Abstract

Existing specialized hardware architectures and implementations for biological sequence alignment are presented. A number of different approaches exist to perform sequence alignment in hardware. Among these, FPGAs are prevalent in both commercial and academic projects, as they provide convenient, inexpensive, dynamically re-configurable implementations of common sequence alignment algorithms in hardware. Specialized VLSI chips have been designed which provide significant performance advantages. Among other architectures, the use of linear computing SIMD instructions (DSPs, Intel MMX, and PowerPC Altivec) can be used to perform sequence alignment computations in parallel on modern commodity PCs, allowing some algorithms to run up to 6 times faster than the naive implementation. The exploitation of Micro-grained parallelism, instruction-level parallelism, and other techniques are also described.

1 INTRODUCTION

As biological sequencing methods and technologies mature, the quantity of new biological data being generated is growing at an exponential rate. GenBank, a database of DNA sequences from more than 100,000 organisms has grown from 78,000 sequences in 1992 to well more than 15 million sequences in 2002, representing a growth of nearly 20,000% in the last ten years [1]. Figure 1 (below) shows the rate of growth of GenBank, a large public repository of DNA and RNA sequences, from 1982 until 2000. In short, Moore’s Law can’t keep up with the explosion of new biological data, and the gap is only expected to widen.

Finding alignments (regions of high similarity) between two or more sequences is critical step in the analysis of any sequence. Aligning sequences allows scientists to discover new genes or locate known genes, which is often critical to drug discovery or disease diagnoses. In addition, sequence alignment allows researcher to derive possible evolutionary relationships between organisms based on similarity of sequenced data. As the various databases of sequences grow, it becomes necessary not only to find good alignments for across a handful of sequences, but to
quickly and efficiently (measured in seconds) match a sequence against vast databases of tens of millions of sequences.

The mathematically rigorous and proven optimal approach to aligning just two sequences, known as *dynamic programming pairwise alignment* [4][5], is computationally expensive and has been shown to have a $O(n^2)$ time complexity.

![Growth of GenBank](image)

**Figure 1: Exponential Growth of GenBank. Source: [1]**

Although dynamic programming can be extended to additional dimensions, it is not computationally feasible for the multiple sequence alignment (MSA) of more than a handful of sequences. Most approaches to multiple sequence alignment and sequence database alignment involve the use of heuristics to cleverly reduce search spaces and quickly produce alignments or
matches which are of a high quality, but not necessarily optimal. In the general case, this appears to be largely acceptable for most applications.

Clearly, performance during alignment is absolutely critical today, and will likely be equally critical in the future. Fortunately, most alignment algorithms are easily parallelizable, as will be described later in this paper. Indeed, most are embarrassingly parallel, including dynamic programming. Many implementations exist which are written to exploit massively parallel supercomputer architectures or distributed computing environments such as relatively low-cost Beowulf clusters.

Additional performance gains are attainable by implementing alignment algorithms in specialized hardware, which is nearly always considerably faster than implementing algorithms in software, on any kind of computer.

Duncan A. Buell [19] writes:

> It is a basic observation about computing that generality and efficiency are in some sense inversely related to one another; the more general-purpose an object is and thus the greater the number of tasks it can perform, the less efficient it will be in performing any of those tasks. Design decisions are therefore almost always compromise…

> To counter the problem of computationally intense problems for which general-purpose machines cannot achieve the necessary performance, special-purpose processors, attached processors, and co-processors have been built for many years…

Along these lines, several academic, research, and commercial groups have developed specialized hardware “bio-accelerators” which focus primarily on the problem of performing high-quality sequence alignment at hardware speeds. Some use programmable FPGAs, some use custom-made VLSI processors while others make full use of new architectures in commodity PC processors to achieve considerable instruction-level parallelism (ILP).

All implementations use algorithms which were first developed in software, and in that sense add no new fundamental insight to the art and science of sequence alignment. However, to a biologist, even just a modest doubling in performance can mean the difference between a feasible and infeasible experiment. It is in this context that hardware acceleration of biological sequence alignment is critically important.

## 2 INTRODUCTION TO DYNAMIC PROGRAMMING

Most of the hardware systems described in this paper implement either the Smith-Waterman pairwise local alignment algorithm [4], which was subsequently improved by Gotoh [2], or the Needleman-Wunsch pairwise global alignment algorithm [5]. Both algorithms are closely related and are simple, mathematically rigorous, and easily parallelized….to a point. Both Smith-
Waterman and Needleman-Wunsch are slightly different forms of a *dynamic programming* algorithm. Sequence comparison algorithms based on dynamic programming have been proven to produce an optimal alignment [4]. The dynamic programming technique has a time complexity of $O(nm)$ where $n$ and $m$ are the lengths of the sequences being aligned. Assuming that we are aligning sequences of similar length, this notation is typically shortened to $O(n^2)$. Further, dynamic programming applied to $n$ sequences has been shown to be an NP-Hard problem, hinting that Smith-Waterman or Needleman-Wunsch is as good as any possible optimal sequence alignment algorithm. Because of this, fast multiple sequence alignment (MSA) algorithms use heuristics to produce alignments which are not guaranteed to be optimal.

Dynamic programming is a mathematical term for an approach to decision problems where each, best step towards the solution is calculated and cached in order to eventually arrive at a complete path which describes an optimal solution.

For the remainder of this section, let us consider local sequence alignments specifically using the *Smith-Waterman* algorithm. When performing pairwise alignments (comparing two sequences), Smith-Waterman is implemented using a 2-dimensional matrix, with one sequence represented across the top and the other sequence represented down the left side of the matrix. The first row and the first column are initialized to 0.

![Figure 2](image)

**Figure 2** – (A) shows a blank, initialized matrix for performing dynamic programming while (B) shows a partially completed matrix

Smith-Waterman uses a scoring system in which residue (character/nucleotide) matches are
given a positive score, mismatches are given a negative score, and gaps are given a large negative score. For matrix (B) in Figure 2, the following scoring table was applied:

\[
\begin{align*}
\text{MATCH} &= +2 \\
\text{MISMATCH} &= -1 \\
\text{GAP PENALTY} &= -2
\end{align*}
\]

More sophisticated scoring mechanisms with a basis in empirical molecular biology can be applied, especially when amino acid (protein) sequences are being compared. In that case substitution matrices are generally used. A notion of the relative frequency of finding one amino acid immediately following another is built into the substitution matrix. This has been determined on an empirical (and sometimes theoretical) basis. The use of substitution matrices ultimately produces alignments which are more biologically meaningful. Examples of real substitution matrices include BLOSUM and PAM [41].

Additionally, affine gap penalties can be used, wherein the penalty of a gap is determined by the number of gaps immediately preceding the current gap. The central idea behind affine gap penalties is that a gap at the end of a long chain of gaps should incur less of a penalty than a gap in the middle of an otherwise well-aligned sequence.

After initializing the 2D matrix as described above, the Smith-Waterman algorithm fills in the matrix. The value of each cell in the matrix is determined by the following recurrence equation:

\[
M_{i,j} = \begin{cases} 
S_{i,j} & \text{Match/Mismatch Score} \\
\text{GAP} & \text{Gap Penalty} \\
M_{i-1,j-1} + S_{i,j} \\
M_{i,j-1} + \text{GAP} \\
M_{i-1,j} + \text{GAP} 
\end{cases}
\]

Figure 3 – Calculating matrix values for dynamic programming. Notice that this is a recurrence equation. The dependencies on previous values in the matrix will impact the amount of parallelism that can be exploited, and will guide the choice of suitable hardware architectures, as described later.

The matrix cell \(M_{i,j}\) in matrix \(M\) with the largest value represents the end of the path to the optimal alignment. Note that with local alignments, there may be more than one cell with an identical maximum value, indicating that there are multiple possible optimal alignments.
Once the matrix has been populated and the cell with the maximum value has been identified, Smith-Waterman then backtracks, using cached pointers along the way representing the path taken to get to the highest score. The backtracking terminates when it hits a cell containing a zero value. The backtracked path represents the optimal alignment in reverse order.

As mentioned previously, sequence alignment using dynamic programming can be applied to \( n \geq 2 \) sequences. With dynamic programming, if \( n \) sequences are being simultaneously compared, then an \( n \)-dimensional matrix must be used, increasing the time and space complexity exponentially.

It is clear that this algorithm relies heavily on addition and comparison instructions. Fortunately, these instructions are easily parallelized. Techniques for instruction level parallelism (ILP) of addition and comparison using SIMD instructions in modern, general-purpose CPUs will be discussed in subsequent sections of this paper.

Every cell in the matrix is dependent on the values of the cells above, to the left, and to the up-and-left diagonal. This creates interesting (unfortunate?) data dependencies which limit the amount of parallelism which can be exploited in this dynamic programming approach. These dependencies are depicted via the antidiagonals shown below in Figure 4.

These dependencies provide interesting constraints on the amount of parallelism that can be exploited. Most dynamic programming hardware implementations use systolic arrays to compute each antidiagonal in the matrix, using the results of previously calculated antidiagonals in a step-by-step parallel approach to fill in the matrix as rapidly as possible. It turns out that systolic arrays provide an excellent model for dealing with the fundamental recurrence relationship between adjacent antidiagonals in the dynamic programming matrix.
3 SYSTOLIC ARRAYS

In terms of hardware architecture, a systolic array is a network of directly interconnected processors which perform specific (usually simple and similar) computations and pass the results of those computations to other similar or identical processors within that network. Systolic arrays are used quite frequently in the domain of massively parallel computing, since they readily map to parallel problems while also limiting communication bottlenecks found in other parallel processor architecture [31].

Designing a Systolic Array involves designing the internal workings of each individual processing element (PE) in addition to designing the dataflow and communication link between processors in the array. Figure 5 shows a simple, unidirectional systolic array for incrementally adding +3 to each number in an input stream.
Systolic Arrays can be implemented in hardware in a variety of ways. They can be implemented directly in hardware as a linear array. The Brown Systolic Array (B-SYS) was implemented as an 8-bit, linear hardware co-processor using 6.9mm x 6.8 mm 2u-CMOS chips, each containing 47 processor units and 48 registers [31]. B-SYS is described in more detail later in this paper.

Systolic Arrays can be implemented in reconfigurable hardware such as the Splash-2 system, in which Hoang [19][20][32] implements both a bi-directional and uni-directional systolic array to perform DNA sequence comparison. Finally, some massively parallel systems can emulate meshes of different dimensions, such as the Connection Machine. Although far less preferable than hardware implementations, systolic arrays can also be implemented in software, as I have done in Appendix A.

Figure 5 – A trivial unidirectional systolic array consisting of 3 processing elements (PEs). The output of one PE is the input of another. The systolic array shown here adds +1 to the input stream at each PE.

There exist several distinct systolic array architectures, but perhaps the most notable architecture is that of the Systolic Shared Register Architecture (SSRA).
3.1 The Systolic Shared Register (SSR) Architecture

According to Hughey, SSRA machines have the following four common characteristics [31]:

- Regular Topology
- SIMD Broadcast Instructions
- Shared Register Banks
- Stream Oriented Communication

Perhaps the most critical features of this systolic architecture are the Shared Registers and Streams.

Figure 6 shows a general and simplistic view of the basic SSRA architecture.

![Figure 6 – The basic linear SSRA architecture, as presented by Hughey [31]. Notice that each pair of PEs is separated by a shared register, which is used for efficient communication between processing elements (PEs). Each operation both computes and moves data along the linear systolic array.](image)

The Brown Systolic Array Co-Processor (B-SYS) is a classic implementation of a linear SSRA architecture, and is described in more detail later in this document.

One of the critical details about a systolic array is its *systolic speed*, which is a measure of the number of steps necessary for data to travel from one PE to another. When designing a systolic data stream, it might be important to introduce delays. This is especially true when you have bi-directional systolic arrays which move data via two or more streams both “left” and “right” (westward and eastward). It may be the case that data moving left needs to flow through the systolic array at a rate which is either faster or slower than the data stream moving to the right, therefore you could describe the relative systolic speeds of two data streams as a ratio. (e.g “3:1 or 5:2”).
Some high-level design tools and low-level programming languages exist to assist in the design of systems based on Systolic Arrays. The New Systolic Language (NSL) is an excellent example of such a system [31].

3.2 New Systolic Language (NSL)

NSL attempts to abstract the lower-level hardware details from the systolic array developer, simplifying the process of designing and developing complex systolic processes. To do this, NSL specifically addresses the distinct concepts of individual `Processing Element` (PE) design and `Dataflow` between PEs.

NSL was originally developed specifically for the Brown Systolic Array (B-SYS) coprocessor, which implements a systolic array using a Systolic Shared Register Architecture (SSRA). The details of the B-SYS system are outlined in the section on Configurable Co-Processors. The original design goal of NSL was to act simply as a high-level interface for B-SYS such that the systolic array designer need not worry about physical connections, hazards, and the actual allocation of the registers and delay elements. Although NSL has strong ties to the B-SYS hardware, it is intended to eventually be adapted to a variety of different reconfigurable hardware systems which could in turn be used to implement systolic arrays.

Interestingly, NSL is implemented as extensions to C++ through the definition of special classes encapsulating the core concepts of systolic arrays (namely, Processing Elements and Dataflow). NSL also relies heavily on C++ operator overloading to achieve its goals. NSL defines classes of objects such as SStream (Systolic Stream), which is arguably the single most important benefit of NSL.

In the paper “Programming Systolic Arrays” [31], Hughey presents NSL source code for the implementation of a Processing Element (PE) for performing sequence comparisons, which is directly applicable to performing sequence alignment of biological sequences. Figure 7 presents Hughey’s NSL source code for sequence comparison. As you can see, the source code is both small and simple, and makes important use of the SStream object type.

One of the interesting features of the NSL program shown in Figure 7 is that it defines two eastward moving data streams, each with different speeds. The first is a stream of characters (which will be compared), and another is a set of scores (weights). The weight stream moves at a speed of “2”, since this score data is required from two steps in the past. This illustrates how NSL can easily handle advanced stream manipulation, being able to easily specify stream speeds and allow PEs to have both forward and reverse look-ahead, at least for small numbers of systems.
Figure 7 – NSL Source code [31] for performing sequence comparison using a linear, unidirectional systolic array. This algorithm outputs the edit distance between two sequences. It should be clear that the implementation of a systolic array using a high-level language such as NSL or C++ greatly simplifies the development systolic arrays in reconfigurable hardware systems.
In conclusion, the systolic array model maps very well to the recurrence equations used in
dynamic programming. Several systems for sequence comparison have been implemented using
systolic arrays, including the Brown Systolic Array (B-SYS) system as well as a system built on
top of the FPGA-based Splash-2 system. Indeed, most of the custom hardware approaches in the
literature use systolic arrays. These systems are described later in this paper.

4 RECONFIGURABLE HARDWARE METHODS

The first class of specialized hardware architectures reviewed is those which use reconfigurable
technology to dynamically reconfigure hardware to support any number of sequence alignment
algorithms directly in hardware. Most of the extant implementations in this class of specialized
hardware use Field Programmable Gate Arrays (FPGAs) to implement the classic Smith-
Waterman dynamic programming sequence alignment algorithm, although other approaches
exist and are described.

4.1 FPGA – Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a form of dynamically reconfigurable computing
platform. They offer speed and programmability, and are often much cheaper than VLSI/ASIC
solutions.

4.1.1 Splash 2

Splash 2 is a second generation implementation of a reconfigurable hardware system for
performing custom computing, and was designed and implemented at the Supercomputing
Research Center (SRC) in San Diego California in 1994-1996. It is, in essence, a multi-board
coprocessor system which is intended to be attached to a host system (specifically, via a Sun
SBus on a Sun SPARCstation 2).

This section is subdivided into three sections. First, an outline of the Splash 2 architecture is
described. Next, I describe an implementation of sequence comparison using Splash 2 in which
the edit distance between two sequences is computed. Finally, I discuss an extension to the
sequence comparison system in which a sequence alignment is computed. In either case, Splash
2 is used to implement a systolic array, as previously described.
4.1.1.1 The Splash 2 Architecture and Implementation

Splash 2 was built using Xilinx XC4010 FPGA chips, each of which contains a 20x20 array of Control Logic Blocks (CLBs). Each CLB has three lookup tables and two flip-flops, and can implement any Boolean function of nine inputs. The basic Xilinx XC4000 CLB architecture [35] is shown in Figure 8 below.

![Figure 8 – The Xilinx XC4000 CLB Schematic. Source: [35]](image)

The routing of data through the 20x20 array on each FPGA is controlled through programmable switching points. There are a few different ways in which the CLBs can be interconnected in the 20x20 array, including:

- Row and Column Oriented Routing between CLBs
- A Ring Topology (called a VersaRing) used to connect to boundary CLBs to I/O components
- A programmable global routing scheme

Each of the 400 CLBs (20x20) per FPGA can be configured and programmed to be a register, adder, comparator, or other similar small processing unit, and some operations such as addition can involve small groups of adjacent CLBs which can share a fast carry bit [19].
Splash 2 is a multi-board system, with as many as 16 boards all attached within a single external chassis and interconnected using a Futurebus+ backplane\(^1\). The Splash 2 boards on the Futurebus+ are attached to a Sun host via a cable going to an SBus interface card on the Sun host. Each Splash-2 card has 17 Xilinx XC4010 FPGA chips. This provides a total of \((17 \text{ chips} \times 16 \text{ boards} \times 400 \text{ CLBs per chip}) = 108800\) CLBs. It should be noted that some of the CLBs are reserved for overhead (communication and I/O) and will not be available for general computing purposes. Nevertheless, it should be apparent that this system provides a great deal of programmable logic in a relatively small enclosure.

As previously mentioned, each of the boards on the Splash 2 system has 17 Xilinx FPGA chips. These FPGA chips are directly connected to 512KB of local RAM, and each FPGA is linearly connected to its neighbor and a crossbar switch. Splash 2 was designed to handle computations in two distinct modes: Data broadcast (SIMD) and Linear (e.g. systolic). Since each FPGA on each board is connected to its neighbor in a linear fashion, and the input to one board is connected to the output of the previous board, every FPGA in the system can be linearly connected in this fashion. Data passes from the Splash 2 interface board to the first FPGA on the first card, and the data passes through each until exiting the last FPGA on the last board. In this way, data can be thought of as a stream, which is a useful way of looking the Systolic solution to Sequence comparison implemented by Hoang on the Splash 2 system.

Data can also be broadcast to all FPGAs on a board by using the novel Splash 2 crossbar scheme. This scheme requires that one of the FPGA chips (X0) serve a dual role. In addition to performing programmed general-purpose computing functions, it is also capable of placing data from the SIMD bus onto the Crossbar, which subsequently allows all of the FPGAs on a Splash-2 board to have simultaneous access to the SIMD data in a broadcast fashion.

Internally, the Splash 2 system has a 36-bit wide data bus. 32 bits are used for ordinary data, but the remaining four bits are used as tags or opcodes, which can have special significance only to the Splash 2 system. This is especially relevant for the first FPGA on each board (X0), which can interpret these opcodes and effect how incoming data is handled (broadcast vs. linear).

\(^1\) Futurebus+ is currently a dead standard.
Figure 9 – The entire Splash 2 layout. Note that the boards have 17 FPGA chips and a crossbar. Chip X0 is especially important, as it controls how incoming data to a board is handled (linear mode or broadcast mode via the crossbar). Missing from this diagram is the attached memory (512K Bytes for each FPGA chip). Source: [36]

4.1.1.2 Sequence Comparison

Hoang implemented a sequence comparison system using Splash-2 which calculates the edit distance between two genetic sequences [19][20][32]. The edit distance between two sequences
is the minimum number of edit operations (insert, delete, or substitute) which is required in order to transform one sequence into the other. Hoang accomplished this by implementing the dynamic programming algorithm as described in previous sections using a systolic array.

As described earlier, dynamic programming for sequence alignment involves recurrence relationships (data dependencies), which limit the amount of parallelization which can be exploited by multi-processor systems. Each antidiagonal in a 2D dynamic programming matrix is dependent on the values in the previous antidiagonal. However, each value along the same antidiagonal can be computed in parallel.

Two different systolic approaches are possible. The first involves a bidirectional systolic array, in which a stream representing one sequence is injected in one end of the array, while another stream is injected on the opposite end of the systolic array. As both sequences march in opposite directions through the array, local edit distances are stored and output. In his other approach, Hoang uses a unidirectional array in which the systolic array is pre-populated with one of the sequences, and the second sequence marches down the array being compared at every step. The advantages and disadvantages of each approach are described.

4.1.1.2.1 The Bidirectional Systolic Array Approach

In the bidirectional systolic array approach, an array of Processing Elements (PEs) are allocated. Two data streams are then passed across this array from both directions (from the West and from the East). Each of the two sequence streams represent a DNA sequence to be compared. As the streams move across the PEs in the array, each PE computes local edit distances which are then transported out of the array on two bidirectional edit streams. This is shown in Figure 10 below.

![Figure 10 – The Bidirectional Systolic Array for computing the edit distance between two sequences. Sequences enter from either direction. When symbols from one sequence meet another, a local edit distance is computed and cached.](image-url)
It should be clear from Figure 10 that since the two sequence streams are marching in opposite directions at each step, they need to be separated by an empty space in the stream for correct timing. This means that half of length of the streams is essentially wasted and then in order to compare sequences of length M and N requires $2 \times \max(M+N)$ PEs to be allocated. This spacing for the sake of timing is an unfortunate inefficiency of the bidirectional approach, and is remedied in the *Unidirectional Systolic Array* approach described below.

The pseudo code for the bidirectional approach, as presented by Hoang, is rather simple:

```plaintext
loop
  if (SCin != \emptyset) and (TCin != \emptyset) then
    PEDist ← \min \left\{ PEDist + c(SCin, TCin),
                             TDin + c(SCin, \emptyset),
                             SDin + c(\emptyset, TCin) \right\}
    SDout ← PEDist
    TDout ← PEDist
  elseif (SCin != \emptyset) then
    SDout ← SDin
    TDout ← SDin
  elseif (TCin != \emptyset) then
    SDout ← TDin
    TDout ← TDin
  else
    SDout ← PEDist
    TDout ← PEDist
  endif
  SCout ← SCin
  TCont ← TCin
endloop
```

Figure 11 – Algorithm for the implementation of a bidirectional systolic array for calculating the edit distance between two sequences. Notice the recurrence equation for calculating edit distance. Also notice that a full implementation in the C language is presented in Appendix A of this paper. Source: [20]

In order to better understand the intricacies of the bidirectional systolic model for sequence comparison, I implemented this bidirectional approach in software in the C language, and it is presented in Appendix A of this paper.

Each bidirectional Processing Element (PE) is connected to its neighbor to the left and right in the array and has an extremely simple block diagram, as shown below in Figure 12.
The bidirectional approach is capable of comparing sequences of length M and N with:

\[
\text{Num of PEs} = 2 \times \text{MAX}(M+1, N+1)
\]

processing elements. It is capable of performing this computation in a time complexity proportional to the length of the array [19][20][32]. In fact, to be more specific, using my software implementation, I found that the minimum number of steps required to compare two sequences is precisely:

\[
\text{MINSTEPS} = \text{Num of PEs} + \text{Length of Longest Sequence}
\]

Obviously, having a finite number of Processing Elements in an array could present a problem for comparing extremely long sequences. Fortunately, the problem of sequence comparison is easily conquered. For long sequence comparisons, which will not fit in a finite number of PEs, the sequences can be subdivided into smaller subsequences which will fit the constraints of the hardware. In the end, it is the responsibility of the host system to re-assemble the partial comparisons into a comprehensive whole.

The unidirectional systolic array approach, which involves only one input stream into the array, is more efficient for a number of reasons, which will be described next.
4.1.1.2.2 The Unidirectional Systolic Array Approach

In the Unidirectional Systolic Array, both of the sequences to be compared are converted into a single stream separated by a single space. A new control stream is introduced called a tag stream, which is aligned with the unified sequence stream and whose tags simply annotate each character in the stream as either a symbol from the source sequence, the target sequence, or a separator between sequences. In this way, the source sequence is loaded once into the systolic array and may be compared against multiple target sequences without having to be loaded again. This scheme is ideal for comparing the source sequence against target sequences in a large database of sequences (a typical application for sequence comparison). The maximum source sequence length is directly equal to the number of Processing Elements (PEs) in the array. There is no constraint to the length of the target sequences. Once the source sequence is preloaded into the systolic array, the array can function at near perfect utilization levels, and is considerably more efficient that its unidirectional counterpart, in both time and space complexity. The only overhead is in the transition between sequences, as denoted by the PR tag as shown in Figure 13 below.

![Figure 13 – The dataflow through a unidirectional systolic array [19][20]. Note the tag stream. Source: [19]](image)

The only tradeoff here between bidirectional and unidirectional systolic arrays seems to be that the unidirectional systolic Processing Elements (PEs) are more complex in their implementation. Each unidirectional PE still has 4 inputs and 4 outputs, but the unidirectional PEs must now not only perform the requisite dynamic programming calculations, but must also interpret and act on the flow of control data from the tag stream. Figure 14 shows the PE algorithm necessary to implement a unidirectional array for edit distance calculations.
Figure 14 – Pseudo code for the PE algorithm necessary to compute edit distance in a unidirectional systolic array [20]. Note the increase in complexity of the algorithm over the simpler (but less efficient) bidirectional approach shown in Figure 11. Source: [20]

```
loop
  if (TAGin = SR) then
    if (SRCch = ∅) then
      SRCch ← CHRin
      CHRout ← ∅
      DSTout ← PDSTin
    else
      CHRout ← CHRin
    endif
    PDSTout ← PDSTin
  endif
  if (TAGin = PR) then
    if (SRCch = ∅) then
      DSTout ← PDSTin
    endif
    PDSTout ← DSTin
    CHRout ← CHRin
  endif
  if (TAGin = TG) then
    if (SRCch ≠ ∅) and (CHRin ≠ ∅) then
      DSTout ← min (PDSTout+c(SRCch,CHRin),
                      DSTin+c(SRCch,empty),
                      DSTout+c(empty,CHRin))
    elseif (SRCch = ∅) then
      DSTout ← DSTin
    endif
    PDSTout ← DSTin
    CHRout ← CHRin
  endif
  TAGout ← TAGin
endloop
```

The unidirectional systolic array approach can compute the edit distance between two sequences of lengths M and N with a time complexity of \( O(M + N) \). The preloaded sequence (source sequence) can be at most as long as the array. Fortunately, as with the bidirectional systolic array, the problem of comparing long sequences can be computed by subdividing the problem. Also, as mentioned before, the unidirectional approach can rapidly compare multiple target sequences against the source (preloaded) sequence, which is useful for comparing sequences against a database.

Finally, Hoang’s tests of unidirectional vs. bidirectional showed that the unidirectional approach was slightly more than twice as fast as the bidirectional approach when calculating the edit distance between the same two sequences.
Calculating edit distance is only one useful measure of the similarity between sequences. Another, more valuable technique is to *align* two sequences such that the most conserved regions between the sequences line up (are aligned). Fortunately, systolic arrays can be used in a second-stage analysis to determine the optimal alignment between two sequences by using the dynamic programming methodology as described previously in Section 2. Hoang’s technique for sequence *alignment* is discussed next.

### 4.1.1.3 Sequence Alignment with Splash 2

Calculating the edit distance between two sequences is of marginal use to a biologist. Even more important than edit distance is the optimal *alignment* between two sequences. Finding a good alignment between sequences allows biologists to look for and find homologous regions which can be of critical importance for finding genes or other key aspects of DNA and/or amino acid sequences. Splash 2 has been used by Hoang to find the optimal alignment between two sequences [32]. Hoang based his sequence alignment system on his previous work in finding edit distances. His implementation is, in fact, a two-phase process in which the calculation of the edit distance (as shown in the previous subsections of this paper) is the first phase and is used to find an optimal alignment in the second phase. And since the systolic array is so massively pipelined, executing the second stage (sequence alignment) takes essentially the same amount of time as just calculating the edit distance.

**NOTE:** The implementation outlined below is based on the original Splash system (not Splash 2) and was built in 1992. However, this is unimportant, since all algorithmic concepts are still applicable to Splash 2 and this system would be (and has been) trivially ported to the more modern Splash 2 system.

The basis of Hoang’s systolic array for sequence alignment is that the edit distances which are computed on each Processing Element (PE) in the systolic array (as described in previous sections) are saved either to local Splash RAM or sent back to the host machine. The amount of storage required is proportional to M x N, where M and N are the lengths of the sequences being compared. Saving the edit distances is accomplished by adding data streams to every PE which are perpendicular to the systolic streams. These new, perpendicular edit streams feed the intermediate table (in this case, in onboard Splash RAM). This intermediate table in RAM is essentially treated as a stack, since it is LIFO. Once the edit distance is computed, the flow of data to this table is reversed as described below.

Saving the edit data is completed once the final edit distance is computed. The data in the intermediate table is then fed into a new systolic array which has some interesting properties. A diagram for this new (phase 2) systolic array is shown in figure 15 below. This systolic array implements the backtracking methodology used in dynamic programming as described in Section 2 of this paper. Notice that the matrix of intermediate edit distances flow down from above in a reversed order such that we start at the cell in the lower right corner of the matrix. By definition of global alignment, this is the end of the path which leads to the optimal global
alignment. At each step, the source and target streams are fed into the systolic array from either side of the array. Notice that this is a bidirectional systolic array, and as such the sequence symbols in the sequence streams must be separated by one null space for timing purposes. The marker stream and marker flag are used to follow the sequences of inserts, deletes, and substitutions. The marker flag starts at a position on the systolic array corresponding to where the lower-right corner of the intermediate matrix intersects the array. When the marker flag shifts to the right, this signifies a deletion. A shift to the left signifies an insertion, and no change indicates a substitution (match or mismatch). These shifts affect the marker bit stream, and represent an encoding of inserts, deletes, and substitutions which will be later interpreted outside the systolic array. It is this marker bit stream that encodes the optimal reverse path through the 2D dynamic programming matrix and therefore represents the optimal global alignment (in reverse) as well. The path is encoded in a relative fashion on the marker stream, and this requires that the marker stream have a *systolic speed* of 2. In other words, it shifts two PEs every step.

![Diagram](image)

**Figure 15 – A bidirectional systolic array for performing global sequence alignment [32].** Note the timing of the incoming data from the intermediate distance table, the marker stream, and the market flag. Essentially this systolic array implements the dynamic programming *backtracking* technique described in Section 2. Source: [32]

Hoang implemented a finite state machine (FSM) within Splash in order to decode the exiting marker bit stream and produce a meaningful alignment. This FSM is shown in Figure 16 below. The key to decoding the exiting bit stream is to realize that the marker bit can shift at most one position (left or right), and thus it is the length of the gap between binary 1’s which represents an insertion, a deletion, or a substitution.
In summary, sequence alignment via dynamic programming can be implemented in two phases via a systolic algorithm. This approach requires an intermediate memory area (such as RAM) which is of size $M \times N$, and where $M$ and $N$ are the lengths of the sequences being compared.

### 4.1.2 JBits

JBits is a programming API, developed by Xilinx, a maker of FPGA chips.\footnote{JBits was announced by Xilinx in 1998, but Xilinx appears to have ceased support of JBits in favor of other toolkits.} The JBits API is essentially a Java tool which allows designers and engineers to write java applets or applications that send information directly to an FPGA. JBits can handle FPGA bit streams and allows for extremely rapid runtime reconfiguration of internal FPGA logic. JBits is intended to work primarily with versions of the Xilinx FPGAs that are based on the Virtex architecture, and it allow for easy FPGA logic reconfiguration at run-time, while still maintaining correct timing information necessary for smooth operation [16].

Steven A. Guccione and Eric Keller from Xilinx Inc., implemented the Smith-Waterman dynamic programming algorithm on a single Xilinx Virtex FPGA [16]. Their implementation was based on unidirectional systolic arrays, as described in previous sections of this paper. However, this new implementation used JBits in order to perform dynamic runtime reconfiguration of the FPGA.
This dynamic reconfiguration was done in basically two ways:

- Folding the scoring matrix values directly into hardware at runtime. This allowed these scoring quantities to be treated as hardwired constants in every Processing Element (PE) in the systolic array. Since there was no memory reference or even register lookup necessary, this helped increase the speed of calculating distances.
- One of the sequences to be compared was dynamically reconfigured into the PE hardware at runtime, essentially treating every character in the sequence as a constant, embedded in the circuit itself at each PE. However, instead of embedding the characters of the sequences, a 2-bit binary representation was used instead for efficiency.

This dynamic reconfiguration optimization rapidly reconfigures a new sequence-specific circuit for every new sequence comparison! For database searches where one compares one sequence against a large database of sequences, this approach is particularly fast, since the circuit need not be reconfigured between searches.

Another optimization used by Guccione and Keller involved the use of the Virtex carry-chain which provides a specific, fast route between adjacent CLBs for the sake of fast (and common) arithmetic carry operations. Using the carry-chain allowed the FPGA to avoid using slower, more general data path routes between CLBs. In this particular application, this was possible because of the way in which distances were encoded.

Lipton and Lopresti [37] found that in the dynamic programming problem, the distances at each step can be encoded modulo 4, requiring only 1 bit to represent the relative distance change at each step in the systolic array. An up-down counter at the end of the distance bit stream on the systolic array translates the relative distances into a final edit distance. It is this fact that allows the faster Virtex carry bit to be used to further improve performance.

Guccione and Keller provided some interesting performance results, showing that the JBits and Virtex FPGA combination exceeded the performance of the Splash 2 implementation by nearly two orders of magnitude. In addition, their implementation required only a single FPGA chip, while Splash 2 used 272 FPGA chips.

### 4.1.3 TimeLogic DeCypher II

The DeCypher II is a commercial product from a company called TimeLogic (http://www.timelogic.com). According to TimeLogic, the DeCypher II is a “bio-accelerator”, and is essentially an FPGA-based co-processor board for PCI-capable Sun servers. The DeCypher II can be dynamically configured to support a wide variety of biologically interesting algorithms, including Smith-Waterman, sequence assembly, sequence database searching (BLAST) and more. The DeCypher II claims that the inner loops of many of their core algorithms have been tuned to execute within one clock cycle in each Processing Element (PE).
Apparentley, the technical details of the DeCypher II product implementation are proprietary, and very few technical implementation details are publicly available.

### 4.2 MGAP – Micro-Grain Array Processor

Manjit Borah et al. from Pennsylvania State University have designed a system for biological sequence comparison by exploiting the pipelined SIMD characteristics found in the massively parallel Micro-Grain Array Processor (MGAP) hardware architecture [10]. Their approach is capable of achieving linear time complexity, given enough processors.

The MGAP architecture is a two-dimensional SIMD array of 16K individual micro-grained processors spread across 32 chips. In theory, multiple groups of chips can be used together on a single co-processor board to achieve hundreds of thousands or even millions of processors. Each processor is very simple and performs only simple instructions such as addition and comparison. In addition, each processor has a very small amount of local memory. All of the processors in the architecture are interconnected using a nearest-neighbor mesh. The MGAP as presented by Borah et al. has a clockspeed of 25MHz and is capable of 0.8 teraops (almost a trillion operations per second).

Processors using the MGAP architecture have been used successfully in the domains of digital signal processing (DSP), linear algebra (matrix transformations), image processing, biological sequence comparison, and more. The architecture lends itself to classes of problems which can be best addressed with the use of pipelined SIMD operations and/or simple parallel computation across matrices [10].

The small local memory that each processor can address is bit-addressable (hence the small or micro-grained nature of the architecture), and a processor is capable of operating only on a single bit or small collection of bits (i.e. 32 bits) called digits. These individual processors are known as digit-processors. Longer operands are handled by applying more processors (in parallel), and grouping them into what is known as word processors. The architecture for the fundamental digit processor is shown in Figure 17 and the four possible word processor configurations are shown in Figure 18.

---

3 Indeed, linear-time complexity is the claim from many hardware solutions to the alignment problem. This is only true for small sequences, where parallelism essentially divides the time complexity by the magnitude of the parallelization (i.e. by the number of concurrent Processing Elements).
Figure 17 – A schematic of the atomic MGAP digit processor. Image reproduced from [10]

In this implementation of sequence alignment, the authors exploited the fact that the MGAP is two-dimensional in order to map this directly to the two-dimensional matrix which is used in computing sequence alignments and edit distances using dynamic programming, as described above in Section 2 (Dynamic Programming).

MGAP is able to compare sequences of lengths M and N using an array of (M X N) processors. Sequence comparison for longer sequences can be divided into sub-sequences, requiring multiple iterations of the sequence comparison process to compare two long sequences.
Figure 18 – The four types of word processor configurations. Each configuration is made up of two or more digit processors. The snake (a), vertical (b), horizontal (c), and local (d). Figure is reproduced from [10].

Since we cannot avoid the recurrence equation (i.e., data dependencies) used in Dynamic Programming, the data flow into the 2D matrix will have to occur in a specific order so that each antidiagonal can be correctly computed.

First, the 2D matrix is initialized by storing the symbols from a source sequence in every row of the matching column, as shown in Figure 18. In addition, the small dynamic programming scoring table (see Figure 2) must be distributed to each word processor in the array. Distributing the source sequence is an unavoidable setup time for the algorithm, and will take a constant amount of time for every new source sequence to be compared. Larger substitution matrices have to be distributed when performing protein alignments, incurring a larger initialization penalty.

Next, the second of the two sequences, the target sequence, is marched down through the matrix in a pipelined fashion such that the timing delays occur and the recurrence relationship between antidiagonals is preserved. This is also shown in Figure 19.

Finally, as each processor receives a character from the second sequence, it computes scores based on our scoring table (see Figure 3), and results from the application of this scoring flows from the processor $P_{i,j}$ to processors $P_{i,j+1}$ (down), $P_{i+1,j}$ (right), and $P_{i+1,j+1}$ (diagonal). This data flow is shown in Figure 20.
Multiple sequences can be compared to the first by pipelining them and leaving only a one step delay between two different sequences. In this way, this approach can be extended to perform rapid database searches in $O(M + N + K)$ time, where $M$ and $N$ are the lengths of your sequences, and $K$ is the number of sequences to compare to the first (pre-loaded) sequence.

**Figure 19** – The $M \times N$ matrix of processors is pre-populated with a sequence $A$ such that every symbol in $A$ maps to one column and all rows in the matrix. In this case, the sequence is $A = \{a_1, a_2, ... a_m\}$. Next, another sequence ($B = \{b_1, b_2, ... b_n\}$) is pipelined and marched through the matrix of processors for computation. This pipelining scheme preserves the data dependencies introduced in the dynamic programming recurrence equation, and lends itself to the pipelined comparison of multiple sequences in $O(M+N+K)$ time complexity. Figure reproduced from [10].
Borah et al. implemented dynamic programming on the MGAP by allocating word processors in a local configuration (as shown in Figure 18 (d)). Each atomic digit processor had 32 bits of local memory, and in a 2 X 2 local configuration, had 128 bits of local memory available to the word processor. However, in practice, Borah’s implementation required only 19 of the 32 bits on each digit processor. Two bits were used to store a symbol from the first sequence (Sequence A). Five bits were used to locally store the scoring table, and additional bits were used to hold the results of the recurrence equation (the results of the additions to be used in the comparisons).

MGAP performance was shown to be very promising, and outperformed the Splash (version 1) systolic array implementation for sequences of 100 characters by a factor of 5X. Note: it would be interesting to compare MGAP to Splash for longer sequences, where partial comparisons must be aggregated by the host system.
5 VLSI HARDWARE ARCHITECTURES AND IMPLEMENTATIONS

Application-specific processors (ASICs), typically made with VLSI manufacturing technology, represent another key way in which custom sequence alignment hardware can be applied. Although VLSI processors are typically expensive and complex to manufacture, they are often much faster than alternative hardware approaches.

5.1 SAMBA – Systolic Accelerators for Molecular Biological Applications

SAMBA (Systolic Accelerators for Molecular Biological Applications) [24] is a system which was developed with Smith-Waterman style sequence comparison in mind, allowing for a speedup of several orders of magnitude beyond conventional general-purpose CPUs for the alignment of pairs of sequences. The Smith-Waterman algorithm is parameterized in SAMBA, meaning that the scoring model and other parameters can be passed to the system dynamically, thus affecting the behavior of the system and offering some degree of flexibility above and beyond a strictly hardwired implementation.

SAMBA was designed with 128 12-bit custom VLSI-based processors in addition to an FPGA-based interface module (programmable driver). Although using FPGAs for interfacing to the host machine, the core functionality of SAMBA is in its array of VLSI processors, and thus I classify SAMBA as a VLSI-based system. One of the key differences between BISP (another VLSI-implemented sequence alignment project) and SAMBA is the structure of the controller. Where BISP uses a general-purpose programmable CPU (the Motorola 68020), SAMBA uses FPGAs. FPGAs in the controller are used to help insure easy programmability as well as fast throughput. In BISP, the driver can and does become the bottleneck.

SAMBA implements a parameterized version of the Smith-Waterman dynamic programming algorithm for sequence alignment. There are five parameters in total which affect the behavior of the Smith-Waterman execution. By tweaking these parameters, the algorithm can be used to globally align two sequences (Needleman-Wunsch [5]), locally align two sequences (conventional Smith-Waterman [4]), or perform database-style searches of sequences within a genomic database (similar to BLAST).

SAMBA implements a unidirectional linear systolic array. In SAMBA, the entire array is pre-populated with one sequence (or a fragment of one sequence, if it won’t fit). Since the array is comprised of 128 processors, it takes a 128 cycles to pre-populate the array at a rate of one character per cycle. Substitution matrices and other cost tables must be distributed across the array as well, which takes additional setup time. As with other unidirectional systolic array approaches, sequences are then streamed across the array from left to right. The sequences are then compared one character at a time, and the resultant alignment is output one character at a
time from the right end of the array, which is then sent via the FPGA controller back to the host
computer for additional processing.

In the original SAMBA implementation, a 40 MHz DECstation 5000 was used as the host
computer, with a TURBOchannel bus. The FPGA array interface adapter component uses a
PAM (Programmable Active Memory) implementation in order to provide a dynamically
configurable hardware interface. The SAMBA interface board is made from a PeRLe-1 FPGA-
based prototype board. This board uses 16 Xilinx XC3090 FPGA chips, with some small
amounts of local memory used for I/O buffering and other uses.

The systolic array itself is composed of 32 VLSI chips on two circuit boards. Since each VLSI
chip has four processors on it, there is a total of 128 processors in the systolic array. Each
processor has a small amount of local memory (32 12-bit words), used for storing substitution
matrices.

SAMBA was shown to have a performance of about 1200 MCOPS (million cell operations per
second) [24], which is relatively slow compared to other hardware-based systems. However,
considering that SAMBA predates many other modern hardware-based approaches to the
systolic alignment of sequences, the results are reasonably impressive. I suspect that the speed
of the system was constrained more by the host machine than by the dedicated hardware
components.

5.2 BioSCAN – Biological Sequence Comparative Analysis Node

The BioSCAN (Biological Sequence Comparative Analysis Node) [7][8] system attempts to
address the challenge of database lookups of biological sequences, which is one major class of
problems directly related to the sequence alignment problem.

BioSCAN also uses systolic arrays, but it differs from other hardware solutions that use systolic
arrays under the guiding design belief that implementing dynamic programming in systolic
arrays produce overly complex processing elements (PEs). When complex logic is implemented
in PEs, the PEs take up more space and thus total processor density is reduced. Instead of
implementing dynamic programming algorithms, BioSCAN implements a linear similarity
algorithm. In this linear comparison approach, segments (subsequences) of two full sequences
are compared to one another in a linear fashion, where the overall linear similarity score of two
sequences can be calculated as follows:

$$S_{x,y}^L = \sum_{k=0}^{L-1} T(A_{x-k}, B_{y-k})$$

[7]

Where $A$ and $B$ are sequences being compared and $T$ is any substitution cost lookup table.
This approach can be easily parallelized using a divide-and-conquer approach where the host system can re-assemble high-scoring partial segments into a full alignment, once they’ve been computed by the co-processor.

One of the primary design goals of the system was to reduce off-chip traffic as much as possible such that multiple chips could be cascaded into a systolic array of arbitrary length without incurring too much penalty from communications.

From [7]:

The ASIC design is modular and consists of four major blocks: clocks, frame, memory, and PEs. The primary clock input to the ASIC is a square waveform from a single input pin. A 2-phase nonoverlapping clock is derived on-chip. The true and complement phases of the two clocks are globally routed using an electronically balanced tree and interlocking circuitry. The final driving state of the four clocking signals are placed in the corners of the die to minimize crosstalk interference with other circuits and power supply lines.

The basic architecture and data path of the BioSCAN chip is shown in Figure 21 below.

![Figure 21– The BioSCAN chip architecture and datapath. Source: [7]](image-url)
Each ASIC chip contains an onboard area of static RAM under the design assumption that the linear sequence comparison will be dominated by memory accesses. The onboard SRAM is fast, but still takes 16 clock cycles to stream 28 16-bit values [7].

Figure 22 shows the internal architecture for each Processing Element (PE). Notice its extreme simplicity, which was a primary design goal in order to achieve an overall higher PE-per-ASIC ratio.

As with other systolic array approaches, BioSCAN must initialize its PEs before alignment can proceed. This initialization step involves the following steps:

- The substitution matrix is distributed to the local SRAM of each PE (for protein alignment, this involves 20 characters).
- The query sequence is distributed across the PEs (one character per PE)

Initialization has a linear time complexity, and increases linearly with the number of PEs and the complexity of the substitution matrix.

![Diagram](image_url)

**Figure 22 – The internal architecture for a BioSCAN Processing Element (PE).** Notice its extreme simplicity. Also note that the characters of the query sequence are stored in an interesting 2-of-8 encoding rather than binary. This was done, since 2-of-8 decoding logic in each PE was much simpler and thus took less space to implement (thereby increasing PE density). Source: [7]
The BioSCAN ASICs were manufactured by Hewlett-Packard, and have the following characteristics:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Die Size</td>
<td>7750 µm x 9050 µm</td>
</tr>
<tr>
<td>Transistors</td>
<td>537,675 (89,736 in RAM)</td>
</tr>
<tr>
<td>Processors</td>
<td>812 (536 transistors per PE)</td>
</tr>
<tr>
<td>Pins/Package</td>
<td>84 pin PGA (42 Vdd/GND)</td>
</tr>
<tr>
<td>Clock</td>
<td>32 MHz @ 50°C</td>
</tr>
<tr>
<td>Voltage/Power</td>
<td>4.75-5.25V (3.0W max)</td>
</tr>
<tr>
<td>Interface</td>
<td>TTL-compatible</td>
</tr>
<tr>
<td>Process</td>
<td>1.2 µm N-well CMOS</td>
</tr>
</tbody>
</table>

Figure 23– Physical manufacturing characteristics of the BioSCAN ASIC. Adapted from [7]

Like other biology coprocessors, BioSCAN involves several components: A host interface, an array interface, and an array of BioSCAN ASICS. The host interface is intended to connect to a Sun Microsystems VME-bus (actually an IEEE standard) and act as an adapter from that host bus to the array interface by simplifying the connection to the host bus. The array interface, which sits between the systolic array (theoretically an arbitrary number of ASICS) and the host interface, is responsible for the management and control of the systolic array and acts as both an array driver and a liaison between the ASIC array and the host interface. Although the ASIC size is logically limitless (in theory), in practice, the physically manufactured board consists of 16 chips.

In my opinion, the BioSCAN ASIC array was well-designed. On the manufactured board, the ASICs were configured into a ring topology which allows a small amount of fault-tolerance: if one ASIC malfunctions, the ASIC ring is reconfigured into a linear array. Two ASIC failures would be catastrophic. Additionally, if necessary, the array can be split and sub-divided into two or more sub-arrays which can perform sequence alignment in parallel. This would be potentially useful if one was aligning particularly small sequences.

Performance comparisons between BioSCAN and other systems is somewhat difficult to do, primarily since BioSCAN does not use Smith-Waterman or Needleman-Wunsch dynamic programming in order to align sequences. Instead, it uses a different approach altogether. Where dynamic programming is guaranteed to provide an optimal alignment relative to a scoring system, the algorithm executed by BioSCAN has no similar guarantees. Indeed, the algorithm is effective and fast (it is the same basic approach used with BLAST), but it is not guaranteed to be optimal. Fortunately, for the biologist, very good yet sub-optimal alignments are usually perfectly acceptable and optimality often offers little to no additional biological meaning (in fact optimal alignments are sometimes not the most biologically meaningful alignment).

To benchmark the system performance of BioSCAN, the BioSCAN designers measured the MCOPS (Millions of Cell Operations per Second) and compared this to the MCOPS of other
systems. Not surprisingly, BioSCAN performance exceeded that of all other measured systems at the time, and was 25 times faster than SPLASH and more than twice as fast as BISP (both systems are described elsewhere in this paper).

The designers of BioSCAN at the University of North Carolina at Chapel Hill have placed a version of BioSCAN online for public access. Since BioSCAN solves the problem of rapid database searching, users can search the entire public DNA/RNA databases or protein databases in a matter of minutes.

5.3 Patzer’s “Lab-on-a-Chip”

Aaron Patzer, from Duke University wrote a recent paper titled *Highly Parallel DNA Sequence Matching and Alignment Processor* [42]. In this paper, he outlines a simple (yet unnamed) VLSI architecture for performing *sequence re-assembly*. Sequence re-assembly is a critical component of the DNA sequencing process. DNA sequencing involves taking many thousands or millions of small pieces (almost always less than 500 base pairs in length), and attempting to find how these sequences best overlap in order to form a coherent, single sequence. This problem is very computationally intensive, as the number of subsequences can be quite large. A complete reassembly process has, at best, a time complexity of $O(N^2 \log_2 N)$ [42].

Patzer’s algorithm is trivially simple, and is simply this:

*Slide one sequence past another looking for the best overlap region.*

A block diagram is shown in Figure 24 below. Shift registers are used to shift DNA sequences past one another, and a prefetch shift buffer is used to avoid fetching data from RAM at each cycle and this avoiding stalls due to memory accesses. A Comparison Block logic unit, also shown in Figure 24, contains comparator logic which implements both XNOR and XOR comparators. XNOR implements strict matching, while the XOR function allows for looser matching based on complementary base pairs (A=T, G=C) [42]. This simple kind of comparator is possible, because for DNA, all four base pairs can be encoded in 2 bits. In this case, a $A^\prime$ = “00”, $T^\prime$=”11”, $C^\prime$=”01”, and $G^\prime$=”10”, so the XNOR and XOR functions apply directly to this bit representation and maps nicely to biological meaning of strict and complementary matching.

Patzer argues that strict matching is not always superior to looser matching, since deletions, insertions, and substitutions are so common in real biological data.
Patzer claims theoretical performance on the order of billions of comparisons per second, mainly due to pipelining and the fact that M simultaneous comparisons can be made with a 2M-bit register. Also, his architecture is inherently pipelined and, because of the prefetch shift buffer, is not susceptible to stalls due to frequent memory access. Finally, Patzer’s VLSI architecture allows for multiple alignments to be compared simultaneously and for the sequences to be shifted at each step by the same number of alignments. This is accomplished via diagonal cross comparisons as shown in Figure 25 below.
Figure 25 – The diagonal cross comparisons which happen in parallel. This allows the sequences to be shifted multiple base pairs at each step of the alignment, allowing for M/2 x P base comparisons to be calculated each clock cycle where M is the number of bits in the shift registers, and P is the number of diagonal cross comparisons possible at each step. Source: [42]

Although Patzer had not fabricated his VLSI design at the time of his paper’s publication, he did fully test the design using accepted design, test, and simulation software systems. His original, conservative design allowed for 8 DNA base pairs x 2 diagonal comparisons to execute every clock cycle. On a 500MHz processor, this equates to:

\[ 8 \text{bp} \times 2 \text{ alignments} \times 500\text{MHz} = 8 \times 10^9 \text{ comparisons per second}. \]

This performance is clearly at least a few orders of magnitude faster than anything possible with general-purpose CPUs.

6 PROGRAMMABLE CO-PROCESSOR

The programmable co-processors described below are programmable, linear, systolic arrays implemented in VLSI which are attached to a host via a bus-attached co-processor board. They are programmable in the sense that the functionality of the individual processing elements (PEs) are configurable via software (often in a limited way), thus providing some flexibility over completely hardwired systems. The systems described below can employ a variety of systolic algorithms, but were originally designed with sequence alignment in mind.

6.1 Kestrel

The Kestrel system is a hardware-based programmable linear systolic array system intended for use with sequence alignment [18]. It is essentially a programmable co-processor, and was
originally based on the B-SYS [38] programmable processor (described in this paper). Kestrel was developed at the University of California at Santa Cruz, and is named after the small California falcon by the same name. Although Kestrel is intended to be specialized to sequence alignment, one of the central design goals of the Kestrel system was to make the system programmable. Programmability (and thus flexibility) was considered critical, since many different approaches exist and algorithms to sequence alignment, many of which can be customized and implemented on Kestrel.

As of 1998, Kestrel had 64 processing elements (PEs) per chip, with 8-16 chips per board, for a total maximum of 1024 PE’s per Kestrel board. Interestingly, since dealing with large quantities of biological data invariably involves accessing disk I/O subsystems, the I/O rates for Kestrel are optimized with this in mind such that Kestrel is never be a bottleneck in an alignment of large sequences or large numbers of sequences, which must be streamed from disk subsystems.

On the VLSI scale, Kestrel implements a linear array of 8-bit SIMD Processing elements. Although an 8-bit data path may seem prohibitively small, the Kestrel designers argue that such a data path is sufficient for most sequence alignment problems, and that the smaller datapath drastically simplifies the VLSI design, implementation (e.g. reduced chip size), especially considering that each PE has its own local RAM (SRAM). Figure 26 below shows the architecture of a single Kestrel PE:

![Figure 26 – The internal architecture of a single Kestrel Processing Element (PE). Note the local 256x8 bit SRAM, the ALU and the left and right shared systolic registers. Reproduced from [18].](image-url)
One of the key features of the Kestrel system is the use of SSR (Systolic Shared Arrays). These SSRs allow for highly efficient inter-PE and inter-chip communications. Since register files are shared and situated between PEs (both on chip and across chip boundaries), there is no need for store and fetch operations, thus eliminating all overhead for communications between PEs. Essentially, communication and computation occur at the same time.

Each PE has a 256-byte pool of static RAM (SRAM) memory. A single value can either be read or written to an SRAM every clock cycle (clock typically runs at 33MHz). Much of the storage capacity of an SRAM is wasted in basic sequence comparisons. However, the largish SRAMs can be useful for more complex sequence alignment algorithms, for example for storing substitution matrices (especially useful for the alignment of amino acid sequences).

The ALUs within each PE in Kestrel are 8-bit units which are capable of performing up to \(2^8 = 256\) distinct arithmetic operations and logic functions. The Kestrel PE ALU is *programmable* in the sense that the behavior of the ALU is determined by two 4-bit fields known as *propagate* and *generate* fields which fully describe the ALU function in terms of carry-outs and results.

One of the Kestrel design goals was to find the middle-ground between performance, size, and cost. This was achieved in the ALU component by making it an 8-bit unit, which conserved space (ultimately increasing PE density) while maintaining relatively high performance.

Other key components of the Kestrel PE are that of the *comparator* and the *bit-shifter*. The comparator plays an obvious and significant role in the selection of minimum or maximum values, which is a critical aspect of the dynamic programming recurrence formula, as described fully in other portions of this paper. As with other aspects of the Kestrel architecture, the comparator handles multi-precision operations.

In addition to the PEs, Kestrel is made up of two additional distinct architectural components:

- A PE Array Controller
- Host bus interface

The array controller controls instruction issue to the array, looping, and communications to the host bus interface.

The host bus interface is a hardware-specific interface between the Kestrel board and the host bus. The current Kestrel implementation is intended to function in a 33MHz or 66MHz PCI bus. Although the PEs are 8-bit entities, from the host’s perspective the Kestrel board itself uses 52-bit long, fixed-length instructions which specify operators, operands, and special commands to the co-processor board. The host-bus adapter and array controller are responsible for parsing, translating, and executing the 52-bit Kestrel instructions from the host. The format for these instructions is shown below in Figure 27.
Figure 27 – The Kestrel microcode and instruction format. Adapted from [11][12].

Performance measurements from a simulated 33MHz Kestrel implementation were very promising. Kestrel outperformed all existing FPGA-based systolic array co-processors (available at that time...such as Splash) by a factor of 15, and outperformed a 16K PE MasPar MP-2 by a factor of 3.

6.2 B-SYS (Brown Systolic Array)

The Brown Systolic Array (B-SYS) [38] was one of the first viable programmable VLSI-based systolic arrays to use Shared Systolic Registers (SSR). Developed at Brown University in the early 1990’s, B-SYS implemented SSR architecture which connected 470 processing elements (PE) across 10 VLSI chips on a single co-processor board with a theoretical sustained performance of 5 GOPS. Although able to solve many problems which map to the systolic array paradigm, B-SYS is best suited to problems involving simple integer operations. Because of these reasons, B-SYS is an excellent platform for performing dynamic programming used in multiple sequence alignment (MSA).

As mentioned before, B-SYS employs a Systolic Shared Register (SSR) architecture. In addition, B-SYS supports the concept of broadcast instructions. Broadcast instructions are instructions which are broadcast and executed (identically) across most/all of the PEs within the array. This was done to simplify design, with the understanding that in most applications, each PE in a linear array performs the same fundamental operations as its adjacent neighbors.

In the B-SYS SSR design, each processing unit in the linear array has access to the registers of both of its neighbors (to the East and West) and each PE maintains no additional local memory registers (more or less). While SSR does not mandate a linear architecture (other topologies are possible), B-SYS itself is strictly limited to linear arrays.

As with Kestrel (as mentioned previously, Kestrel was based on B-SYS), each PE in B-SYS is an 8-bit unit.
In the early B-SYS tests, the system showed very promising results. On one-to-many style sequence alignment (looking for a sequence within a large database of sequences), B-SYS outperformed its host computer by nearly two orders of magnitude. B-SYS outperformed existing Splash implementations, and was comparable to the P-NAC system. In addition, at the time B-SYS was slower, but comparable to the Cray-2 supercomputer.

As mentioned above, Kestrel’s design was based on B-SYS.

7 SEQUENCE ALIGNMENT ON MASSIVELY PARALLEL SUPERCOMPUTERS

Several systems have been implemented on special, massively parallel super computers. Although not strictly custom hardware systems, they do take full advantage of the underlying computer architecture of the massively parallel systems.

One such system, *BLAZE*, is briefly described below.
7.1 BLAZE on the MasPAR

A group of researchers from the Biochemistry department at Stanford University implemented a system for fast sequence alignment using a MasPar MP1104 massively parallel computer [9]. The Stanford researchers implemented the Smith-Waterman dynamic programming algorithm for sequence database searching. Noting that Smith-Waterman is typically too time-complex to effectively use for database searching, BLAZE was able to perform complete database searches of the Swiss-Prot protein database in less than 15 seconds.

As described earlier, Smith-Waterman computes optimal local sequence alignments, and is therefore more sensitive than any other approach. Other sequence database searching systems employ heuristic approaches to work around the time complexity problems inherent in the dynamic-programming approach. In effort to produce both a fast and sensitive search, the Stanford researches opted to stick with Smith-Waterman, but on exceedingly fast hardware.

The MasPar MP1104 has 4096 4-bit processors, and the system had 256MB of RAM (total). At the time (1993), 256MB was enough to hold the GenBank and/or SwissProt databases in memory at the same time. This helped with performance results significantly, since disk I/O was not involved.

Unfortunately, the Stanford BLAZE team did not fully describe the implementation details of their system. No further details are currently available.

8 EXPLOITING PARALLELISM IN MODERN GENERAL-PURPOSE CPUs

General-purpose, commodity CPUs currently have SIMD (Single Instruction Multiple Data) functional units and corresponding SIMD instructions. Intel announced their MMX-enabled Pentium brand processors in 1997. Other commodity processors such as the PowerPC and Sun SPARC now have vector processing capabilities.

In addition to using SIMD technology, other software techniques exist to exploit the low-level hardware architecture when performing sequence alignment. These are described as well.

---

4 This informal benchmark refers to the Swiss-Prot database as of 1993. Since the database growth has been significantly outstripping the speed increasing computational capacity of modern computers, I suspect that this claim is no longer true. Moore’s law is not keeping up with bioinformatics. ☺
8.1 Basic Code Optimizations

Alpern et al. presented an interesting set of ways to speed up the task of multiple sequence alignment, based on an understanding of modern CPU architectures [30]. Some of these optimizations are simple code optimizations which can increase performance incrementally over naïve implementations.

The first optimization involves *blocking* with respect to caches. Alpern et al. recommend that computation should be blocked in favor of invoking cache misses while executing Smith-Waterman style dynamic programming. Because of the inherent nature of the recurrence equation used in dynamic programming, the cache needs for a large alignment are also relatively large (especially for large matrices). This is even more important for multi-dimensional dynamic programming, where the space complexity grows exponentially. The suggested blocking solution involves re-ordering the nested looping structure for the dynamic matrix to maximize spatial locality while also performing the matrix fill operation in *swathes*, whose length is determined by the CPU architecture (e.g. things like cache size are important when determining an optimal swath size).

Other simple code optimizations are presented, including improvements to the inner loop structure for the naïve implementation. For example, instead of using loops which use double subscripts for 2-dimensional matrix operations, it might be possible to instead use singly-subscripted arrays and get around the required use of an indirect reference and limit the amount of register copying involved.

The resulting code from these optimizations is a little more complex (and harder to understand!) but Alpern reported that these simple hardware-aware programming changes produced a noticeable improvement in performance as shown in the graph shown below in Figure 29.

<table>
<thead>
<tr>
<th>Query Sequence Length</th>
<th>500 residues</th>
<th>2000 residues</th>
<th>5000 residues</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Original Code (nanoseconds)</em></td>
<td>954ns</td>
<td>988ns</td>
<td>1077ns</td>
</tr>
<tr>
<td><em>Optimized Code</em></td>
<td>886ns</td>
<td>862ns</td>
<td>859ns</td>
</tr>
<tr>
<td><em>Percent Speedup</em></td>
<td>7.6%</td>
<td>15%</td>
<td>25%</td>
</tr>
</tbody>
</table>

Figure 29 – Speedup after loop and cache optimizations on an Intel i860 CPU. Table adapted from [30].

As Figure 29 (above) clearly shows, simply modifying the code with respect to the hardware architecture and organization helps significantly. However, some of the most significant optimizations occur with parallelization, as will be discussed next.
8.2 Vector Processors (SIMD – MMX, AltiVec, etc.)

Modern CPUs such as the Intel Pentium (MMX) and the Motorola/IBM PowerPC (AltiVec) currently have integrated vector sub-processors [3][26][39][40]. These are effectively SIMD (Single Instruction Multiple Data) subcomponents with their own functional units and specialized instruction set. Originally intended for signal processing and multimedia applications, these vector sub-processors can be exploited to perform parallelized biological sequence analysis as well.

The following table (Figure 30) describes some of the vendors and products offering vector-processing (SIMD) on-chip.

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Microprocessor</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AMD</strong></td>
<td>K6/K6-2/K6-III</td>
<td>MMX/3DNow!</td>
</tr>
<tr>
<td></td>
<td>Athlon</td>
<td>Extended MMX</td>
</tr>
<tr>
<td><strong>Chromatics</strong></td>
<td>MPact</td>
<td></td>
</tr>
<tr>
<td><strong>HP (Compaq/DEC)</strong></td>
<td>Alpha</td>
<td>MVI (Motion Video Instruction)</td>
</tr>
<tr>
<td><strong>HP</strong></td>
<td>PA-RISC</td>
<td>MAX(-2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multimedia Acceleration eXentions</td>
</tr>
<tr>
<td><strong>HP/Intel</strong></td>
<td>Itanium (Merced)</td>
<td>SSE</td>
</tr>
<tr>
<td><strong>Intel</strong></td>
<td>Pentium MMX/II</td>
<td>MMX (Multimedia eXtension)</td>
</tr>
<tr>
<td></td>
<td>Pentium III</td>
<td>SSE (Streaming SIMD Ext.)</td>
</tr>
<tr>
<td><strong>MicroUnity</strong></td>
<td>MediaProcessor</td>
<td></td>
</tr>
<tr>
<td><strong>Motorola</strong></td>
<td>PowerPC G4</td>
<td>Velocity Engine (AltiVec)</td>
</tr>
<tr>
<td><strong>Philips</strong></td>
<td>TriMedia</td>
<td></td>
</tr>
<tr>
<td><strong>SGI</strong></td>
<td>MIPS</td>
<td>MDMX (MIPS Digital Media eXtensions)</td>
</tr>
<tr>
<td><strong>Sun</strong></td>
<td>SPARC</td>
<td>VIS (Visual Instruction Set)</td>
</tr>
</tbody>
</table>

Figure 30 – Current commodity microprocessors which include SIMD technology. Table adapted from [29].

The Intel Pentium with MMX technology has been exploited to perform Smith-Waterman dynamic programming and achieve a 600% overall speedup. Using C/C++ and inline assembly code, a group from Oslo was able to get a 500MHz Intel Pentium III (MMX) to perform 150,000,000 cell updates per second, and at the time this was the fastest implementation of Smith-Waterman on a general-purpose uniprocessor [29].
The Oslo group was able to achieve parallelization by exploiting the natural parallel nature of the dynamic-programming matrix. In other words, as with the systolic-array approaches, the anti-diagonals of the dynamic-programming matrix could be computed in parallel for maximum parallelization, while simultaneously respecting the intrinsic recurrence relationship between diagonals. As shown below, the Oslo group decided not to compute the anti-diagonals, but computed the verticals instead!

The group achieved 8-way parallel processing with 8-bit values using Intel MMX instructions, which were written entirely in assembly code. Interestingly, the group decided that although computing the anti-diagonals in groups of 8 values would eliminate dependencies, they decided instead to perform the 8-way parallel processing in a vertical fashion, parallel to one of the sequences in the comparison. The reasoning here was that although some data dependencies would reduce the amount of pure parallelization, the memory references would be much easier and faster to perform in this vertical nature than in an anti-diagonal fashion. The data dependencies would then be resolved internally within each vertical group of 8 values. Their novel approach is shown in Figure 31 below.

![Figure 31](image)

**Figure 31** – Instead of computing the anti-diagonals (left), they compute the verticals (right) using 8-way parallel processing. Source: [29]

To offset the dependency problem along the vertical groups, Rognes et al. used a Smith-Waterman optimization introduced by Green in 1993 known as SWAT. SWAT is a shortcut technique for essentially pruning the search space while performing the dynamic programming
matrix fill operation. Since every cell is dependent on the cell above it, SWAT allows you to quickly determine if small vertical groups of values need to be calculated in the traditional way or simply ignored. The Oslo team executed the SWAT optimization in parallel, and if all eight cells did not have to be calculated, then they would be ignored. If any one cell needed to be calculated, then the algorithm was required to perform the time-consuming fill operation for the eight values in the vertical group. However, SWAT works because it is usually the case that the cells do not need to be computed exhaustively, but can be ignored without adversely affecting the algorithm. Refer to the paper by Green here for more details on this shortcut.

The Oslo team decided to divide up the Pentium/MMX SIMD registers into 8-bit units, which is the smallest amount possible [26] and allows for maximum parallelization. Interestingly Intel MMX allows for integer operations to occur in a saturated mode, which prevents overflow and wrap-around conditions (i.e. keeps 8-bit values between 0 and 255). This was readily exploited since there are many situations in the Smith-Waterman algorithm where values which would ordinarily go negative should be set to zero. So, rather than writing the instructions to do this, MMX handles this naturally, increasing performance somewhat.

Since the most common operation in the inner loop is to compute the maximum (or minimum) of a small set of numbers, the PMAXUB instruction was used. Alternatively, this max operation can be calculated using the two MMX instructions PSUBUSB (unsigned saturated subtraction) and PADDUSB (unsigned saturated addition).

By using these MMX parallelization and optimization techniques, the Oslo group was able to report a six-fold (600%) speedup over the naïve implementation. These tests were done on a Pentium-III which has 64-bit SIMD registers. The PowerPC’s Velocity Engine (Altivec) uses 128-bit SIMD registers, and Intel is claiming to do the same. 128-bit SIMD registers would allow for twice the parallelization, on even faster modern processors.

8.3 Exploiting Z-Buffer Instructions via Microparallelism

One way to achieve parallelism on a processor is to use microparallelism. Microparallelism can be achieved through packing several meaningful numbers into one natural word (i.e. 32-bit or 64-bit word).

In Smith-Waterman or Needleman-Wunsch dynamic programming, the core of the algorithm (the recurrence equation) performs two simple instructions: ADD and MIN/MAX. In an effort to make the common case fast, parallelizing these operations as much as possible while still executing within the constraints of the dynamic programming recurrence equation will clearly be beneficial.

Some CPU architectures include parallel Z-Buffer instructions and functional units which are intended to speed-up 3D graphics applications. The Intel i860, for example, has parallel Z-
Buffer instructions which simultaneously handle up to four two-byte fields of an 8-byte doubleword [30]. For example, the FZCHKS instruction will perform 4 MIN operations on parallel, and the FIADD.DD instruction will add four 16-bit values in parallel.

<table>
<thead>
<tr>
<th>Query Sequence Length</th>
<th>500 residues</th>
<th>2000 residues</th>
<th>5000 residues</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-Way Microparallel</td>
<td>186ns</td>
<td>188ns</td>
<td>168ns</td>
</tr>
<tr>
<td>Speedup Over Optimized</td>
<td>4.76 X</td>
<td>4.59 X</td>
<td>5.11 X</td>
</tr>
<tr>
<td>Percent Speedup</td>
<td>5.13 X</td>
<td>5.26 X</td>
<td>6.41 X</td>
</tr>
</tbody>
</table>

Figure 32 – Speedup after using parallel MIN and ADD Z-Buffer instructions on the Intel i860 CPU. Note that this is a superlinear speedup! Alpern et al. suggested that the superlinear speedup was due to the removal of an inner-loop conditional as part of this parallel optimization. Table adapted from [30].

It should be noted that Alpern et al. had to manually massage the assembly code to make use of these Z-Buffer instructions, since not even the Intel compiler that they were using created assembly code with these optimized instructions. This is most likely a common problem, even with commonplace processors such as the Intel Pentium (MMX) or PowerPC (AltiVec).

8.4 64-Bit Integer Microparallelism

Even processors without Z-Buffer instructions can be parallelized.

Similar to the Z-Buffer instructions, it is also possible to pack four meaningful numbers into a single 64-bit natural word for 64-bit machines such as the DEC Alpha [30].

This is done in a very straightforward fashion if four 15-bit binary numbers are packed into a single 64-bit integer, where every 15-bit number is separated with a 0 bit. Once this is done, normal 64-bit ADD instruction will add four numbers simultaneously (with the extra 0-bit used for carries). MAX can similarly be implemented, with a little additional work. Machines with 32-bit natural words should also work, but with half the parallelism (i.e. two 15-bit numbers can be added simultaneously with the nature 32-bit ADD instruction).
9  APPENDIX A

A Software Implementation of a Bi-directional Systolic Array for Computing the Minimum Edit Distance Between DNA Sequences

Below, I implement a software-based systolic array for computing the minimum edit distance between two DNA sequences. This program is written in the C language, and is a direct software version of the Splash-2 hardware (FPGA) implementation presented by D.T. Hoang [19][20][32].

I implemented a bi-directional systolic array, which is capable of computing the minimum edit distance using roughly 2*(max(m+1, n+1)) processing elements (PE), where m and n are the lengths of the DNA sequences to be compared. This algorithm computes the solution in time complexity proportional to the length of the array, which is approximately:

$$\text{array\_length} + (\max(2*\text{stream\_length1}, 2*\text{stream\_length2})).$$

Note that this implementation finds the minimum edit distance between two sequences, but does not output the optimal alignment between the two sequences. It is not difficult to extend this technique to output the alignment as well as the edit distance, as shown in [32].

```c
#include <stdio.h>

#define SEQ_SOURCE "ACTATTATAA"
#define SEQ_TARGET "ACTATAATA"

/* Define the Sequences To Compare */
#define SEQ_SOURCE "ACTATTATAA"
#define SEQ_TARGET "ACTATAATA"
*/

#include <stdio.h>

/* Define the Sequences To Compare */
define SEQ_SOURCE "ACTATTATAA"
define SEQ_TARGET "ACTATAATA"
*/
/ Edit distance penalties.
* Here, there is no penalty for a residue match.
* Insertions and Deletions (Gaps) are assigned
  * a penalty of 1 point, while residue substitutions
  * (mismatches) are given a penalty of 2.
* It is not uncommon to penalize a mismatch less than
  * a gap. However, I am sticking with the penalty values
  * used in Hoang's FPGA implementation for clarity
  *
  */
#define MISMATCH 2
#define GAP 1
#define MATCH 0

/*

* The PE is the "Processing Element", which
* is the basic building block of the systolic
* array. Using reconfigurable hardware such as
* FPGA's, these would be implemented directly in
* hardware. In Hoang's implementation on
* Splash-2, a single-board Splash-2 system with
* 16 Xilinx FPGAs implemented 24 PE's per FPGA
* chip, for a total of 384 PEs per board.
*
*/
typedef struct struct_PE {
    char SCin; /* source character input */
    char SCout; /* source character output */
    char TCin; /* target character input */
    char TCout; /* target character output */
    int SDin; /* source distance input */
    int SDout; /* source distance output */
    int TDin; /* target distance input */
    int TDout; /* target distance output */
    int PEDist; /* The stored edit distance */
} PE;

/*

* Data Stream element
*
* This is the input data stream structure. We start
* with a character from our DNA sequence, followed by
* an initial distance value which is essentially the
* index of the sequence character in the stream.
* The streams get allocated and initialized dynamically
* as arrays of STREAM structures.
*
*/
typedef struct struct_STREAM {
    char c;
    int d;
} STREAM;

/* some function prototypes */
STREAM *init_stream(int *length, char *seq);
int cost(char a, char b);
void printPE(PE *P);

/*
 * main() -
 * Allocate and initialize our source streams.
 * Allocate and initialize our array of PEs
 * Iterate through and calculate the total minimum
 * edit distance between the two provided sequences.
 * Output the state of each PE at each iteration,
 * and output the edit distance at the end.
 */
int main(int argc, char *argv[]) {
    STREAM *source_stream;
    STREAM *target_stream;
    int source_length, target_length;
    int num_PE, index, i;
    int final_edit_distance;
    PE *PEarray, *P;

    /* initialize source stream */
    source_stream = init_stream(&source_length, SEQ_SOURCE);
    /* initialize destination stream */
    target_stream = init_stream(&target_length, SEQ_TARGET);

    /* determine size of array and allocate */
    num_PE = 2*max(strlen(SEQ_SOURCE)+1, strlen(SEQ_TARGET)+1)-1;
    PEarray = (struct struct_PE *)calloc(num_PE, sizeof(PE));

    /* Do the real work here */
    /* Hoang states "The number of steps required to compute the edit
     * distance and to transport it out of the array is
     * proportional to the length of the array"
     *
     * To be more specific, I determined that the number of required
     * iterations is equal to the number of PE elements in your system
     * PLUS the length of the largest sequence being compared.
     */
    for(index=0;index<num_PE+max(source_length, target_length);index++) {
        /* print a marker between iterations */
        printf("------------------------------------\n");

        /* inject streams on both end */
        if(index<source_length) {
            PEarray[0].SCin = source_stream[index].c;
            PEarray[0].SDin = source_stream[index].d;
        } else {
            PEarray[0].SCin = 0;
            PEarray[0].SDin = 0;
        }

        if(index<target_length) {
            PEarray[num_PE-1].TCin = target_stream[index].c;
            PEarray[num_PE-1].TDin = target_stream[index].d;
        } else {
            PEarray[num_PE-1].TCin = 0;
            PEarray[num_PE-1].TDin = 0;
        }
    }
    printf("\n");
}

50
for(i=0;i<num_PE;i++) {
    /* get a shortcut pointer for more readable code */
    P = &PEArray[i];

    /* In both directions, the input of one PE is equal to
    * the output of its neighbor in a linear fashion
    */
    if(i>0) {
        P->SCin  = PEArray[i-1].SCout;
        P->SDin  = PEArray[i-1].SDout;
    }
    if(i<num_PE-1) {
        P->TCin  = PEArray[i+1].TCout;
        P->TDin  = PEArray[i+1].TDout;
    }
}

/* Here we iterate through each PE in the systolic
 * array, calculating the stored "edit distance"
 * for each PE. This is the core of the algorithm,
 * where dynamic programming is being implemented.
 * Each PE essentially represents an anti-diagonal
 * in a dynamic programming matrix, where there exists
 * a recurrence relationship (dependency) between
 * each anti-diagonal and its predecessor (to the upper
 * left).
 * All entries along an anti-diagonal can be computed
 * in parallel, while successive anti-diagonals are
 * data-dependent.
 */
for(i=0;i<num_PE;i++) {
    P = &PEArray[i];  /* a shortcut variable */
    if(P->SCin && P->TCin) {
        P->PEDist = min(P->PEDist + cost(P->SCin, P->TCin),
                        P->TDin   + GAP,
                        P->SDin   + GAP);
    } else if (P->SCin) {
        P->PEDist = P->SDin;
    } else if (P->TCin) {
        P->PEDist = P->TDin;
    }

    /* progress the systolic array here */
    P->SCout = P->SCin;
    P->TCout = P->TCin;
    P->SDout = P->PEDist;
    P->TDout = P->PEDist;
    printPE(P);
}

final_edit_distance = P->TDout;
printf("Minimum Edit Distance: %d\n", final_edit_distance);
/* de-allocate what we've allocated */
free(source_stream);
free(target_stream);
free(PEarray);
exit(0);
}

/*
 * printPE() -
 * This function simply pretty-prints the state of
 * a PE
 */
void printPE(PE *P) {
    char SCin, SCout, TCin, TCout;
    SCin = P->SCin;
    if(!SCin) {
        SCin = '-';
    }
    SCout = P->SCout;
    if(!SCout) {
        SCout = '-';
    }
    TCin = P->TCin;
    if(!TCin) {
        TCin = '-';
    }
    TCout = P->TCout;
    if(!TCout) {
        TCout = '-';
    }
    printf("PEDist = %d\n", P->PEDist);
    printf("SCin = %c, SCout = %c\n", SCin, SCout);
    printf("SDin = %d, SDout = %d\n", P->SDin, P->SDout);
    printf("TCin = %c, TCout = %c\n", TCin, TCout);
    printf("TDin = %d, TDout = %d\n", P->TDin, P->TDout);
    printf("\n");
    return;
}

/*
 * cost() -
 * Here, we calculate the cost of a match/mismatch
 * by essentially determining whether there was a match or
 * a mismatch and returning the appropriate penalty.
 */
int cost(char a, char b) {
    if(a == b) {
        return MATCH;
    } else {
return MISMATCH;
}
}

/*
 * max() -
 * Give the maximum of two integer values
 */
int max(int a, int b) {
    if(a>=b) {
        return a;
    } else {
        return b;
    }
}

/*
 * min() -
 * Compute the minimum of three values, used
 * in the core of the main algorithm to
 * implement dynamic programming
 */
int min(int a, int b, int c) {
    if(a<=b) {
        if(a<=c) {
            return a;
        } else {
            return c;
        }
    } else {
        if(b<=c) {
            return b;
        } else {
            return c;
        }
    }
}

/*
 * init_stream() -
 * This routine initializes an input stream,
 * given a DNA sequence. It inserts a space
 * between each character, and sets default
 * edit distances for each character, based on
 * its position within the array
 */
STREAM *init_stream(int *length, char *seq) {
    int i, l, j;
    STREAM *stream;

    *length = l = strlen(seq)*2-1;
    stream = (STREAM *)calloc(l, sizeof(STREAM));
    j = 0;
    for(i=0;i<l;i++) {
        if(i%2==0) {
            stream[i].c = seq[j++];
            stream[i].d = j;
        } else {
            stream[i].c = ' ';
            stream[i].d = 0;
        }
    }
    return stream;
}
} else {
    stream[i].c = 0;
    stream[i].d = 0;
}
}

/*
for(i=0;i<l;i++) {
    printf("[%c,%d]\n", stream[i].c, stream[i].d);
} */

return(stream);
BIBLIOGRAPHY:

[9] Douglas L. Brutlag et al., BLAZE: An Implementation of the Smith-Waterman Sequence Comparison Algorithm on a Massively Parallel Computer, Department of Biochemistry, Stanford Medical School
[12] David Dahle et al., The UCSC Kestrel General Purpose Parallel Processor, 1999, Department of Computer Engineering, Jack Baskin School of Engineering, University of California, Santa Cruz
[13] Barry Fagin and J. Gill Watt, A Special-Purpose Processor for Gene Sequence Analysis, Thayer School of Engineering, Dartmouth College
[16] Steven A. Guccione and Eric Keller, Gene Matching using JBits, Xilinx, Inc
[20] Dzung T. Hoang, Searching Genetic Databases on Splash 2, Department of Computer Science, Brown University
[22] Richard Jaenicke, Programming Applications for the AltiVec Architecture, Mercury Computer Systems
[23] D. Lavenier and J.-L. Pacherie, Parallel Processing for Scanning Genomic Data-Bases, IRISA, Campus de Beaulieu
[24] Dominique Lavenier, SAMBA – Systolic Accelerators for Molecular Biological Applications, IRISA, Campus Universitaire de Beaulieu
[25] Bing-Chang Lai and Phillip John McKerrow, Programming the Velocity Engine, School of Information Technology and Computer Science, University of Wollongong
[26] Programming with the Intel MMX Technology, Intel Corporation
[28] Raj K. Singh et al., A Scalable Systolic Multiprocessor System for Analysis of Biological Sequences, Department of Computer Science, Program in Molecular Biology and Biotechnology, University of North Carolina at Chapel Hill
[30] Bowen Alpern et al., Microparallelism and High-Performance Protein Matching, Supercomputing ’95
[36] Sea H Choi et al., Signal Processing Applications using VHDL on Splash 2, Dept. of Computer Science, Michigan State University Dept. of Electrical Engineering
[37] Richard Lipton, Daniel Lopresti, A systolic array for rapid string comparison, Chapel Hill Conference on Very Large Scale Integration (VLSI), pp. 363-376, 1985
[38] Hughey and Daniel P. Lopresti, B-SYS: A 470-Processor Programmable Systolic Array, Department of Computer Science, Brown University
[42] Aaron Patzer, Highly Parallel DNA Sequence Matching and Alignment Processor, Electrical and Computer Engineering Department, Duke University