Fast Parallel DNA-Based Algorithms for Molecular Computation: Quadratic Congruence and Factoring Integers

Weng –Long Chang

***Abstract*—Assume that n is a positive integer. If there is an integer $0 < M < n$ such that $M^2 \equiv C \pmod{n}$, i.e., the congruence has a solution, then C is said to be a quadratic congruence $\pmod{n}$. If the congruence does not have a solution, then C is said to be a quadratic noncongruence $\pmod{n}$. The task of solving the problem is central to many important applications, the most obvious being cryptography. In this article, we describe a DNA-based algorithm for solving quadratic congruence and factoring integers. In additional to this novel contribution, we also show the utility of our encoding scheme, and of the algorithm's submodules. We demonstrate how a variety of arithmetic, shifted and comparative operations, namely bitwise and full addition, subtraction, left shifter and comparison perhaps are performed using strands of DNA.**

***Index Terms*—Biological cryptography, biological parallel computing, DNA-based supercomputing, factoring integers, molecular-based supercomputing, quadratic congruence, the RSA public-key cryptosystem.**

I. Introduction

THIS PAPER IS organized as follows: in Section II we introduce DNA models of computation proposed by Adleman and his coauthors in detail. In Section III we give a high-level description of our quadratic congruence algorithm. By breaking this down into submodules in Section IV, we prove the operation of the various novel algorithms for arithmetic, shifted, and comparative operations. In Section V, based on our quadratic congruence algorithm, we also give a high-level description of our factoring integer algorithm. In Section VI we demonstrate that the time complexity of our algorithm is square on the input size. In Section VII, we prove that our proposed algorithm is currently the fastest method to factor integers, and we conclude with a brief discussion in Section VIII.

II. Background

In this section we present the basic structure of the DNA molecule, and the techniques for dealing with DNA that will be used to solve quadratic congruence and factoring integers.

Manuscript received April 15, 2010; revised June 30, 2011; accepted August 01, 2011. Date of publication September 12, 2011; date of current version March 13, 2012. This work was partly supported by the National Science Foundation of Republic of China under Grants No. 99-2221-E-151-030- and 99-2622-E-151-021-CC3.

The author is with Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences, Kaohsiung City 807-78, Taiwan (e-mail: changwl@cc.kuas.edu.tw).

Digital Object Identifier 10.1109/TNB.2011.2167757

A. The Structure of DNA

From [1], [2], DNA (*DeoxyriboNucleic Acid*) is the *molecule* that plays the main role in DNA based computing. Each *deoxyribonucleotide* contains three components: a *sugar*, a *phosphate* group, and a *nitrogenous* base. The sugar has five carbon atoms, and the carbons of the sugar are numbered from 1' to 5'. The phosphate group is attached to the 5' carbon, and the base is attached to the 1' carbon. Within the sugar structure there is a *hydroxyl* group attached to the 3' carbon. As stated in [4], the base is made of one of four distinct nucleotides, which are *adenine*, *guanine*, *cytosine* and *thymine* that are, respectively abbreviated A, G, C, and T. Because nucleotides are distinguished solely from their bases, they are simply represented as A, G, C, or T nucleotides, depending upon the sort of base that they have.

B. Adleman's Experiment for Solution of a Satisfability Problem

Adleman *et al.* [6], [7] performed experiments that were applied to respectively solve a 6-variable 11-clause formula and a 20-variable 24-clause 3-conjunctive normal form (3-CNF) formula. A Lipton encoding was used to represent all possible variable assignments for the chosen 6-variable or 20-variable **SAT** problem. For each of the 6 variables $x_1, \ldots, x_6$ two distinct 15 base value sequences were designed. One represents *true* (T), x_k^T, and another represents *false* (F), x_k^F for $1 \leq k \leq 6$. Each of the 2^6 truth assignments was represented by a *library sequence* of 90 bases consisting of the concatenation of one value sequence for each variable. DNA molecules with library sequences are termed *library strands* and a combinatorial pool containing library strands is termed a *library*. The 6-variable library strands were synthesized by employing a mix-and-split combinatorial synthesis technique [6], [7]. The library strands were assigned library sequences with x_1 at the 5'-end and x_6 at the 3'-end ($5' - x_1 - x_2 - x_3 - x_4 - x_5 - x_6 - 3'$). Thus synthesis began by assembling the two 15 base oligonucleotides with sequences x_6^T and x_6^F. This process was repeated until all 6 variables had been treated. The similar method also is applied to solve a 20-variable of 3-SAT [7]. (For more discussions of the relevant biological technologies refer to [6], [7]).

C. DNA Manipulations

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], [2]:

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. *Detect*. Given a tube P, if P includes at least one DNA molecule we have "yes," and if P contains no DNA molecule we have "no."
4. *Discard*. Given a tube P, the operation will discard P.
5. *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.
6. *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.
7. *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.
8. *Read*. Given a tube P, the operation is used to describe a single molecule, which is contained in tube P. Even if P contains many different molecules each encoding a different set of bases, the operation can give an explicit description of exactly one of them.

III. Quadratic Congruence Algorithm

Given a well-defined notion of the remainder one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(d \bmod n) = (b \bmod n)$, we write $d \equiv b \pmod{n}$ and say that d is equivalent to b, modulo n. The integer can be divided into n equivalence classes according to their remainders modulo n. The equivalence class modulo n containing an integer d is $[d]_n = \{dr\}$. The set of all such equivalence classes is $\boldsymbol{Z}_n = \{[d]_n : 0 \leq d \leq n-1\}$. One often sees the definition $\boldsymbol{Z}_n = \{0, 1, \ldots n-1\}$ [3]. The *greatest common divisor* of two integers d and n, not both zero, is the largest of the common divisors of d and n; it is denoted $\gcd(d, n)$. Because the equivalence class of two integers uniquely determines the equivalence class of their product, thus, we define multiplication modulo n, denoted $*_n$, as follows: $[d]_n *_n [h]_n = [d * h]_n$. Using the definition of multiplication modulo n, we define the multiplicative group modulo n as $(\boldsymbol{Z}_n^*, *_n)$, where $\boldsymbol{Z}_n^* = \{[d]_n \in \boldsymbol{Z}_n : \gcd(d, n) = 1\}$.

Assume that the length of M is k bits. Also suppose that M is represented as a k-bit binary number, $m_{k-1} \ldots m_0$, where the value of each bit m_j is either 1 or 0 for $0 \leq j \leq k-1$. The bits m_{k-1} and m_0 represent the most significant bit and the least significant bit for M, respectively. Therefore, the form of an expression, $M^2 \pmod{n}$, can be transformed into another form

$$(3-1) : (\ldots((M * m_{k-1}) \bmod n) * 2) \bmod n)$$
$$+(((M * m_{k-2}) \bmod n) * 2) \bmod n) + \cdots + ((M * m_0) \bmod n).$$

The following pseudo algorithm is applied to solve quadratic congruence and factoring integers.

1) Method 1: Solving quadratic congruence and factoring integers.

(1) Every computation of $M^2 \pmod{n}$ for $0 < M < n$ is simultaneously performed on a molecular computer.
(2) Find four solutions that are, respectively, x, $n - x$, y, and $n - y$ for $M^2 \equiv 1 \pmod{n}$.
(3) The integer n can be factored as $p*q$, where $p = \gcd(x - y, n)$ and $q = \gcd(x + y, n)$.

EndMethod

Proof: Step (1) in Method 1 is employed to simultaneously perform every computation of $M^2 \pmod{n}$ for for $0 < M < n$. This indicates that every value of $M^2 \pmod{n}$ for for $0 < M < n$ is determined after Step (1) is finished. Then, Step (2) in Method 1 is used to search four integer solutions so that M^2 is equivalent to 1, modulo n. Next, from four solutions of $M^2 \equiv 1 \pmod{n}$, Step (4) is used to factor the integer n into $p * q$. Therefore, it is inferred from Method 1 that quadratic congruence and factoring integers can both be solved. ■

The following DNA algorithm is applied to figure out solution space of quadratic congruence that is to perform Step (1) in Method 1.

Algorithm 3–1: Figure out solution space of quadratic congruence.

(1) $\mathbf{Init}(T_0)$.
(2) $\mathbf{SelectQuadraticCongruence}(T_0, T_\theta)$.
(3) $\mathbf{ModularValue}(T_n)$.
(4) $\mathbf{ModularMultiplication}(T_0, T_n)$.

EndAlgorithm

Theorem 3–1: From **Algorithm 3–1**, the problem of quadratic congruence can be solved.

Proof: On the execution of Step (1), it calls $\mathbf{Init}(T_0)$ to construct library sequences for 2^k possible solutions for quadratic congruence. This means that tube T_0 includes library sequences encoding 2^k possible solutions for quadratic congruence. Next, the execution of Step (2) calls $\mathbf{SelectQuadraticCongruence}(T_0, T_\theta)$ to perform selection of legal solutions for quadratic congruence. This implies that legal solutions for quadratic congruence are encoded in tube T_0. On the execution of Step (3), it calls $\mathbf{ModularValue}(T_n)$ to encode n. This indicates that tube T_n contains a library sequence encoding n. Next, the execution of Step (4) calls $\mathbf{ModularMultiplication}(T_0, T_n)$ to finish computation of $M^2 \pmod{n}$. After those steps are processed, every library sequence in tube T_0 performs computation of $M^2 \pmod{n}$. Therefore, solutions of quadratic congruence can be computed from those steps in **Algorithm 3–1**. ■

Algorithm Modules

We now describe, in detail, the various modules that are combined to form the overall quadratic congruence algorithm.

A. A Library for Solving Quadratic Congruence

From [1], [2], for every bit m_j in M denoted in Section III, two *distinct* 15 base value sequences are designed to respectively represent the value "0" for m_j and the value "1" for m_j. Assume that m_j^1 denotes the value of m_j to be 1 and m_j^0 defines the value of m_j to be 0. Each of the 2^k different values for M was represented by a *library sequence* of $(15 * k)$ bases consisting of the concatenation of one value sequence for each bit.

Library sequences are also termed *library strands* and a combinatorial pool containing library strands is termed a *library*. The following procedure is used to construct a library to solve quadratic congruence.

Procedure $\mathbf{Init}(T_0)$

(1) **For** $j = 0$ **to** $k - 1$
 (1a) $\mathrm{Amplify}(T_0, T_1, T_2)$.
 (1b) $\mathrm{Append\text{-}head}(T_1, m_j^1)$.
 (1c) $\mathrm{Append\text{-}head}(T_2, m_j^0)$.
 (1d) $T_0 = \cup(T_1, T_2)$.
 EndFor
EndProcedure

Lemma 4–1: A library for solving quadratic congruence can be constructed from $\mathbf{Init}(T_0)$.

B. Selection of a Library for Solving Quadratic Congruence

Because the largest element in $\boldsymbol{Z}_n^*$ is equal to $n - 1$, suppose that $n - 1$ is represented as a k-bit binary number, $\theta_{k-1} \ldots \theta_0$, where the value of each bit θ_j is either 1 or 0 for $0 \leq j \leq k - 1$. The bits θ_{k-1} and θ_0 is used to represent the most significant bit and the least significant bit for $n - 1$, respectively. From [1], [2], for every bit θ_j, two *distinct* 15 base value sequences are designed to respectively represent the value "0" for θ_j and the value "1" for θ_j. Assume that θ_j^1 denotes the value of θ_j to be 1 and θ_j^0 defines the value of θ_j to be 0. The following algorithm, $\mathbf{SelectQuadraticCongruence}(T_0, T_\theta)$, is proposed to construct a library sequence for encoding $n - 1$ and select library strands encoding those values which ranges are from 0 through $n - 1$ from tube T_0, generated by the algorithm $\mathbf{Init}(T_0)$.

Procedure $\mathbf{SelectQuadraticCongruence}(T_0, T_\theta)$

(1) **For** $j = 0$ **to** $k - 1$
 (1a) $\mathrm{Append\text{-}head}(T_\theta, \theta_j)$.
EndFor
(2) **For** $j = k - 1$ **to** 0
 (2a) $T_0^{ON} = +(T_0, m_j^1)$ and $T_0^{OFF} = -(T_0, m_j^1)$.
 (2b) $T_\theta^{ON} = +(T_\theta, \theta_j^1)$ and $T_\theta^{OFF} = -(T_\theta, \theta_j^1)$.
 (2c) **If** $(\mathrm{Detect}(T_\theta^{ON}) == \text{“}yes\text{”})$ **then**
 (2d) $T_0^{=} = \cup(T_0^{=}, T_0^{ON})$ and $T_0^{<} = \cup(T_0^{<}, T_0^{OFF})$.
  Else
 (2e) $T_0^{>} = \cup(T_0^{>}, T_0^{ON})$ and $T_0^{=} = \cup(T_0^{=}, T_0^{OFF})$.
  EndIf
 (2f) $T_\theta = \cup(T_\theta^{ON}, T_\theta^{OFF})$.
 (2g) $T_0 = \cup(T_0, T_0^{=})$.
 (2h) $\mathrm{Discard}(T_0^{>})$.
 EndFor
(3) $T_0 = \cup(T_0, T_0^{<})$.
EndProcedure

Lemma 4–2: The algorithm $\mathbf{SelectQuadraticCongruence}(T_0, T_\theta)$ can be applied to encode $n - 1$ and select library strands encoding those values which ranges are from 0 through $n - 1$ from tube T_0, generated by the algorithm $\mathbf{Init}(T_0)$.

C. A Library Sequence for the Second Operand of a Modular Operation

Assume that the length of n denoted in Section III is k bits. Also suppose that n is represented as a k-bit binary number, $n_{k-1} \ldots n_0$, where the value of each bit n_j is either 1 or 0 for $0 \leq j \leq k - 1$. The bits n_{k-1} and n_0 represent the most significant bit and the least significant bit for n, respectively. From [1], [2], for every bit n_j, two *distinct* 15 base value sequences are designed to respectively represent the value "0" for n_j and the value "1" for n_j. Assume that n_j^1 denotes the value of n_j to be 1 and n_j^0 defines the value of n_j to be 0. The following algorithm $\mathbf{ModularValue}(T_n)$, is proposed to construct a library sequence for encoding n.

(1) **For** $j = 0$ **to** $k - 1$
 (1a) $\mathrm{Append\text{-}head}(T_n, n_j)$.
EndFor
EndProcedure

Lemma 4–3: A library sequence for encoding n can be constructed from $\mathbf{ModularValue}(T_n)$.

D. The Algorithm for Computation of a Modular Multiplication

For any positive integer M, Blakley [4] proposed the fastest method to perform computation of (3–1) denoted in Section III for $(M * M) \pmod{n}$. Blakley's algorithm is described below.

Blakley's algorithm: Perform computation of $(M * M) \pmod{n}$.

Input: Two positive integers M and n.
Output: The answer of $(M * M) \pmod{n}$, Y.
Method:
(1) $Y = 0$
(2) **For** $j = k - 1$ **down to** 0
 (2a) $Y = Y * 2$
 (2b) **If** $(Y \geq n)$ **then**
  (2c) $Y = Y - n$
  EndIf
 (2d) **If** $(m_j == 1)$ **then**
 (2e) $Y = Y + M$
 (2f) **If** $(Y \geq n)$ **then**
 (2g) $Y = Y - n$
   EndIf
  EndIf
 EndFor
EndAlgorithm

From Blakley's algorithm, it is indicated that adder and subtractor of at most $(4 * k)$ times are applied to perform computation of $(M * M) \pmod{n}$. From Blakley's method, Y is finally obtained after at most updating $(4 * k + 1)$ times of the value for Y. Assume that the length of Y is k bits. Also suppose that Y is represented as a k-bit binary number, $y_{f,k-1} \ldots y_{f,0}$, where the value of each bit $y_{f,g}$ is either 1 or 0 for $1 \le f \le (4 * k + 1)$ and $0 \le g \le k - 1$. The bits, $y_{f,k-1}$ and $y_{f,0}$, represent the most significant bit and the least significant bit for Y, respectively. If updating of the f^{th} time for Y is finished through an adder, then two binary numbers $y_{f,k-1} \ldots y_{f,0}$ and $y_{f+1,k-1} \ldots y_{f+1,0}$ represent the augends and the sum of the f^{th} updating, respectively. If updating of the f^{th} time for Y is finished through a subtractor, then two binary numbers $y_{f,k-1} \ldots y_{f,0}$ and $y_{f+1,k-1} \ldots y_{f+1,0}$ represent the minuend and the difference of the f^{th} updating, respectively.

From [1], [2], for every bit $y_{f,g}$, two *distinct* 15 base value sequences were designed to respectively represent the value "0" for $y_{f,g}$ and the value "1" for $y_{f,g}$. Assume that $y^1_{f,g}$ denotes the value of $y_{f,g}$ to be 1 and $y^0_{f,g}$ defines the value of $y_{f,g}$ to be 0.

In Blakley's algorithm, it uses successive operations of left shifter, subtraction and addition to perform computation of $(M * M) \pmod n$. The procedure, $\mathbf{ModularMultiplication}(T_0, T_n)$, is applied to perform all of the steps to computation of $(M * M) \pmod n$. This implies that each step in Blakley's algorithm is performed through the procedure, $\mathbf{ModularMultiplication}(T_0, T_n)$.

Procedure $\mathbf{ModularMultiplication}(T_0, T_n)$

(1) $\mathbf{InitialValue}(T_0)$.
(2) **For** $j = k - 1$ **down to** 0
 (2a) $\mathbf{ParallelLeftShifter}(T_0, (k - 1 - j) * 4 + 1)$.
 (2b) $\mathbf{ParallelComparator}(T_0, T_n, T_0^>, T_0^=, T_0^<, (k - 1 - j) * 4 + 2)$.
 (2c) $T_0 = \cup(T_0^>, T_0^=)$.
 (2c1) **If** $(\mathrm{Detect}(T_0) ==$ "yes"$)$ **then**
 (2d) $\mathbf{BinaryParallelSubtractor}(T_0, (k - 1 - j) * 4 + 2)$.
 EndIf
 (2d1) **If** $(\mathrm{Detect}(T_0^<) ==$ "yes"$)$ **then**
 (2e) $\mathbf{ReservedValue}(T_0^<, (k - 1 - j) * 4 + 2)$.
 EndIf
 (2f) $T_0 = \cup(T_0, T_0^<)$.
 (2g) $T_0 = +(T_0, m^1_j)$ and $T_1 = -(T_0, m^1_j)$.
 (2h) **If** $(\mathrm{Detect}(T_0) ==$ "yes"$)$ **then**
  (2i) $\mathbf{BinaryParallelAdder}(T_0, (k - 1 - j) * 4 + 3)$.
  (2j) $\mathbf{ParallelComparator}(T_0, T_n, T_0^>, T_0^=, T_0^<, (k - 1 - j) * 4 + 4)$.
  (2k) $T_0 = \cup(T_0^>, T_0^=)$.
  (2k1) **If** $(\mathrm{Detect}(T_0) ==$ "yes"$)$ **then**
  (2l) $\mathbf{BinaryParallelSubtractor}(T_0, (k - 1 - j) * 4 + 4)$.
  EndIf
  (2l1) **If** $(\mathrm{Detect}(T_0^<) ==$ "yes"$)$ **then**
  (2m) $\mathbf{ReservedValue}(T_0^<, (k - 1 - j) * 4 + 4)$.
  EndIf
  (2n) $T_0 = \cup(T_0, T_0^<)$.
 EndIf
 (2o) **If** $(\mathrm{Detect}(T_1) ==$ "yes"$)$ **then**
  (2p) $\mathbf{ReservedValue}(T_1, (k - 1 - j) * 4 + 3)$.
  (2q) $\mathbf{ReservedValue}(T_1, (k - 1 - j) * 4 + 4)$.
 EndIf
 (2r) $T_0 = \cup(T_0, T_1)$.
EndFor
EndProcedure.

Lemma 4–4: The algorithm $\mathbf{ModularMultiplication}(T_0, T_n)$ can be used to finish computation of $(M * M) \pmod n$.

E. A Library Sequence for an Initial Value to Computation of a Modular Multiplication

The $\mathbf{ModularMultiplication}(T_0, T_n)$ module uses, as a submodule, a parallel initial-valued assignment. We describe the construction of a parallel initial-valued assignment for bit-strings of arbitrary length. The following algorithm is used to construct a library sequence to encode an initial value to computation of a modular multiplication.

Procedure $\mathbf{InitialValue}(T_0)$

(1) **For** $g = 0$ **to** $k - 1$
 (1a) Append-head$(T_0, y^0_{1,g})$.
EndFor
EndProcedure

Lemma 4–5: Library strands for initial values to computation of a modular multiplication for M of k bits can be constructed from the algorithm $\mathbf{InitialValue}(T_0)$.

F. The Construction of a Left Shifter

The $\mathbf{ModularMultiplication}(T_0, T_n)$ module uses, as a submodule, a parallel left shifter. We describe the construction of a parallel left shifter for bit-strings of arbitrary length. A left shifter is an instruction of two operands of k bits that the second operand is applied to represent the number of the left shift to the first operand. Suppose that a binary number $y_{f,k-1} \ldots y_{f,0}$ denoted in Section IV-D, represent the first operand of a left shifter. Because computation of $(M * M) \pmod n$ denoted in Section IV-D only needs to perform left shift of one time, the second operand actually is equal to one. The following algorithm is used to construct a parallel left shifter.

Procedure $\mathbf{ParallelLeftShifter}(T_0, f)$

(1) Append-head$(T_0, y^0_{f+1,0})$.
(2) **For** $j = 0$ **to** $k - 2$
 (2a) $T_1 = +(T_0, y^1_{f,j})$ and $T_2 = -(T_0, y^1_{f,j})$.
 (2b) Append-head$(T_1, y^1_{f+1,j+1})$.
 (2c) Append-head$(T_2, y^0_{f+1,j+1})$.
 (2d) $T_0 = \cup(T_1, T_2)$.
EndFor
EndProcedure

TABLE I
TRUTH TABLE OF A ONE-BIT SUBTRACTOR

Minuend bit	Subtrahend bit	Previous borrow bit	Differenc e bit	Borrow bit
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Lemma 4–6: The algorithm $\mathbf{ParallelLeftShifter}(T_0, f)$ can be applied to finish the function of a parallel left shifter.

G. The Construction of a Parallel Comparator

The $\mathbf{ModularMultiplication}(T_0, T_n)$ module uses, as a submodule, a parallel comparator. We now describe its construction in detail. A one-bit parallel comparator is a Boolean function that performs compared operation of the two input bits. From compared results in a one-bit parallel comparator, DNA strands encoding those pairs $(y_{f,g}, n)$ with compared results "$>$", DNA strands encoding those pairs $(y_{f,g}, n)$ with compared results "$=$" and DNA strands encoding those pairs $(y_{f,g}, n)$ with compared results "$<$" are, respectively, put into three different tubes.

Therefore, the submodule, $\mathbf{OneBitComparator}(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$ is presented to compute the function of a one-bit parallel comparator. The first parameter and the second parameter, T_0 and T_n, respectively, contain those DNA strands that respectively encode $y_{f,g}$ and n. The third parameter, $T_0^{>}$, includes those DNA strands with the comparative result of greater than ("$>$") between $y_{f,g}$ and n. The fourth parameter, $T_0^{=}$, contains those DNA strands with the comparative result of equal ("$=$") between $y_{f,g}$ and n. The fifth parameter, $T_0^{<}$, consists of those DNA strands with the comparative result of less than ("$<$") between $y_{f,g}$ and n. The sixth parameter, f, is applied to represent the f^{th} compared operation in parallel comparator of a k-bits. The seventh parameter, g, is used to represent the compared operation of the g^{th} time for a one-bit parallel comparator from the f^{th} compare operation in parallel comparator of a k-bits. The eighth parameter, j, is employed to represent the j^{th} bit of n to be compared. The module, $\mathbf{ParallelComparator}(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f)$ also is proposed to finish the function of a k-bit parallel comparator.

Procedure $\mathbf{OneBitComparator}(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$

(1) $T_0^{ON} = +(T_0, y_{f,g}^1)$ and $T_0^{OFF} = -(T_0, y_{f,g}^1)$.
(2) $T_n^{ON} = +(T_n, n_j^1)$ and $T_n^{OFF} = -(T_n, n_j^1)$.
(3) **If** $(\text{Detect}(T_n^{ON}) == \text{"}yes\text{"})$ **then**
  (3a) $T_0^{=} = \cup(T_0^{=}, T_0^{ON})$ and $T_0^{<} = \cup(T_0^{<}, T_0^{OFF})$.
 Else
  (3b) $T_0^{>} = \cup(T_0^{>}, T_0^{ON})$ and $T_0^{=} = \cup(T_0^{=}, T_0^{OFF})$.
 EndIf
(4) $T_n = \cup(T_n^{ON}, T_n^{OFF})$.
EndProcedure

Lemma 4–7: The algorithm $\mathbf{OneBitComparator}(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, g, j)$ can be applied to finish the function of a one-bit parallel comparator.

Procedure $\mathbf{ParallelComparator}(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f)$

(1) **For** $j = k - 1$ **to** 0
 (1a) $\mathbf{OneBitComparator}(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f, j, j)$.
 (1b) **If** $(\text{Detect}(T_0^{=}) == \text{"no"})$ **then**
 (1c) Terminate the execution of the loop.
  Else
 (1d) $T_0 = \cup(T_0, T_0^{=})$.
  EndIf
 EndFor
(2) $T_0^{=} = \cup(T_0^{=}, T_0)$.
EndProcedure

Lemma 4–8: The algorithm $\mathbf{ParallelComparator}$ $(T_0, T_n, T_0^{>}, T_0^{=}, T_0^{<}, f)$ can be used to finish the function of a k-bit parallel comparator.

H. The Construction of a Binary Parallel Subtractor

The $\mathbf{ModularMultiplication}(T_0, T_n)$ module uses, as a submodule, a parallel subtractor. We first describe the construction of a parallel subtractor for a single bit, and then show how this may be used as a building block for a subtractor using bit-strings of arbitrary length. A one-bit subtractor is to finish the arithmetic subtraction of three input bits. It consists of three inputs and two outputs. Two of the input bits represent minuend and subtrahend bits to be subtracted. The third input represents the borrow bit from the previous higher significant position. The first output gives the value of the difference for minuend and subtrahend bits to be subtracted. The second output gives the value of the borrow bit to minuend and subtrahend bits to be subtracted.

Suppose that the two one-bit binary numbers $y_{f,g}$ and $y_{f+1,g}$ denoted in Section IV-D, represent the first input and the first output of a one-bit subtractor for $1 \le f \le (4 * k + 1)$ and $0 \le g \le k - 1$. Also suppose a one-bit binary number n_j denoted in Section IV-C, represents the second input of a one-bit subtractor for $0 \le j \le k - 1$, and two one-bit binary numbers $b_{f,g}$ and $b_{f,g-1}$ represent the second output and the third input of a one-bit subtractor. From [1], [2], two *distinct* DNA sequences are designed to encode every bit $b_{f,g-1}$ and $b_{f,g}$. Assume that $b_{f,g}^1$ contains the value of $b_{f,g}$ to be 1 and $b_{f,g}^0$ contains the value of $b_{f,g}$ to be 0. Similarly, also suppose that $b_{f,g-1}^1$ contains the value of $b_{f,g-1}$ to be 1 and $b_{f,g-1}^0$ contains the value of $b_{f,g-1}$ to be 0. The following algorithm is proposed to finish the function of a parallel one-bit subtractor.

Procedure $\mathbf{ParallelOneBitSubtractor}(T_0, f, g, j)$

(1) $T_1 = +(T_0, y_{f,g}^1)$ and $T_2 = -(T_0, y_{f,g}^1)$.
(2) $T_3 = +(T_1, n_j^1)$ and $T_4 = -(T_1, n_j^1)$.
(3) $T_5 = +(T_2, n_j^1)$ and $T_6 = -(T_2, n_j^1)$.
(4) $T_7 = +(T_3, b_{f,g-1}^1)$ and $T_8 = -(T_3, b_{f,g-1}^1)$.
(5) $T_9 = +(T_4, b_{f,g-1}^1)$ and $T_{10} = -(T_4, b_{f,g-1}^1)$.
(6) $T_{11} = +(T_5, b_{f,g-1}^1)$ and $T_{12} = -(T_5, b_{f,g-1}^1)$.

(7) $T_{13} = +(T_6, b^1_{f,g-1})$ and $T_{14} = -(T_6, b^1_{f,g-1})$.
(8a) **If** (Detect(T_7) == "yes") **then**
(8a1) Append-head$(T_7, y^1_{f+1,g})$ and Append-head$(T_7, b^1_{f,g})$.
EndIf
(9a) **If** (Detect(T_8) == "yes") **then**
(9a1) Append-head$(T_8, y^0_{f+1,g})$ and Append-head$(T_8, b^o_{f,g})$.
EndIf
(10a) **If** (Detect(T_9) == "yes") **then**
(10a1) Append-head$(T_9, y^0_{f+1,g})$ and Append-head$(T_9, b^0_{f,g})$.
EndIf
(11a) **If** (Detect(T_{10}) == "yes") **then**
(11a1) Append-head$(T_{10}, y^1_{f+1,g})$ and Append-head$(T_{10}, b^0_{f,g})$.
EndIf
(12a) **If** (Detect(T_{11}) == "yes") **then**
(12a1) Append-head$(T_{11}, y^0_{f+1,g})$ and Append-head$(T_{11}, b^1_{f,g})$.
EndIf
(13a) **If** (Detect(T_{12}) == "yes") **then**
(13a1) Append-head$(T_{12}, y^1_{f+1,g})$ and Append-head$(T_{12}, b^1_{f,g})$.
EndIf
(14a) **If** (Detect(T_{13}) == "yes") **then**
(14a1) Append-head$(T_{13}, y^1_{f+1,g})$ and Append-head$(T_{13}, b^1_{f,g})$.
EndIf
(15a) **If** (Detect(T_{14}) == "yes") **then**
(15a1) Append-head$(T_{14}, y^0_{f+1,g})$ and Append-head$(T_{14}, b^0_{f,g})$.
EndIf
(16) $T_0 = \cup(T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.
EndProcedure

Lemma 4–9: The algorithm $\mathbf{ParallelOneBitSubtractor}$ (T_0, f, g, j) can be applied to finish the function of a parallel one-bit subtractor.

The one-bit subtractor just described calculates the difference bit and the borrow bit for two input bits and a previous borrow. Two k-bit binary numbers can finish subtractions of k times by means of this one-bit subtractor. A binary parallel subtractor is to finish arithmetic subtraction for two k-bit binary numbers. The following algorithm is proposed to finish the function of a binary parallel subtractor.

Procedure $\mathbf{BinaryParallelSubtractor}(T_0, f)$

---

(1) Append-head$(T_0, b^0_{f,-1})$.
(2) **For** $j = 0$ **to** $k - 1$
(2a) $\mathbf{ParallelOneBitSubtractor}(T_0, f, j, j)$.
EndFor
EndProcedure

Lemma 4–10: The algorithm $\mathbf{BinaryParallelSubtractor}(T_0, f)$ can be applied to finish the function of a binary parallel subtractor.

I. Library Strands for Intermediate Values to Computation of a Modular Multiplication

The $\mathbf{ModularMultiplication}(T_0, T_n)$ module uses, as a submodule, a parallel assignment operator. We describe the construction of a parallel assignment operator for using bit-strings of arbitrary length. Blakley's algorithm denoted in Section IV-D is used to finish computation of $(M * M) \pmod n$. In Blakley's algorithm, it uses successive operations of addition, subtraction and left shifter to perform computation of $(M * M) \pmod n$. The procedure, $\mathbf{ReservedValue}(T_2, f)$, is used to reserve the result to intermediate computation of $(M * M) \pmod n$. The intermediate result will be used through next intermediate computation for $(M * M) \pmod n$.

Procedure $\mathbf{ReservedValue}(T_2, f)$

---

(1) **For** $g = 0$ to $k - 1$
(1a) $T_3 = +(T_2, y^1_{f,g})$ and $T_4 = -(T_2, y^1_{f,g})$.
(1b) **If** (Detect(T_3) == "yes") **then**
(1c) Append-head$(T_3, y^1_{f+1,g})$.
EndIf
(1d) **If** (Detect(T_4) == "yes") **then**
(1e) Append-head$(T_4, y^0_{f+1,g})$.
EndIf
(1f) $T_2 = \cup(T_3, T_4)$.
EndFor
EndProcedure

Lemma 4–11: The algorithm $\mathbf{ReservedValue}(T_2, f)$ can be applied to finish the function of reserving the intermediate result for computation of $(M * M) \pmod n$.

J. The Construction of a Binary Parallel Adder

The $\mathbf{ModularMultiplication}(T_0, T_n)$ module uses, as a submodule, a parallel adder. We first describe the construction of a parallel adder for a single bit, and then demonstrate how this perhaps is applied as a building block for a parallel adder by means of using bit-strings of arbitrary length. A one-bit adder is to perform the arithmetic sum of three input bits. It consists of three inputs and two outputs. Two of the input bits represent augends and addend bits to be added, respectively. The third input represents the carry from the previous lower significant position. The first output gives the value of the sum for augends and addend bits to be added. The second output gives the value of the carry to augends and addend bits to be added. The truth table of the one-bit adder is shown in Table II.

Suppose that two one-bit binary numbers denoted in Section IV-D, $y_{f,g}$ and $y_{f+1,g}$, represent the first input of a one-bit adder for $1 \leq f \leq (4 * k + 1)$ and $0 \leq g \leq k - 1$, and the first output of a one-bit adder, respectively, a one-bit binary number denoted in Section III, m_j, represents the second input of a one-bit adder for $0 \leq j \leq k - 1$, and two one-bit binary numbers, $z_{f,g}$ and $z_{f,g-1}$, represent the second output and the

TABLE II
TRUTH TABLE OF A ONE-BIT ADDER

Augend bit	Addend bit	Previous carry bit	Sum bit	Carry bit
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

third input of a one-bit adder, respectively. Two *distinct* DNA sequences are designed to encode the value "0" or "1" for every bit $z_{f,g-1}$ and $z_{f,g}$ to $1 \leq f \leq (4*k+1)$ and $0 \leq g \leq k-1$. For the sake of convenience in our presentation, assume that $z^1_{f,g}$ contains the value of $z_{f,g}$ to be 1 and $z^0_{f,g}$ contains the value of $z_{f,g}$ to be 0. Also suppose that $y^1_{f+1,g}$ denotes the value of $y_{f+1,g}$ to be 1 and $y^0_{f+1,g}$ defines the value of $y_{f+1,g}$ to be 0. Similarly, assume that $z^1_{f,g-1}$ contains the value of $z_{f,g-1}$ to be 1 and $z^0_{f,g-1}$ contains the value of $z_{f,g-1}$ to be 0. The following algorithm is proposed to finish the function of a parallel one-bit adder.

Procedure $\mathbf{ParallelOneBitAdder}(T_0, f, g, j)$

(1) $T_1 = +(T_0, y^1_{f,g})$ and $T_2 = -(T_0, y^1_{f,g})$.
(2) $T_3 = +(T_1, m^1_j)$ and $T_4 = -(T_1, m^1_j)$.
(3) $T_5 = +(T_2, m^1_j)$ and $T_6 = -(T_2, m^1_j)$.
(4) $T_7 = +(T_3, z^1_{f,g-1})$ and $T_8 = -(T_3, z^1_{f,g-1})$.
(5) $T_9 = +(T_4, z^1_{f,g-1})$ and $T_{10} = -(T_4, z^1_{f,g-1})$.
(6) $T_{11} = +(T_5, z^1_{f,g-1})$ and $T_{12} = -(T_5, z^1_{f,g-1})$.
(7) $T_{13} = +(T_6, z^1_{f,g-1})$ and $T_{14} = -(T_6, z^1_{f,g-1})$.
(8a) **If** $(\text{Detect}(T_7) ==$ "yes"$)$ **then**
  (8a1) $\text{Append-head}(T_7, y^1_{f+1,g})$ and $\text{Append-head}(T_7, z^1_{f,g})$.
 EndIf
(9a) **If** $(\text{Detect}(T_8) ==$ "yes"$)$ **then**
  (9a1) $\text{Append-head}(T_8, y^0_{f+1,g})$ and $\text{Append-head}(T_8, z^1_{f,g})$.
 EndIf
(10a) **If** $(\text{Detect}(T_9) ==$ "yes"$)$ **then**
  (10a1) $\text{Append-head}(T_9, y^0_{f+1,g})$ and $\text{Append-head}(T_9, z^1_{f,g})$.
 EndIf
(11a) **If** $(\text{Detect}(T_{10}) ==$ "yes"$)$ **then**
  (11a1) $\text{Append-head}(T_{10}, y^1_{f+1,g})$ and $\text{Append-head}(T_{10}, z^0_{f,g})$.
 EndIf
(12a) **If** $(\text{Detect}(T_{11}) ==$ "yes"$)$ **then**
  (12a1) $\text{Append-head}(T_{11}, y^0_{f+1,g})$ and $\text{Append-head}(T_{11}, z^1_{f,g})$.
 EndIf
(13a) **If** $(\text{Detect}(T_{12}) ==$ "yes"$)$ **then**
  (13a1) $\text{Append-head}(T_{12}, y^1_{f+1,g})$ and $\text{Append-head}(T_{12}, z^0_{f,g})$.
 EndIf
(14a) **If** $(\text{Detect}(T_{13}) ==$ "yes"$)$ **then**
  (14a1) $\text{Append-head}(T_{13}, y^1_{f+1,g})$ and $\text{Append-head}(T_{13}, z^0_{f,g})$.
 EndIf
(15a) **If** $(\text{Detect}(T_{14}) ==$ "yes"$)$ **then**
  (15) $\text{Append-head}(T_{14}, y^0_{f+1,g})$ and $\text{Append-head}(T_{14}, z^0_{f,g})$.
 EndIf
(16) $T_0 = \cup(T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14})$.
EndProcedure

Lemma 4–12: The algorithm $\mathbf{ParallelOneBitAdder}(T_0, f, g, j)$ can be applied to finish the function of a parallel one-bit adder.

IV. FACTORING INTEGER ALGORITHM

The RSA public-key cryptosystem can be used to encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted message will not be able to decode them. One must be an element in $\boldsymbol{Z}^*_n$ denoted in Section III and must be quadratic residue of modulo n. Therefore, assume that one is represented as a k-bit binary number, $y^0_{(4*k+1),k-1} \cdots y^1_{(4*k+1),0}$, denoted in Section IV-D. For using the same library sequence to encode one, the main advantage is to reduce the time-complexity of the algorithm for solving the RSA public-key cryptosystem. An eavesdropper only needs to use the following algorithm to factor integers. This implies that the RSA public-key cryptosystem can be broken from the following algorithm.

Algorithm 5–1: Breaking the RSA public-key cryptosystem.

(1) Call **Algorithm 3–1**.
(2) $T_1 = +(T_0, y^1_{(4*k+1),0})$ and $T_2 = -(T_0, y^1_{(4*k+1),0})$.
(3) $\text{Discard}(T_2)$.
(4) $T_0 = \cup(T_0, T_1)$.
(5) **For** $j = 1$ **to** $k-1$
 (5a) $T_1 = +(T_0, y^0_{(4*k+1),j})$ and $T_2 = -(T_0, y^0_{(4*k+1),j})$.
 (5b) $\text{Discard}(T_2)$.
 (5c) $T_0 = \cup(T_0, T_1)$.
 EndFor
(6) **If** $(\text{Detect}(T_0) ==$ "yes"$)$ **then**
 (6a) $\text{Read}(T_0)$.
 EndIf
(7) Assume that four integer solutions for $M^2 \equiv 1 \pmod{n}$ are, respectively, x, $n-x$, 1 and $n-1$.
(8) Through a *digital* computer, two large prime numbers p and q are determined, where $p = \gcd(x-1, n)$ and $q = \gcd(x+1, n)$.

(9) Through a *digital* computer, the corresponding secret key d for the public key e is determined, where $e * d \equiv 1 \ (\mathrm{mod}(p-1)*(q-1))$.
EndAlgorithm

Theorem 5–1: From those steps in **Algorithm 5–1**, an eavesdropper can break the RSA public-key cryptosystem.

V. Complexity Assessment

Theorem 6–1: Suppose that the length of M is k bits. The RSA public-key cryptosystem can be broken with $\mathrm{O}(k^2)$ biological operations of laboratory techniques from solution space of library sequences.

Proof: Refer to **Algorithm 3–1** and **Algorithm 5–1**. ■

Theorem 6–2: Suppose that the length of M is k bits. The RSA public-key cryptosystem can be broken with $\mathrm{O}(2^k)$ library strands in biological operations of laboratory techniques from solution space of library sequences.

Proof: Refer to **Algorithm 3–1** and **Algorithm 5–1**. ■

Theorem 6–3: Suppose that the length of M is k bits. The RSA public-key cryptosystem can be broken with $\mathrm{O}(c)$ tubes in biological operations of laboratory techniques from solution space of library sequences, where c is a constant value.

Proof: Refer to **Algorithm 3–1** and **Algorithm 5–1**. ■

Theorem 6–4: Suppose that the length of M is k bits. The RSA public-key cryptosystem can be broken with the longest library strand, $\mathrm{O}(k^2)$, in biological operations of laboratory techniques from solution space of library sequences.

Proof: Refer to **Algorithm 3–1** and **Algorithm 5–1**. ■

VI. The Faster Method for Factoring Integer

Algorithm 3–1 is used to solve the problem of quadratic congruence. With the result generated by **Algorithm 3–1**, **Algorithm 5–1** is applied to factor integers and its ultimate aim is to break the RSA public-key cryptosystems. The following theorem is employed to prove that time complexity of **Algorithm 5–1** is currently the fastest method to factor integers.

Theorem 7: With biological operations of laboratory techniques from solution space of library sequences, time complexity of **Algo**rithm 5–1 is the optimal solution of breaking the RSA public-key cryptosystems.

Proof: Algorithm 5–1 is used to factor a big integer into primes by quadratic congruence, and its ultimate aim is to break the RSA public-key cryptosystems. Blakley's algorithm is the best method to perform computation of $M^2 \ (\mathrm{mod}\ n)$ and it is implemented in $\mathbf{ModularMultiplication}(T_0, T_n)$. Thus, it is inferred that time complexity of **Algorithm 5–1** is currently the fastest method to factor integers. ■

VII. Conclusion

The number of steps any classical computer requires in order to find the prime factors of a k-bit integer n increases exponentially with k, at least by means of using algorithms [5] known at present. Shor's *quantum* factoring algorithm [6] contains that the two main components, modular exponentiation (computation of $a^x \bmod n$) and the inverse quantum Fourier transform (**QFT**) take only $\mathrm{O}(k^3)$ operations. Vandersypen and his coauthors [7] report an implementation of the simplest instance of Shor's algorithm: factorization of $n = 15$ (whose prime factors are 3 and 5). The previous relative work [8] theoretically proves that the problem of factoring integers can be solved with $\mathrm{O}(k^3)$ biological operations. In this article, Our *molecular* factoring algorithm demonstrate theoretically how basic biological operations can be used to solve the problem of factoring integers with $\mathrm{O}(k^2)$ biological operations. Both of Shor's quantum factoring algorithm and our molecular factoring algorithm need to simultaneously deal with 2^{1024} bit information to find the prime factors for an integer n of 1024 bits used in the current RSA public-key cryptosystem. However, due to current many technical difficulties, therefore, the two algorithms currently do not in fact find the prime factors for an integer n of 1024 bits. This implies that if a quantum computer and a molecular computer are *really* constructed in the future (perhaps after many years), then Shor's quantum factoring algorithm and our molecular factoring algorithm have very high feasibility for solving the problem of factoring integers.

References

[1] R. S. Braich, C. Johnson, P. W. K. Rothemund, D. Hwang, N. Chelyapov, and L. M. Adleman, "Solution of a satisfiability problem on a gel-based DNA computer," in *Proc. 6th Int. Conf. DNA Comput.*, 2001, vol. 2054, Lecture Notes in Computer Science, pp. 27–42.
[2] L. M. Adleman, R. S. Braich, C. Johnson, P. W. K. Rothemund, D. Hwang, and N. Chelyapov, "Solution of a 20-variable 3-SAT problem on a DNA computer," *Science*, vol. 296, no. 5567, pp. 499–502, Apr. 2002.
[3] N. Koblitz, *A Course in Number Theory and Cryptography*. New York: Springer-Verlag, 1987.
[4] G. R. Blakley, "A computer algorithm for calculating product AB modulo M," *IEEE Trans. Comput.*, vol. C-32, no. 5, pp. 497–500, 1983.
[5] D. E. Knuth, *The Art of Computer Programming,: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1998, vol. 2.
[6] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.
[7] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, vol. 414, pp. 883–887, 2001.
[8] W.-L. Chang, M. Guo, and M. Ho, "Fast parallel molecular algorithms for DNA-based computation: Factoring integers," *IEEE Trans. NanoBiosci.*, vol. 4, no. 2, pp. 149–163, Jun. 2005.

Weng-Long Chang received the Ph.D. degree in computer science and information engineering from National Cheng Kung University, Taiwan, in 1999.

He is currently a full Professor in the Department of Computer Science and Information Engineering, National Kaohsiung University of Applied Sciences, Taiwan. His research interests include quantum algorithms, quantum-molecular algorithms, DNA-based algorithms, and languages and compilers for parallel computing.