Formal Development of a Reconfigurable Tool for Parallel DNA Matching

A. E. Abdallah
Centre for Applied Formal Methods
School of Computing
South Bank University
103 Borough Road
London SE2 0AA
Email: A.Abdallah@sbu.ac.uk

G. Simiakakis
Harocopio University
70 Eleftheriou Venizelou St
17671 Athens, Greece
Email: gsimiak@hua.gr

T. Theoharis
Department of Informatics
The University of Athens
Panepistimioupolis 15784, Athens, Greece
Email: theotheo@di.uoa.gr

Abstract

DNA matching is a computationally demanding task. The Human Genome Project is producing huge quantities of data, which have to be analyzed. A formal description of the task of searching a DNA sequence is given and an efficient parallel algorithm is derived using formal methods. The algorithm is implemented on an FPGA using Handel-C, a language that enables the compilation of high-level algorithms directly into gate level synchronous hardware [9], thus reducing the development time. The designed algorithm makes no assumptions about DNA transformations, and is therefore a very powerful tool. It can be used in conjunction with an expert system to automatically detect patterns of interest in the DNA.

Keywords: FPGAs, Reconfigurable computing, formal methods, DNA, string matching, CSP, Handel-C, pipeline.

1 Introduction

There is an increasing electronic availability of DNA sequences from human and other species. The ambitious Human Genome Project has recently decoded the DNA sequence of chromosomes 21 and 22 and it is expected that the whole human genome will be completed within the next three years in an admirable world-wide collaboration. The amount of data produced is huge; chromosome 22 (the smallest one) consists of approximately 32Mbytes of base data.

Scientists who wish to use these data to test their hypotheses electronically are faced with an overwhelming amount of processing effort. Often this effort involves searching DNA data for traces of viruses or other organisms that have been incorporated in the genome over the long track of evolution. Also, DNA sequences of different species may be compared in order to verify common ancestry or detect protein homology. Detection of sequences of interest is a very complex task and several software solutions exist [4, 10, 17, 14].

DNA sequences are available as text documents. In computing terms the operations that we need to perform on DNA sequences are similar to well known string problems, specifically the edit-distance [18] and the longest common subsequence problem. The first measures the distance between two strings as a function of the number of insertions, deletions and substitutions necessary in order to go from one string to the other. The second, as the name implies, determines the longest common substring of two given strings.

The strings involved are however enormous (millions of places in length). Efficient string processing is therefore a requirement. Both the above problems have been well studied in computer science and efficient implementations on systolic architectures have been proposed [13, 15]. Such special purpose archi-

0-7803-6542-9/00/$10.00 © 2000 IEEE

tectures, although very efficient, can be hard and expensive to build. A recent innovation is reconfigurable computing, or computing systems whose hardware can be modified by software to match a certain application [16]. The main component of such systems is the *Field Programmable Gate Array* (FPGA).

The FPGA [8] is a silicon chip that can rapidly implement a significant amount of digital hardware under software control. For example, Xilinx and Altera are well known FPGA manufacturers. Dramatic performance gains, in comparison to a microprocessor, can be achieved by implementing a certain algorithm directly onto hardware and exploiting natural hardware parallelism while at the same time reducing overheads such as instruction fetches. Furthermore one can vary the function of an FPGA as the problem at hand varies.

Until recently, the main problem with using hardware has been the long time required and the large expense involved in producing it. Even FPGA's required training in extremely specialized tools. Then *hardware compilation* came along which allows ordinary computer programs to be turned automatically into hardware designs (FPGA's), thus promoting the programmer to a hardware designer. A well known and successful such tool is Handel-C available from ESL [9].

Apart from development efficiency, hardware compilation offers another important opportunity, namely the potential to produce provably correct hardware at the same cost as the production of provably correct software (assuming of course the correctness of the hardware compiler). In this paper we make a step in this direction by beginning with a functional specification of the problem and deriving an efficient pipelined implementation which can readily be programmed in Handel-C and thus compiled onto an FPGA. We consider a variation of the longest common subsequence problem which is useful for DNA operations and use well known mathematical transformations [7, 1, 2, 3] in the derivation that follows. The algorithm has been implemented in Handel-C and performance figures are given.

2 Functional Notation

We give a brief summary of the functional notation used in this paper. The reader should refer to [5, 6] for a fuller account of this notation. Lists are finite sequences of values of the same type. The list concatenation operator is denoted by $+\!\!+$ and the list construction operator is denoted by :. The elements of a list are displayed between square brackets and separated by commas. Function composition is denoted by $\circ$. Functions are usually defined using higher order functions or by sets of recursive equations. The operator $*$ (pronounced "map") takes a function on the left, a list on the right, and applies the function to each element of the list. Informally, we have:

$$f * [a_1, a_2, \cdots, a_n] \quad = \quad [f(a_1), f(a_2), \cdots, f(a_n)]$$

The operator / (pronounced "reduce") takes an associative binary operator on the left, a list of values on the right and returns the "summation" of all the elements of the list. This can be informally described as follows

$$(\oplus)/\,[a_1, a_2, \cdots, a_n] \quad = \quad a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

In order to formulate the pattern matching problem, we will make use of the following list manipulation functions. The function $inits_+$ returns the list of non-empty initial segments (prefixes) of a list in *increasing* order of length.

$$\begin{array}{l} inits_+\ [a_1, a_2, \cdots, a_n] \\ = [\ [a_1], [a_1, a_2], \cdots, [a_1, \cdots, a_{n-1}], [a_1 \cdots [a_n]\ ] \end{array}$$

Similarly, the function $fins_+$ returns the list of non-empty final segments (post-fixes) of a list in *increasing* order of length. Thus, informally, we have

$$\begin{array}{l} fins_+\ [a_1, a_2, \cdots, a_n] \\ = [\ [a_n], [a_{n-1}, a_n], \cdots, [a_2, \cdots, a_n], [a_1, \cdots, a_n]\ ] \end{array}$$

Finally, the function $segs_+$ which returns all the possible combination of consecutive elements of a list is defined as a composition of three functions.

$$segs_+ = (+\!\!+/) \ \cdot \ inits_+ * \ \cdot \ fins_+$$

3 Starting Specification

DNA matching can be translated to string matching in computing terms, where each string element is one of the 4 characters $\{A, C, G, T\}$ which represent the 4 DNA bases. For example, the DNA sequence of human chromosome 22 is downloaded into a huge text file of the form:

```
TTTGGCTAAAACCGAAATCAATTATGAAGC AAAGG AAGTG....
```

This sequence represents one of the two strands of the chromosome; the other is complementary. Complementary base pairs are $A - T$ and $C - G$. This huge sequence must then be searched to detect, for example, a (smaller) DNA string belonging to a virus or to reveal

similarities with a chromosome of a particular African monkey.

A mathematical specification of the problem will be given using list notation. Lists are finite sequences of a certain base type which in this case is the set:

$$\Sigma = \{A, C, G, T\}$$

Let Q and Q be lists of type $[\Sigma]$ representing the query DNA sequence (for example virus) and the bank DNA sequence (for example chromosome 22) respectively.

First, we define a function *llcp* which takes two lists and returns the length of the longest common prefix of both lists. For example, the length of the longest common prefix of `ATCCTG` and `ATCTTG` is 3 being the length of `ATC`. A formal definition of *llcp* is given recursively as follows:

$$\begin{array}{lllll} llcp\ s\ [] & = & 0 & & \\ llcp\ []\ t & = & 0 & & \\ llcp\ (a:s)\ (b:t) & = & 1 + llcp\ s\ t, & \text{if} & a = b \\ & = & 0, & & \text{otherwise} \end{array}$$

At first, it may be suggested that it is all about simple string matching, the type that a word processor uses when trying to find a certain word in a document. While this is often true, DNA operations usually have certain peculiarities. First, *mutations* take place which change some bases of the subject string (e.g. $C \rightarrow T$), thus rendering impossible its exact detection. Second, *deletions* can occur, which remove a significant part of the subject string altogether. And third, *insertions* introduce a new piece of DNA within the subject DNA (e.g. the DNA of a virus particle), see Figure 1. Combinations of these operations are possible.

```
C C G A A A T C
|  :  |  |  :  |  |  |      Mutation(:)
C T G A C A T C

C C G A A A T C
|  |  |  .  .  |  |  |      Deletion(.)
C C G        A T C

C C G        A T C
|  |  |  .  .  |  |  |      Insertion(.)
C C G C T A T C
```

Figure 1. Mutation, deletion and insertion (| denotes a perfect match).

It is possible to introduce a similarity measure function between two strings that incorporates the above concepts of deletion, insertion and mutation and handles them all simultaneously, but this will be at a significant cost of increasing the complexity of the specification and making it less general. Instead we consider a simpler approach which is based on the observation that all the above cases (plus some more) can be handled by detecting *the longest common substrings* in B and Q of at least a minimum length. Thus for example, if we detect two common substrings of length 40 and 50 separated by a single element which is different in B and Q, then this points to a mutation. Similarly, two common substrings that are separated in B (resp. Q) by another string, point to an insertion (resp. deletion).

In addition many other useful DNA transformations can be detected in this manner, such as the transposition of parts of the DNA sequence. Of course, at the end of the day it will be the biologist who will decide what is of significance as well as its categorization. All that is therefore required is a tool which will compare two strings B and Q and detect equality of substrings within them, regardless of where they occur, in B and Q.

A string c is said to be a maximal common segment of two strings s and t if and only if s can be expressed in the form $s_1\ ++c\ ++s_2$, t can also be expressed in the form $t_1\ ++c\ ++t_2$ and c cannot be extended to the right (that is, when s_2 and t_2 are non-empty lists, their first elements must be different. i.e. $hd\ s_2 \neq hd\ t_2$)

We need to identify the maximal common segments of s and t. The main insight for achieving this is the observation that a maximal segment c of two lists s and t is the longest common prefix of a final segment of s (i.e. $c\ ++s_2$) and a final segment of t (i.e. $c\ ++\ t_2$). Therefore, a suitable solution to this problem is reduced to computing the longest common prefixes of each final segment of s with each final segment of t. Hence, the lengths of maximal common segments, *lmcs* can be determined by computing the following table:

$$lmcs\ s\ t = [\ llcp\ a\ b\ \mid\ a \leftarrow fins_+\ s; b \leftarrow fins_+\ t\ ]$$

Given that the size of s is n and the size of t is m (where $m \leq n$) there are $n \times m$ entries in the table; each entry requires $O(m)$ calculation but by using a simple dynamic programming technique this can be reduced to $O(1)$.

The lengths of maximal common segments for the strings `ATCCATGTCATC` (horizontal query) and `CTATCTCATCG` (vertical data bank) are given by the following table. Note that both strings are displayed in reverse order.

The entry at position (i, j) of the above table computes the length of the longest prefix,

	C	T	A	C	T	G	T	A	C	C	T	A
G	0	0	0	0	0	1	0	0	0	0	0	0
C	1	0	0	1	0	0	0	0	1	1	0	0
T	0	2	0	0	2	0	1	0	0	0	2	0
A	0	0	3	0	0	0	0	2	0	0	0	3
C	1	0	0	4	0	0	0	0	3	1	0	0
T	0	2	0	1	5	0	1	0	0	0	2	0
C	1	0	0	1	0	0	0	0	1	1	0	0
T	0	2	0	0	2	0	1	0	0	0	2	0
A	0	0	1	0	0	0	0	2	0	0	0	3
T	0	1	0	0	1	0	1	0	0	0	1	0
C	1	0	0	1	0	0	0	0	1	1	0	0

Figure 2. Table of maximal segments for two strings.

llcp (*drop i s*) (*drop j t*), where (*drop i s*) denotes the postfix of s obtained by removing the first i elements of s. It may be worthwhile to note that list indexing starts from 0. The first column of the table in Figure 2 displays *llcp* for the string `C` with each final segment of the bank sequence; i.e. `G`, `CG`, `TCG`, `ATCG`, `CATCG`, `TCATCG`, `CTCATCG`, `CTCATCG`, `TCTCATCG`, `ATCTCATCG`, `ATTCTCATCG`, `CTATCTCATCG`. Similarly, the last column of the table displays *llcp* for the whole query with each final segment of the bank sequence. The length of the longest common segment is 5, the appropriate segment can be retrieved as `TCATC` by simply following the diagonal in the table starting at 5.

4 Derivation of a Pipelined Algorithm for FPGA's

4.1 Transformation to CSP

We have that *lmcs* is expressed as a pipe pattern; using transformational techniques from [3, 2, 1], we can formally synthesize a pipeline network of communicating sequential processes expressed in Hoare's CSP [12] for solving this problem as follows:

$$LMCS(s,t) = INIT(t) \gg ((\gg)/(STAGE * s))$$

$$\begin{aligned} INIT([]) &= !eot \rightarrow SKIP \\ INIT(a:s) &= !(a,0) \rightarrow INIT(s) \end{aligned}$$

$$\begin{aligned} STAGE(c) &= ?(x,n) \rightarrow !(x,n) \rightarrow F(c,[x \oplus_c n]) \\ F(c, r:rs) &= ?\text{“}eot\text{”} \rightarrow !eot \rightarrow PRD_{down}(r:rs) \\ &\quad | \\ &\quad ?(x,n) \rightarrow !(x,r) \rightarrow F(c,(x \oplus_c n):r:rs) \end{aligned}$$

$$x \oplus_c n = \text{if } x = c \text{ then } n+1 \text{ else } 0$$

Note that $\gg$ denotes the piping CSP operator. The synthesis of the above processes from their functional description is a straightforward task using the refinements rules in [1, 2, 3]. The above parallel version requires $O(m+n)$ time steps to terminate and uses $O(m)$ parallel processes in the pipeline.

4.2 Pseudo Handel-C description

The algorithm is implemented as a bidirectional pipeline, depicted in Figure 3, of length $m+1$ where m is the length of string B. The current lengths of matching strings flow to the right while final results lengths (coulmns) flow downwards.

Each stage in the pipeline holds one element c of the query sequence Q and compares it at each clock cycle with the incoming element x of the bank sequence B and the value v of longest prefix in the preceeding segment. If these two elements match, the length of the longest common prefix r starting with x is incremented otherwise, it is reseset to 0. In both cases this value is stored in a local array which will eventuallymake a column in the global table. After the first cycle the most recent value of r is passed to an external channel *down*, coupled with the incoming element of the data bank and passed to the right. For space reason the Handel-C version is ommitted.

5 Conclusion and Further Work

A fast FPGA implementation of DNA matching has been obtained based on a formal description of the problem which revealed a generally useful algorithm. Speed is extremely important in this application domain because DNA databases, which are now widely available, are huge. The presented algorithm makes little to no assumptions on the transformations that may have taken place on the DNA, and is therefore able to detect all cases of interest.

It is important to observe that hardware performance thus becomes available out of a general purpose FPGA card. What is striking is the short development time that a hardware compiler like Handel-C allows. Although current FPGA chips allow for large lengths for string Q, in the present algorithm the maximum length for string Q is always limited by the size of the FPGA. Larger strings should be broken up into segments and processed sequentially or by multiple FPGAs. Resulting matches that span over multiple segments could then be combined.

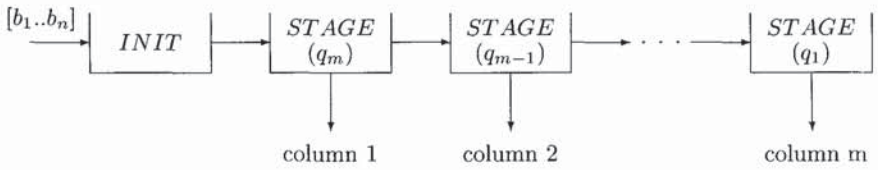

Figure 3. The matching pipeline process $LMCS$.

Acknowledgements

The authors would like to thank The British Council in Greece and the University of Athens for funding their joint research. We would like also to thank Dr. Nikos Yannakouris and Dr. George Dedoussis for their helpful comments on genetics.

References

[1] A.E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to CSP Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag,1995) 67-96

[2] A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'96*, LNCS **1124**, (Springer Verlag, 1996), pp 911-920.

[3] A. E. Abdallah, Functional Process Modelling. In K Hammond and G. Michealson (eds), *Research Directions in Parallel Functional Programming*, (Springer Verlag, October 1999). pp339-360.

[4] Altshul, S.F. et al., Basic Local Alignement search tool, *Journal of Molecular Biology*, pp.403-410, 1990.

[5] R. S. Bird, An Introduction to The Theory of Lists, in: M. Broy, ed., *Logic of Programming and Calculi of Discreet Design*, (Springer, Berlin, 1987) 3-42.

[6] R. S. Bird, *Introduction to Functional Programming*, (Prentice-Hall, 1998).

[7] R. S. Bird, J. Gibbons and G. Jones, Formal Derivation of a Pattern Matching Algorithm, *Science of Computer Programming* 12 (2) (Elsevier, 1989), pages 93-104.

[8] Chan, P.K. and S. Mourad, *Digital System Design Using Field Programmable Gate Arrays* Prentice Hall, 1994.

[9] *Handel-C Language Reference Manual*, Embedded Solutions Limited, 7/8 Milton Park, Abingdon, Oxfordshire, OX14 4RT, United Kingdom, 2000.

[10] Gaasterland, T. and C.W. Sensen, MAGPIE: Automated genome interpretation, *Trends Genet.*, Vol. 12, pp.76-78, 1996.

[11] Guerdoux-Jamet P.,Lavenier D., Wagner C. and Quinton P., Design and Implementation of a Parallel Architecture for Biological Sequence Comparison, in L. Bouge and P. Fraigniaud and A. Mignotte and Y. Robert (eds), Proceedings of the *European Conference on Parallel Processing, EuroPar'96*, LNCS **1123**, (Springer Verlag, 1996), pp 11-24.

[12] C. A. R. Hoare, *Communicating Sequential Processes*. (Prentice-Hall, 1985).

[13] Hoang D.T. and D.P. Lopresti, *FPGA Implementation of Systolic Sequence Alignment*, LNCS **705**, 1993, pp. 183-191.

[14] Reese, M.G. et al., Genome Annotation Assessment in Drosophila melanogaster, *Genome Research*, Vol. 10, No. 4, pp.483-501, 2000.

[15] Sastry R. and N. Ranganathan, A Systolic Array for Approximate String Matching, *IEEE International Conference on Computer Design*, Cambridge, Massachusetts, 1993, pp. 402-405.

[16] Schewel, J. et al (Eds), Reconfigurable Technology: FPGAs for Computing and Applications, *Proceedings of Spie*, 20-21 September 1999, Boston, Massachusetts.

[17] Stormo, G.D., Gene-Finding Approaches for Eukaryotes,*Genome Research*, Vol. 10, No. 4, pp.394-397, 2000.

[18] Wagner R.A. and M.J. Fisher, The String-to-String Correction Problem, *J. ACM*, 21(1), 1974, pp. 168-173.

[19] Waterman M. S., *Mathematical Methods for DNA Sequences*. (CRC Press, 1989).