A Scalable Computer Architecture and Algorithm for Sequence Alignment

Bin Wang, Utah State University, Computer Science, bin.wang@m.cc.utah.edu
(Current Address: Celera Diagnostics, 1401 Harbor Bay Parkway, Alameda, CA 94502)
Donald H. Cooley, Utah State University, Computer Science, don.cooley@usu.edu
Nicholas J. Macias, Cell Matrix Corporation, nmacias@cellmatrix.com
Lisa J.K. Durbeck, Cell Matrix Corporation, ld@cellmatrix.com

Keywords: DNA, sequence alignment, Cell Matrix, parallel architecture, big-O

INTRODUCTION

Generally, the algorithm used by sequence alignment tools is a dynamic programming algorithm introduced by Needleman and Wunsch in 1970 [1]. Previous sequential implementations of this algorithm have a time complexity of $O(n^2)$, where n is the length of the input sequences. In this paper, we present a cellular-based hardware organization to implement Needleman and Wunsch's dynamic programming algorithm. The actual implementation uses a recently developed parallel computing architecture called the Cell Matrix™. The Cell Matrix is an architecture that takes advantage of the anticipated massive cellular implementation possibilities of nanotechnology. By trading hardware for software, we show that this organization achieves a time complexity of O(n) in finding the optimal alignment score for two DNA sequences. Thus, while the execution time for a software-based implementation increases by a factor of four for a doubling of sequence lengths, this approach sees only a linear increase of two.

SEQUENCE ALIGNMENT ALGORITHM

The process of aligning two sequences, represented as strings of symbols, involves inserting spaces at appropriate points in either of the sequences in order to maximize the number of matches or score. For an alignment, this score represents a measure of the similarity between the sequences. A typical scoring algorithm is computed as:

+1 for each symbol match
-1 for each symbol mismatch
-2 if either of the two characters is a space

Thus, the optimal alignment problem is to find the maximal score of all possible alignments between two sequences. This score is closely related to the edit distance concept introduced by Levenshtein in 1966 [2].

For two sequences, the number of alignment possibilities increases as 2^n where n is the length of the shorter of the two sequences. The Needleman and Wunsch dynamic programming algorithm for comparing sequences [1] is the basic algorithm for most pair-wise sequence alignment tools used by molecular biologists. This algorithm solves an instance of the problem by taking advantage of computed solutions for smaller instances of the same problem. To find an optimal alignment score F[i, j] of two sequences s[1...i] and t[1...j], we divide the process into three smaller problems:

- Find the optimal alignment score F[i, j-1] of s[1...i] and t[1...j-1], and
- Find the optimal alignment score F[i-1, j-1] of s[1...i-1] and t[1...j-1], and
- Find the optimal alignment score F[i-1, j] of s[1...i-1] and t[1...j].

Knowing the solutions of these three smaller problems, the optimal alignment score F[i, j] of s[1...i] and t[1...j] can only be one of the following:

- Align s[1...i] with t[1...j-1] and match a space with t[j], or
- Align s[1...i-1] with t[1...j-1] and match s[i] with t[j], or
- Align s[1...i-1] with t[1...j] and match s[i] with a space.

The alignment score for each of these solutions is:

- F[i, j-1] – 2 (-2 for a gap between a space and t[j])
- F[i-1, j-1] ± 1 (+1 for match if s[i] = t[j], -1 for mismatch if s[i] ≠ t[j])
- F[i-1, j] – 2 (-2 for a gap between s[i] and a space)

Thus, the optimal alignment score for s[1...i] and t[1...j] is

(1)

This algorithm uses a score matrix F of size (i+1)*(j+1). F[m, n] is the optimal alignment score for s[1...m] and t[1...n]. The values for F[0, 0], F[0, n] and F[m, 0] are known:

- F[0, 0] = 0 because it is the score of the alignment between two empty sequences.
- F[0, n] = -2*n, F[m, 0] = -2*m because they are the scores for aligning an empty sequence to another sequence of length n or m, resulting in a gap of length n or m.

The score matrix can be filled from F[1, 1], row by row, left to right in each row, or any other order that makes sure F[m, n-1], F[m-1, n-1] and F[m-1, n] are available when computing F[m, n]. After the entire score matrix is filled, F[i, j] will be the score for the optimal alignment of s[1...i] and t[1...j]. The optimal alignment can be recovered from F by tracing which of the three choices was chosen to compute F[m, n] from F[i, j] all the way back to F[1, 1]. In Figure 1, the alignment table is shown for two simple sequences. A more complete description of this algorithm is presented by Setubal and Meidanis [3].

Score Matrix F:

		G	A	C	G	G	A	T
	0	-2	-4	-6	-8	-10	-12	-14
G	-2	**1**	-1	-3	-5	-7	-9	-11
A	-4	-1	**2**	0	-2	-4	-6	-8
T	-6	-3	**0**	1	-1	-3	-5	-5
C	-8	-5	-2	**1**	0	-2	-4	-6
G	-10	-7	-4	-1	**2**	1	-2	-4
G	-12	-9	-6	-3	0	**3**	1	-1
A	-14	-11	-8	-5	-2	1	**4**	2
A	-16	-13	-10	-7	-4	-1	2	**3**

Optimal alignment:

```
GA-CGGAT
||  ||||
GATCGGAA
```

Score for optimal alignment:

F[7,8]=3

Figure 1. Example of the Needleman-Wunsch algorithm for DNA sequence alignment.

This algorithm has a space complexity proportional to the size of the score matrix F, which is O(i * j). Since the cells in the score matrix F must be filled one by one, this algorithm also has a time complexity of O(i * j), or O(i^2) if the two sequences have nearly the same length. In truth, the algorithm can be implemented with a space complexity of O(i). The scoring matrix can be filled row by row, such that after filling row i, values of row i-1 can be discarded. Therefore, one only needs to keep at any time, in memory, two rows. However, if this method is used, retrieval of the optimal alignment is complicated since one cannot trace back to F[1,1] with only two rows saved. Recognize also, that such an implementation still has a time complexity of O(i*i).

DNA sequences encoding protein domains typically contain about 10^3 base pairs. For long sequences, the time complexity of such an algorithm is significant. In this paper we show that using an appropriate parallel architecture, the time complexity can be reduced to O(i).

PARALLEL ARCHITECTURE IMPLEMENTATION

In a serial implementation, the cells of the table would be computed sequentially along diagonals formed by a wave front moving from the upper-left of the table (F[0,0]) to the lower right (F[i,j]). Note that after F[1,1] is computed, either F[1,2] or F[2,1] can be computed. This is because F[1, 1] F[0, 1] F[0, 2] are available for F[1, 2], and F[2, 0] F[1, 0], F[1, 1] are available for F[2, 1]. Thus, F[1,2] and F[2,1] can be computed in parallel, along with the other entries on the same diagonal line, F[3,0] and F[0,3]. If the entries in each diagonal of the wave front are computed in parallel, the computation time complexity is reduced to

$$O\left(\sqrt{i^2 + j^2}\right)$$

This is the length of the diagonal of the score matrix F. If the two sequences have similar length, this bound is O(i), or, more commonly, O(*n*), where n is the length of either sequence. Note, however, that the space complexity is increased at this lower time complexity, because it requires many more processors operating in parallel.

In this paper we present an implementation that has time complexity of O(*n*) and a space complexity of O(n^2). To date, such a time complexity has not yet been achieved, and no research has reported such a sequence alignment machine.

The approach taken in this work is to develop a simple custom processor that takes in {F[i, j-1], F[i-1,j-1], F[i-1,j]} and computes and outputs the score F[i,j]. Parallelism is achieved by linking these processors in a 2D array that corresponds closely with the score matrix F in Figure 1. Each processor computes its value based on its inputs, and then outputs the result to the next processor. Sequence data are input through the top and left edges of the 2D array. When the processor array is stabilized, the right bottom unit contains the optimal alignment score.

An alternative to this approach is to code a parallel implementation of the algorithm for a multiprocessor machine or a cluster of machines. However, to achieve the desired time complexity, an extremely large number of processors is needed: 10^6 processors for a typical DNA sequence length of 10^3. Implementation issues aside, processor latency and inter-processor communication delay are expected to become a serious problem at that magnitude. The approach taken in this work allows a system design that closely resembles the original algorithm. And, by carefully

designing the custom processor, latency and inter-processor communication times are minimized. This allows for a better constant factor for the time complexity calculation. Finally, because of the simplicity of the custom processor, the hardware requirement is minimized. This is permitted because the actual operations required in parallelizing such an algorithm are relatively simple; if one were to instead use an array of microcomputers, considerably more processing power would be available than is needed.

The custom processor designed for this work was designed and simulated using the Cell Matrix architecture and tool set. Cell Matrix hardware is extremely fine-grained reconfigurable logic formed from a cellular array of gate-level processors. Implementation of the custom processor was achieved by configuring a 25x27 array of Cell Matrix cells as a custom processor to perform the function of computing an individual value in the score matrix F. These were then tiled out on a large array of Cell Matrix cells to achieve the 2D array of custom processors corresponding with the score matrix F.

This implementation choice was made because it fit the requirements of the current stage of the project and is also a good candidate for ultimately being able to construct a large hardware/software system for fast DNA sequence pairing. At this early stage of research, simulated hardware is appropriate, because of the focus on coming up with the design and the need to iterate through many design cycles to arrive at the final design. It also simplifies debugging by limiting the diagnosis of problems to those stemming from the design itself. The design and simulation tools used give an accurate, cell-level simulation of the design and timing information. Debugging is made easier by the fact that we could dictate the physical layout of the processors, and have it closely correspond with our conceptual layout of both the processor internals and the score matrix. Because the design and implementation are compartmentalized and modular, it is possible to build a small system to design and test the approach; because of the modularity, it is also plausible that much of this work will translate to a larger-scale system. At later stages of the project, it is theoretically possible to use etremely dense manufacturing technologies, even nanotechnology, to build the large Cell Matrix hardware array needed. This appears possible because of the regular, redundant structure of the hardware and its local fault isolation, as well as the software level's distributed configuration and control mechanisms [4] that lead to reasonable configuration setup times and scalable control systems. Initial work done by the authors in automatic local fault detection and handling [5, 6] may also lead to a means to handle the high fault incidence rate in such a large system.

Cell Matrix Background

The Cell Matrix architecture is a programmable cellular hardware array with very fine-grained, distributed software control [7,4]. Unlike traditional CPU/memory architectures, the Cell Matrix hardware is homogeneous and regular in structure. The basic element of a Cell Matrix is a cell. Cells are interconnected to form a matrix of cells. Each cell accepts inputs from its neighbor cells, processes the inputs based on its internal reconfigurable memory, and outputs the result to its neighbor cells. All cells in the architecture are physically identical. The contents of each cells' memory and its control inputs determines its functionality.

An example of a Cell Matrix cell is shown in figure 2. Such four-sided cells can be connected to form a two dimensional Cell Matrix with each cell connected to four neighbors. Other topologies are also possible: six-sided cells can form a honeycomb like structure, or a cuboid. Each cell has a data input, a data output, a control input and, a control output on each side. A 4-sided cell also has an internal memory of 16 rows (inputs) by 8 columns (outputs). A cell operates in one of two modes: Data mode or Control mode. In Data mode, all of the control inputs are 0 and the cell is a combinatorial logic circuit. In this mode, it uses its four data inputs and the internal memory to determine its outputs. To be in Control mode, at least one of the four control inputs must be 1. In this mode, the cell's internal memory can be reconfigured by serially shifting the data input into the cell's internal memory according to a system wide clock. When the cell returns to Data mode, the reconfigured memory gives it a new functionality.

A cell or group of cells can be configured to perform many different functions, such as a piece of wire, or a logic gate, or a more complex function. The cell shown in Figure 2 has been configured as full adder (Figure 3). It adds data input from the north and south lines, with a carry input from the west line. It outputs the sum to the south and carry to the west lines. Several such cells can be linked to form an n-bit adder. Certainly, more complex circuits can also be implemented on this architecture.

Because the cells can be individually configured, the architecture supports a one-problem, one-machine model of computing. The circuit can be specifically designed for the particular problem to be solved. This architecture also provides a platform for distributed and parallel computing.

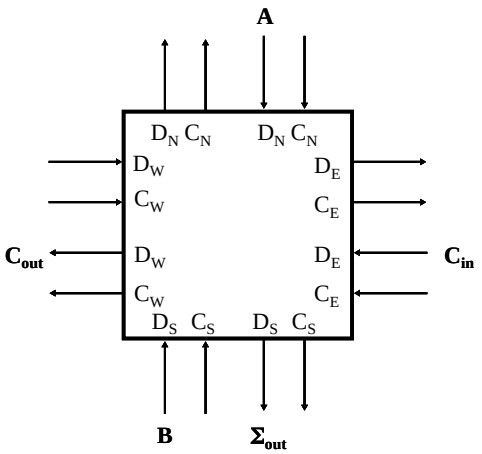


Figure 2. Cell Matrix cell Diagram

Inputs				Outputs							
D_N	D_S	D_W	D_E	C_N	C_S	C_W	C_E	D_N	D_S	D_W	D_E
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	1	0	0
0	1	0	0	0	0	0	0	0	1	0	0
0	1	0	1	0	0	0	0	0	0	1	0
0	1	1	0	0	0	0	0	0	1	0	0
0	1	1	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0
1	1	0	1	0	0	0	0	0	1	1	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	1	1	0

Figure 3. A Cell Matrix cell configured as a 1-bit full adder.

Different parts of a matrix can be configured to perform different tasks, or similar tasks with different data. Thus, a problem can be distributed spatially rather than temporally achieving reductions in processing time. Because of its support for distributed and parallel computation, an extremely large number of switches can be efficiently utilized in a Cell Matrix architecture. An example is a DES cracker constructed using a Cell Matrix [4]. This design can achieve O(1) run time in finding an encryption key. Another important feature of this architecture is that a cell's internal memory can be reconfigured by other neighboring cells. This feature allows a designer to build dynamic, self-modifying circuits. Some interesting examples of this feature include a parallel, distributed genetic algorithm [8], evolvable hardware, and run-time fault tolerant hardware [5, 6].

Custom Processor Design

Each custom processor is responsible for the value of one cell in the score matrix. Figure 4 shows the basic design of a single custom processor unit. The processor unit has five inputs, two of which contain the sequence data s[i] and t[j]. The other three inputs are used in computing score matrix cell values from the neighbor processors. In each case, the processor compares the sequence data inputs s[i] and t[j], and adds or subtracts one from F[i-1, j-1] according to the comparison result. It also subtracts two from F[i, j-1] and F[i-1, j]. These three results are then compared, and the largest result is used as the value of F[i, j]. Finally, F[i,j] is passed to its neighbor. There are several basic circuitries in this processor cell. Each circuit executes a simple function, and they work together to perform the function of computing a value for that cell of the score matrix. These basic circuitries are shown as boxes inside the processor in Figure 4. Because of space limitations, only the sequence comparison unit is shown in Figure 5.

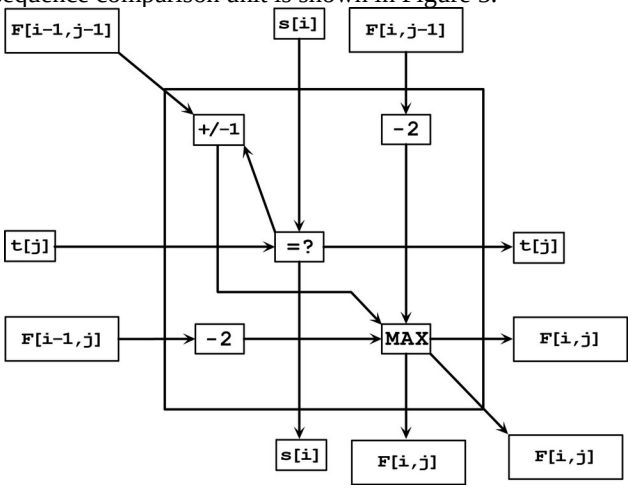
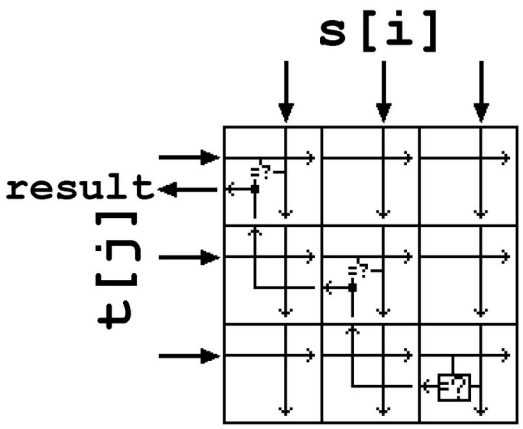


Figure 4. Processor Architecture



DS=N DE=W DW=(N&W + ~N&~W) & S	DS=N DE=W	DS=N DE=W
DS=N DE=W DN=E	DS=N DE=W DW=(N&W + ~N&~W) & S	DS=N DE=W
DS=N DE=W	DS=N DE=W DN=E	DS=N DE=W DW=N&W + ~N&~W

Figure 5. Sequence Comparison Unit

The sequence comparison unit compares the two sequence data values s[i] and t[j]. The result of this comparison is used to control the +/-1 unit, which adds or subtracts one from F[i-1, j-1] according to the comparison result. Two –2 units are used to subtract 2 from F[i, j-1] and F[i-1, j]. And finally, a MAX unit is used to select the largest value and pass the value to its neighbor processors.

This design requires that the processor exchange information with its six neighbors, two of which are on its diagonal, F[i-1, j-1] and F[i+1, j+1]. However, four sided cells in a Cell Matrix cannot directly pass information diagonally, because data I/O ports are only on the sides of the cell. Therefore, this information exchange must be routed through neighbors as shown in figure 6.

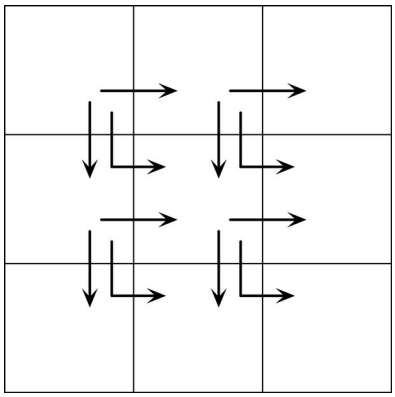


Figure 6. Cell Array Routing Map

In this design, different values in the sequence data are represented using 3 bits: 000 for A, 001 for G, 010 for C and 011 for T. The fourth bit is reserved for other possible sequence data values, such as U (uracil) in a RNA sequence, or N for bad sequence read out. Values in the score matrix F are represented using 9 bits. The first bit is the sign bit. Negative numbers are represented in 2's complement format. This gives a range of –256 to +255, which is enough to align two DNA sequences of up to 128 base pairs. Such an encoding scheme can be readily scaled to accommodate longer sequences.

Implementation on Cell Matrix Architecture

As shown in Figure 5, the sequence comparison unit is a 3x3 block of Cell Matrix cells. Sequence data inputs are on the top and left edges of the unit. Corresponding bits of the sequence data are compared, and the output of the unit is the logical AND of the three individual bit comparison results. This sequence comparison unit's output is 1 when two input sequence data are the same, and 0 otherwise. This unit also allows sequence data to pass through it, so that the custom processor can pass the sequence data to its neighbor processors. Since all negative numbers are represented in 2's complement format, all addition and subtraction can be performed using adders.

The custom processor designed for this application requires a nine-bit adder. The +/-1 unit and –2 unit are both based on this nine-bit adder.

The +/-1 unit adds 1 or –1 to an input value according to the output of the sequence comparison unit. The sequence comparison unit outputs 1 when s[i]=t[j], and the processor adds 1 to F[i-1, j-1]. Otherwise, the sequence comparison unit outputs 0, and the processor adds –1 to F[i-1, j-1]. In the +/-1 unit, F[i-1, j-1] is one of the two inputs to the nine-bit adder. The other input of the adder is either 1 or –1. Which value to input is controlled by the sequence comparison unit. The input is 1 when the sequence comparison unit's output is 1 (s[i]=t[j]), and –1 when the

output is 0 (s[i]≠t[j]). This can be achieved by setting the eight higher order bits to the complement of the output of the sequence comparison unit and lowest order bit to 1, because 1 is 000000001 and –1 is 111111111 when represented using nine bits in 2's complement form. A block of nine cells can be configured to perform such a function. These nine cells send a nine-bit value to the adder on its left. The control input is on the top and passed to all nine cells. The eight higher order bits are the reverse of the control input, and the lowest bit is always set to one. The nine-bit value sent to the adder is 000000001 (=1) when the control input is 1, and 111111111 (=-1) when the control input is 0. The adder adds this value to the F[i-1, j-1] input on its left, and sends the result to its right.

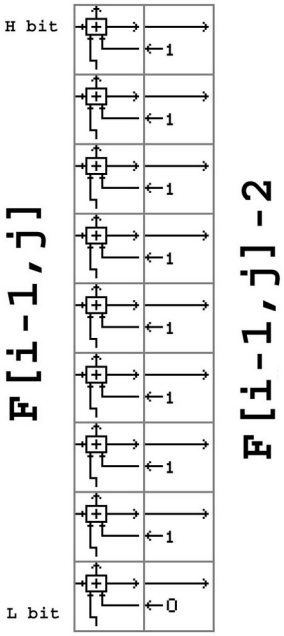


Figure 6. Vertically oriented –2 unit

The –2 unit is very similar to the +/-1 unit. Because it always subtracts 2 from its input, there is no need for a control input. The adder in this unit will always have an input of –2. The 2's complement of –2 is 111111110. The nine cells used to send +/-1 to the adder can be configured to send this value to the adder at all times. The other input of the adder will be F[i-1, j] from the neighbor process to the left, or F[i, j-1] from the neighbor processor above. A vertically configured –2 unit with high order bit on the top and low order bit on the bottom can be used to form F[i-1, j], which is passed to the unit from the left (Figure 6). For the F[i, j-1] that comes from the top, a horizontally configured -2 unit can be used. The input is on the top of the unit with the high order bit on the left and low order bit on the right, and output is on the bottom of the unit.

The sign bit of A-B can be used to compare A and B. If the sign bit of A-B is 0, then A-B is positive and A≥B. If the sign bit of A-B is 1, then A-B is negative and A<B. A-B can be performed using an adder by representing B in 2's complement form. Again, a simple adder can be organized to perform this operation.

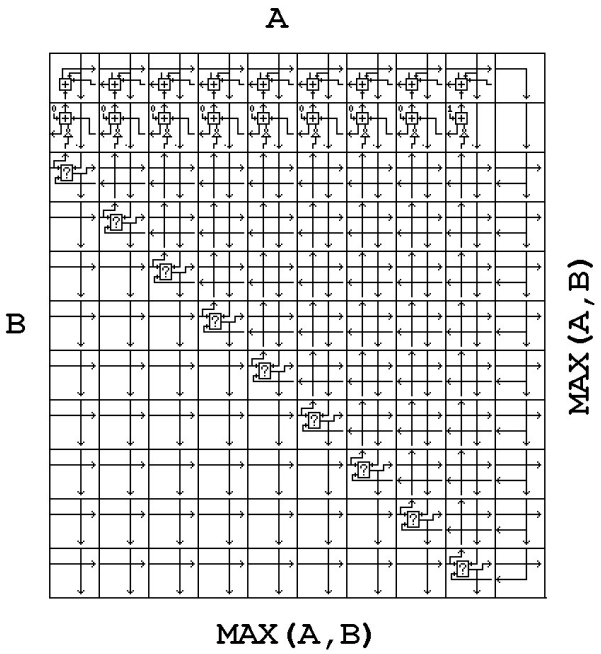


Figure 7. Max selection Unit

Figure 7 shows a MAX selection unit. This unit is composed of a 9x2 A-B unit on the top and a 9x9 selection unit on the bottom. Two data inputs A and B are on the top and left of the unit. Data B is passed to the A-B unit by the selection unit. The sign bit of A-B is passed to the selection unit as a selection signal. This signal is 0 when A≥B and 1 when A<B. The cells on the diagonal have three inputs: A from the top, B from the left and the selection signal from right. It also passes B from its left to the A-B unit on the top. The output of these cells depends on the selection signal. If A≥B, they will receive selection signal 0 and output A. Otherwise, they output B. So this MAX unit can be used to select larger number from two inputs. To select the largest value from three numbers, as required in this custom processor, two of this MAX unit can be linked together. The output of the first MAX unit, the larger one of the first two numbers, is passed to the second MAX unit to be compared with the third number. The output of the second MAX unit is then the largest value of the three numbers.

A Cell Matrix implementation of a single custom processor is shown in figure 8. The underlying hardware is a 25x27 block of Cell Matrix cells, and these have been configured with the appropriate truth tables to implement the subcomponents, and wiring, of the custom processor. The

implementation is composed of the basic processing elements discussed previously. The processor has six inputs and six outputs. Sequence data s[i] and t[j] are compared in the sequence comparison unit at the top left corner of the processor. The comparison result is then linked to the control input of the +/-1 unit. F[i-1, j-1] input is on the middle of the left edge and linked to the +/-1 unit. The output of +/-1 unit, F[i-1, j-1]±1, is connected to the MAX selection unit. F[i, j-1] inputs is on the middle of the top edge, it is connected to a horizontally configured –2 unit. F[i, j-1]-2 is then connected to the MAX selection unit. F[i-1, j] inputs is also connected to a –2 unit, F[i-1, j]-2 is the third input of the MAX selection unit. The output of the processor is the output of the MAX selection unit. On the top right corner is a direct pass for F[i, j-1]. This is necessary as processor [i, j-1] and [i+1, j] cannot communicate directly. Further implementation details are available [9].

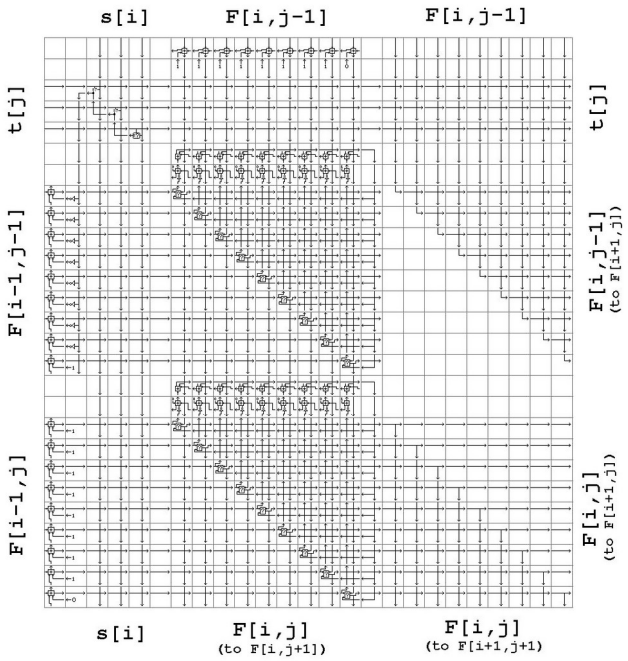


Figure 8. Cell Matrix Processor Implementation

Such a 2D array of these custom processors can be used to perform DNA sequence alignment. As stated previously, the algorithm requires that F[0, 0], F[0, n] and F[m, 0] be initialized to F[0, 0]=0, F[0, n]= -2*n and F[m, 0]= -2*m. Cell Matrix circuitry can be added to the top and left edges of the 2D processor array to initialize these values.

An important feature of this 2D processor array circuit is its symmetry and redundancy. This permits the use of a previously developed method of configuring the hardware that exploits the local self-configuring mechanism to be used to set up the array of processors in parallel, again in a wave front manner, in time

$$O\left(\sqrt{i^2 + j^2}\right)$$

This becomes important when the full-scale system can be built, configured and used.

TEST RESULTS

The custom processor design and layout was developed and tested using software tools previously developed by Cell Matrix Corporation and made freely available for noncommerical use. The design was laid out using the preliminary version of the graphical Layout Editor[10]. It was simulated, modified and debugged using version 2 of the graphical simulator[11]. These tools provide a GUI interface for developers to quickly build and test circuits.

A sample 2D processor array was constructed using the Layout Editor. Its correct function was tested using a batch-mode, command-line version of the simulator that provides timing information. This simulator runs considerably faster than the graphical simulator when simulating a large number of cells: for this project $675 \times n^2$ cells were used for each sample 2D processor array, where n is the sequence length. Sequences of length 240, 398, 559, and 730 base pairs were tested. This simulator has to simulate a large number of parallel operations at any timestep on a sequential desktop computer, and in a large simulation, the time required becomes prohibitive. It is expected that in the not too distant future, chip-level versions of the Cell Matrix will become available that can be used directly for prototyping Cell Matrix circuits.

The testing of the design and implementation was done by first testing the individual low-level components of the processor, and then testing the processor array using a representative sample of input sequences. Correct operation of the array was determined by the match score output by the array, which could be compared against the expected outcome. Once the design was verified as correct, timing tests were also run.

Timing of the Alignment

The goal of this research was to design a Cell Matrix circuit that could align two DNA sequences in O(n) time. To obtain timing information for the sequence alignment circuit, it was first allowed to stabilize, and then a new pair of sequences were input. The time for a sequence alignment to be determined is simply the time for the circuit to stabilize. The unit of time for this measurement was the

propagation delay of a single Cell Matrix cell, or the time for a Cell Matrix cell to respond to an input change.

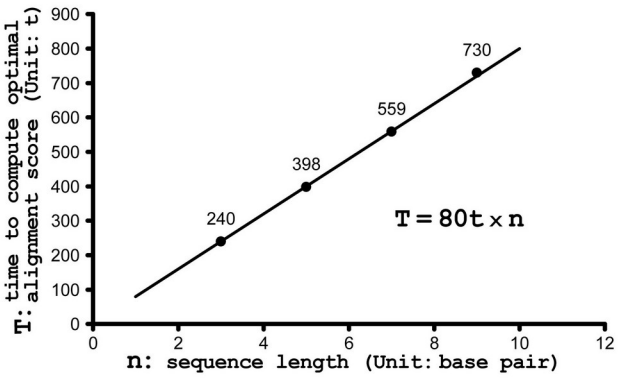


Figure 9. Base Pair Length versus Time to Align

As noted, in a DNA sequence alignment circuit, the optimal alignment score is obtained when the processor array is stabilized. Thus, the time for the circuit to stabilize is the time for the circuit to compute the optimal alignment score. Measuring this time for processor arrays of different sizes shows how computation time increases with sequence length.

The timing results are shown in Figure 9. As shown in the plot, the time to compute the optimal alignment score increases linearly with the length of the input sequences: $T \approx 80t \times n$, where T is total computing time, t is the typical delay for a single Cell Matrix cell, and n is the length of the input sequences. The results verify that the design archived the goal of finding an optimal alignment score for two sequences in O(n) time.

DISCUSSION

This implementation of the dynamic programming algorithm is the first sequence alignment system that achieves O(n) time complexity. The major cost here is hardware, not compute time. Instead of computing the values in the score matrix F one by one using a single processor, a processor array of size $n \times n$ is used. The problem of finding the optimal alignment score is broken down into smaller tasks. These smaller tasks are then distributed spatially over the processor array and performed in parallel by individual processors. As the length of the sequences increase, the size of the processor array, or the hardware cost, also increases. Because the hardware cost increases with a complexity of O(n^2), hardware cost becomes an issue for long sequence alignments.

DNA sequences encoding protein domains are typically about 10^3 base pairs long, aligning two such sequences would require a processor array of size 10^6. In this design, each processor is assembled using 675 Cell Matrix cells. Thus, a processor array for alignment of two sequences of length 1,000 each would require about 7×10^8 cells. Although the state of the art is rapidly progressing, current silicon techniques permit only about 500,000 cells on a chip, which is enough to align two sequences of only about 30 base pairs each. Multi-chip Cell Matrices can be built, but not to the scale required here. To effectively utilize the Cell Matrix architecture, much denser manufacturing techniques are required. With the advent of nanotechnology, biology-based computing and other molecular engineering techniques, extremely large Cell Matrix configurations will be both possible and practical [4]. The development of the nanocircuit was named the breakthrough of the year (2001) by Science magazine [12]. Examples of the experimental nanocircuit include carbon nanotube transistors [13] and logic gates made of nanowires [14]. Although current nanocircuits are still very simple and rudimentary, they portend a bright future for nanocomputing.

The timing result given by the Cell Matrix simulator is in terms of an abstract unit t, which is the typical delay of a single Cell Matrix cell. This sequence alignment system is a pure combinatorial circuit, and therefore this computing time is a function only of the propagation delay t. This time will depend on the underlying technology of the circuits. For a small 128 cell feedback loop Cell Matrix circuit built with a Xilinx Spartan-II FPGA it has been shown that the single cell propagation delay is approximately 3.9 nanoseconds. The computing time for each base pair alignment is ~80t, or about 3 microseconds. If a sufficiently large Cell Matrix circuit were available, it could align two sequences of 3 million base pairs each in under one second. This is orders of magnitude faster that that attainable on the fastest PC performing a standard serial alignment.

FUTURE WORK

The goal of this work was to trade hardware for execution time, and thus obtain a better growth rate on sequence alignment time. The algorithm presented in this work only finds the optimal alignment score, not the optimal alignment. The obvious next step is to maintain information at each step and output what that optimal pairing is. Additional hardware would be required to perform this task, but the process would not add significantly to the processing time. The next step in this work will be to incorporate that capability into a Cell Matrix-based implementation. It would also be interesting to include some improvements to the alignment algorithm such as a non-linear gap cost function, and more complex scoring system.

While nanotechnology progresses, it has not progressed to the point that a large Cell Matrix array can be built. Meanwhile, we are working with traditional methods such as silicon fabrication techniques to build the Cell Matrix-based DNA alignment circuit presented here. With it

we will be able to see whether we can demonstrate in real hardware the same O(*n*) scalability we saw in simulated hardware. We will also be able to determine the single cell propagation delay t with precision for a particular hardware implementation. We expect this to be better on native hardware than the propagation delay recorded for cells implemented as circuits on an FPGA. We also intend to investigate minimizing the cell usage in the processor design: the design presented here was layed out by hand in the way most obvious and easy to generate and debug; significant reduction in the number of cells required to implement the processor is likely from optimizing the layout for minimal size.

[1] S.B. Needleman and C.D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." Journal of Molecular Biology, 48:443-453, 1970.

[2] V.I. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals." Soviet Physics Doklady, 6:707-710, 1966

[3] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. Boston: PWS, 1997.

[4] L. Durbeck and N. Macias. "The Cell Matrix: an architecture for Nanocomputing." Nanotechnology 12:217-230, 2001

[5] L. Durbeck and N. Macias. "Defect-tolerant, fine-grained parallel testing of a Cell Matrix." Proc. SPIE ITCom 2002 Series 4867 ed J Schewel, P James-Roxby, H Schmit and J McHenry 71-85, 2002

[6] N. Macias and L. Durbeck. "Self-Assembling Circuits with Autonomous Fault Handling." Proc. The 2002 NASA/DOD Conference on Evolvable Hardware ed A Stoica, J Lohn, R Katz, D Keymeulen and R Salem Zebulum 46-55, 2002.

[7] N. Macias. "The PIG paradigm: the design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. Proceedings of the First NASA/DoD Workshop on Evolvable Hardware." ed A Stoica, D Keymeulen and J Lohn 175-180, 1999

[8] N. Macias. "Ring around the PIG: a parallel GA with only local interactions coupled with self-reconfigurable hardware platform to implement an O(1) evolutionary cycle for EHW." The 1999 Congress on Evolutionary Computing 1067-1075, 1999

[9] B. Wang. "Implementation of a Dynamic Programming Algorithm for DNA Sequence Alignment on the Cell Matrix Architecture." M.S. Thesis, Utah State University, 2002.

[10] http://www.cellmatrix.com/entryway/products/software/layoutEditor.html, accessed Oct 15, 2002

[11] http://www.cellmatrix.com/entryway/products/software/simulator.html, accessed Oct 15, 2002

[12] R. Service. "Molecules get wired." Science 294:2442-2443, 2002

[13] A. Bachtold, P. Hadley, T. Nakanishi, and S. Dekker. "Logic circuits with carbon nanotube transistors." Science 294:1317-1320, 2001

[14] Y. Huang, X. Duan, Y. Cui, L. Lauhon, K. Kim, and C. Lieber. "Logic gates and computation from assembled nanowire building blocks." Science 294:1313-1317, 2001.

BIOGRAPHY

Bin Wang is a PhD candidate in Biochemistry at the University of Utah and is currently finishing a dissertation on using Nuclear Magnetic Resonance and other biochemical techniques to study protein structure and protein ligand interaction, one component to the effort to understand the structure and functioning of the HIV Virus. Bin received an M.S. degree from Utah State University in Computer Science in 2002 for the work described in this paper.

Dr. Donald H. Cooley is

Nicholas J. Macias and Lisa J.K. Durbeck are the inventors of the Cell Matrix architecture, along with Murali Dandu Raju and Lawrence B. Henry III. As directors of Cell Matrix Corporation, and as Adjunct Professors at Utah State University, Macias and Durbeck have been actively engaged in the research, development, education and advocacy needed to bring the architecture into common use, by researchers within a wide variety of disciplines.