting from the qth bit. Now the length of each strand adds up to $r+s_1+\cdots+s_{n-1}+n+n\times n+2n-1-1$ bits. Step (3) shifts the product by calling ParallelShifter (T_0, n, p) and will add $(n-1)(n-2)/2 + n \times n$ bits to every strand. Generally speaking, doing parallel division one time need increasing the strands $D = s_1 + \cdots + s_{n-1} + n + M$ bits.

From ParallelDivision (T_0, n, p, q, r), it takes $O(n^3)$ extract operations, $O(n^3)$ append operations, $O(n^3)$ merge operations, and $O(n)$ test tubes to finish the function of parallel division. □

3.11. The construction of a parallel adder of two points on elliptic curve

By far the addition, subtraction (the same to addition), multiplication, and division operations over $GF(2^n)$ have been solved. Now consider how to execute addition of two points on an elliptic curve $y^2 + xy = x^3 + ax^2 + b$ in biological ways. We should consider five different cases: case 1, the first point is O and the point of sum equals to the second point; case 2, the second point is O and the point of sum equals to the first point; case 3, one point is the inverse of the other one, and the point of sum is O; case 4, one point equals to the other one, and computes the sum as the formula in part 3.1; case 5, computes the sum using the formula in part 3.1.

The algorithm AddTwoNode ($T_0, n, x1, y1, x2, y2, r$) is used to add two points. The first one's position x starts from the $(x1)$th bit and position y starts from the $(y1)$th bit, and the second one's position x starts from the $(x2)$th bit and y starts from the $(y2)$th bit. The parameter r represents the current bit. In the procedure, it calls Picking01 ($T_0, n, x1, y1, x2, y2, T_{11}$) to pick out the strands on which the first point is O and store them in T_{11}, Picking02 ($T_0, n, x1, y1, x2, y2, T_{12}$) to pick out the strands on which the second point is O and store them in T_{12}, PickingInverse ($T_0, n, x1, y1, x2, y2, r, T_2$) to pick out the strands on which one point is the inverse of the other one and store them in tube T_2, and PickingEqual ($T_0, n, x1, y1, x2, y2, T_3$) to pick out the strands on which one point equals to the other one and store them in tube T_3. These four algorithms are easy to design, so omitted here. Note that PickingInverse ($T_0, n, x1, y1, x2, y2, r, T_2$) will increase n bits to every strand in T_0 (see Algorithm 7).

Lemma 7. *The algorithm AddTwoNode ($T_0, n, x1, y1, x2, y2, r$) can compute the sum of two points on elliptic curve.*

Proof. Step (5) employs append operation to append M bits 0 to all strands in tube T_0. Step (6) to step (8) are operations on tube T_0 to get μ of strands in T_0. Step (9) to step (13) are operations on tube T_3 to get μ of strands in T_3. Step (14) pours T_0 and T_3 to T_0 and the length of every strand in T_0 now is $r + n + n + M + n + D - 1$ bits. Step (15) to step (21) accomplish to compute the position x of point of sum and the result is from $(r + X)$th bit. The execution of steps (22) to (26) is used to get the position y of the sum which equals to $\mu(x1 + x3) + x3 + y1$. And now, the position y of the sum is from $(r + X + Y)$th bit and the length is $r + n + X + Y - 1$ bits.

Steps (27) and (28) are applied to append the value of positions x and y of the sum of two points to the last of every strand. Now, the length of every strand in T_0 is $r + n + X + Y + 2n - 1$ bits with the value x from the $(r + n + X + Y)$th bit and y from the $(r + n + X + Y + n)$th bit.

From AddTwoNode ($T_0, n, x1, y1, x2, y2, r$), it takes $O(n^3)$ extract operations, $O(n^3)$ append operations, $O(n^3)$ merge operations, and $O(n)$ test tubes to finish the function of a parallel adder for points on elliptic curve. □

3.12. Breaking the elliptic curve cryptosystem

We have constructed the algorithm above for parallel computing the point of the sum of two points. Then, we can solve elliptic curve discrete logarithm problem as follows: consider point P and Q are given, and l is what we want to get which matches $Q = lP$. First, we amplify P into two tubes and add P in one tube. Check if $2P$ equals to Q; if not, note down the value of $2P$ and pour two tubes together. Then, amplify the tube into two tubes and add $2P$ in one tube. Check if any point equals to Q; if not, note down the value of $4P$ and pour two tubes together, or we get the value of l. Then, amplify the tube into two tubes and add $4P$ in one tube, . . . , while this loop executes n times, the value from 1 to 2^n for l will have been checked, and the elliptic curve cryptosystem has been broken by the solved elliptic curve discrete logarithm problem.

4. CONCLUSION

This paper is the first effort in literature that demonstrates that the difficult problem for elliptic curve discrete logarithm can be solved on a DNA-based computer. While Chang's

```
(1) Picking01 $(T_0, n, x1, y1, x2, y2, T_{11})$
(2) Picking02 $(T_0, n, x1, y1, x2, y2, T_{12})$
(3) PickingInverse $(T_0, n, x1, y1, x2, y2, r, T_2)$
(4) PickingEqual $(T_0, n, x1, y1, x2, y2, T_3)$
(5) For $j = 0$ to $M - 1$
    (5a) Append $(T_0, x^0_{r+n+j})$
EndFor
(6) ParallelAdder $(T_0, n, y1, y2, r + n + M)$
(7) ParallelAdder $(T_0, n, x1, x2, r + n + M + n)$
(8) ParallelDivision $(T_0, n, r + n + M + n, r + n + M, r + n + M + 2n)$
(9) For $j = 0$ to $n - 1$
    (9a) $T_m = +(T_3, x^1_{x1+j})$ and $T_s = -(T_3, x^1_{x1+j})$
    (9b) Append $(T_m, x^1_{r+n+j})$ and Append $(T_s, x^0_{r+n+j})$
    (9c) $T_3 = \cup(T_m, T_s)$
EndFor
(10) ParallelMultiplier $(T_3, n, r + n, r + n)$
(11) ParallelShifter $(T_3, n, r + n + n + n \times n)$
(12) ParallelAdder $(T_3, n, y1, r + n + n + M - n, r + n + n + M)$
(13) ParallelDivision $(T_3, n, x1, r + n + n + M, r + n + n + M + n)$
(14) $T_0 = \cup(T_0, T_3)$
SUPPOSE $U = n + M + n + D$
(15) ParallelMultiplier $(T_0, n, r + n + U - n, r + n + U - n)$
(16) ParallelShifter $(T_0, n, r + n + U + n \times n)$
(17) ParallelAdder $(T_0, n, r + n + U - n, r + n + U + M - n, r + n + U + M)$
(18) ParallelAdder $(T_0, n, r + n + U + M, x1, r + n + U + M + n)$
(19) ParallelAdder $(T_0, n, r + n + U + M + n, x2, r + n + U + M + n + n)$
(20) For $j = 0$ to $n - 1$
    (20a) Append $(T_0, x_{a,r+n+U+M+3n+j})$
EndFor
(21) ParallelAdder $(T_0, n, r + n + U + M + 2n, r + n + U + M + 3n, r + n + U + M + 4n)$
SUPPOSE $X = U + M + 5n$
(22) ParallelAdder $(T_0, n, x1, r + n + X - n, r + n + X)$
(23) ParallelMultiplier $(T_0, n, r + n + U - n, r + n + X)$
(24) ParallelShifter $(T_0, n, r + n + X + n + n \times n)$
(25) ParallelAdder $(T_0, n, r + n + X + n + M - n, r + n + X - n, r + n + X + n + M)$
(26) ParallelAdder $(T_0, n, r + n + X + n + M, y1, r + n + X + n + M + n)$
SUPPOSE $Y = n + M + n + n$
(27) For $j = 0$ to $n - 1$
    (27a) $T_m = +(T_0, x^1_{r+X+j})$ and $T_s = -(T_0, x^1_{r+X+j})$
    (27b) Append $(T_m, x^1_{r+n+X+Y+j})$ and Append $(T_s, x^0_{r+n+X+Y+j})$
    (27c) $T_0 = \cup(T_m, T_s)$
EndFor
(28) For $j = 0$ to $n - 1$
    (28a) $T_m = +(T_0, x^1_{r+n+X+Y-n+j})$ and $T_s = -(T_0, x^1_{r+n+X+Y-n+j})$
    (28b) Append $(T_m, x^1_{r+n+X+Y+n+j})$ and Append $(T_s, x^0_{r+n+X+Y+n+j})$
    (28c) $T_0 = \cup(T_m, T_s)$
EndFor
(29) Append $n + X + Y$ bits 0 to each strand in $T_{11}$ and $T_{12}$. Then, append values of $x2$ and $y2$ of each strand to
        the end in $T_{11}$, and append values of $x1$ and $y1$ of each strand to the end in $T_{12}$
(30) Append $X + Y + 2n$ bits 0 to every strand in $T_2$
(31) $T_0 = \cup(T_{11}, T_{12}, T_2, T_0)$
EndProcedure
```

ALGORITHM 7: Procedure AddTwoNode $(T_0, n, x1, y1, x2, y2, r)$.

work makes great progress in application of DNA computing in cryptoanalysis [16], which is breaking RSA by factoring integer, this paper proposes application of DNA computing in another popular cryptosystem, ECC, which is more complex and has more challenge in cryptoanalysis. Though the algorithm is somewhat complex, it takes a series of steps that is polynomial in the input size, so it is feasible in theory and inspirits the development of DNA computing. Simultaneously, the paper also shows that humans' complex mathematical operations can directly be performed with basic biological operations. The property for the difficulty of elliptic curve discrete logarithm is the basis of elliptic curve cryptosystems. However, this property seems to be incorrect on a molecular computer. This indicates that the elliptic curve cryptosystems are perhaps insecure if the technique of DNA computing is skillful enough to run the algorithms efficiently as discussed in this paper.

ACKNOWLEDGMENTS

The authors thank Daniel Howard (QinetiQ, Malvern Technology Centre, UK) for his great efforts in correcting this paper. This work is supported by National Natural Science Foundation (60603053) of China.

REFERENCES

[1] L. M. Adleman, "Molecular computation of solutions to combinatorial problems," *Science*, vol. 266, no. 5187, pp. 1021–1024, 1994.

[2] E. Csuhaj-Varjú, L. Kari, and G. Paun, "Test tube distributed systems based on splicing," *Computers and Artificial Intelligence*, vol. 15, no. 2-3, pp. 211–232, 1996.

[3] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[4] N. Koblizt, *Introduction to Elliptic Curves and Modular Forms*, Springer, New York, NY, USA, 1984.

[5] S. Lang, *Elliptic Curves: Diophantine Analysis*, Springer, New York, NY, USA, 1978.

[6] V. S. Miller, "Use of elliptic curves in cryptography," in *Proceedings of the 5th Annual International Cryptology Conference (CRYPTO '85)*, Santa Barbara, Calif, USA, August 1985.

[7] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance," in *Proceedings of the 2nd Workshop on Advances in Cryptology: Theory and Application of Cryptographic Techniques (EUROCRYPT '84)*, pp. 224–314, Springer, Paris, France, April 1985.

[8] R. P. Feynman, "There's plenty of room at the bottom," in *Minaturization*, D. H. Gilbert, Ed., pp. 282–296, Reinhold, New York, NY, USA, 1961.

[9] R. J. Lipton, "DNA solution of hard computational problems," *Science*, vol. 268, no. 5210, pp. 542–545, 1995.

[10] R. R. Sinden, *DNA Structure and Function*, Academic Press, New York, NY, USA, 1994.

[11] G. Paun, G. Rozenberg, and A. Salomaa, *DNA Computing: New Computing Paradigms*, Springer, New York, NY, USA, 1998.

[12] W.-L. Chang, M. S.-H. Ho, and M. Guo, "Molecular solutions for the subset-sum problem on DNA-based supercomputing," *Biosystems*, vol. 73, no. 2, pp. 117–130, 2004.

[13] M. Guo, M. S.-H. Ho, and W.-L. Chang, "Fast parallel molecular solution to the dominating-set problem on massively parallel bio-computing," *Parallel Computing*, vol. 30, no. 9-10, pp. 1109–1125, 2004.

[14] M. S.-H. Ho, "Fast parallel molecular solutions for DNA-based supercomputing: the subset-product problem," *Biosystems*, vol. 80, no. 3, pp. 233–250, 2005.

[15] D. Boneh, C. Dunworth, and R. J. Lipton, "Breaking DES using a molecular computer," Tech. Rep. CS-TR-489-95, Princeton University, Princeton, NJ, USA, 1995.

[16] W.-L. Chang, M. Guo, and M. S.-H. Ho, "Fast parallel molecular algorithms for DNA-based computation: factoring integers," *IEEE Transactions on Nanobioscience*, vol. 4, no. 2, pp. 149–163, 2005.

[17] J. Watson, M. Gilman, J. Witkowski, and M. Zoller, *Recombinant DNA*, Freeman, San Francisco, Calif, USA, 2nd edition, 1992.

[18] F. Eckstein, *Oligonucleotides and Anologues*, Oxford University Press, Oxford, UK, 1991.

[19] J. Watson, N. Hoplins, J. Roberts, J. Steitz, and A. Weiner, *Molecular Biology of the Gene*, Benjamin/Cummings, Menlo Park, Calif, USA, 1987.

[20] G. M. Blackburn and M. J. Gait, *Nucleic Acids in Chemistry and Biology*, IRL Press, Washington, DC, USA, 1990.

[21] M. Wiener and R. Zuccherato, "Faster attacks on elliptic curve cryptosystems," in *Selected Areas in Cryptography*, vol. 1556 of *Lecture Notes in Computer Science*, pp. 190–200, Springer, New York, NY, USA, 1999.