MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1588 October 1996 Revised

Parallel Function Application on a DNA Substrate

Andrew J. Blumberg

Abstract

In this paper I present a new model that employs a biological (specifically DNA-based) substrate for performing computation. Specifically, I describe strategies for performing parallel function application in the DNA-computing models described by Adelman [1, 2], Cai *et. al.*[4], and Liu *et. al.*[8]. Employing only DNA operations which can presently be performed, I discuss some direct algorithms for computing a variety of useful mathematical functions on DNA, culminating in an algorithm for minimizing an arbitrary continuous function. In addition, computing genetic algorithms on a DNA substrate is briefly discussed.

Keywords: DNA Computing, Parallel Computation, Parallel Architecture

Contents

1 Introduction

Recently there has been some very exciting work in performing computation using DNA and some tools of molecular biology. For the most part, this research has focused on a constraint-based model for solving search problems. That is, most of the computation models and results have operated in the following manner—DNA strands are synthesized to contain encodings of all possible solutions of the problem in question. Then, the multi-set of strands is filtered to remove all strands not satisfying some constraint. This process is repeated until the only strands that have satisfied all of the constraints (and hence remain in the test-tube or on the adhesion surface) are solutions to the problem. The advantage in this model of the DNA substrate is that the number of filtering operations depends on the complexity of the problem (*e.g.* the depth of the circuit or the size of the graph) but not on the possible number of solutions tested.

Unfortunately, for a number of reasons I believe that this conception of DNA computation will (at least in its present form) be of relatively limited utility. Some of the reasoning is simple—for problems which suffer exponential growth in the space of possible solutions as the size of the problem grows (*e.g.* NP-complete problems), the size of the possible solution space rapidly outstrips the maximum number of DNA strands that can fit in a milliliter of solution (roughly 10^{20}). For example, in the case of the satisfiability problem (deciding on the existence of a satisfying assignment for a logical expression in CNF form), a milliliter of solution will suffice to check at most about 70 variables ($2^{70} \approx 10^{20}$), assuming ideal conditions. Increasing working volume will not help, as the exponential growth of the problem will rapidly outstrip the linear gains to be had from the expanded computing volume and additionally the increased volume will decrease the reaction rates because of increased path length between molecules in solution.

Furthermore, the traditional model of DNA computation phrases computation as a passive process of filtering away incorrect solutions until the correct one is reached. For many problems, it is vastly easier to express the computation as a process of applicative function computation. For example, consider the problem of attempting to minimize a continuous function. It is not clear how to express this in a constraint formalism. However, when phrased in terms of function computation, many solutions are possible on parallel hardware.

As such, it is extremely important to develop a model of DNA computation which allows easy implementation of general purpose parallel computation. Quite recently there were some steps in this direction [5] in which an algorithm for addition with DNA was demonstrated. While very clever, this algorithm suffers from a number of operational defects. First of all, it is not a parallel algorithm in the sense that in a given test-tube only one addition can occur. This appears to throw away the greatest asset of DNA based computation, massive parallelism. Furthermore, the output of the algorithm is not in the same form as the input; a nontrivial transformation has to occur before the algorithm can be run again. This paper is intended to serve as a guideline for research in this area. We develop a superior model of parallel computation which could be implemented using today's technology. However, I do not claim that this is necessarily the best way to do this–but it is one way and perhaps elements of this solution will prove useful in constructing a final solution.

In the following sections I detail an abstract model of DNA computation in which the process of DNA computation is envisioned as a process of transformation of the strands; that is, parallel independent function evaluation. In the model, the same function is evaluated in parallel on different data (where the strands encode the data). Thus, this model can be thought of as embodying SIMD (single-instruction multiple-data computation).

I wish to emphasize the fact that the abstract model to be presented in the following sections is bound to actual DNA computation in the sense that all of the operations described have already been performed in an experimental setting and discussed in the literature. This model is a specification for computation bound to an implementation substrate, not simply a metaphor for computation at a DNA level. Of course, the model is not restricted to work only on the traditional methods of manipulating DNA—but it can be implemented in that domain (and I expect that the initial set of experiments to confirm and explore the models described herein will be executed in that domain).

2 DNA Computing Model

The base instruction set of the computational model will be grounded on a DNA substrate as follows. A "legal" strand of DNA will correspond to a bit-string using the variable-length word encoding method discussed in Adleman [1] (where the strand consists of *<bit-label-0> <bit-value-0> <bit-label-0> <bit-label-1> <bit-value-1>* ...). In general, a strand will correspond to a specific processing element.

Any tube of DNA manipulated in the following discussions is assumed to contain a multiset populated by legal stands of DNA distributed according to the specific problem. The following operations can be performed on such a tube of DNA :

1. Separate via constraints : In this operation a single tube of DNA is split into two tubes based on the given constraint; all strands satisfying the constraint in one tube, all strands failing to satisfy the constraint in the other tube. These constraints at the physical level are phrased in terms of the presence or absence of base pairs in the strand. However, we will typically consider "higher-order" constraints which refer to bits in the representation encoded on the strand, for conciseness.
2. Merge : Two tubes are mixed together to form one tube.
3. Append (bit or tag): Appends to all strands in a tube the specified object. Typically either bits or tags are appended. A bit is a triple consisting of <bit-label> <bit-value> <bit-label> as detailed above. A tag is simply a unique sequence of base pairs which we use to distinguish different parts of the strand. In discussions to follow, tags will be referred to with names like "NEW". These names are for convenience of discussion only and do not make any implications about the bases which make up the tag.

   These both are higher-order operations built on the basic physical operation of ligating a base.
4. Cut on tag : Cuts all strands in a tube along the specified tag. Actually performed using restriction enzymes.

5. Detect : Detects if there is any DNA present in a tube at all.

All of these operations can presently be performed in an experimental setting—*e.g.* this model can be implemented with current DNA technology. Also, although these operations (and the rest of this discussion) are phrased in terms of Adleman's [1, 2] solution-based embodiment of DNA computing, it is straightforward to translate to the surface-based embodiment discussed in Liu *et. al.*[8] and Cai *et. al.*[4]. In this case, the merge operation is unnecessary and separate is replaced by mark (with the other operations being specialized to operate only on marked strands). The algorithms provided below must be changed trivially to handle these changes to the underlying model.

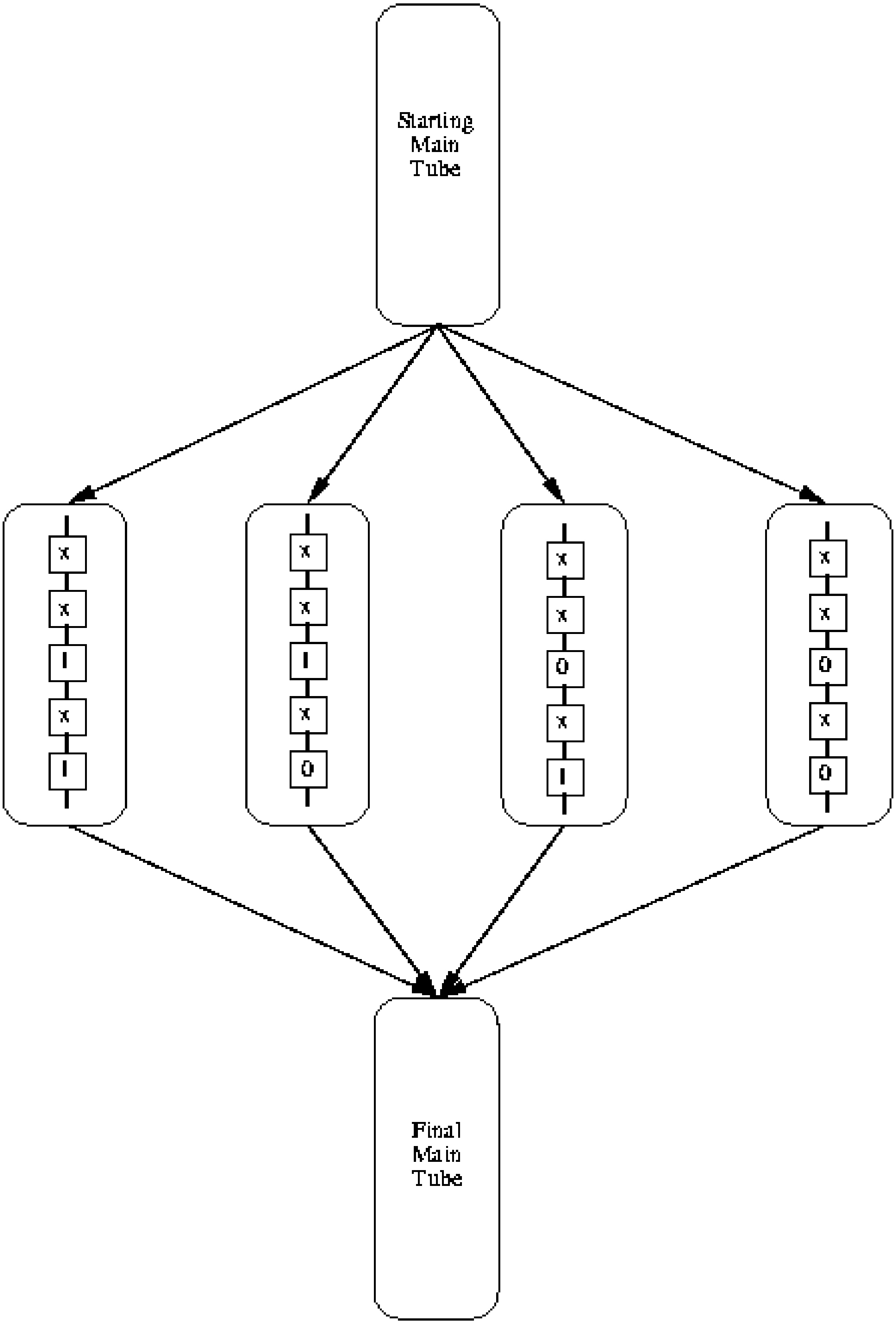

Figure 1

3 Basic Algorithmic Structure

To describe the algorithms for computation, I need to clarify the exact level at which the algorithm is operating. In the traditional model, the instruction stream of the algorithm is embodied wholly in the physical processes going on. The data varies across strands, but the same thing is done to all strands at every step of the computation–the instruction stream is fixed.

Implementing traditional parallel computation seems to depend on being able to modify the instruction stream depending on certain properties of the computational element. Since this is not practical at the physical level (we don't want to be testing the tube and then performing different physical operations) the modification of the instruction stream has to be done by separating the set of strands into a small number of different tubes. For each of these sub-tubes, a distinct activity will be performed and then the tubes are merged together again. This notion turns out to be very powerful–different things happen depending on specifics of the strands, but it is never necessary to explicitly investigate the nature of any of the strands.

Essentially, this process takes advantage of localization of computational dependence. Consider data represented as a string of bits. If we can know a priori that the result of some computation on this string of bits only depends on the values of bits n and m, then we can perform this computation in the DNA model easily as follows. We separate all of the strands into four tubes, depending on the pairs of values of bits n and m. Then, for each of the four sub-tubes we perform whatever operation is indicated by the bit values which we know hold over all strands in the given sub-tube. Typically this will involve concatenating some value to the end of the strands. Finally, we merge all of the sub-tubes together again and perform the next operation.

The number of separations performed depends only on the algorithm and the length of the computation, not on any specific details of strands. Of course, the number of sub-tubes required increases exponentially in the number of dependent bits, so it is advantageous to keep the number of bits needed small. Fortunately, this turns out not to be a particularly restrictive constraint.

3.1 Logical Operators

Specify a logical operator, a multiset of n bit strings, and a single n bit string. The following operation will implement the combination (via the logical operator) of the single bit string with each member of the multiset.

Begin with the main tube containing strands representing the multiset of n bit strings.

1. Let $count = 0$ (note this is a bookkeeping step - no tube operations are involved).
2. Append a tag ("NEW") to all strands.

3. Separate into two tubes (A and B) based on the value of bit *count* (0 and 1 respectively).

4. (a) Compute a bit by applying the logical operator to 0 and bit *count* of the argument. Append this bit to all strands in tube A.
   (b) Compute a bit by applying the logical operator to 1 and bit *count* of the argument. Append this bit to all strands in tube B.

5. Merge the tubes A and B.

6. Increment *count*.

7. If $count < n$, goto step 3. Otherwise, fall through to step 8.

8. Cut all strands at the tag "NEW".

9. Separate into tubes OLD and NEW based on the presence or absence or tag "NEW"

In the tube labelled NEW will be the result of applying the logical operator and the argument to all of the strands in parallel. Note that the number of steps in this algorithm is $O(n)$ in the number of bits in the string but independent of the number of strands. The tube NEW can now be used as the starting tube (perhaps after amplification using PCR to ensure enough copies of strands) for another operation.

It is possible to convert this algorithm so as to take both arguments from the strands—*e.g.* let each strand consist of the concatenation of the arguments. In this case there are four separation tubes (based on the possible pairings of values at each bit-position) but otherwise the algorithm remains the same. In this case the output will need to have the next argument concatenated to it to continue (although we can apply the uniform operation with global argument to the output at this point with no further processing).

3.2 Addition

The logical operation algorithm was perhaps the simplest kind of algorithm of the class we are presenting, in that the value of an output bit depended only on the corresponding input bits. The addition algorithm is somewhat more involved, as we must propagate a carry bit throughout the computation. Nonetheless, it is fundamentally similar.

The specification is identical to the specification of the logical operator algorithm (except that the operator here is fixed to be addition). Start with the working tube filled with strands representing the multiset.

1. Let $count = 0$ (note this is a bookkeeping step - no tube operations are involved).

2. Append a tag ("NEW") to all strands.

3. Separate into two tubes (C and notC) based on the presence of the tag "CARRY".

4. Separate both C and notC into two tubes based on the value of bit *count* (0 and 1 respectively).

5. (a) To the two tubes from C, remove the tag "CARRY" (this can be done by cutting and separating, since nothing follows this tag). Add 1, bit *count* of the argument, and 0 or 1 (depending on the tube) to get a result bit (and a carry bit). Append this result bit to the tube. Append the tag "CARRY" if the computed carry bit is 1.

   (b) To the two tubes from notC, add bit *count* of the argument and 0 or 1 (depending on the tube) to get a result bit (and a carry bit). Append this result bit to the tube. Append the tag "CARRY" if the computed carry bit is 1.

6. Merge all tubes.

7. Increment *count*.

8. If $count < n$, goto step 3. Otherwise, fall through to step 8.

9. Separate into two tubes (C and notC) based on the presence of the tag "CARRY".

10. Remove the tag "CARRY" from tube C (done as described above). Append a 1 to all strands.

11. Merge tubes.

12. Separate into tubes OLD and NEW based on the presence or absence or tag "NEW"

Again note that the number of separations is $O(n)$ in the number of bits in the strings and independent of the number of parallel additions carried out (*i.e.* the number of strands of DNA). It is clear that this algorithm could also be transformed to perform additions where both arguments come from the bit string (as described in the logical operations algorithm), although this new algorithm will be rather involved.

3.3 Multiplication and Polynomials

Multiplication can be carried out in a number of fashions. There is a direct implementation in the style of the addition algorithm. This is not described herein as it is very messy and not particularly illuminating (the construction should be fairly obvious). Another possibility is to use the traditional algorithm for multiplication taught in schools - compute all the sum terms (appropriately shifted) by multiplying by each individual bit of one of the multiplicands. Concatenate all of these terms onto the end of the strand. Then run the addition algorithm repeated to perform pairwise add and concatenate operations - this will eventually result in the right answer in time $O(n)$ in the length of the string. A more clever way to do this is to implement a carry-save adder (a classical parallel adder design) [7].

Define the squaring operation according to the simple algorithm of concatenating the value of each strand to itself and then performing the multiplication algorithm. Using this squaring operation, it is easy to implement exponentiation. Consider the recursive expansions $b^n = (b^{\frac{n}{2}})^2$ if n is even, $b^n = b(b^{n-1})$ if n is odd [AS85]. Using these relations, we can represent the computation of any exponent of b in terms of two operations; squaring the current value and multiplying the current value by b. This can implemented in the model so long as we store the original argument value as well as a current "working value" on the strand. Then we multiply the working value by the original value or square the working value to form the new working value repeatedly as specified by the compiled recursive decomposition.

Given these operations, we can now evaluate arbitrary polynomials in parallel. One should note that via approximation methods this permits us to compute arbitrary continuous functions—for example, trigonometric functions (via Taylor expansion).

3.4 Observations and Comments

A few points about these algorithms need to be emphasized. First of all, the algorithms are all parallel (over the strands) with time dependent only on the number of bits computed with. The data varies (according to each individual strand) but the instruction is the same over all strands—hence the invocation of the metaphor of SIMD models of computation. However, keep in mind that the number of separation tubes increases exponentially with the number of "separation bits" (*e.g.* the addition algorithm required 4 separation tubes at each step as it depended on 2 separation bits, whereas the logical operators algorithm required merely 2 separation tubes).

The fact that the format of the data is preserved means we can perform abstraction of operations in this model. The construction of the exponentiation algorithm demonstrates how this abiliity to compose these operations permits the high-level construction of functions from primitives far-removed from the DNA level which can subsequently be compiled down to DNA (although one should keep in mind that this composability is sensitive to errors).

Also observe that the existence of these algorithms implies that this model of DNA computation supports the implementation of universal Turing machines. This is not surprising, but it useful to be aware of.

Finally, it should be noted that this model of computation is quite well suited for reversible computation—it is possible to encode the process of the computation on the strand as the computation progresses, thereby permitting full reversibility.

3.5 Applications to Minimization

One direct application of the algorithms discussed in this paper is to the minimization of an arbitrary continuous function. Via the Stone-Weierstrauss theorem, we can approximate arbitrary continuous functions using polynomials. So to perform the minimization of a function, compute the representation by polynomials (to desired accuracy). Then evaluate this polynomial function over a tube employing the algorithm described above; this is equivalent to evaluating the function over approximately 10^{20} random (uniformly distributed) arguments. However, modify the evaluation algorithms slightly so as to preserve a copy of the original argument at the beginning of the string at all times (*e.g.* write a copy of the argument to the "NEW" string before performing the algorith, etc.). This is for read-out purposes. Having evaluated the function over the tubes, we now need to extract the minimal string(s).

For this, we need the $< x$ function. This returns all strands representing numbers less than the argument x. To do this, simply separate repeatedly according to the following algorithm.

1. Let *count* = the most significant nonzero bit of the argument.
2. Separate the tube based on the value of bit *count*; if *count* is 0 merge into the result tube, if *count* is 1 keep in the working tube.
3. Set *count* to be the next most significant nonzero bit of the argument. If there are no more nonzero bits, return the result tube.
4. Otherwise, goto step 2.

Now using binary search we can find the set of minimum strands rather easily.

1. Set *upper-bound* = the largest number represented in the current scheme, *lower-bound* = 0, *test* = *upper-bound*.
2. Compute $<$ *test* into tube A.
3. (a) If tube A is empty, set *lower-bound* to *test*. Goto step 4.
   (b) If tube A is not empty, set *upper-bound* to *test*. Goto step 4.
4. Set *test* to $\frac{upper-bound-test}{2}$. If *upper-bound* $-$ *lower-bound* is within some tolerance level, stop and return $<$ *upper-bound* as the result.

I should point out that this algorithm employs techniques from both the function evaluation paradigm discussed in this paper and the constraint-filtering paradigm for DNA computation. I suspect that a close coupling of the two techniques (*e.g.*evaluate a function and then filter based on the computed values) will be a useful programming model for this class of machines.

3.6 Genetic Algorithms

As discussed in the introduction, the utility of using DNA computation for exponential search problems is limited. However, search problems (in particular exponential ones) are of critical importance. Thus, it behooves us to attempt to provide methods for enhancing the solution of such problems even when the solution space is too large for direct attack. The genetic algorithms methodology has proven useful in the context of traditional computers for such problems. Herein I present a simple implementation strategy for genetic algorithms within our model.

The essential loop in a genetic algorithm is as follows.

1. Initialize the population of organisms (typically represented by bit-strings of "genes").
2. Evaluate the fitness of the population according to some fitness function determined by the nature of the problem.
3. Throw out all but the most fit organisms (according to some percentage criterion) and fill the population by combining the "surviving" organisms according to some scheme attempting to mimic sexual reproduction.
4. Goto step 1.

But we can fairly easily implement this in our DNA-based model. We represent each organism as a bit-string representing its genome. We have already described herein how to implement the fitness function and cull out the bottom performers (same as the maximization procedure). Now, given our resulting organisms, we can generate new organisms as follows.

1. Make a copy of the parent tube to work with.
2. Perform arbitrary concatenation of pairs in the tube. This can be performed by adding tags to each strand so that each strand has only one "sticky" end, thus enforcing pairwise (but random) concatenation.
3. Now, perform the following logical operation on the pairs. Iterate over the bits in the pairs. If both bits are the same, append this bit to the end of the strand. If the bits are different, append a random bit to the strand. The random bit can be added by slightly modifying the operation in which a fixed bit is added. In that operation, many copies of an oligonucleotide representing the bit value in question are added to the working tube and ligated. To add a random bit, add to the working tube (in roughly equal proportions) copies of the oligonucleotides for both bit values. Then the specific value ligated to a specific strand is essentially random.
4. Cut off the offspring, and mix with the original parent tube to produce the new working tube to repeat the process on (*i.e.* evaluate the fitness function, cull, *etc.*).

No doubt it is possible (and desirable) to construct algorithms corresponding to more elaborate recombination schemes—but this algorithm should be indicative of the basic strategy and take more as proof of concept than as a recommended genetic algorithm.

4 Conclusion

In this paper I have presented and discussed a new model of DNA computation in which computation is viewed as process of parallel function evaluation. I believe that this model will be superior to the constraint-based model simply in that it permits the utilization of many algorithms from mainstream computer science to be implemented in terms of the DNA computing model.

However, the fact that it can be performed in the present metaphor for manipulation of DNA in biological labs should not obscure the essential fact that these metaphors for manipulating DNA are driven by the considerations of biological problems and not by computational problems. We have described our model in terms of a few basic operations on the underlying DNA representation (separation and concatenation being the major ones) which can be performed using PCR and traditional separations. However, there is a great deal of room to consider researching new models to which these operations can be bound. This could occur in the context of the traditional models (*i.e.* better separation methods and PCR methods optimized for the kinds of problems that arise in this class of endeavors) as well as in the context of general biological computing.

We wish to emphasize this last point—the operations we require for this kind of processing are embedded in some of the tasks carried out by the biological machinery found in living organisms. A fair amount of interesting research should be devoted to discovering how the human body handles the implementation of these operations. For example, there must be fairly sophisticated error-handling machinery in living systems to deal with copying DNA successfully—a basic operation. We should harness this machinery to build our computers out of DNA.

Future research will be directed at achieving an implementation of many of these processes in the existing metaphors for manipulation of DNA as well as developing new means to perform the basic operations.

5 Acknowledgements

This paper benefited greatly from the direction and guidance of Thomas Knight, Jr. The suggestions of Dr. Knight were invaluable in developing this research. The support and suggestions of John C. Mallery helped enable this research to occur. This paper was improved by comments from Gerald Sussman, Carlos Bustamante, and William A.M. Blumberg. This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number MDA972-93-1-003N7. The views and conclusions are those of the authors, not the Massachusetts Institute of Technology or the United States Government. Any errors or shortcomings are the responsibility of the authors.

References

[1] Adleman, L. M. Molecular computation of solutions to combinatorial problems. *Science* 266:1021-1024, 1994.

[2] Adleman, L. M. On constructing a molecular computer. Manuscript, Computer Science Department, University of Southern California, 1995.

[3] Abelson, H. and G. J. Sussman. *The Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[4] Cai, W., Condon, A. E., Corn, R. M., Glaser, E., Fei, Z., Frutos, T., Guo, Z., Lagally, M., Liu, Q., Smith, L. M., and A. Thiel. The power of surface-based DNA computation. Manuscript, Chemistry Department, University of Wisconsin, Madison, 1996.

[5] Guarnieri, F., Fliss, M., and C. Bancroft. Making DNA Add. *Science* 273:220-223, 1996.

[6] Hillis, W. D. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[7] Knight, T. Personal Communication. 1996.

[8] Liu, Q., Guo, Z., Condon, A. E., Corn, R. M., Lagally, M. G., and L. M. Smith. A surface-based approach to DNA computation. Manuscript, Chemistry Department, University of Wisconsin, Madison, 1996.