MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1589 October 1996 Revised

General Purpose Parallel Computation on a DNA Substrate

Andrew J. Blumberg

Abstract

In this paper I describe and extend a new DNA computing paradigm introduced in Blumberg [4] for building massively parallel machines in the DNA-computing models described by Adelman [1, 2], Cai *et. al.*[5], and Liu *et. al.*[8]. Employing only DNA operations which have been reported as successfully performed, I present an implementation of a Connection Machine [7], a SIMD (single-instruction multiple-data) parallel computer as an illustration of how to apply this approach to building computers in this domain (and as an implicit demonstration of PRAM equivalence). This is followed with a description of how to implement a MIMD (multiple-instruction multiple-data) parallel machine. The implementations described herein differ most from existing models in that they employ explicit communication between processing elements (and hence strands of DNA).

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.

Now consider the operational constraints imposed by this traditional model of DNA computation–there are grave limitations imposed by the very nature of the model. At the algorithm level, the major flaw in the conventional model of DNA computing is that there is no facility for communication of information between strands of DNA; *e.g.*between processing elements.

It is easy to see that this operation is extremely useful–the arguments are analogous to the arguments for parallel processing computers in general. For example, consider the area of image processing. For concreteness, let us take the example of filtering an 1024 X 1024 image. This is a relatively trivial undertaking given communication between processors–each processor assumes responsibility for a pixel, and the computation can be performed using a small number of local interactions. But in the constraint model, each strand would have to represent a possible solution to the filtering problem–requiring each strand to be at least a million base pairs long (depending on the efficiency of the encoding). Furthermore, an unfeasibly large number of constraints would have to be applied to filter the pool down to the correct filtered solution.

For other examples, consider problems in simulation–*e.g.*air-flow calculations, circuit simulation, *etc.*Given intercommunication between "processing elements", there exist simple algorithms for performing all of these on massively parallel hardware. Furthermore, even were it possible to do these things simply in the traditional model, a great deal of effort would have to be spent developing the algorithms. But why disregard the vast amount of prior research on parallel computing? It seems a much better solution to modify the model to allow us to tap into the rich existing body of parallel algorithms rather than require a whole new discipline of algorithm construction.

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 [6] 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 originally described in [4]. This model is employed to perform the implementation of SIMD computing machines on the DNA substrate. I will present the mapping between the DNA substrate and the operation of a Connection Machine [7], a massively parallel SIMD computer (which implies a mapping between the DNA substrate and a PRAM). The analogy seems a natural one; the original Connection Machine was composed of a very large number of very simple processors. Thus it is ideal for implementation in the DNA domain. This mapping allows software developed for the Connection Machine as the machine instruction level to be compiled down to DNA operations and carried out in the molecular setting. This of course provides the advantage of enormously greater parallelism than was ever possible in silicon.

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.*[5]. 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.

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 (and in particular communication between processing elements) seems to depend on being able to modify the instruction stream. 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.

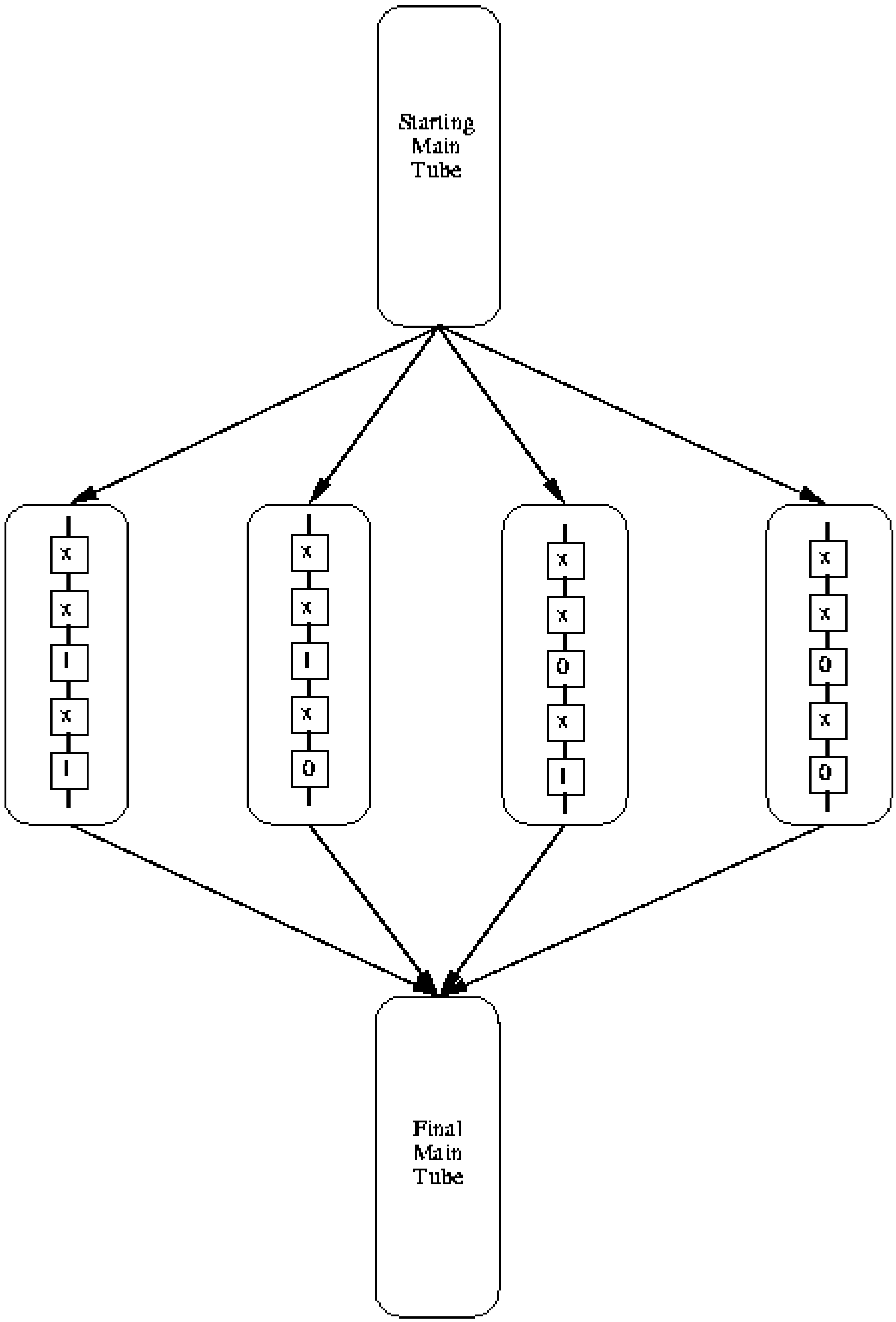

Figure 1

4 Implementation of the Connection Machine

It is possible to employ this model to design an infinite variety of specialized algorithms, and in fact this is done in [4]. However, I would now point out that there has been a great deal of work in the past ten years in the field of SIMD algorithm design, perhaps best exemplified by the Connection Machine [7]. Therefore, rather than attempt to further reinvent the wheel, I present now a mapping enabling the implementation of a Connection Machine in a computational framework supporting the basic operations we discussed above - in particular, in the DNA computing domain.

4.1 Processors

The Connection Machine consists of a large number of very simple processors. Each processor consists of memory, some flags, and simple logical circuitry. At each step, the processor reads two bits from memory and one flag bit and combines them to produce two bits of output and a flag output.

So let each strand of DNA represent a single processor (for the time being). The basic format for the strands will be as follows—a unique tag specifying the name of the processor, a sequence of bits representing the memory, and a sequence of bits representing the flags. This format will be expanded slightly during the following discussions as needed.

The instruction sequence is implemented at the level of the physical DNA manipulation operations. Given an instruction which specifies two read locations from memory, a flag to read, two write locations in memory, and a flag to write, we perform the following operation :

1. Separate the CM tube into an active tube and an inactive tube depending on the values of certain flag bits (this enables processors to be "turned off" to permit simulation of a wide variety of processor topologies). We will now only work with the active tube.
2. Separate the active tube into eight sub-tubes depending on the values of the read locations and the read flag; there are three bits which control the nature of the operation performed.
3. For each sub-tube, externally perform the logical operation and write the result bits to the write locations and the write flag. That is, the logical operation performed is specified at the instruction (physical) level. So for each sub-tube we compute externally the results and apply this by writing the correct bits and flag value.
4. Merge all of the tubes to reform the general CM tube.

I need to expand on the algorithm for performing the write operation described above. There are a number of ways to write values into the strands. The method we will describe here re-writes each strand, incorporating the new values.

1. Write the tag "NEW" to the end of each strand in the tube.
2. Set *count* = 0 (this is a bookkeeping step external to the tube).
3. If *count* is not equal to one of the two write locations, separate into two tubes based on the value of bit *count*.
4. In this case, we are simply copying the current bit-values.
   (a) For tube 0, append the value 0 to the end of the strand.
   (b) For tube 1, append the value 1 to the end of the strand.
   (c) Merge tubes.
5. If *count* is equal to one of the write locations, append the bit value to be written to all strands.
6. Increment *count*. If *count* < the size of the processor strand (total number of bits), goto step 3.
7. Cut strands at "NEW" tag.
8. Separate based on presence of "NEW" tag. Throw away old strands.

At the completion of this algorithm, the strands are in the same format as before and so the operation can be iterated (presuming we have a strategy for controlling the inevitable errors which will arise in this computation).

However, there is one more feature of the processor operation in the Connection Machine we have overlooked–the global bit. Each processor can set a global bit, the OR of which is available at the instruction level. This is easily implemented in the DNA model by simply filtering for strands with the proper bit set and performing the detection operation to determine if any such strands exist.

This conception of SIMD computing uses the DNA itself simply as memory—the instruction operations are all carried out at the level of tube operations. This is possible since the power of our operation set enables the strands of DNA to simply contain the data which varies over the processors—there is no need to encode information about how to compute at each processor. This is actually a substantial improvement in space efficiency over the silicon Connection Machine (which had a small general logic unit in each processor).

4.2 Interconnect

In many ways, the interesting part of the construction we are discussing is the algorithm for communication between strands. More than anything else, this distinguishes the computing model discussed herein from traditional views on DNA computing. Furthermore, it is the most useful aspect of this construction, for it can be generalized and abstracted to be employed as a solution in many different architectures on DNA substrates.

However, certain new problems must be addressed in our discussion of this algorithm. For the most part in the preceding discussion, I have been working with an idealized model of the actual DNA processing in which it is assumed that all of the operations have 100% efficiencies and introduce no errors. However, when discussing the interconnection algorithm we need to consider certain pragmatic issues in order to be assured that our algorithms are correct.

Specifically, we have the following problem. In a realistic implementation of the machine described herein, there will necessarily be redundancy—several thousand identical DNA strands corresponding to a given processor of the Connection Machine. As long as we ensure that at no time do any processors disappear due to loss of DNA, then we can be assured that most of the processors will be in identical and correct states. However, communication between processors raises new and grave dangers. Specifically, any communication algorithm we consider must ensure that each copy of a given processor receives the same information in a given time-step; otherwise we risk divergent behaviors. Although it will probably be interesting to consider the nondeterministic model of computation supported by having processors which diverge at run-time, this is inconsistent with the traditional conception of computing algorithms and is certainly inappropriate in the context of simulating the connection machine.

First, I will present an algorithm whereby a sender can transmit a message to a recipient *which is willing to accept messages from that particular sender* . This algorithm will then be used as the basic building-block of a protocol which will guarantee consistency. In the following algorithm description, the dash will indicate concatenation and the function *WC* will indicate the Watson-Crick complement. Further, assume each processor has been given an unique ID tag.

1. Concatenate to the sending processors the string *"MESSAGE1" (a tag)–WC(ID of recipient)–WC(ID of sender)–WC(data to be sent)–"MESSAGE2" (a tag)* .
2. Cut the strands after the "MESSAGE1" tag.
3. The messages (denoted by the "MESSAGE2" tag) are separated from the tube.
4. (Optional) The messages are amplified.
5. Remove the "MESSAGE2" tag from the messages.
6. Remove the "MESSAGE1" tag from the strands.
7. Concatenate to a given processor its ID tag and the ID tag of the processor it is presently willing to receive data from.
8. The messages are merged with the processor strands. Since the message consists of two ID tags and the data, this will result in the message sticking to the end of a recipient processor strand (with appropriate ID tags) with the message data hanging off the end. Adding polymerase will then result in the copying of the complement of the data onto the processor strand.
9. The messages are melted off the processors and separated out.

10. The message is incorporated into the appropriate location of the strand by using an algorithm mostly identical to the memory write algorithm (re-copying) described above.

Employing this algorithm, I will now describe how we can build hypercubes and thereby implement traditional packet-routing algorithms on our DNA Connection Machine. Let us further expand the representation for each processor such that a given processor incorporates arbitrarily ordered lists of the n processors it can receive from and the n processors it can send to.

Now, perform routing by an $2n$ stage process. During the first n stages, each processor can generate a message to be sent to the nth processor on the list it can send to. Then, in each of the next n stages the processors concatenate with the ID tag of the nth processor they can receive from and then hybridize with appropriate messages. Since at each receiving stage only a single class of message can hybridize to a given type of processor, we are guaranteed that all processors of a specific type will receive the same data (presuming a sufficient quantity of message strands).

Let's clarify this a little bit via an example. Consider process number 11. This processor can receive messages from processors 1, 3, and 6 and can send messages to 1, 2, and 4. So now we will walk through a step of operation focused on this processor. First, the processor performs some computation depending on the global instruction and its flags and memory bits (as described above)–this involves a variety of separations and mixings, but we aren't particularly interested in this process presently. At the conclusion of the computation stage, we enter the communication stage.

Now we go through three sending stages. At the first stage, processor 11 has the option to send a message intended for processor 1. If its program specifies, it will generate a message (as described in the algorithm above) which contains its ID (11), the ID of the recipient processor (1), and some data. At the second stage, 11 can send a message intended for processor 3, and at the last stage it can send a message intended for processor 6. This three-stage event is performed for all processors simultaneously.

Finally we have three receiving stages. At the first stage, processor 11 generates an appropriate tag at its end such that it will hybridize with messages sent from processor 1 to processor 11. All of the messages generated in the sending stage are mixed with the processor strands and allowed to hybridize. If something does hybridize to processor 11, we know this is a message from processor 1 and it can be incorporated into the memory of processor 11 (via the described copying process) appropriately. Next, processor 11 generates the tag to accept messages from processor 2 to processor 11 (having removed the old tag) and similarly incorporates such messages if they arrive. Finally, processor 11 accepts messages from processor 4.

The reason for the incorporation into the recipient of both its own ID and the ID of the processor it is willing to receive from is that this guarantees that at any time, all of the copies of a given processor which receive a message will receive only the same message–even if multiple messages for that particular processor exist. In an implementation in which the processor simply received messages based on its ID, if two different processors sent to it different copies could receive different messages and there would be no local way to determine who had received what.

This scheme is very well-suited for the implementation of a hypercube. At any given routing step a processor can receive information from all of the processors "connected" to it and send information to all of the processors "connected" to it. Expanding to include more processors is trivial—the only consequence is an increase in the size of a given processor strand. The capacity of the lines is dependent on the amount of space on a strand the implementor is willing to devote to message storage.

Also, note that since the lists of processors are stored in "memory" on the strand, it should be possible to implement dynamic connection changes depending on software.

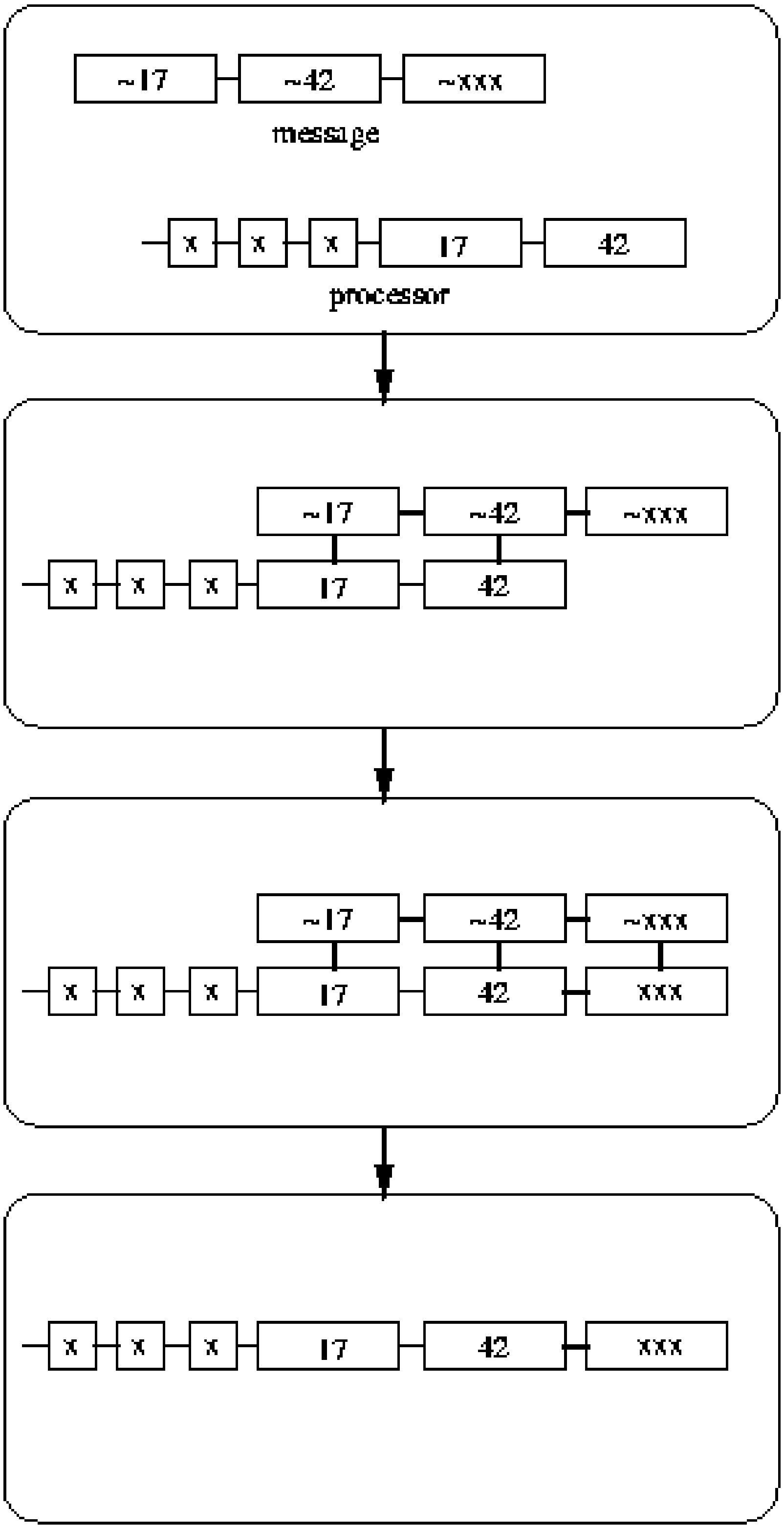
~17
~42
~xxx
message
x
x
x
17
42
processor
~17
~42
~xxx
x
x
x
17
42
~17
~42
~xxx
x
x
x
17
42
xxx
x
x
x
17
42
xxx

Figure 2

4.3 Initial Generation

The discussion of the algorithms for computation and communication above all presume the existence of appropriately configured processor strands. I have avoided discussion of this because this is strictly an operational problem–not an algorithmic one–and I because I believe that this can be readily addressed with current tools. There exist methods for reasonably synthesizing all of the n-bit strings [1,2,4]. These can be easily adapted to produce the random variation of parts of the processor strands, such as the ID's, as well as what might be called the controlled variation of the lists of communication processors. These processes will suffice for construction of the entire strand when the initial data of the processors is assumed to be essentially random.

It is a somewhat more challenging problem to consider sending specific data to each processor–as would be the case in the image processing example we discussed. There is really no way to do this without at some point synthesizing a strand representing the data (either directly in the processor or as some large message strand which will be sent to each processor). Although this may be quite slow under current technologies, it is certainly technically feasible and we expect that this will be ameliorated by accelerations in process speed. In the interim such a model may be practically limited to random initial data for most processors or initial data generated by computation from the smaller set of processors which can receive data from the outside.

5 Interim Observations and Comments

The previously described algorithms will allow a Connection Machine SIMD computer to be implemented on a substrate supporting a fairly simple set of operations (in particular the DNA substrate). This machine will be comparatively slow in terms of global operations per second (limited by the time taken for separation technology) but will have an enormous number of processors to compensate for this relative sloth. We wish to emphasize the point that this model provides a coupling between algorithms written for the Connection Machine and the DNA substrate—via the mapping defined above, all such algorithms can be compiled into DNA-based operations. Furthermore, all of the DNA-based operations are feasible under current technology and explicit performance of all of them has been described in the literature.

6 Building a MIMD machine

The Connection Machine is a SIMD machine—each processor executes the same instruction during a given time step. And while this seems a particularly apt metaphor for DNA-based computation, it is certainly possible to implement a machine in which each processor implements different instructions during a given time step–a MIMD machine. This can be done at a number of levels. One possibility is to perform emulation of a MIMD machine at the software level on top of the Connection Machine substrate—this emulation will incur merely a constant slowdown [7]. Alternatively, we can perform the implementation directly—simply by separating the active machine tube into different tubes depending on bits (stored on each strand) which encode specific instructions. However, since we need a different tube for each distinct instruction, there are practical pressures to keep instruction sets fairly small (*e.g.*RISC size instruction sets) or retain the SIMD characteristics.

7 Error Handling

For the most part, the matter of handling the inevitable errors that will arise in the computation has been largely ignored. Strategies for handling error are described in a separate paper. Briefly, I claim that redundancy coupled with repeated separations to increase probability of success, PCR amplification to counter loss of processor, and voting schemes for output are sufficient. An elaboration of this claim as well as detailed arguments in support of it will be found in the separate paper.

8 Conclusion

Having established the insufficiency of the traditional methods of DNA computation, I have presented an implementation of a Connection Machine in the DNA substrate. Since the Connection Machine implements a PRAM, this is an implicit demonstration the present technologies for DNA computation support the implementation of a PRAM in a relatively direct fashion. Although a central component of my claim is that this machine can be implemented on present technology, my intent is less to promote a specific architecture and more to promote a paradigm. That is, I intend this paper to provide a guideline by example as to the kinds of machine models that should be considered for implementation in the DNA substrate. For DNA computation to be viable, it is imperative that research be conducted on active computation as described in this paper, rather than on the passive constraint pruning model.

9 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] Blumberg, A. J. Parallel Function Application on a DNA Substrate. MIT Artificial Intelligence Laboratory Memo 1588, 1996.

[5] 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.

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

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

[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.