Methods and Architectures for Realizing Fast Phylogenetic Computation Engines Using VLSI Array Based Logic

James P. Davis and Sreesa Akella
Department of Computer Science and Engineering
jimdavis@cse.sc.edu

Peter Waddell
Department of Biological Sciences
Department of Statistics

University of South Carolina
Columbia, SC 29208

ABSTRACT

Evaluating phylogenetics trees is an endeavor fundamental to comparative genomics and a core discipline of Bioinformatics. However, with single trees taking up to a week on the fastest processor under general models of evolution and the number of trees growing exponentially with the number of sequences analyzed, this is an exceptionally computationally intensive endeavor. There has been much work recently to develop more efficient algorithms, as well as to parallelize algorithms for execution on fast and/or distributed computing platforms. However, it is apparent that these approaches can only lessen the problem, as computational run-times remain on the order of weeks for likelihood-type approaches in particular, which are a favorite because of their biological and statistical appeal. At present, phylogenetics algorithms are still executing code running on many layers of other software, before actually executing on the underlying computational hardware. We propose a design method plus a set of architecture patterns for implementing phylogenetics algorithms directly into hardware, thus eliminating much overhead and allowing for on-chip parallelization. The solution formulations use a direct mapping to application-specific, VLSI array-based custom computing machines design specifically for phylogenetics data processing tasks. Such VLSI computing "engines" are realized as specialized computer chips that can either be integrated with standard computing platforms (e.g., with PCs as add-on cards) or used to construct specialized computing environments for bioinformatics and genomics data processing.

Keywords
Phylogenetics Least Squares Trees, VLSI, Reconfigurable Computing, Computer Architecture, RTL Design Methods.

1. Introduction

Phylogenetics algorithms cover three main classes of problem (Swofford et al. 1996): *parsimony*, which is like the vertex coloring problem of classical graph theory; *distance methods*, which aim to find the tree whereby the path-distances of the tree best match the observed distances (e.g. by least squares); and, *likelihood methods*, where the likelihood of the data is typically calculated using Markov transition matrices and summing over all possible character state assignments at internal nodes. Each approach possesses distinct problems in terms of where computational bottlenecks occur. Precisely what these bottlenecks are has so far not been systematically explored (Swofford et al. 1996). Indeed it is only recently that time optimal algorithms have been described for one large class of distance based phylogenetic problem (Bryant and Waddell 1998).

The promise of putting phylogenetics algorithms into hardware includes: (1) eliminating levels of intervening software--such as the operating system--which slows down the execution of C code tremendously (e.g. 1 or 2 orders of magnitude); and, (2) parallelizing or pipelining algorithm functions by exploiting the natural capabilities of VLSI circuits. The latter allows a VLSI circuit designed for the focused purpose of executing specialized algorithms to do far more work-per-cycle even than code written directly in a native instruction set on a general-purpose computer chip. Speedup in this second aspect can often be 1 or 2 orders of magnitude as well. Part of the strategy in such designs is to avoid bottlenecks in a purposeful way by exploiting the capabilities of VLSI custom logic that general architectures cannot hope to match.

We begin by illustrating how two of the most simple, but useful, phylogenetic heuristics can be conceptualized as a preparation for realization directly in VLSI custom logic circuits. The design steps to put these algorithms into circuits on a reconfigurable platform follow this discussion. At this level, bottlenecks become apparent and heuristic circuit design can be run many times to explore finding near optimal circuit designs. At that point, the problem is well enough defined to consider fabricating or "programming" a special-purpose VLSI logic chip containing problem-solving “engines” implementing the algorithms. Already, with the types of FPGA boards that can be added onto standard desktop machines, we are seeing architectures that can potentially speed up computing solutions to these problems by a factor of 10 to 1000 times.

2. Architecture of Phylogenetics in Silicon

The basic premise for our work in moving phylogenetic algorithms directly into hardware “engines” is that we can improve the throughput and speed of processing by realizing the structures directly in custom VLSI logic circuits, as opposed to running the algorithms on a multi-user computer or network of such machines. We describe the nature of phylogenetic processing that we are concerned with, along with a rationale for considering architecture and realization of the key algorithms associated with these problems directly into VLSI custom logic.

Bottlenecks in UPGMA and Neighbor Joining

The algorithms considered in our research are UPGMA and NJ. Both have relevancy beyond phylogenetics, since these are two of the hierarchical clustering methods that are both fast and useful with gene expression or micro-array data (Eisen et al. 1998, Waddell and Kishino 2000). They combine a limited search through tree space with a rapidly-evaluated local optimality criteria (in this case related to least squares). Although each is already order n^2, in microarray applications, n (the number of objects or taxa), is frequently in the range of 10,000 to 50,000.

Thus, even though phylogenetics applications may run these methods at rates such as 1 second each for n=100, there is a time increase of > ~10,000 in microarray applications without concerns about memory bottlenecks. In addition, the most tractable statistics for determining stability of clusters use Monte-Carlo, Bootstrap or Jackknife procedures that require the whole process to be repeated ~100 to 1000 times. With potentially dozens of distance transforms to consider, the problem rapidly escalates to one that may take days to complete. However, we emphasize that it is the simplicity of these algorithms that makes them immediate candidates for direct realization in VLSI custom logic circuits. For complexity, a single tree of 100 taxa under a general codon-based model will take more than 1 week on even the fastest CPU.

For the UPGMA algorithm (Sokal and Michiner, 1957, Swofford et al. 1996), the input data are typically distances with a statistical accuracy of ~3-8 decimal places, allowing for single precision data. Both algorithms are time optimal and have two obvious bottlenecks. The first is in deciding which of the $n(n-1)/2$ pairwise distances is minimal at each step of the star-decomposition clustering.

Following this, the data matrix is reduced in dimension by 1 due to the clustering of two objects and this introduces the second bottleneck with the need to calculate an average distance between the two objects (*i* and *j*) as a single cluster (*k*) and all other objects. For UPGMA, these new distances are calculated as $d_{kl} = (d_{il}|C_i| + d_{jl}|C_j|)/(|C_i| + |C_j|)$, where $|C_j|$ is the number of objects in cluster *j*, and these values replace the distances from *i* and *j* in the clustering distance matrix.

The next sections describe how these steps of the algorithm can be converted into efficient computational engines in hardware. NJ uses a slightly different algorithm to UPGMA, but the bottlenecks are the same: the need to find the minima of a matrix of values, followed by the need to compress two rows and columns into a single one.

Rationale for VLSI Custom Logic Architecture

As shown in Figure 1, we can consider the decision of how to realize an algorithm as being across a continuum of potential architectures. At one end, using standard microprocessor-based architectures (such as those that comprise PCs and Unix-based workstations and networks), we have technology that is designed for a high degree of programmability (evidenced by the programming languages and environments supported), generality and backwards compatibility.

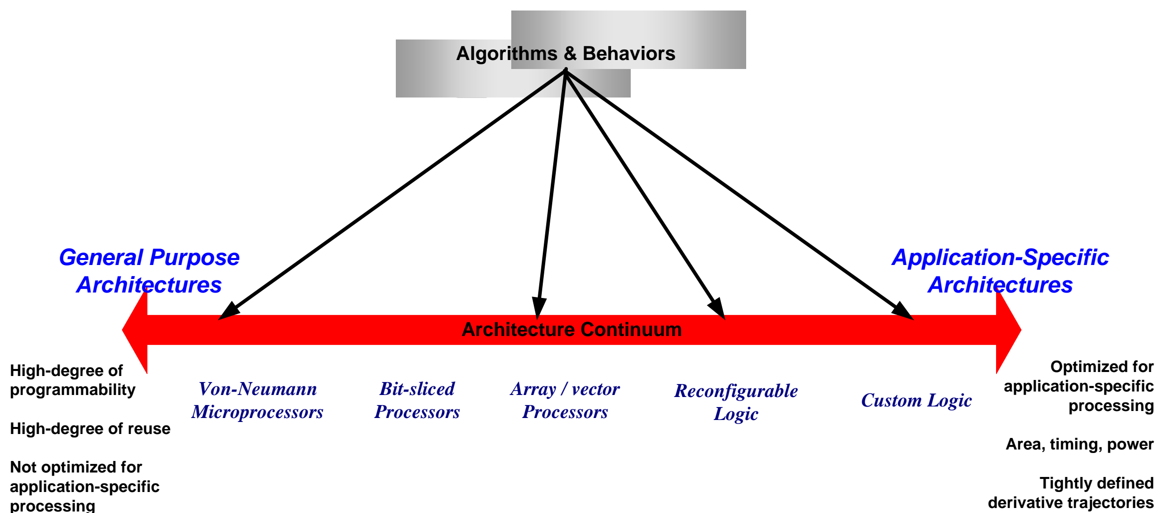

Figure 1. Continuum of architectures for algorithm realization.

Most algorithmic work in bioinformatics is done with architectures to the left of Figure 1 in mind. The principal task is thus to optimize the execution of the algorithm, written in a software language such as C, on the selected general-purpose architecture. Attempts are also made to parallelize execution by spanning the execution across multiple, networked systems of similar class (whether PCs, workstations, or supercomputers). Although there are many variations on

this theme, the execution model underlying all of these architectures is based on the Von Neumann architecture and its variants, which has been in use for 50 years (Tanenbaum, 1997). Furthermore, most classes of these architectures execute a layer of "virtual machines" consisting of various software components (operating system, drivers, application programs, etc.) designed to make the architecture of general use to a broad class of algorithms and application programs.

At the other end of the continuum are architectures based on the principal of designing application-specific VLSI integrated circuits to implement custom logic functions and algorithms. Solutions of this type tend to be highly specialized to the algorithms used in the application, and allow specialized packaging (such as with cellular phones, which use a high content of special-purpose VLSI circuits to achieve low power, high functionality, portability and low cost). What these architecture options on the right side of Figure 1 sacrifice in terms of generality, they trade off in speed and efficiency for the computational problem at hand.

Thus, the basis for our work is the recognized efficiencies to be gained by realizing the specialized functionality of a class of phylogenetic algorithms, using the understood benefits in selecting such architectures as practiced by systems designers in other domains such as telecommunications. Many of these architectural benefits of custom logic solutions have been discussed elsewhere (Camposano et al. 1991, Thomas et al. 1993, Gajski et al. 1994, Buell et al. 1996).

Therefore, given this premise, the problem treated in this paper becomes one of coming up with an effective and reasonably accessible means to move from specifying algorithms for phylogenetic problem-solving so that they can be quickly realized in VLSI custom logic circuits using engineering practices common to this endeavor. To do this, we present a methodology for transitioning from algorithm to custom logic architecture that can be built using array-based VLSI circuits such as FPGAs and ASICs[1]. Streamlining of this methodology of getting to efficient custom logic circuits is important, since there are a wide variety of phylogenetic methods, and biologists are wedded to a wide variety of these (Swofford et al. 1996).

3. The Methodology Explained

The notation and methodology presented in this section allow algorithm designers to take certain classes of algorithms, partition them into concurrently defined sub-units, and transform them into register-level abstractions amenable to a VLSI chip design stream—thus allowing the algorithm to be implemented directly as a array-based custom logic circuit. There are important differences from the usual approach for realizing an algorithm via implementation of software code.

Moving from Algorithm to VLSI Hardware

Fundamental to our approach is the basic series of representations, and their respective transformations, as shown in Figure 2.

---

[1] The acronyms FPGA and ASIC stand for *field programmable gate array* and *application-specific integrated circuit*, respectively. These are the leading technological approaches (with many variations in device architecture and implementation choice) currently being used in the semiconductor industry, and form the basic device choices under consideration as part of our research.

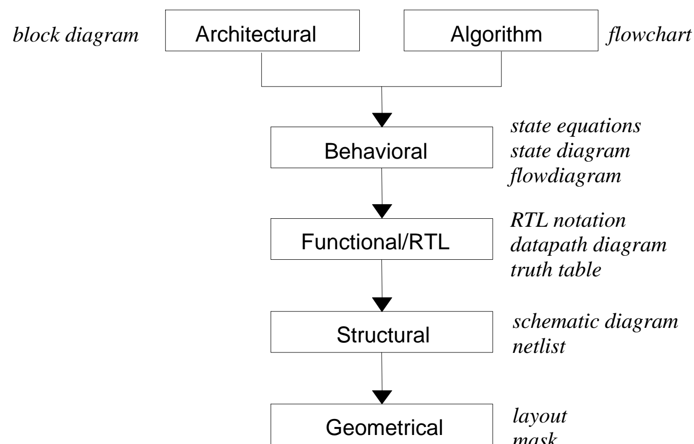

Figure 2. Hierarchical Transformations of Abstract Representations.

For creating algorithm specifications that can easily be realized in custom digital VLSI logic, we start with a representation of the algorithm. It is common in the scientific computing domains to jump directly into writing algorithms in a language such as C, once the algorithm has been appropriately mathematically formulated. In our approach, we take the algorithm description and create a graphical flowchart-like representation, explained later in the paper.

Next, we add to this flowchart-like set of structures a set of "bindings" to each of the algorithm "steps". The steps are organized and ordered according to when they might be expected to execute in real-time, based on the specification of appropriate clocking signals operating at some clock frequency. The basic principle of organization conforms to the definition of a Finite State Machine (FSM), as represented in a state diagram. The structure of the state diagram is captured as a refinement of the flowchart, turning it into a "flowdiagram" (Davis, 1995). Through this incremental refinement, we capture the control flow of the algorithm, and can make partitioning decisions as to which parts of the algorithm might be performed concurrently in the hardware realization.

The next step is to add the actual data manipulation statements associated with the algorithm. These are individually specified as Register-Transfer Level (RTL) statements involving standard arithmetic, Boolean and steering logic functions that are well-understood abstractions in digital logic and computer architecture design.

In this paper, we focus on these levels of the abstraction hierarchy—from algorithm to RTL representation, as we are working with automated tools and methods that allow us to capture the phylogenetics algorithm specifications and refine them through incremental and iterative enhancement. We use a VLSI design tool set, called flowHDL®, allowing us to create the specification and subsequent refinements and partitioning of the algorithm. Other steps in the hierarchical transformation from algorithm to VLSI circuit realization are based on the standard design process steps associated with such design for application-specific circuits in other domains (Camposano et al. 1991). As such, they are not considered further in this paper.

Introduction to the Algorithm Representation

In considering algorithm design for on-chip systems, we make several assumptions about the "typical" organization of such systems. Researchers and practitioners have developed these assumptions over time, and they form the basis for designers to understand VLSI-based systems. First, system components can be partitioned and defined in terms of their control and data behaviors (Mead and Conway, 1978). For on-chip systems, we can think of an arbitrary system block implementing an algorithm in terms of its control path and data path. In addition, we include memory in the description when we are interested in representing behaviors involving manipulation of multi-dimensional structures such as matrix arrays. The partitioning of on-chip system components is illustrated in Figure 3.

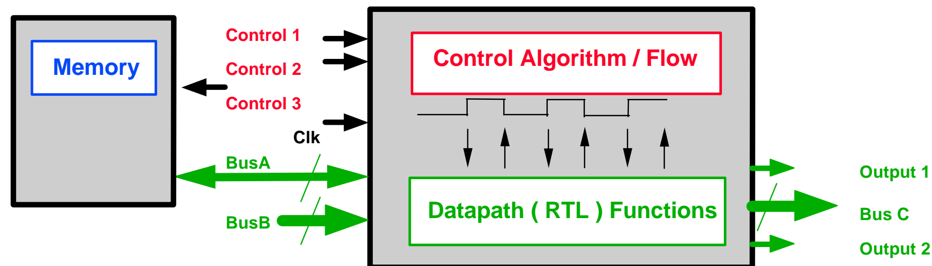

Figure 3. Partitioning an algorithm block into control path, data path and memory.

Second, each block in a design can be represented as a group of control path and data path *units*. Designers model a control path unit with a Finite State Machine model, to specify the sequential behavior for controlling data path operations. Designers model a data path unit as a partially ordered set of register-transfer operations.

Notational constructs of the flowdiagram

Once we create a hierarchical structure of functionally partitioned blocks, we further refines each of the bottom-level blocks with the second graphical notation. We use a different representation, a graphical *flowdiagram*, to create a partially ordered temporal sequence of data operations. Initially, this sequence represents the precedence of data operations in each system block of the algorithm. The designer would refine and extend the flowdiagram with explicit clocking information later in the specification process. The flowdiagram is an enhancement of the algorithmic state machine (ASM) notation (Claire, 1973). It has been enhanced to include explicit references to data operations, sequenced and scheduled according to control-step behavior in the design. An example flowdiagram is shown in Figure 4.

A flowdiagram has a graphical symbol set similar to the traditional flowchart. It is capable of representing the basic constructs of control flow common to algorithmic design: *sequence*, *selection*, and *iteration*. One extension to the ASM chart is the addition of a notation for describing data operations. This notation is based on the concept of register-level assignment, as defined in (Mead and Conway, 1978). In a concurrent design, one flowdiagram thread may model the interaction of both control path and data path units.

An algorithm designer constructs a flowdiagram model to represent the control flow for each function block in the design. Multiple flowdiagram "threads" can be created to model concurrent behavior for a given block. The threads interact with one another using standard mechanisms for

concurrent systems, such as synchronization and cooperation on shared tasks (using polling, handshaking or interrupts), and competition for shared system resources (using arbitration).

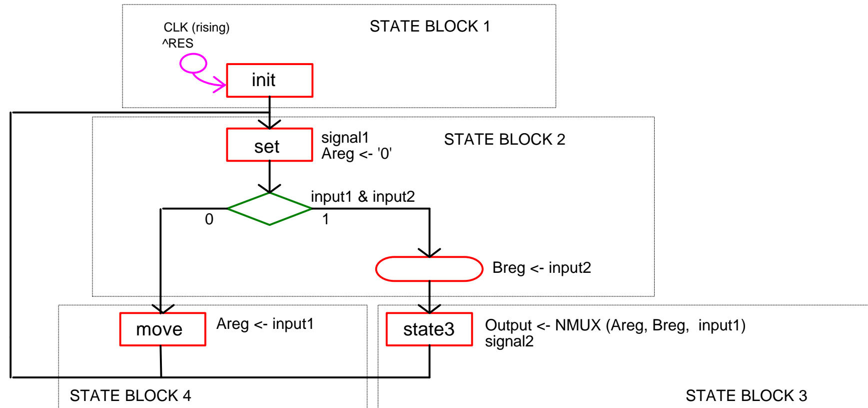

Figure 4. State block structure of a flowdiagram.

4. The Methodology Illustrated with UPGMA

We can discuss other aspects of the notation by looking at the specific UPGMA algorithms and their realization as state machines and register-transfer data operations in the following flowdiagrams. We look at operation sequencing, operation scheduling and resource allocation—essential steps in transforming an abstract algorithm into a form that can be directly realized in VLSI custom logic circuits (Davis, 1995).

Specifying operation sequencing

An algorithm designer specifies data operations in the flowdiagram notation using RTL-level expressions. An expression can be an assignment of some value or signal to another signal. In addition, expressions can have explicit data operations, represented as functional transformations, specified as part of their right-hand side. Expressions, represented as RTL notation, are attached to specific states in the flowdiagram, indicating that these operations are sequenced and scheduled once execution reaches that state.

The example of Figure 6 is for an arithmetic macro-function used in one of the UPGMA Clustering processing blocks, based on the citing in (Durbin et al., 1998). The RTL expressions model concurrent actions that are sequenced in a single state in a flowdiagram. The structural description of the data path unit shown in the figure can be inferred from the expressions using information contained in the flowdiagram in Figure 9.

$$d_{kl} = \frac{d_{il}\ |Ci| + d_{jl}\ |Cj|}{|Ci| + |Cj|}$$

```
MUL_result1 <- MUL( DiL, Cardinality_Ci)
MUL_result2 <- MUL (DjL, Cardinality_Cj)
ADD_result1 <- ADDNC ( MUL_result1, MUL_result2)
ADD_result2 <- ADDNC( Cardinality_Ci, Cardinality_Cj)
Dij2L <- DIV(ADD_result1, ADD_result2)
```

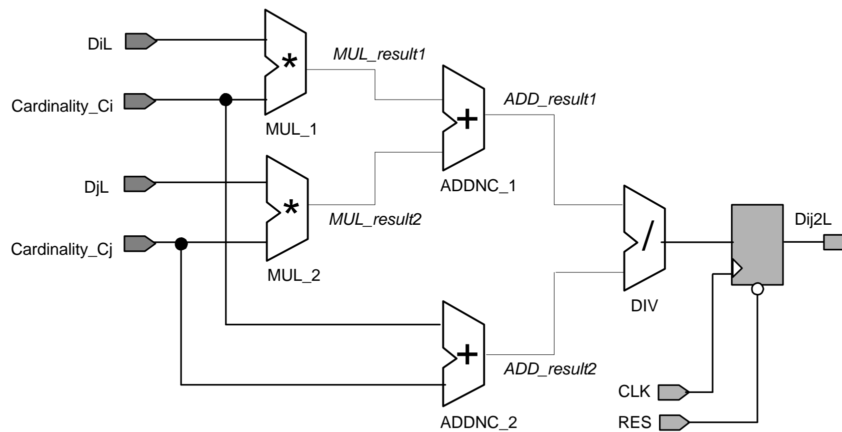

Figure 5. A structure of a data path fragment for a flowdiagram's RTL notation.

<u>Specifying operation scheduling using the state machine</u>
Scheduling information is annotated to the evolving flowdiagram representation using two mechanisms. First, once the designer expresses individual data operations in terms of defined *bus* structures, these buses can be "bound" to specific resource types. The designer will take each bus and declare it as being a *register*, *latch*, *wire* or other bus structure. These are shown

Second, the designer must select a specific clocking scheme, based on requirements for moving data through the data path. Referring to our earlier notion of on-chip systems architecture, we have both a control path unit and data path unit (including memory arrays). In synchronous sequential designs, both units are sensitive to a clock event. A designer can define this clock event using the "system clock" for the overall system, or he may use some other "aperiodic" triggered signal as the clock, generated by another component in the design.

Once a designer annotates the flowdiagram with clocking information, the operations attached to individual states become scheduled. The behavior of each data operation, and the availability of each result on the output of the data path unit, now depends on two things: (1) the relationship between the selected clocking schemes of the control and data paths, and (2) the selected bus structures of the data paths involved in each operation.

<u>Specifying RTL resource allocation</u>
As in many engineering design disciplines, VLSI design has its basic set of high-level primitive building blocks, from which larger units can be built. At each level of VLSI design abstraction, there is a well-defined set of such primitives. For example, designers use individual flip flops and logic gates at the gate level, and bus-oriented logical and arithmetic functions at the register-transfer level. The flowdiagram notation is built around an implicit understanding of these primitives.

The basic register unit is a *bus*, which is an abstract signal path through the design. Buses are read from--and written to--in expressions of data operations in the flowdiagram. This corresponds to a variable appearing on the left and right hand sides of assignment statements in conventional programming languages such as C.

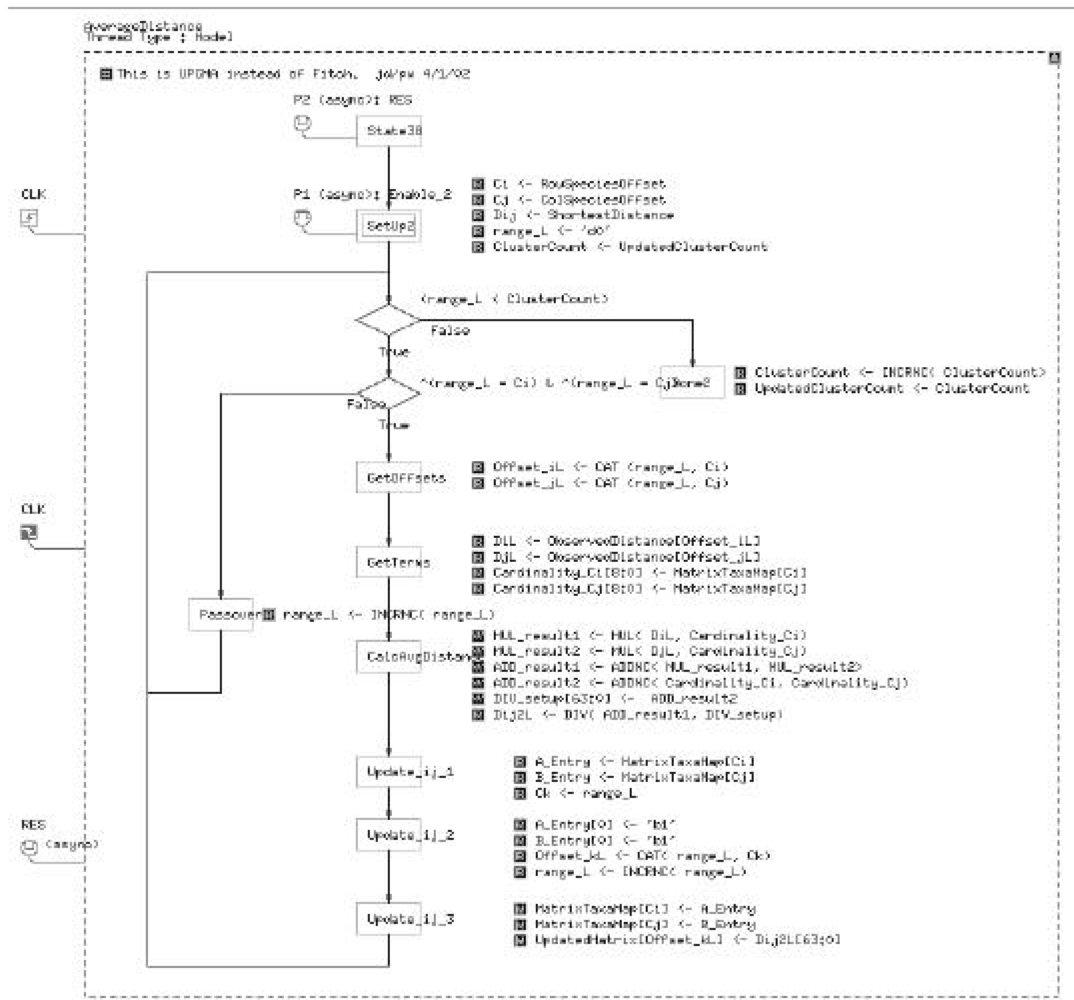

Figure 6. Average distance computation flowdiagram.

A flowdiagram incorporates the notion of *macro-functions* directly into the notation, in that the flowHDL® tool environment supporting flowdiagram creation includes a library of macro-function primitives. Such macros are "functions" in a mathematical sense, taking a set of input arguments, applying a transformation operation on the inputs and returning a result. Macros are used in assignment expressions to specify the behavior of individual data path units, as shown in Figure 6 for the computation of the UPGMA average distance.

<u>Specifying lifetime of data values</u>

One important aspect of algorithm realization in digital logic is determining the lifetime of specific data values (Camposano et al., 1991). In other words, we wish to know how long a specific data value will be valid on the signal path. From a circuit perspective, this lifetime indicates the number of clocking cycles the signal is to be driven to its assigned value before the value is scheduled to change.

Using the flowdiagram notation, we can assign data values to bus, in the same manner as making assignments in a conventional programming language. However the difference is that the data value is held on the bus, across some number of explicit control cycles, until the bus is driven to a different value through a subsequent assignment. In addition, we can "assert" a signal,

indicating that the signal is to go to its "high" value for the duration of the scheduled control cycle. At the conclusion of the current cycle, the signal returns to its default value for subsequent cycles, or for any cycle where not explicitly driven.

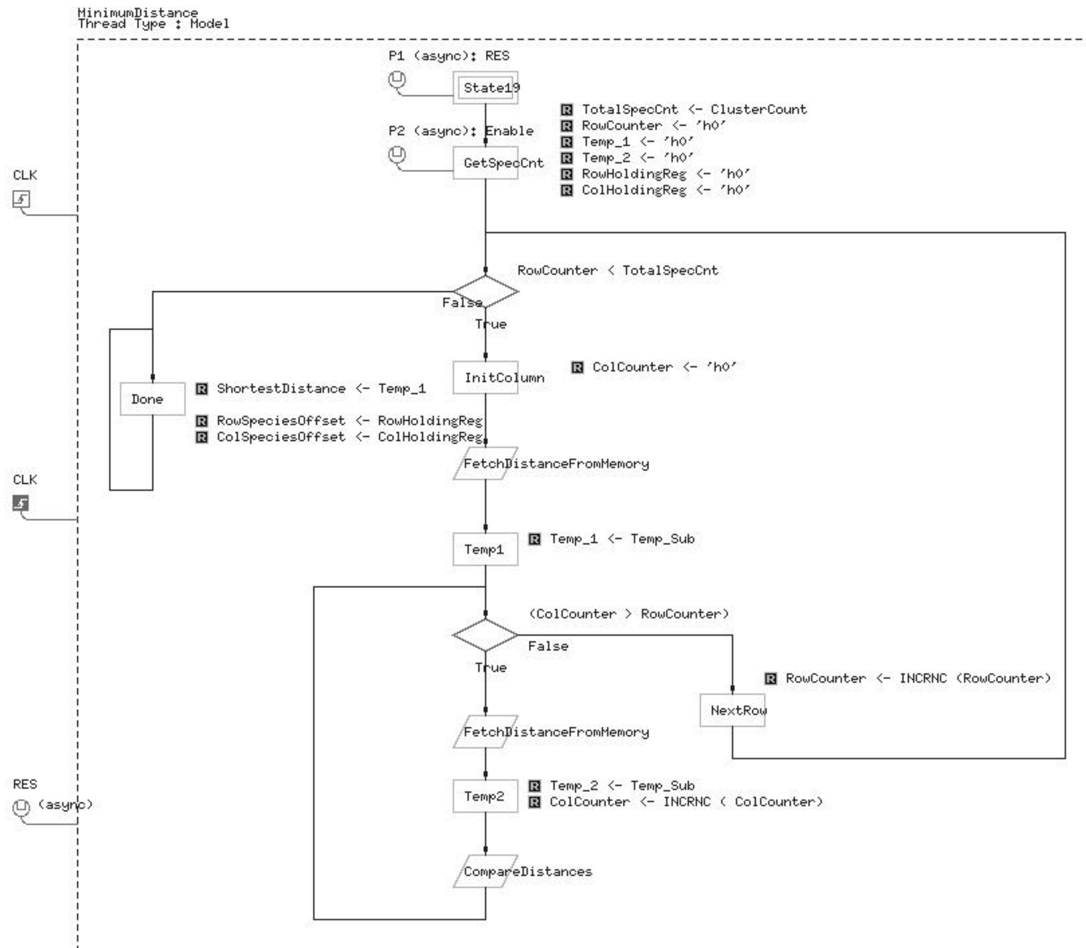

Figure 7. Minimum distance calculation flowdiagram.

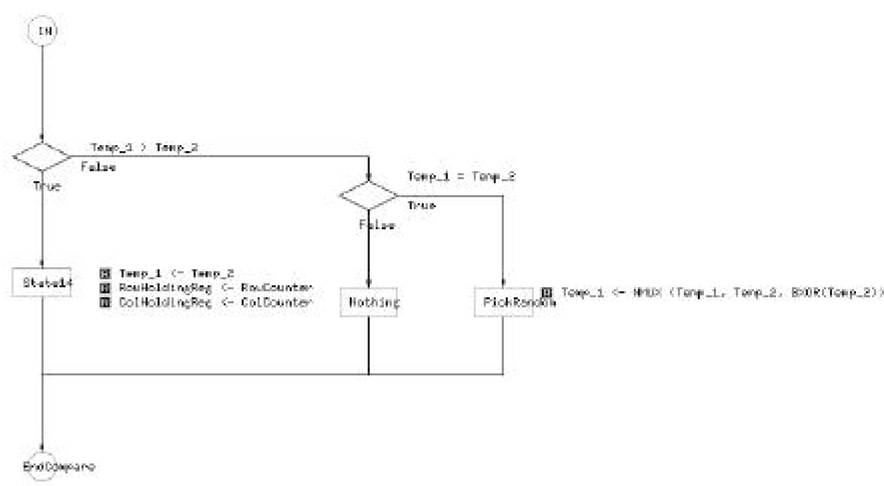

Figure 8. Compare Distances sub-flowdiagram.

5. Summary

In this paper, we have presented an architecture and methodology for realizing phylogenetic algorithms in custom-logic VLSI circuits, and have discussed using these as “engines” to optimize the processing of UPGMA and NJ as one class of computational phylogenetic problems that could benefit from the speedup of moving from general-purpose computing architectures to custom-logic VLSI circuits. We have not discussed the algorithms themselves, but rather, have focused the discussion on how to realize algorithms of this type--although we have shown the flowdiagrams for several key components of the UPGMA clustering method algorithm.

At present we are optimizing the performance of our computational engines, which constitute by far the bulk of the processing time for UPGMA and NJ. One of the key issues arising here involves defining heuristic design approaches that result in the most optimal physical circuits for different classes of algorithms. However, we are already experiencing more than an order of magnitude improvement in computational throughput over standard software-based algorithm realization, and see continued improvements with more optimal circuit designs and increased parallelism through the use of reconfigurable computing techniques (Buell at al. 1996).

One important consideration in applying the approach of this paper to real-world bioinformatics problem spaces is scalability. Memory usage scales as $O(n^2)$ for both UPGMA and NJ. The chips we are presently using have 2 x 256Kb of what is equivalent to Level 1 cache RAM (although with much less overhead due to the absence of fetch/decode/execute times associated with the CPU architectures of general-purpose computing systems). For NJ or UPGMA, this allows storage on a single Xilinx Vertex® FPGA chip of the distance matrix (with 32-bit single precision storage) for the square root of 256,000 or 506 taxa. This is ample for many data sets encountered in phylogenetics and systematics.

However, the resources required for 10,000, thus more genes from a series of microarray experiments are on the order of 400 times this, or requiring greater than 100 Mb of local memory. This can be achieved in two ways, by using resources of multiple custom logic chips or by adopting chip technology simply with more resources. Thus the phylogenetic problems described here fall near one end of a range where available memory local to the algorithm processing is most critical.

On the other hand, for more criteria-oriented distance problems, such as least squares and minimum evolution (Bryant and Waddell 1998), the problems tend to be formulated with 30-300 taxa, where the emphasis is more on intensive computations on these taxa. These problems are also known to be time optimal and appear to be prime candidates for even larger speed increases than those found for UPGMA.

6. References

1. Bryant, D., and P.J. Waddell. (1998). Rapid evaluation of least squares and minimum evolution criteria on phylogenetic trees. Molecular Biology and Evolution 15: 1346-1359.

2. Buell, D.A., J.M. Arnold, W.J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, Chapter 8, “Searching Genetic Databases on Splash 2,” D. T. Hoang, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 97-109.
3. Camposano, R. and W. Wolf (eds.), *High-level VLSI Synthesis*, Kluwer Academic Publishers, 1991.
4. Clare, C.R., *Designing Logic Systems Using State Machines*, McGraw-Hill Publishing Co., 1973.
5. Davis, J., “High-level Design-for-Synthesis: the Next Step”, in *EDA&T 95: Conference on Electronic Design Automation and Test*, Asian Sources Media Group, 1995.
6. Davis, J., S. Nagarkar, and J. Matthewes, “High-level Design of On-chip Systems for Integrated Control and Datapath Applications”, in *Design Supercon-96: On-Chip System Design Conference*, Hewlett Packard Company, Inc., 1996.
7. Durbin, R., S.R. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, 1998.
8. *flowHDL® Reference Manual*, © 1998 KBS Corporation.
9. Gajski, D., and L. Ramachandran, "Introduction to High-level Synthesis", *IEEE Design & Test of Computers*, Vol. 11, No. 4, Winter 1994, pp. 44-54.
10. Hayes, J., *Computer Architecture and Organization*, McGraw-Hill Publishing Company, Inc., 1979.
11. Mead, C. and L. Conway, *VLSI Systems Design*, Addison Wesley Publishing Co., 1978.
12. Swofford, D.L., G.J. Olsen, P.J. Waddell, and D.M. Hillis (1996). Phylogenetic Inference. In: "Molecular Systematics, second edition" (ed. D. M. Hillis and C. Moritz), pp 450-572. Sinauer Assoc, Sunderland, Mass.
13. Tenenbaum, A., Structured Computer Organization, 3rd Edition, Prentice-Hall Publishers, Inc., 1997.
14. Thomas, D., J. Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign", *IEEE Design & Test of Computers*, Vol. 10, No. 3, September 1993, pp. 6-15.
15. Waddell , P.J., and H. Kishino. (2000). Cluster Inference Methods and Graphical Models evaluated on NCI60 Microarray Gene Expression Data. Genome Informatics Series 11: 129-141.