A Complete Programming Environment for DNA Computation

Steven Carroll {scarroll@uiuc.edu}
University of Illinois at Urbana-Champaign

Abstract

The Sticker model is one of the most complete models for DNA computation. This model provides a small number of primitive operations that can be combined to form powerful programs. This paper describes a language, compiler and simulator for developing, debugging and evaluating DNA algorithms. The language is based on ANSI C with extensions for modifying test tubes and manipulating DNA bit strings. A discussion of the future role of compiler technology in DNA computation is also provided.

Introduction

Many researchers are now convinced that the much-heralded end of the line for silicon based computing is finally approaching. Many alternatives have been proposed in the literature, but one of the most promising of these non-silicon technologies is biological computation. In this paper, we will focus on computation based on DNA manipulations. DNA computation is attractive because it would be possible to operate on many strands simultaneously, resulting in massive parallelism. The relatively small size of the DNA molecules is another major advantage. Looking even farther in the future, biological computers are naturals for interfacing with living organisms.

Within the realm of DNA computation, one of the most complete and well-defined computation models is the Sticker Model for DNA Computation that was proposed by Roweis and Winfree[1]. Individual strands of DNA are used to represent bit streams and biological processes allow these strands to be manipulated in parallel. In general, all of the possible solutions are represented in a tube and operations are performed to weed out invalid ones. There are only a small number of elementary operations are possible in the sticker model, but these operations can be combined to form a fairly robust set of bit string manipulations. However, one of the major problems that must be dealt with in DNA algorithm design is the lossy nature of biological computation.

Therefore, any DNA algorithm designer must be concerned with two major problems: how to synthesize complex algorithms out of only a few basic operations, and how to evaluate the performance of those algorithms in the presence of errors to assure that an acceptable solution will be found in most executions. The goal of this work is to create tools to assist the DNA computer programmer in solving these two problems. First, a simulator was designed to simulate a sticker model based DNA computer on traditional hardware. The design of a proposed machine architecture presented in the original sticker model paper was flushed out to form a complete ISA. The common errors that would occur during execution of a DNA program are also simulated and the probabilities of these errors can be changed to reflect improvements in the reliability of the machine apparatus and the biological processes.

In addition to the simulator, a high level language called DNA C was developed for expressing the manipulation of the DNA strands in the machine in a more intuitive way. A compiler was constructed to create assembly files that could be read by the simulator by translating the higher-level tube manipulations into low-level operations.

The compiler and simulator were tested using a solver for the NP-Complete problem "minimal set covering". This algorithm, which has no known polynomial time solution in traditional computational hardware, can be solved in polynomial time in the DNA computer because all possible solutions can be operated on simultaneously. A DES decryption algorithm's components were also taken into account when the language was designed [2]. Therefore we are confident that the language is expressive enough to describe most currently proposed DNA algorithms.

The organization of this paper is as follows: In the Background section, the details of the sticker model, the sticker model computer and the error models will be presented. Next, the design of the simulator will be detailed in the Simulator section. The Language Design section describes the modifications to the ANSI C specification that define the DNA C language. The Compiler Design section presents the details of the implementation of the DNACC compiler and the translation of the high level expressions to the basic operations of the simulator. The Results section gives the simulation results of the minimal set cover solver. Potential

extensions and lessons learned from this work will be evaluated in the Discussion section.

Background

Sticker Model

DNA Computation is still an immature field and far from producing viable machines, however in this section we will examine the basic biological processes that allow DNA computation to take place. DNA is composed of strands of 4 bases (Adenine, Thymine, Cytosine, and Guanine) and each of these bases has a complement: A matches T, T matches A, C matches G, and G matches C. Figure 1 depicts bonding between single stranded DNA molecules. Notice that the second strand aligns itself with the first strand at a position such that they are exactly complementary. This bonding process is the primary mechanism used in the Sticker model of DNA computation.

The sticker model breaks the DNA strand into bits where each bit can be composed of several bases. The subsequences that make up each bit of the computation must be unique and the bit subsequences cannot appear anywhere else in the strand. A sticker is the complement of one of the bit subsequences. Since the bit subsequence is not repeated, if a sticker is introduced to the test tube with the DNA strand, it is guaranteed to bond only to the bit it complements. An example of a subdivided DNA strand for computation is given in Figure 1. Since a sticker is attached at bit positions 0 and 2, this configuration would represent the binary value 101. The properties of comma-free codes, commonly used in reliable communications can be used to design DNA strands with the necessary properties.

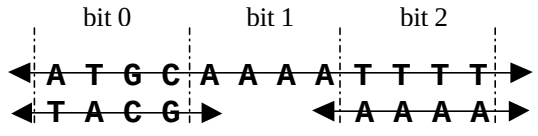

Figure 1- DNA strand divided into bits with stickers

The sticker model is desirable for computation because many such strands could be contained within a single tube. There is also massive data parallelism inherent in this system because all of the strands within a tube can be operated on at once. For example, by adding a large quantity of stickers for a given bit to the tube, that bit would become set on every strand in the tube simultaneously. By constructing one strand for each possible solution to a problem and operating on all possible solutions simultaneously we are effectively trading exponential time computation for exponential space computation. However, the exponential space is more desirable because the size of the data molecules is so small. As we shall see, this system would allow the evaluation of NP complete problems to be solved in a polynomial number of operations.

There are a small number of basic operations that can be performed in the sticker model, but these operations can be combined to form a very robust set of computations. The first operation, SET, was discussed briefly above. A large number of sticker molecules for a given bit are introduced into the tube containing the data molecules. Over a period of time, these stickers will adhere to the corresponding bit in the data molecules. The complement operation to SET, RESET, is also available. There has been some debate over the best physical process to utilize for the RESET operation, but for this paper we will assume that so-called "anti-stickers" can be added to the tube causing any stickers at a given bit to fall off.

Next, the SEPARATE operation separates all the strands with a given bit set from those with the bit un-set. So, at the beginning of the operation we have one tube with all data molecules in it. At the end, we have two tubes; one has all the molecules with the given bit set and the other tube has all the molecules that don't have that bit set. The opposite operation to SEPARATE is COMBINE. The COMBINE operation takes two tubes and combines the molecules in the two tubes into a single tube. A related operation is the MOVE operation, which takes all the contents of one tube and transfers them to another tube.

Sticker Model Machine Architecture

The proposed Sticker DNA Computer is depicted in Figure 2. There are 3 tubes connected in series and a pump to drive the contents around the loop. Each data tube has one end that is open and one end that will block the DNA molecules from escaping.

There are three types of data tubes: Data Tubes, Sticker Tubes, Anti-Sticker Tubes. Data Tubes contain the normal DNA molecules and are the operands of most DNA operations. A Sticker tube contains a large number of sticker molecules for a given bit. An Anti-Sticker Tube contains a large number of anti-sticker molecules for a given bit. There are also three different kinds of operator tubes: Blank Tubes, Probe Tubes, and Filter Tubes. Blank Tubes are simply open on both ends and allow DNA molecules to pass through unmolested. Filter Tubes allow stickers and anti-stickers to flow through but

block the data strands from passing. Probe Tubes contain a number of probes, which attach to any molecule that is not set at a given bit thereby preventing it from passing through. The basic operations described above are implemented in the proposed machine as shown in Table 1.

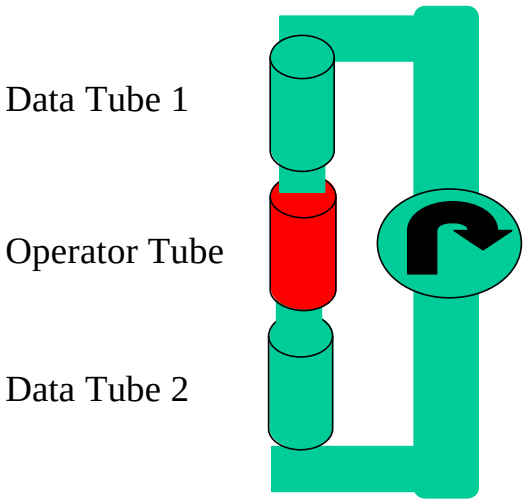

Figure 2 - Proposed Machine Architecture

Error Models

It is important to note that the physical mechanisms involved in the performance of the above operations are not as error free as the basic operations in traditional silicon based computation. DNA molecules can be stuck to the walls of the test tubes and probes. If the stuck molecule is the correct solution to the problem, the outputted solution to the algorithm will be either incorrect or suboptimal. Many other kinds of errors are possible as well. For instance, sometimes stickers will adhere to areas of the strand that are close but not exactly the same as the subsequence they are meant to adhere to. Sometimes stickers will simply come unbound introducing bit errors into the computation.

Simulator Design

The DNA Simulator was developed in Java with the Swing API. It is implemented as a Java Applet so that it can eventually be deployed on a web page that will describe the basics of DNA computation and allow users to submit their own test programs to it. The Simulator design matches the DNA Machine Architecture described above as was described in the original Sticker model paper, but the design is fleshed out with a complete ISA for controlling it. This section will present the details of the complete architecture and its ISA.

Table 1-Basic Operations

Operation	Data Tube 1	Operator Tube	Data Tube 2
SET	Data	Filter	Sticker
RESET	Data	Filter	Anti-Sticker
COMBINE	Data (dest)	Blank	Data
MOVE	Data (dest)	Blank	Data (src)
SEPARATE	Data (off)	Probe	Data (src)

The simulation environment also contains a RISC-like processor simulation for controlling the pump and tubes, a rack of tubes, and a robot arm to manipulate those tubes and put them in the main computer. The rack of tubes consists of data tubes that are addressable by an index number. In addition to the data tubes, there are fixed locations for the operator tubes including one probe tube for each bit position, a blank tube, and a filter tube. There is also one sticker tube and one anti-sticker tube for each bit. Because the total number of SET and RESET operations cannot necessarily be determined in advance at runtime, we assume that each sticker and anti-sticker tube is positioned in the rack beneath vats that can fill the empty sticker tube after it has been used. We believe this design is superior to the alternative, which would just have a sufficiently large number of sticker and anti-sticker tubes. Certainly it makes it simpler to program as the bit position can be used as the index into the sticker tube rack.

There are two separate memory banks: one for data memory and one for instruction memory. The assembly input file sets the size of the banks in the current implementation to simplify code generation. The number of registers is set at 64, which was chosen to be large enough to handle all reasonable programs to simplify register allocation.

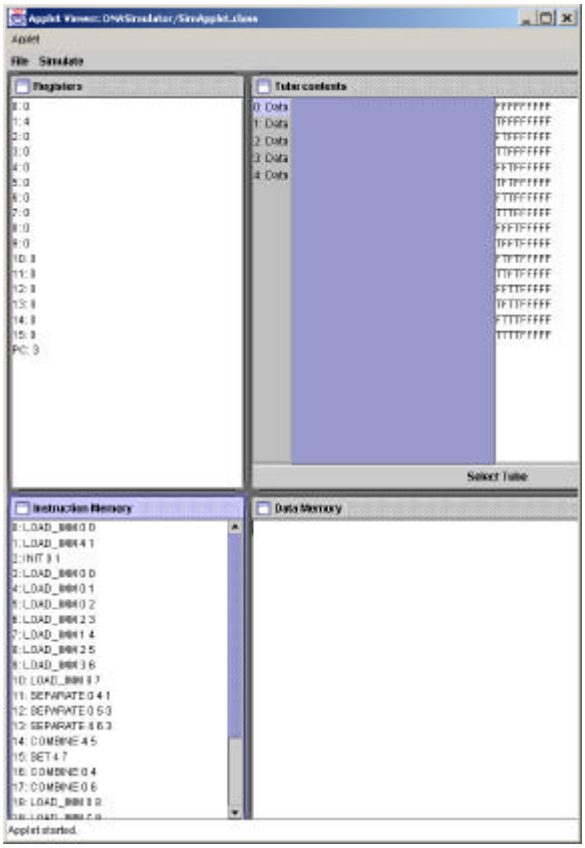

Figure 3- GUI Screenshot

Figure 3 is a screenshot of the DNA Simulator Applet. In the upper left, window is the register information. The register contents and the program counter (PC) are updated after each instruction is executed. The lower left hand window is the Instruction Memory. It displays each instruction stored there and each instruction is assumed to take up only one spot in the memory. The most useful feature in the instruction memory window is that a user can click on any statement (or groups of statements) and set them as breakpoints for debugging purposes. The Data memory is similar to the Register window. Each slot holds one integer value (64 bit). The most interesting window is the tube window. Any data tube can be selected in tube rack and its contents will be displayed to the right side of the window. The right side of the window displays the bits of the strand using an 'F' is no sticker is attached to that bit position and a 'T' if a sticker is attached. If the tube is empty, the word "Empty" is displayed in the window instead.

The Simulate menu allows has three options: Execute runs the entire program from the beginning (stopping at the end or at breakpoints), Step executes only the next instruction, and Continue restarts the program from the current PC after a breakpoint was reached.

Instruction Set Architecture

The processor for the machine has a load-store architecture and all of the standard logical and arithmetical instructions exist. The conditional control flow operations are always preceded by the COMPARE instruction, which sets a series of flags (equal, less than, greater than). The conditional branch instructions then make the control flow decision based on the value of the flags.

There are 6 tube operations in the ISA. Each operand is a register that contains an index into the tube array. MOVE and COMBINE simply take two registers that are indices into the data tube rack and move/combines the second into the first. If the two registers match, the operation maps to a NOP. The SET and RESET operations also take two registers. The first is an index into the data tube rack. The second is an index into the (anti-)sticker rack. SEPARATE takes three register operands: source tube, "on" tube, and probe tube. The source tube has the initial data set. The on tube gets all the data with the bit turned on. The probe tube is an index into the probe tube rack. Finally, the data strands with the given bit (specified by the probe tube) are moved back to the source.

The final operation is INIT, which takes two register operands, one is the index of a data tube, the second holds the number of bits (m, where n is the number of bits in the strand and $m <= n$) to initialize. The simulator then adds 2^m strands where the last (m-n) bits are zero. In other words, if $m = 3$ and $n = 6$, the tube would contain the following eight strands: (000 000), (001 000), (010 000), (011 000), (100 000), (101 000), (110 000), and (111 000). This instruction is used to create an initial data set to start the computation. It is not clear from the original paper whether this would be synthesized from basic operations or if it would be assembled outside of the machine and simply place the appropriate tube in the data tube rack.

Language Design

The name of the high level language developed for this project is DNA C. It is a variant of the ANSI C specification. A C variant was chosen because it would be the easiest for most C programmers to learn and the compilation techniques for C-like functional languages are well understood. A number of C concepts have been removed to make the compiler development easier for our prototype. (structs, unions, floats, and pointers). The size of tube arrays must be static because it would be difficult to actually create a rack of an unknown number

of test tubes. The rest of the C structures behave exactly as they do in a normal C program.

One new type has been added to represent data tubes. Each data tube has an array-like syntax called bit references, "`<|`*`integer-expression`*`|>`", for operating on the bits of the strands contained within it, but there is no way to access the individual strands within the tube because this is physically impossible in the machine described above. Otherwise, the tubes are treated much like any other type. Arrays of any number of dimensions can be defined with tube elements. In this section, we will examine all of the new tube expressions that have been added for DNA C. The translation of these expressions into simulator ISA operations will be discussed in the Compiler Design section.

Tube Declarations

Ex:

```
tube t<|8|>;
tube tube_array[3]<|8|>;
tube t_no_length; // illegal
```

Each tube must have a length in bits declared with it. Arrays are declared normally with a trailing size expression.

Tube Initialization

Ex:

```
t1 init 3;
```

This expression maps directly to the simulators INIT opcode.

Bit Assignment

Ex:

```
t<|I|> = (A > 35);
t<|0|> = 0;
t<|1|> = 1;
```

Bit assignments statements are any statement where the left hand side is a bit reference and the right hand side is a Boolean expression. The Boolean expression is evaluated and if the value is true (non-zero), the bit is set in every strand in the tube. Otherwise that bit is cleared in every strand.

Bit Copy

Ex:

```
t<|3|> = t<|5|>;
```

Bit copy statements are any statement where the left and right hand side are both bit references. For each strand in the tube, the value of the right hand bit is copied into the value on the left.

Bit Logic Operations

Ex:

```
t<|3|> = t<|5|> ^ t<|6|>;
t<|2|> = t<|3|> | t<|i|>;
t<|0|> = t<|I|> & t<|I+1|>;
t<|5|> = !t<|2|>;
```

These examples represent XOR, bitwise OR, bitwise AND, and NOT operations, respectively. In the first example, the fifth bit of the tube is XORed with the sixth bit and the result is copied into the third bit for each strand in the tube. More complex expressions are also legal and the operator precedence is the same as these operators have for integers in C.

Tube Combinations

Ex:

```
t1 <- t2;
t3 = t1 + t2;
t1 += t2;
```

The first example adds the contents of `t2` into `t1`. The second example combines the contents of two tubes and stores it in a third tube. The final example is equivalent to the first example. The benefit of the first and last is that the outputted compiled code lacks an extra MOVE operation.

Tube Moves

Ex:

```
t1 = t2;
```

This expression maps directly to the simulators MOVE opcode. The contents of t2 are moved to t1. (It is assumed that t1 is either empty initially, otherwise this instruction is equivalent to a combination.)

Tube Separation

Ex:

```
t_src<|bit|> -> t_on : t_off;
```

The source tube is split into two separate tubes based on the bit number of the bit reference. The strands in the source tube with that bit set are deposited in tube t_on and the rest are put in t_off. There is currently a restriction that the "off" tube must be distinct from the source tube.

Compiler Design

The DNACC compiler is a fairly traditional compiler design. The freely available programs flex and bison (the GNU versions of the traditional compiler tools lex and yacc) were used to construct a lexical analyzer and syntax analyzer, respectively. We will not dwell too much on the details of the compiler implementation, but briefly, a lexical analyzer takes an input program and splits it into tokens (keywords, identifiers and constants). The next phase of the compiler takes that stream of tokens and recognizes the grammatical structures of that stream. The input grammar is a modified version of a freely available ANSI C grammar. As each rule in the grammar is recognized, the actions defined on that grammar structure builds the elements of an Abstract Syntax Tree (AST) and symbol table information. Syntax based translation is performed on the AST to break it up into a linear sequence of simple machine operations. Finally, the assembly file generator takes this sequence and outputs it in the assembly file format that is recognized by the simulator.

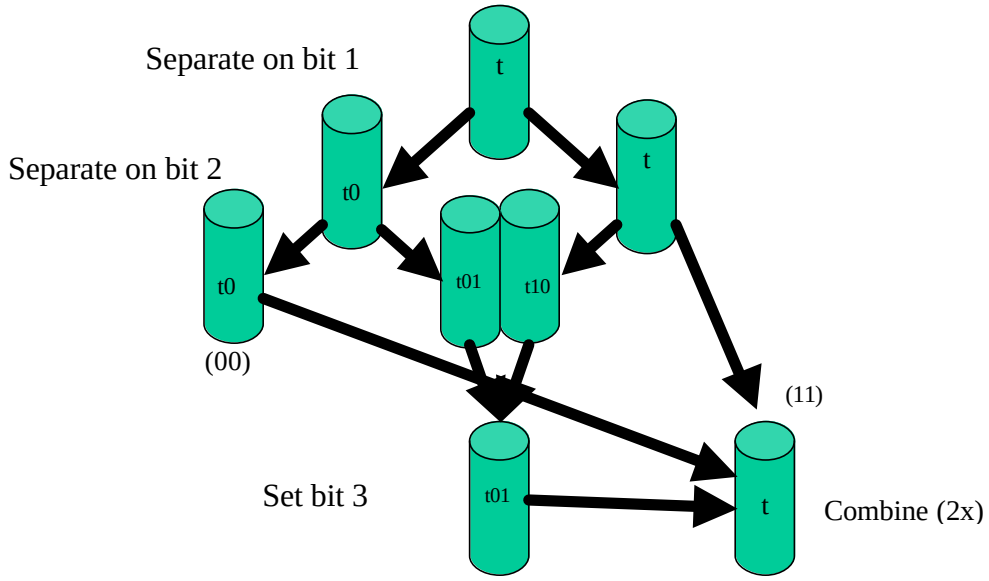

Figure 4 - XOR Translation

Logical Operation Translation

It is not obvious how logical bit operations should be implemented using the sticker model's basic operations, and so the automatic translation of these operations by the compiler really improves the understandability of the DNA C source code significantly. The translation of an XOR operation (`t<|3|> = t<|1|> ^ t<|2|>`) is depicted in Figure 4. First, a separation operation is performed on the bit 1 creating two tubes, one with bit 1 off (`t0`) and one with bit 1 on (`t`). Next, we perform a separation on each of these tubes based on bit 2. Now, we have 4 tubes: `t0` has all the strands with bits 1 and 2 equal to (0,0), `t01` has (0,1) strands, `t10` has (1,0) strands, and the tube `t` has (1,1) strands. Therefore, the tubes `t01` and `t10` contain all the tubes that should evaluate to true for the XOR, so we combine them into a single tube. All of these strands have bit 3 set in them. All the tubes are combined into the original tube t and the operation is complete. The modification of this translation to handle the OR and AND is trivial by simply changing which tubes are combined and set in the last step. It is worth mentioning that the above algorithm assumes that the bit position where the result is being stored has not been "written" to before (i.e. it is unset in all of the strands). If this condition is not met, an extra reset operation must be performed on all of the tubes that have a false result for the logical operation. Whether or not the bit has been written previously can be determined statically be the compiler in some cases, but in our implementation, it is conservatively assumed that all strand spaces have previously been written.

For more complex operations involving several logical operations (i.e. `t<|5|> = t<|3|> & (t<|4|> | t<|7|>)`, the intermediate results of the computation must be stored somewhere. Fortunately, the compiler can easily handle this automatically for the user. Scratch space is allocated at the end of the data strand for compiler temporary storage bits. Other statements in the same program can reuse this storage space for their computations. This reuse is crucial because increasing the size of the strand increases the size of each molecule. Bigger tubes may be necessary as the size of the strand grows.

A bit copy operation is also very similar to the XOR operation listed above. The tube is split into two based on the source bit. The tube with the unset strands has the destination bit reset.

The bit assignment operation is slightly different from the logical operations. First, the Boolean expression on the right hand side is evaluated like any normal integer operation and the result is stored into a register. If that value is non-zero, the destination bit is set, otherwise it is reset, so a simple if-then hammock sequence of operations is emitted. Separation operations and combination operations map directly to machine operations with a few extra instructions to calculate tube rack indices if the

operands are tube arrays. Combinations of the form A = B + C are mapped to a combination (C into B) and a move (B to A).

Results / Status

In this section, we present some run time information for the simulator to give some idea of its performance and usefulness. The presented runtimes are for the minimal set covering algorithm presented in the original Sticker model paper[1]. It finds the minimum number of subsets that must be selected to cover a complete set of objects. The number of bits column is the total number of subsets. For this example, the number of subsets is also the same as the number of total items. Therefore, each of the strand lengths is twice the number of bits. The total number of data strands in the computation is equal to $2^{(number\ of\ bits)}$. The runtimes for the simulated program are given in Table 1. The runtimes are exponentially increasing with the number of bits as expected which puts a limit on the usefulness of the simulator for big computations. Ideally, the simulator will need to be re-implemented to run on a large-scale parallel machine to run bigger simulations. However, the simulator would still be useful for working on smaller data sets for debugging before scaling it up to the full data set when it runs on the actual hardware. For instance, instead of decoding a full 56 bit DES encryption, it would be possible to execute test runs on smaller key size to make sure the error profiles are acceptable and the program is correct.

# of bits	Runtime (secs)
4	27
8	126
16	1306
32	15348

Table 2- Runtime of Minimal Set Cover Program

Discussion/Future Work

In this section, we will discuss the lessons learned about DNA computation from this project and present ideas for future work in this area such as improvements to the machine architecture, the compiler algorithms, etc.

One of the aspects of the language design that seems particularly lacking is the semantics of the "=" operator when dealing with tubes. When C programmers writes the statements "`int j = 4; int i = j;`" they expect that after the execution of the statements, both `i` and `j` will have the value of 4 in them. However, in the tube statement "`tube t_new<|64|> = t_old;`", after the execution of the statement, the tube `t_new` contains all of the strands that were originally in `t_old` and `t_old` is empty. Clearly, the tube assignment does not behave in exactly the way that a C programmer will expect it to behave. A better syntax perhaps would be to simply use the "`<-`" operator that was previously used as a combine operator. If the tube were initially empty, the meaning of the statement "`t_new <- t_old`" would be equivalent to the original move syntax, but if `t_new` had strands in it to before execution, the statement would be a combine. In either case the symbol "`<-`" does a good job of visually suggesting the movement of data from one location to the other instead of the mathematical equivalence that the "=" operator suggested.

The optimization of code in the DNA computer is different than that of a traditional computer. In the traditional compiler, the key optimizations focus on such ideas as register allocation, instruction level parallelism, and strength reduction (replacing expensive operations with less expensive equivalent operations). However, in the DNA computer, the computation time of the biological processes are many orders of magnitude slower than any of the processor instructions' execution time. Therefore, the contribution of the processor's speed to the total execution time is negligible and therefore optimizing the traditional instruction part of the program is irrelevant. Even among the tube operations, the move and the combines are simple mechanical operations that can be completed in a fraction of the time that the separate, set and reset operations can be completed in. Therefore, when optimizing for the DNA computer, the only important factors are the number of separates, sets, resets, and the length of the DNA strand. Keeping the number of tubes to a minimum can be beneficial in keeping the size of the program small. Test tubes can be large (especially for big data sets) and so keeping them to a minimum will reduce the size of the overall machine and probably the machine's cost.

One of the most important areas for optimization is what we will call tube level parallelism (TLP). Let's examine the XOR operations again. The complete operation consists of 3 separates and 1 set operations. There will also be 1 more reset operation if the bit has been previously set. Without optimization, one XOR would require 5 biological steps. The second and third separations can be done simultaneously. The set and reset can also be done simultaneously. By introducing the tube level parallelism, we have reduced the number of steps to 3

biological steps. However, in order to support this idea, the simulator will need to be extended to issue the two instructions simultaneously. The best way to do this is probably to use explicit parallelization by marking the instructions that can be issued simultaneously.

The key to reducing the number of data tubes in the machine is to reuse the temporary data tubes that are allocated by the compiler. For instance, in the XOR operation detailed above, 3 temporary tubes, `t0`, `t10`, and `t01` must be allocated for holding the strands temporarily during the XOR operation. Its important to reuse these same tubes in all of the logic operations to avoid unnecessary extra tubes.

One of the main missing components in the current implementation that will need to be expanded upon before the simulator is complete is a strand generator unit [3]. The pattern of the strands must be carefully chosen so that a good tradeoff is made between strand length, and the number of bases per bit. The strand length directly effects the size of the DNA molecules and therefore the total size of the data set. However, the more bases there are in each bit, the more unique each bit sequence is and therefore the number of errors in sticker bondings will go down. However, the more bases there are per bit, the longer the strand will be for the same number of bits. A good tradeoff algorithm could be implemented in the compiler so that it automatically generates a strand sequence for the program based on the total number of bits that are used in the program (user defined bits + compiler generated temporary scratch bits). The outputted strand could then be created and duplicated for the data tubes. There is currently no strand generator in the DNA compiler.

Another important optimization role that the compiler could fulfill would be automatically inserting the redundancy necessary to assure low error rates. For instance, it could create two or more copies of each strand tube and operate on them separately (but in parallel, taking advantage of the tube level parallelism). Only solutions that appear in all copies would be considered correct. Copies of bits within the same strand would also be valuable. A simple XOR expression would allow all copies with errors between the two bits, which should be identical to be separated out and discarded.

Conclusion

In this paper, we have presented a simulator, high-level language and compiler for DNA computation. These tools are very useful for evaluating the usefulness of the DNA sticker computation model for performing different kinds of computation. The DNA computation model is unique in that none of the individual strands can be examined during the computation, operations can only be performed only on entire tubes. I believe it is yet to be determined whether or not the model as it stands is robust enough to do more than special purpose operations. The ease of use of DNA C will allow researchers to develop DNA algorithms and see if they suit their needs.

Acknowledgments

This work was supported in part by NSF grant EIA 99-75019 with research support from NSA, and a grant from Intel Corp. The views of this paper do not necessarily reflect the views of the funding agencies.

References

[1] A Sticker Based Model for DNA Computation by Sam Roweis, Erik Winfree, Richard Burgoyne, Nickolas V. Chelyapov, Myron F. Goodman, Paul W. K. Rothemund, and Leonard M. Adleman. In DNA Based Computers: DMACS Workshop, 1996.

[2] On Applying Molecular Computation To The Data Encryption Standard by Leonard M. Aldeman, Paul W. K. Rothemund, Sam Roweis, and Erik Winfree. In DNA Based Computers: DMACS Workshop, 1996.

[3] "DNA Sequences Useful for Computation," by Eric Baum In DNA Based Computers: DMACS Workshop, 1997.