数理解析研究所講究録 1375 巻 2004 年 188-194

Procedures for Multiple Input Functions with DNA Strands

上尾 智史 (Satoshi Kamio) 藤原 暁宏 (Akihiro Fujiwara)
九州工業大学大学院 情報工学研究科
Department of Computer Science and Electronics
Kyushu Institute of Technology

Abstract

In recent works for high performance computing, computation with DNA molecules, that is, DNA computing, has considerable attention as one of non-silicon based computing. In this paper, we propose two procedures for computing multiple input functions. We first propose a simple procedure for computing AND function. The procedure runs in $O(1)$ steps using $O(mn)$ DNA strands for n binary numbers of m bits. We next propose a procedure for EX-OR function. The procedure runs in $O(1)$ steps using $O(mn^2)$ DNA strands, and is also applicable to other functions, such as majority and threshold functions.

1 Introduction

In recent works for high performance computing, computation with DNA molecules, that is, DNA computing, has considerable attention as one of non-silicon based computing. The DNA has two important features, which are Watson-Crick complementarity and massive parallelism. Using the features, we can solve an NP-complete problem, which usually needs exponential computation time on a silicon based computer, in a polynomial number of steps with DNA molecules. There are a number of works with DNA molecules for combinatorial NP-complete problems[1, 2, 9, 10, 15].

However, procedures for primitive operations, such as logic or arithmetic operations, are needed to apply DNA computing on a wide range of problems. A number of procedures have been proposed for the primitive operations with DNA molecules[3, 4, 5, 6, 7, 12]. Fujiwara et al.[4] have proposed addressable procedures for the primitive operations. They first showed a DNA representation of n binary numbers of m bits, and they proposed procedures which compute logic operations and additions of pairs of two binary numbers. The procedures run in $O(1)$ steps using $O(mn)$ DNA strands for n pairs of two binary numbers. Recently, Kamio et al.[8] proposed three procedures for computing the maximum of n binary numbers of m bits. The first procedure consists of a repetition of checking on m bit positions, and runs in $O(m)$ steps using $O(n)$ different DNA strands. The second procedure consists of a repetition of parallel comparisons of two numbers, and runs in $O(\log n)$ steps using $O(mn)$ different DNA strands. The third procedure mainly consists of $O(n^2)$ parallel comparisons, and runs in $O(1)$ steps using $O(mn^2)$ different DNA strands.

In this paper, we propose two procedures for computing multiple input functions. An input of the function is a set of n binary numbers of m bits, and an output is a binary number of m bits, which is defined by a n input logic function. We first show a simple procedure for computing AND function. The procedure runs in $O(1)$ steps using $O(mn)$ different DNA strands for n binary numbers. The procedure is also applicable to other simple logic functions, such as OR, NAND and NOR. We next propose a procedure for EX-OR function. The procedure runs in $O(1)$ steps using $O(mn^2)$ DNA strands, and is also applicable to other functions, such as majority and threshold functions.

2 Preliminaries

2.1 Computational model for DNA computing

A number of theoretical or practical computational models have been proposed for DNA computing[1, 6, 7, 9, 12, 13, 14]. A computational model used in this paper is the same model as [4]. We briefly introduce the model in this subsection.

A *single strand* of DNA is defined as a string of symbols over a finite alphabet Σ. We define the alphabet $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{m-1}, \overline{\sigma}_0, \overline{\sigma}_1, \ldots, \overline{\sigma}_{m-1}\}$, where the symbols $\sigma_i, \overline{\sigma}_i$ $(0 \leq i \leq m-1)$ are *complements*. Two single strands form a *double strand* if and only if the single strands are complements of each other. A double strand with $\sigma_i, \overline{\sigma}_i$ is denoted by $\begin{bmatrix} \sigma_i \\ \overline{\sigma}_i \end{bmatrix}$.

The single or double strands are stored in a *test tube*. For example, $T_1 = \{\sigma_0\sigma_1, \overline{\sigma_1\sigma_0}\}$ denotes a test tube in which two kinds of single strands $\sigma_0\sigma_1$, $\overline{\sigma_1\sigma_0}$ are stored.

Using the DNA strands, the following eight DNA manipulations are allowed on the computational model. Since these eight manipulations are implemented with a constant number of biological steps for DNA strands[11], we assume that the complexity of each manipulation is $O(1)$. (See [4] for details of the manipulations.)

(1) *Merge*: Given two test tubes T_1, T_2, $Merge(T_1, T_2)$ stores the union $T_1 \cup T_2$ in T_1.

(2) *Copy*: Given a test tube T_1, $Copy(T_1, T_2)$ produces a test tube T_2 with the same contents as T_1.

(3) *Detect*: Given a test tube T, $Detect(T)$ outputs "yes" if T contains at least one strand, otherwise, $Detect(T)$ outputs "no".

(4) *Separation*: Given a test tube T_1 and a set of strings X, $Separation(T_1, X, T_2)$ removes all single strands containing a string in X from T_1, and produces a test tube T_2 with the removed strands.

(5) *Selection*: Given a test tube T_1 and an integer L, $Selection(T_1, L, T_2)$ removes all strands, whose length is L, from T_1, and produces a test tube T_2 with the removed strands. (The length of a strand is the number of symbols in the strand.)

(6) *Cleavage*: Given a test tube T and a string of two symbols $\sigma_0\sigma_1$, $Cleavage(T, \sigma_0\sigma_1)$ cuts each double strand containing $\begin{bmatrix} \sigma_0\sigma_1 \\ \overline{\sigma_0\sigma_1} \end{bmatrix}$ in T into two double strands as follows.

$$\begin{bmatrix} \alpha_0\sigma_0\sigma_1\beta_0 \\ \alpha_1\overline{\sigma_0\sigma_1}\beta_1 \end{bmatrix} \Rightarrow \begin{bmatrix} \alpha_0\sigma_0 \\ \alpha_1\overline{\sigma_0} \end{bmatrix}, \begin{bmatrix} \sigma_1\beta_0 \\ \overline{\sigma_1}\beta_1 \end{bmatrix}$$

(We assume that *Cleavage* can only be applied to some *specified symbols* over the alphabet Σ.)

(7) *Annealing*: Given a test tube T, $Annealing(T)$ produces all feasible double strands from single strands in T. (The produced double strands are still stored in T after *Annealing*.)

(8) *Denaturation*: Given a test tube T, $Denaturation(T)$ dissociates each double strand in T into two single strands.

In addition to the above, we add a manipulation to clarify description of this paper. The complexity of the manipulation is also $O(1)$.

(9) *Empty*[1]: Given a test tube T, $Empty(T)$ sets $T = \phi$.

2.2 Representation of binary numbers with DNA strands

In this subsection, we explain a data structure for storing a set of n binary numbers using DNA strands. Let us consider a number x such that $x = \sum_{j=0}^{m-1} x_j * 2^j$, where $x_{m-1}, x_{m-2}, \ldots, x_0$ are binary bits. We assume that the most significant bit x_{m-1} is a sign bit, and a negative number is denoted using two's complement notation. A representation of each bit is the same as that in [4], and is briefly described in the following.

[1] $Empty(T)$ is equivalent to $Copy(\phi, T)$.

We first define the alphabet Σ as follows.

$$\begin{aligned} \Sigma = \{ & A_0, A_1, \ldots, A_n, B_0, B_1, \ldots, B_{m-1}, \\ & C_0, C_1, D_0, D_1, 1, 0, \sharp, \\ & \overline{A_0}, \overline{A_1}, \ldots, \overline{A_n}, \overline{B_0}, \overline{B_1}, \ldots, \overline{B_{m-1}}, \\ & \overline{C_1}, \overline{C_2}, \overline{D_1}, \overline{D_2}, \overline{1}, \overline{0}, \overline{\sharp}\} \end{aligned}$$

In the above alphabet, $A_0, A_1, \ldots, A_n$ denote addresses of numbers, and $B_0, B_1, \ldots, B_{m-1}$ denote bit positions. C_0, C_1 and D_0, D_1 are specified symbols cut by *Cleavage*. Symbols "0" and "1" are used to denote values of bits, and "$\sharp$" is a special symbol for *Separation*.

Using the above alphabet, a value of a bit, whose address and bit position are i and j, is represented by a single strand $S_{i,j}$ such that

$$S_{i,j} = D_1 A_i B_j C_0 C_1 V_{i,j} D_0,$$

where $V_{i,j} = 0$ if a value of the bit is 0, otherwise, $V_{i,j} = 1$.

We call each $S_{i,j}$ a *memory strand*, and use a set of $O(mn)$ different memory strands to denote n binary numbers of m bits, that is, a number x stored in address i is represented by a set of memory strands $\{S_{i,m-1}, S_{i,m-2}, \ldots, S_{i,0}\}$, which denote binary bits $x_{m-1}, x_{m-2}, \ldots, x_0$, respectively. We assume that V_i denotes the number stored in address i as follows.

$$V_i = \sum_{j=0}^{m-1} V_{i,j} * 2^j.$$

We also assume that $S_{i,j}(V)$ denote a memory strand whose value is V, that is,

$$S_{i,j}(0) = D_1 A_i B_j C_0 C_1 0 D_0, S_{i,j}(1) = D_1 A_i B_j C_0 C_1 1 D_0$$

2.3 Primitive operations

In this paper, four operations *ValueAssignment*, *LogicOperation*, *Subtraction* and *MaxOperation* are used as primitive operations. The $ValueAssignment\_V(T_{input}, T_{output})$ is an operation which assigns the same value $V(\in \{0,1\})$ to all memory strands in a test tube T_{input}. The $LogicOperation(T_{input}, L, T_{output})$ is an operation which executes logic operations, which are defined by single strands in a test tube L, for pairs of two memory strands in T_{input}. The $Subtraction(T_{input}, R, T_{output})$ is an operation which executes subtractions, which are defined by single strands in a test tube R, for pairs of memory strands in T_{input}. The $MaxOperation(T_{input}, T_{output})$ is an operation which computes the maximum of values which are stored memory strands in T_{input}. The results of all operations are stored in a test tube T_{output}.

For the above four primitive operations, the following lemmas are obtained in [4] and [8].

Lemma 1 [4] *The $ValueAssignment(T_{input}, T_{output})$ can be executed in $O(1)$ steps using $O(1)$ kinds of DNA strands.* □

Lemma 2 [4] *The $LogicOperation(T_{input}, L, T_{output})$ and $Subtraction(T_{input}, R, T_{output})$, which are for $O(n)$ pairs of m-bit binary numbers, can be executed in $O(1)$ steps using $O(mn)$ kinds of DNA strands.* □

Lemma 3 [8] *The $MaxOperation(T_{input}, T_{output})$, which is for $O(n)$ binary numbers of m bits, can be executed in $O(1)$ steps using $O(mn^2)$ kinds of DNA strands.* □

2.4 Input of procedures

We assume that an input of a function is a set of n binary numbers of m bits, and given by a test tube T_{input} such that,

$$T_{input} = \{S_{i,j} \mid 0 \leq i \leq n, 0 \leq j \leq m-1\},$$

where $\{S_{i,j} \mid 0 \leq i \leq n-1, 0 \leq j \leq m-1\}$ is a set of memory strands which denote n input binary numbers, and $\{S_{n,j} \mid 0 \leq j \leq m-1\}$ is a set of memory strands in which an output of the procedures is stored. (All memory strands are stored in T_{input} again at the end of each procedure.) We also assume that f is a given logic function such that $V_n = f(V_0, V_1, \cdots, V_{n-1})$.

In this paper, we consider multiple input functions such that an output value of each bit does not depend on output values of the other bits. For example, we consider a multiple input function f, and assume that $\{V_0, V_1, \ldots, V_{n-1}\}$ and V_n are input and output binary numbers such that $V_i = \sum_{j=0}^{m-1} V_{i,j} * 2^j$, respectively. Then, we can define a function f_j whose input and output are of Boolean values in j-th bit, that is, $V_{n,j} = f_j(V_{0,j}, V_{1,j}, \cdots, V_{n-1,j})$.

3 Procedure for AND function

The logic function AND for multiple input is defined as follows.

$$V_{n,j} = f_j(V_{0,j}, V_{1,j}, \cdots, V_{n-1,j})$$
$$= V_{0,j} \wedge V_{1,j} \wedge \cdots \wedge V_{n-1,j}$$

A strategy for computing AND is simple. We first separate output memory strands $\{S_{n,j} \mid 0 \leq j \leq m-1\}$ from a test tube T_{input}, store the memory strands in a test tube T_1, and assign 1 to all memory strands in T_1. We next choose output memory strands, whose output value must be 0. An output value of AND function is 0 if there exists one 0 in input values. We separate the output memory strands using input memory strands and additional strands, which indicate a feature of AND function. We move the separated output memory strands from T_1 to T_0, and assign 0 to all memory strands in T_0. Finally, we return memory strands in T_0 and T_1 into T_{input}.

We now summarize an overview of the procedure AND, which computes AND function in $O(1)$ steps.

Procedure AND

Step 1: Separate memory strands whose addresses are A_n from T_{input} to a test tube T_1. Then, execute $ValueAssignment\_1$ for T_1.

Step 2: Separate memory strands whose output value must be 0, from T_1 to T_0. Then, execute $ValueAssignment\_0$ for T_0.

Step 3: Return memory strands in T_1 and T_0 to T_{input}.

(End of the procedure)

We now describe details of the procedure step by step. At the beginning, we show test tubes used in the description.

T_{tmp}: Memory strands are temporarily stored in T_{tmp}.

$T_{1\to 0}$: Single strands, which separate memory strands from T_1 to T_0, are stored in $T_{1\to 0}$.

T_0, T_1: Memory strands, whose value must be 0 and 1, are stored in T_0 and T_1, respectively.

T_{trash}: Unnecessary strands are discarded into T_{trash}.

Step 1 consists of the following manipulations.

```
Step 1
    $Separation(T_{input}, \{A_n\}, T_1)$
    $ValueAssignment\_1(T_1, T_1)$
```

In Step 2, we merge the following test tube $T_{1\to 0}$ with the input test tube.

$$T_{1\to 0} = \{D_1\#D_0\} \cup \{\overline{S_{i,j}(0)D_1\#D_0D_1A_nB_j} \mid 0 \leq i \leq n-1, 0 \leq i \leq m-1\}$$

Then, we execute manipulations *Annealing*, *Cleavage*, *Denaturation* and *Separation*.

After execution of the manipulations, the test tube contains a single strand $D_1\#D_0D_1A_nB_j$ if and only if $V_{n,j}$ must be 0. (If there is at least one 0 in input of AND function, its output must be 0.)

We next execute *Annealing* and *Denaturation* for the above single strand and memory strand in T_1. Then, we obtain a single strand $D_1\#D_0S_{n,j}$ if and only if $V_{n,j}$ must be 0.

We separate the single strand from T_1 to T_0 and remove an unnecessary part $D_1\#D_0$ from the single strand using *Cleavage*. Finally, we assign 0 to the memory strands in T_0. The Step 2 is summarized below.

```
Step 2
    $Empty(T_{tmp})$
    $Copy(T_{input}, T_{tmp})$
```

```
$Merge(T_{tmp}, T_{1\to 0})$
$Annealing(T_{tmp})$
$Cleavage(T_{tmp}, D_0D_1)$
$Denaturation(T_{tmp})$
$Separation(T_{tmp}, \{C_0C_1, \overline{C_0C_1}\}, T_{trash})$

$Merge(T_1, T_{tmp})$
$Annealing(T_1)$
$Denaturation(T_1)$
$Separation(T_1, \{D_1\#D_0\}, T_0)$
$Separation(T_1, \{\overline{D_1\#D_0}\}, T_{trash})$

$Merge(T_0, \{\overline{D_0D_1}\})$
$Cleavage(T_0, D_0D_1)$
$Separation(T_0, \{D_1\#D_0, \overline{D_0}, \overline{D_1}\}, T_{trash})$

$ValueAssignment\_0(T_0)$
```

In Step 3, all memory strands in T_0 and T_1 are returned to T_{input}. This step consists of the following manipulations.

Step 3

```
$Merge(T_{input}, T_0)$
$Merge(T_{input}, T_1)$
```

We now consider complexity of the above procedure. Each step consists of a constant number of DNA manipulations, which are described in Section 2. In addition, $O(mn)$ kinds of strands are used in the procedure. Then, we obtain the following theorem.

Theorem 1 *Procedure AND, which computes the result of AND operation of n binary numbers of m bits, runs in $O(1)$ steps using $O(mn)$ DNA strands.* □

The procedure *AND* are easily modified to be applied to other simple logic functions, such as, OR, NAND and NOR. Thus, we can compute the logic functions with the same complexity.

4 Procedure for EX-OR function

The logic function EX-OR for multiple input is defined as follows.

$$V_{n,j} = f_j(V_{0,j}, V_{1,j}, \cdots, V_{n-1,j})$$
$$= V_{0,j} \oplus V_{1,j} \oplus \cdots \oplus V_{n-1,j}$$

We describe an overview of the procedure for computing EX-OR function intuitively. (Details of the procedure is slightly different from the following overview.) An output value of EX-OR function is "1" if and only if the number of "1" in its input is odd, otherwise, the output value is "0". Thus, we compute the number of "1" in its input by concatenating memory strands whose values are "1". To compute the number in $O(1)$ steps, we concatenate all possible pairs of memory strands in ascending order. More precisely, a memory strand $S_{i,j}$ can be concatenated with one of memory strands $S_{i+1,j}, S_{i+2,j}, \cdots, S_{n-1,j}$ in case that $\{S_{0,j}, S_{1,j}, \cdots, S_{n-1,j}\}$ is a set of memory strands in which input values are stored. For example, let $\{S_{0,0}(1), S_{1,0}(1), S_{2,0}(0), S_{3,0}(1)\}$ be a set of the input memory strands. Then, we obtain the following set of single strands after the concatenation.

$$\{S_{0,0}(1),\ S_{1,0}(1),\ S_{3,0}(1),\ S_{0,0}(1)S_{1,0}(1),\ S_{0,0}(1)S_{3,0}(1),\ S_{1,0}(1)S_{3,0}(1),\ S_{0,0}(1)S_{1,0}(1)S_{3,0}(1)\}$$

In the above single strands, a length of the longest strand means the number of "1" in its input. However, we cannot compute the length directly using $O(1)$ manipulations.

To obtain the length of the longest strand, we use additional single strands, which denote length of strands. The additional strand consists of two parts. The first part consists of dummies to make a length of the single strand constant, and the second part consists of concatenated memory strands which denote the length of the strand. (The length is represented by a binary number.) We assume that α is a single strand whose length is equal to length of a memory strand. The single strand α is used as a dummy, and is concatenated according to a binary number in the second part. (Let k_α be the number of α in the first part. Then, a binary number stored in the second part is $n - k_\alpha$.) Figure 1 shows a set of additional single strands for the above example, where j denote the bit number.

We concatenate the above two kinds of single strands, and separate single strands whose length is $(n + (\lceil \log_2(n+1) \rceil)) \times |\alpha|$, where $|\alpha|$ is a length of a memory strand. In the above example, single strands in Figure 2, whose length is $(4+3) * |\alpha| = 7|\alpha|$, are separated.

By cutting the separated single strand into first and second parts, we obtain length of single strands as binary numbers in the second parts. Since we can compute the maximum of n binary numbers using *MaxOperation*, which is described in Section 2, we obtain a binary number which denotes length of the longest single strand in $O(1)$ steps. Then, finally, we find whether the length is odd or even by checking the lowest bit of the binary number.

We now summarize an overview of a procedure *EX-OR*, which computes EX-OR function in $O(1)$ steps. Some substeps are added to complete the procedure.

Procedure *EX-OR*

Step 1:

(1-1) Separate memory strands whose addresses are A_n from T_{input} to T_n. Then, execute $ValueAssignment\_0$ for T_n.

(1-2) Separate memory strands which denote a value "1" from T_{input} to T_1.

$$\begin{aligned}
\{\boxed{\alpha}\boxed{\alpha}\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)j+n+1+0,2}(0)S_{(n+1)j+n+1+0,1}(0)S_{(n+1)j+n+1+0,0}(0),\\
\boxed{\alpha}\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)j+n+1+1,2}(0)S_{(n+1)j+n+1+1,1}(0)S_{(n+1)j+n+1+1,0}(1),\\
\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)j+n+1+2,2}(0)S_{(n+1)j+n+1+2,1}(1)S_{(n+1)j+n+1+2,0}(0),\\
\boxed{\alpha}\,& S_{(n+1)j+n+1+3,2}(0)S_{(n+1)j+n+1+3,1}(1)S_{(n+1)j+n+1+3,0}(1),\\
& S_{(n+1)j+n+1+4,2}(1)S_{(n+1)j+n+1+4,1}(0)S_{(n+1)j+n+1+4,0}(0)\}
\end{aligned}$$

Figure 1: An example of additional single strands.

$$\begin{aligned}
\{S_{0,0}(1)\boxed{\alpha}\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)\ast 0+n+1+1,2}(0)S_{(n+1)\ast 0+n+1+1,1}(0)S_{(n+1)\ast 0+n+1+1,0}(1),\\
S_{0,1}(1)\boxed{\alpha}\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)\ast 1+n+1+1,2}(0)S_{(n+1)\ast 1+n+1+1,1}(0)S_{(n+1)\ast 1+n+1+1,0}(1),\\
S_{2,4}(1)\boxed{\alpha}\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)\ast 4+n+1+1,2}(0)S_{(n+1)\ast 4+n+1+1,1}(0)S_{(n+1)\ast 4+n+1+1,0}(1),\\
S_{0,0}(1)S_{1,0}(1)\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)\ast 0+n+1+2,2}(0)S_{(n+1)\ast 0+n+1+2,1}(1)S_{(n+1)\ast 0+n+1+2,0}(0),\\
S_{0,0}(1)S_{3,0}(1)\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)\ast 0+n+1+2,2}(0)S_{(n+1)\ast 0+n+1+2,1}(1)S_{(n+1)\ast 0+n+1+2,0}(0),\\
S_{1,2}(1)S_{3,2}(1)\boxed{\alpha}\boxed{\alpha}\,& S_{(n+1)\ast 2+n+1+2,2}(0)S_{(n+1)\ast 2+n+1+2,1}(1)S_{(n+1)\ast 2+n+1+2,0}(0),\\
S_{0,3}(1)S_{1,3}(1)S_{3,3}(1)\boxed{\alpha}\,& S_{(n+1)\ast 3+n+1+3,2}(0)S_{(n+1)\ast 3+n+1+3,1}(1)S_{(n+1)\ast 3+n+1+3,0}(1)\}
\end{aligned}$$

Figure 2: An example of single strands after concatenation.

(1-3) Concatenate memory strands in T_1 in ascending order, and move the concatenated strands from T_1 to T_{tmp_2}.

Step 2:

(2-1) Concatenate additional single strands in T_{tmp_2}, and separate single strands, whose length are $(n + (\lceil\log_2(n+1)\rceil)) \times |\alpha|$, from T_{tmp_2} to T_{tmp_1}.

(2-2) Cut single strands into memory strands in T_{tmp_1}, and execute *MaxOperation* in T_{tmp_1}. Then, store memory strands, which denote the maximum, in T_{out}.

Step 3:

(3-1) Merge from T_{out} to T_n, and execute *LogicOperation* for T_n, and the output is stored in T_n.

(3-2) Separate memory strands, which denote the output from T_n to T_{tmp_1}, and return all memory strands in T_{tmp_1} to T_{input}.

(End of the procedure)

We now describe details of the procedure step by step. At the beginning, we summarize test tubes used in the description.

T_{tmp_1}, T_{tmp_2}**:** Memory strands are temporarily stored in T_{tmp_1} and T_{tmp_2}.

T_0, T_1**:** Memory strands, whose value must be 0 and 1, are stored in T_0 and T_1, respectively.

T_{trash}**:** Unnecessary strands are discarded into T_{trash}.

$T_{con_1}, T_{con_2}, T_{con_3}$**:** Single strands which concatenate memory strands are stored in T_{con1}, T_{con2} and T_{con_3}.

T_{dummy}**:** Additional memory strands for *MaxOperation* are stored in T_{dummy}.

T_{delete}**:** Memory strands which remove unnecessary memory strands are stored in T_{dummy}.

T_{detect}**:** Single strands which denote the lowest bit are stored in T_{detect}.

First of all, substeps (1-1) and (1-2) consist of the following manipulations.

Substep (1-1)
Separation$(T_{input}, \{A_n\}, T_n)$

Substep (1-2)
Copy(T_{input}, T_{tmp_1})
Separation$(T_{tmp_1}, \{1\}, T_1)$

In Substep (1-3), we first merge the following test tube T_{con_1} to T_1, and execute *Annealing* and *Denaturation* for T_1. This operations concatenate an auxiliary single strand A_iB_j to a memory strand $S_{i,j}(1)$ so that the memory strands can be concatenated in ascending order. (We obtain a single strand $S_{i,j}(1)A_iB_j$ after the operations.)

$$\begin{aligned}
T_{con_1} = \{&\overline{A_iB_jC_0C_11D_0A_iB_j},\\
&A_iB_j \mid 0 \le i \le n-1, 0 \le j \le m-1\}
\end{aligned}$$

Then, we move single strands, whose length k, from T_1 to T_{tmp_1}, where k is the length of a single strand $S_{i,j}(1)A_iB_j$.

Next, we merge the following test tube T_{con_2} to T_{tmp_1}, and execute *Annealing* and *Denaturation* for T_{tmp_1}.

$$T_{con_2} = \{\overline{A_xB_jD_1A_yB_j} \mid 0 \le i \le n-1, 0 \le j \le m-1\}$$

Since single strands in T_{con_2} concatenate the above strands in ascending order, we can concatenate all possible pairs of the above strands as a long single strand. Then, using *Separation* with a symbol D_1, we move the strands from T_{tmp_1} to T_{tmp_2}. Details of the substep are given below.

Substep (1-3)

```
    $Empty(T_{tmp_1})$
    $Merge(T_1, T_{con_1})$
    $Annealing(T_1)$
    $Denaturation(T_1)$
    $Selection(T_1, k, T_{tmp_1})$

    $Merge(T_{tmp_1}, T_{con_2})$
    $Annealing(T_{tmp_1})$
    $Denaturation(T_{tmp_1})$
    $Empty(T_{tmp_2})$
    $Separation(T_{tmp_1}, \{D_1\}, T_{tmp_2})$
```

In Substep (2-1), we first merge the following test tube T_{con_3} to T_{tmp_2}, and execute *Annealing* and *Denaturation* for T_{tmp_2}.

$$T_{con_3} = \{\overline{B_j\alpha^iD_0\beta_{i,j}},\ \alpha^iD_0\beta_{i,j} \mid 0 \le i \le n-1, 0 \le j \le m-1\}$$

In the above description, α is a single strand whose length is equal to $S_{i,j}A_iB_j$, and α^i means a single strand $\alpha\alpha\cdots\alpha$ such that α is repeated i times. In addition, $\beta_{i,j}$ is a single strand such that,

$$\beta_{i,j} = S_{(n+1)j+2n+1-i,\log_2 n}S_{(n+1)j+2n+1-i,\log_2 n-1} \cdots S_{(n+1)j+2n+1-i,0},$$

and means a concatenated memory strands which denote a binary value $n-i$ such that $V_{(n+1)j+2n+1-i} = n-i$. Then, we move single strands whose length of $|\alpha| \times n + |\beta|$ from T_{tmp_2} to T_{tmp_1}. where $|\alpha|$ and $|\beta|$ are length of α and β, respectively. We summarize the substep as follows.

Substep (2-1)

```
    $Empty(T_{tmp_1})$
    $Merge(T_{tmp_2}, T_{con3})$
    $Annealing(T_{tmp_2})$
    $Denaturation(T_{tmp_2})$
    $Selection(T_{tmp_2}, |\alpha| \times n + |\beta|, T_{tmp_1})$
```

In Substep (2-2), we first merge $\{\overline{D_0D_1}\}$ to T_{tmp_1}, and execute *Annealing*, *Cleavage* and *Denaturation* for T_{tmp_1}. Then, we move single strands, whose length is k', from T_{tmp_1} to T_{max}, where k' is the length of a memory strand.

Next, we use the following test tubes T_{dummy} and T_{delete}. The memory strands of all addresses are stored in the following test tube T_{dummy}.

$$T_{dummy} = \{S_{i,j}(0) \mid n \le i \le (n+1)m+n-1, 0 \le j \le \log_2(n-1)\}$$

The test tube T_{dummy} is prepared because all addresses are required to execute *MaxOperation*. The values of all memory strands in T_{dummy} are set to 0. In this substep, we remove memory strands $S_{i,j}(0)$ in T_{dummy} such that there is a memory strands $S_{i,j}(1)$ in T_{max}, and then, merge memory strands in T_{dummy} into T_{max}. To achieve this procedure, we use single strands stored in the following test tube T_{delete}.

$$T_{delete} = \{\overline{\#D_1A_iB_jC_0C_1 1} \mid n \le i \le (n+1)m+n-1, 0 \le y \le \log_2(n-1)\}$$

Details of this substep is as follows. We first copy T_{max} to a test tube T_{tmp_1}, and then, move memory strands, whose values are 1, from T_{tmp_1} to T_{tmp_2}. Next, we merge T_{delete} into T_{tmp_2}, and execute *Annealing*, *Cleavage* and *Denaturation* for T_{tmp_2}. After the above operation, there exists a single strand $\overline{\#D_1A_iB_jC_0}$ if and only if $S_{i,j}(1)$ is in T_{max}. Finally, we merge the above single strands to T_{dummy}, and execute *Annealing*, *Denaturation* and *Separation*. Then, we can remove memory strands $S_{i,j}(0)$ in T_{dummy} such that there is a memory strands $S_{i,j}(1)$ in T_{max}.

Finally, we execute *MaxOperation* for addresses A_n, A_{n+1}, $\cdots$,$A_{(n+1)m+n-1}$, and store the output in T_{out}. Details of the substep are given below.

Substep (2-2)

```
    $Merge(T_{tmp_1}, \{\overline{D_0D_1}\})$
    $Annealing(T_{tmp_1})$
    $Cleavage(T_{tmp_1}, \{D_0D_1\})$
    $Denaturation(T_{tmp_1})$
    $Selection(T_{tmp_1}, l, T_{max})$

    $Empty(T_{tmp_1}), Empty(T_{tmp_2})$
    $Copy(T_{max}, T_{tmp_1})$
    $Separation(T_{tmp_1}, \{1\}, T_{tmp_2})$
    $Merge(T_{tmp_2}, T_{delete})$
    $Annealing(T_{tmp_2})$
    $Cleavage(T_{tmp_2}, \{C_0C_1\})$
    $Denaturation(T_{tmp_2})$
    $Empty(T_{tmp_1})$
    $Selection(T_{tmp_2}, x, T_{tmp_1})$
```

$S_{(n+1)j+n,\log_2 n}$	$S_{n,j}$	$S_{(n+1)j+n,\log_2 n}$	$S_{n,j}$
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

Figure 3: A truth table for *LogicOperation*

$Merge(T_{dummy}, T_{tmp_1})$
$Merge(T_{dummy}, \{\#\})$
$Annealing(T_{dummy})$
$Denaturation(T_{dummy})$
$Separation(T_{dummy}, \{\#, \overline{\#}\}, T_{trash})$
$Merge(T_{max}, T_{dummy})$

$MaxOperation(T_{max}, T_{out})$

In Substep (3-1), we first merge the test tube T_{out} to T_n, and execute *LogicOperation*, which is defined by a truth table in Figure 3, for T_n. The output of *LogicOperation* is stored in T_n.

Substep (3-1)
$Merge(T_n, T_{out})$
$LogicOperation(T_n, L, T_n)$

In Substep (3-2), we first perform *Separation* with a symbol A_n in order to extract the output strands, we move the strands from T_n to T_{tmp_1}. Finally, we merge the test tube T_{tmp_1} to T_{input}.

Substep (3-2)
$Empty(T_{tmp_1})$
$Separation(T_n, \{A_n\}, T_{tmp_1})$
$Merge(T_{input}, T_{tmp_1})$

We now consider complexity of the above procedure. Each step of the procedure consists of a constant number of DNA manipulations described in Section 2, and $O(mn^2)$ kinds of strands are used in the procedure. Then, we obtain the following theorem.

Theorem 2 *Procedure EX-OR, which computes EX-OR function of* n *binary numbers of* m *bits, runs in* $O(1)$ *steps using* $O(mn^2)$ *different DNA strands.* □

The procedure *EX-OR* are easily applied to other multiple input functions, such as, majority and threshold functions by modifying substep (3-2). Thus, we can compute such functions with the same complexity.

5 Conclusions

In this paper, we propose procedures for computing multiple input logic. We first show a simple procedure for computing AND function, which runs in $O(1)$ steps for n binary numbers. We next propose a procedure for the EX-OR function, which runs in $O(1)$ steps for n binary numbers.

However, every DNA manipulation used in the model has been already realized in lab level, and some procedures can be implemented practically. Since logic and arithmetic operations are primitive and important, we believe that our results play an important role in the future DNA computing.

References

[1] L.M. Adleman. Computing with DNA. *Scientific American*, Vol. 279, No. 2, pp. 54–61, 1998.

[2] E.B. Baum and D. Boneh. Running dynamic programming algorithms on a DNA computer. *Proceedings of the Second Annual Meeting on DNA Based Computers*, 1996.

[3] P. Frisco. Parallel arithmetic with splicing. *Romanian Journal of Information Science and Technology(ROMJIST)*, Vol. 2, No. 3, pp. 113–128, 2000.

[4] A. Fujiwara, K. Matsumoto, and W. Chen. Addressable procedures for logic and arithmetic operations with DNA strands. *5th Workshop on Advances in Parallel and Distributed Computational Models*, 2003. (to appear)

[5] F. Guarnieri, M. Fliss, and C. Bancroft. Making DNA add. *Science*, Vol. 273(5272), pp. 220–223, 1996.

[6] V. Gupta, S. Parthasarathy, and M.J. Zaki. Arithmetic and logic operations with DNA. *Proceedings of the 3rd DIMACS Workshop on DNA Based Computers*, pp. 212–220, 1997.

[7] H. Hug and R. Schuler. DNA-based parallel computation of simple arithmetic. *Proceedings of the 7th International Meeting on DNA Based Computers(DNA7)*, pp. 159–166, 2001.

[8] S. Kamio, A. Takehara, A. Fujiwara. Procedures for Computing the Maximum with DNA strands. *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, Vol. 1, pp. 351–357, 2003.

[9] R.J. Lipton. DNA solution of hard computational problems. *Science*, Vol. 268, pp. 542–545, 1995.

[10] Q. Ouyang, P.D. Kaplan, S.Liu, and A. Libchaber. DNA solution of the maximal clique problem. *Science*, Vol. 278, pp. 446–449, 1997.

[11] G. Păun, G. Rozeberg, and A. Salomaa. *DNA computing*. Springer-Verlag, 1998.

[12] Z.F. Qiu and M. Lu. Arithmetic and logic operations for DNA computers. *Proceedings of the Second IASTED International conference on Parallel and Distributed Computing and Networks*, pp. 481–486, 1998.

[13] Z.F. Qiu and M. Lu. Take advantage of the computing power of DNA computers. In *Proceedings of the Third Workshop on Bio-Inspired Solutions to Parallel Processing Problems, IPDPS 2000 Workshops*, pp. 570–577, 2000.

[14] J.H. Reif. Parallel biomolecular computation: Models and simulations. *Algorithmica*, Vol. 25, No. 2-3, pp. 142–175, 1995.

[15] H. Yoshida and A. Suyama. Solution to 3-SAT by breadth first search. *American Mathematical Society*, pp. 9–22, 2000.