To be presented at: The 1986 International Workshop on Systolic Arrays
University of Oxford, July 2-4, 1986

Comparing Long Strings on a Short Systolic Array

Richard J. Lipton
Daniel Lopresti

CS-TR-026-86

Department of Computer Science
Princeton University
Princeton, NJ 08544

February 12, 1986

Abstract

In this paper we demonstrate two techniques for comparing strings of arbitrary length on a systolic array we have proposed and implemented. The first is an application of algorithm partitioning, the second an implementation of a string matching heuristic for which we prove performance bounds. We analyze the time and number of processors required in each case and determine the processor utilizations.

This research was supported in part by DARPA under contract N00014-82-K-0549.

Comparing Long Strings on a Short Systolic Array

Richard J. Lipton
Daniel Lopresti

1 Introduction

Rapid advances in very large scale integration have encouraged researchers to implement the inner loops of many algorithms in special-purpose parallel hardware. The unique demands of VLSI technology favor architectures which have a large number of identical processing elements with only local communication requirements; Kung has termed these "systolic arrays." In the literature, most such arrays are presented as requiring a number of processors proportional to the size of the problem. This is, perhaps, an unrealistic assumption as interesting computations would often require hundreds or thousands of processors. Recently, however, authors have begun to investigate the mapping of algorithms onto fixed-size arrays [Chen85a][Chen85b][Hwan81][Mold84][Mold86]. In this paper we demonstrate two techniques for comparing strings of arbitrary length on a systolic array we have proposed and implemented [Lipt85]. The first is an application of algorithm partitioning, the second an implementation of a string matching heuristic.

Of other published processor arrays for string comparison [Fost80][Liu84][Chen85b], only [Chen85b] considers the partitioning problem. The given array, however, is 2-dimensional and requires a complicated data recirculation.

1.1 Quantifying String Similarity

There are a number of different measures for quantifying the similarity between two strings [Hall80]. Perhaps one of the most intuitive is the "edit distance," which can be defined as the minimum cost of a sequence of insertions, deletions, and substitutions needed to transform one string, the source, into the other, the target.

For example, one possible transformation of the string *systolic* into the string *symbolic* is

$$systolic \vDash delete\ s \Rightarrow\ sytolic \vDash delete\ t \Rightarrow\ syolic \vDash insert\ m \Rightarrow\ symolic \vDash insert\ b \Rightarrow\ symbolic$$

which requires two deletions and two insertions. Another possibility, using only substitutions, is

$$systolic \vDash substitute\ m\ for\ s \Rightarrow\ symtolic \vDash substitue\ b\ for\ t \Rightarrow\ symbolic$$

If we assign an insertion or a deletion a cost of one, and a substitution a cost of two (a substitution can be considered an insertion and a deletion), then these are both minimum cost transformations, yielding an edit distance of four. We will assume these cost assignments for the rest of this paper.

1.2 A Dynamic Programming Algorithm

Wagner and Fischer present a dynamic programming solution to the problem of determining edit distance [Wagn74]. Given source and target strings

$$s_1 s_2 \cdots s_m \quad and \quad t_1 t_2 \cdots t_n$$

the algorithm builds an $m \times n$ table by comparing successively longer subsequences.

Define $d_{i,j}$ to be the edit distance between $s_1 s_2 \cdots s_i$ and $t_1 t_2 \cdots t_j$. Then the initial conditions for the computation are

$$d_{i,0} = i \qquad 0 \le i \le m \qquad \text{and} \qquad d_{0,j} = j \qquad 0 \le j \le n$$

and the value $d_{i,j}$ is determined by the recurrence.

$$d_{i,j} = min\left[d_{i,j-1} + 1, d_{i-1,j} + 1, \begin{cases} d_{i-1,j-1} & if s_i = t_j \\ d_{i-1,j-1} + 2 & otherwise \end{cases} \right] \qquad 1 \le i \le m, 1 \le j \le n$$

That is, $d_{i,j}$ depends only on its three neighboring values up and to the left and on the appropriate source and target characters. When the algorithm terminates the value $d_{m,n}$ is the edit distance between the two strings.

The example of the previous section would result in the table shown in figure 1.2.1. The arrows

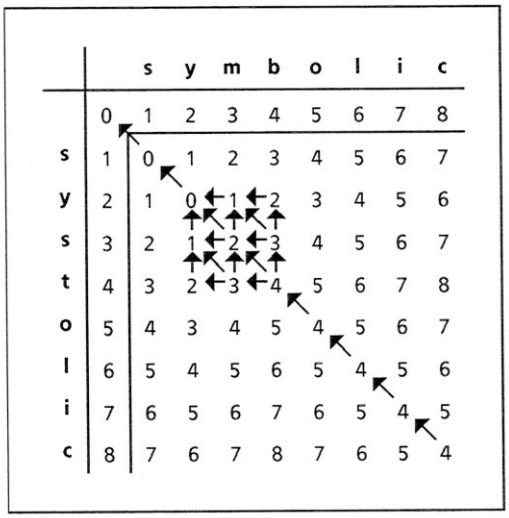

figure 1.2.1 the table

point back along the paths of optimal decisions. Notice that because the strings are quite similar these paths never stray far from the main diagonal.

On a serial computer this algorithm takes time $O(mn)$. Although the entire table requires storage $O(mn)$, the state of the computation necessary for processing a column, row, or diagonal consists only of the previous column, row, or two diagonals respectively, and the source and target strings. Hence, the space complexity is $O(\max(m, n))$.

1.3 A Systolic Array for Determining Edit Distance

There is a great deal of potential concurrency in the construction of an edit distance table; all of the values on any forty-five degree diagonal can be calculated simultaneously. We realize this parallelism by mapping the algorithm onto a linear systolic array [Lipt85]. The data flow can be

visualized by rotating the index set of the recurrence and collapsing one dimension in time [Capp84], as depicted in figure 1.3.1.

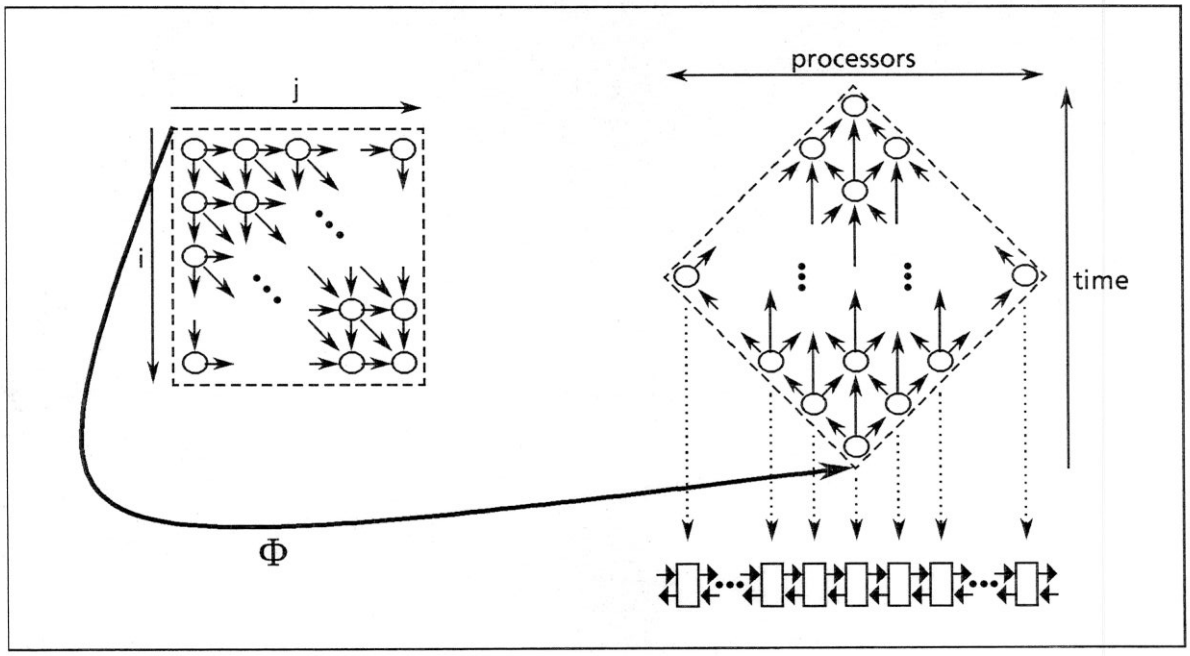

figure 1.3.1 mapping the recurrence onto a linear array

For a rigorous analysis, we adopt the formalism of Moldovan and Fortes [Mold86]. The index set I for the recurrence is

$$I = \left\{ \begin{bmatrix} i \\ j \end{bmatrix} : 1 \le i \le m, 1 \le j \le n \right\}$$

and the dependency vectors D are

$$D = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

Since we want to transform this algorithm into one suitable for implementation on a one-dimensional array we shall, for simplicity, view the first index of the image as time and the second as space. We seek a transformation

$$\Phi = \begin{bmatrix} T \\ P \end{bmatrix}$$

on the index set, where T determines the timing and P determines the processor allocation. That is, the computation indexed by $[i\ j]^t$ will be performed during clock cycle $T[i\ j]^t$, in processor number $P[i\ j]^t$. If we limit each processor to only one calculation per clock, the transformation must be one-to-one.

The choice of T is further restricted by the requirement

$$Td > 0 \quad \textit{for all } d \in D$$

That is, the timing of the transformed algorithm must guarantee that the dependencies can be satisfied; any calculation which depends on the result of another must be scheduled after that one. Of the many possibilities, we choose

$$T = [\ 1\ \ 1\ ]$$

For P we require

$$Pd = -1, 0, or 1 \qquad for all\ d \in D$$

That is, we permit only nearest-neighbor communication among the processors. If the processors are numbered from left to right -1 corresponds to a left connection, 0 to a self-connection (or stored state), and 1 to a right connection. We choose

$$P = [\ -1\ \ 1\ ]$$

The array that results from applying the transformation

$$\Phi = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

to the index set I is that of figure 1.3.1.

As figure 1.3.2 demonstrates, the source and target strings shift in simultaneously from the left

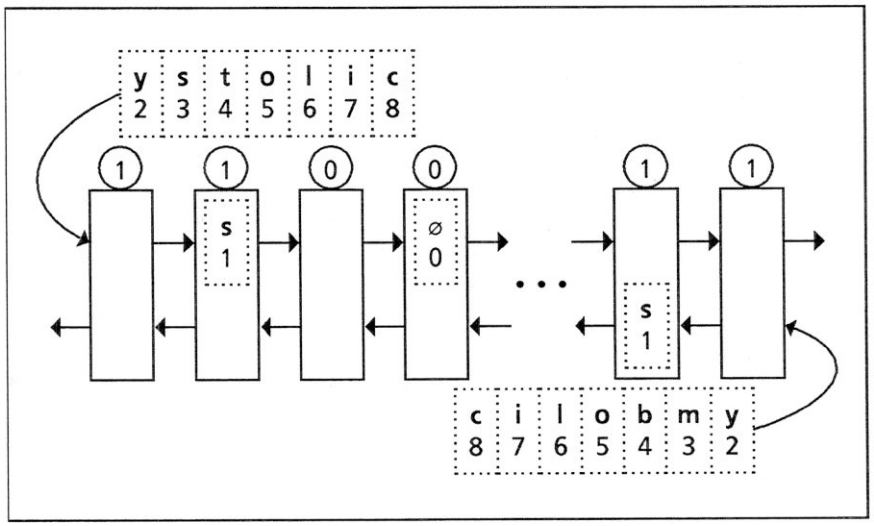

figure 1.3.2 a systolic array for approximate string matching

and right respectively; two new characters are input every other clock cycle. Traveling along with the characters are initializing data values which are the first column and row of the dynamic programming matrix. As these values pass through the array they will be transformed in to the last column and row, and hence the answer.

A single processing element and the program which it executes each clock cycle are shown in figure 1.3.3. When two non-null characters enter from opposite directions, a comparison is performed and the processor determines a new state based on the result of the comparison, the two input values, and its previous state, exactly as in the dynamic programming algorithm.

Before a processor can take part in the computation, its state must be initialized appropriately. This can be accomplished by first flushing the array with null characters (∅), then having each processor assume the state of the last non-null character it has seen, until it performs its first comparison. This initialization is not shown in the figures.

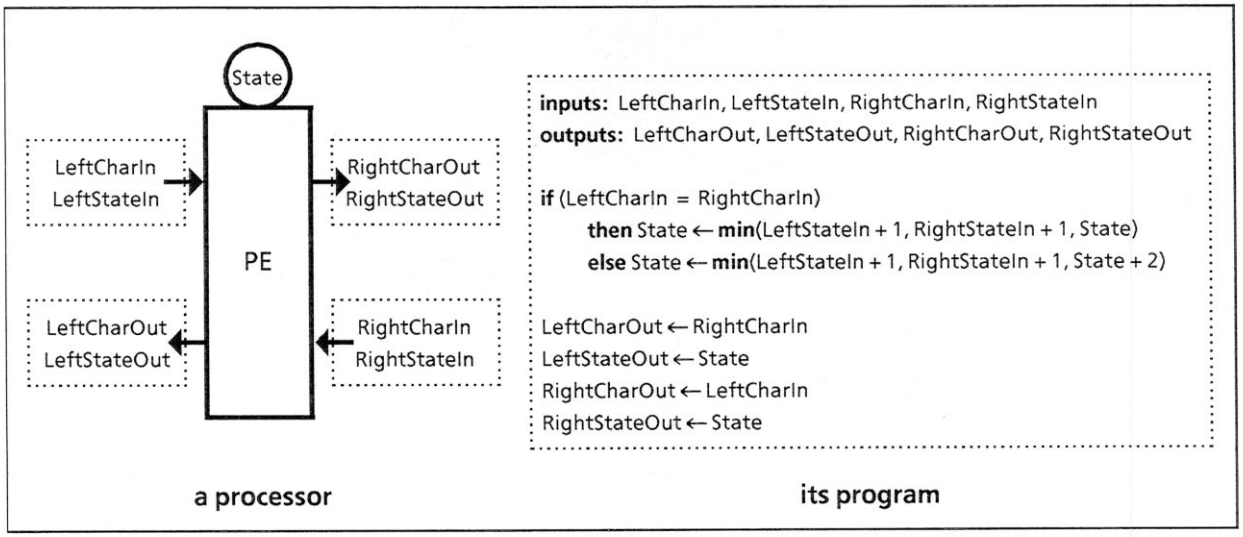

figure 1.3.3 one processing element

Since, under the timing [1 1], the first index point to be processed is $[1 \;\; 1]^t$ and the last is $[m \;\; n]^t$ the basic computation requires

$$T\begin{bmatrix} m \\ n \end{bmatrix} - T\begin{bmatrix} 1 \\ 1 \end{bmatrix} + 1 = m + n - 1$$

clock cycles. It is not difficult to show that this timing achieves the minimum number of cycles necessary to perform the calculation.

The allocation $[-1 \;\; 1]$ maps the point $[m \;\; 1]^t$ into the leftmost processor and $[1 \;\; n]^t$ into the rightmost, so we require

$$P\begin{bmatrix} 1 \\ n \end{bmatrix} - P\begin{bmatrix} m \\ 1 \end{bmatrix} + 1 = m + n - 1$$

processors.

A common efficiency measure for parallel algorithms is processor utilization, the ratio of the serial processing time to the parallel processing time multiplied by the number of processors used. First we must remark that the above analysis measures the interval between the first update and the last and ignores the time needed to shift values into and out of the array. This adds another $(m+n-1)$ cycles to the processing time and is the price we pay for allowing input to and output from the array only at the two end processors. With this in mind, we obtain the expression

$$PU = \frac{mn}{2(m+n-1)(m+n-1)}$$

For source and target strings which are the same length, m, this simplifies to

$$PU = \frac{1}{8 - \dfrac{8}{m} + \dfrac{2}{m^2}}$$

Thus, the processor utilization is effectively 1/8 for long strings. Note that perfect utilization of the processors under this alternating scheme is 1/2.

There are allocations which require fewer processors (for example [0 1]) and hence have higher processor utilizations. Still, this P is attractive because, as was shown, it results in an array which can be implemented with only one mode of operation, shifting data. The array for P equals [0 1] requires that the target string be shifted into the array, then held there while the source string streams by.

1.4 Minimizing Processor Complexity

Since the edit distance between two strings can be as large as $m+n$, it appears that each processor requires an adder and a comparator for numbers $O(\log(m+n))$ bits long. There is, however, an important observation that makes manipulating such large values unnecessary; a constant two bits will suffice. We remark that, at any instant in time, adjacent processors' edit distances are close in magnitude, hence it is only necessary to compute low order bits. As the truncated values shift out of the right side of the array, the full edit distance is reconstructed. An up/down counter is preloaded with the length of the target string and a simple finite state machine increments or decrements it appropriately. Figure 1.4.1 illustrates this. See [Lipt85] and [Lipt86] for details.

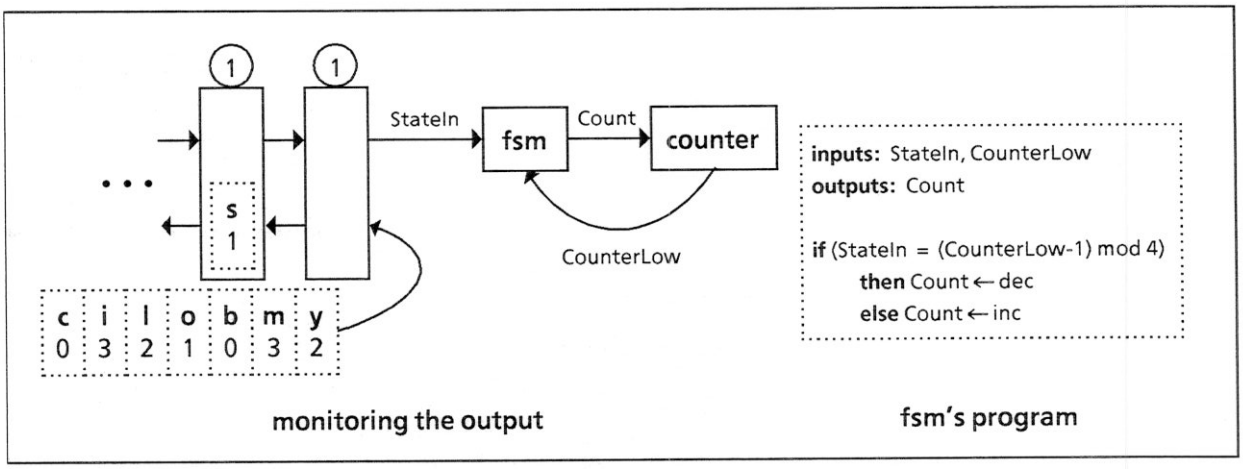

figure 1.4.1 reconstructing the edit distance

This result is significant. Our implementation of this systolic array is unlimited in the length strings that it can compare. Any number of chips can be placed side-by-side as there is no maximum edit distance hardwired in silicon; only the up/down counter, an inexpensive and cascadable TTL part, is length dependent.

2 Comparing Long Strings in Multiple Passes

Despite the unrestricted expandability of our design, it may not always be cost effective to build an array long enough to handle each possible comparison. For example, our current chip is designed to compare DNA sequences, which range in length from tens to tens of thousands of characters. To perform every comparison in a single pass would require tens of thousands of processors, when often

only a hundred or so are actually used. Fortunately, it is possible to partition the problem into blocks so that strings of any length can be compared on an array of fixed length. In this case, rows or columns at earlier points in the computation are recirculated to initialize later calculations.

Figure 2.1 depicts a partitioning of the dynamic programming matrix shown earlier for a systolic array which has seven processors. The calculation is done in four separate passes. The intermediate rows and columns must be stored, external to the array, for their later use.

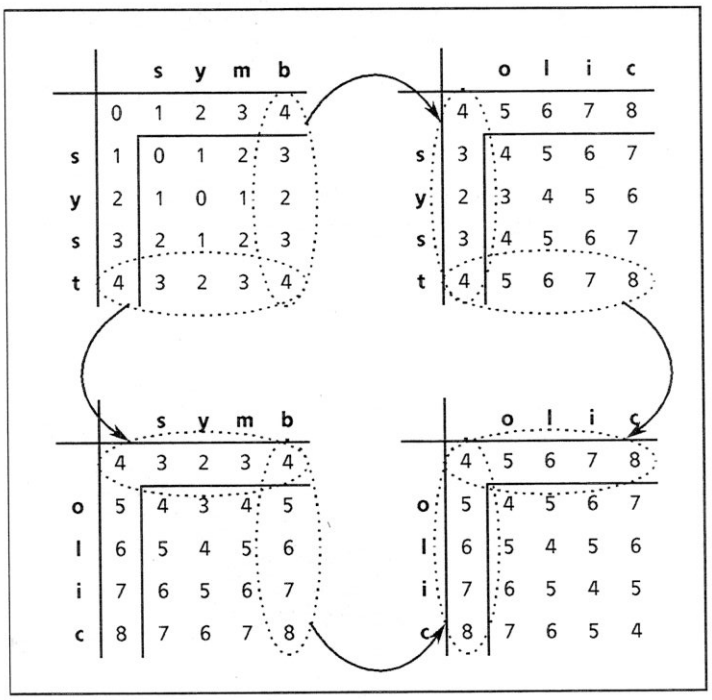

figure 2.1 the partitioned tables

In general, say that we have $2p-1$ processors in a linear array and hence can compare two p-long string segments in one pass. Further, for simplicity, assume that m/p is m_p and n/p is n_p. We divide the computation into $m_p n_p$ blocks, each of size $p \times$p. The scheduling of this secondary computation can be handled just as the primary case was. The index set is

$$I_p = \left\{ \begin{bmatrix} i_p \\ j_p \end{bmatrix} : 1 \le i_p \le m_p, 1 \le j_p \le n_p \right\}$$

and the dependency vectors are

$$D_p = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}$$

Now we are looking for a transformation

$$\Phi_p = \begin{bmatrix} T_p \\ P_p \end{bmatrix}$$

for the block calculations. Again, we require that

$$T_p d_p > 0 \qquad for all\ d_p \in D_p$$

However, since we assume that we have only one array, there is no "processor" allocation, so P_p is [0 0]. Hence, to guarantee that index points are mapped one-to-one, we require the transformation T_p to yield a complete ordering in time. To guide the choice of T_p we observe that the interval between the time a column is generated and when it is used is $T_p[0\ 1]$. This is a measure of the amount of external storage required to hold intermediate column values. Similarly for rows, $T_p[1\ 0]$ measures the time between when a row is generated and when it is used. Hence, to minimize external storage we want the T_p which minimizes

$$T_p\begin{bmatrix} 0 \\ 1 \end{bmatrix} + T_p\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Thus, we are lead to choose $[1\ \ m_p]$ for T_p if the source string is shorter than the target (i.e. $m_p < n_p$), and $[n_p\ \ 1]$ otherwise. Of course, in reality it is far more practical to fix the architecture and define the target string to be the shorter of the two.

Figure 2.2 depicts a systolic array for comparing long strings in multiple passes, illustrated by the source and target strings of the earlier example. The computation as pictured is beginning its second pass, which corresponds to the block in the upper right corner of figure 2.1. The longer queue provides

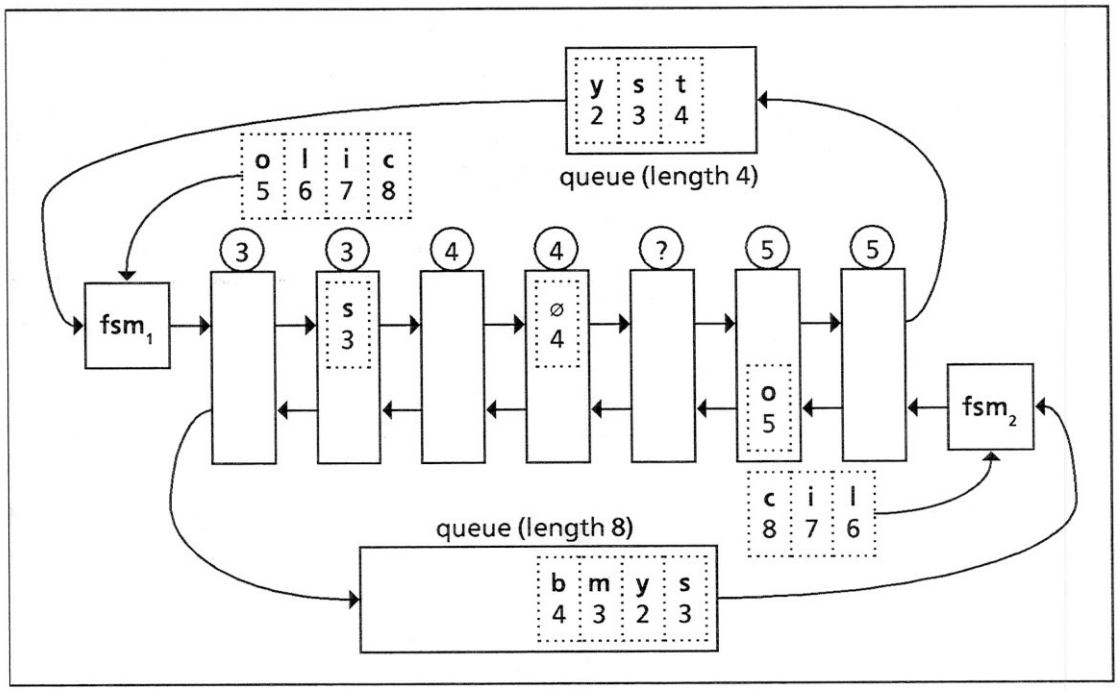

figure 2.2 a multiple pass array

the temporary external storage discussed earlier; its length is that of the shorter string. The two finite state machines fsm_1 and fsm_2 determine when new characters are input and when old ones are recirculated; their programs are shown in figure 2.3. The routine *FlushArray*() simply shifts null characters through to force the source and target segments out and reinitialize the array for the next pass. The shorter queue is necessary to hold values during this flushing. Of course, the state encoding trick of section 1.4 still applies and can be realized by augmenting the program for one of the state machines accordingly.

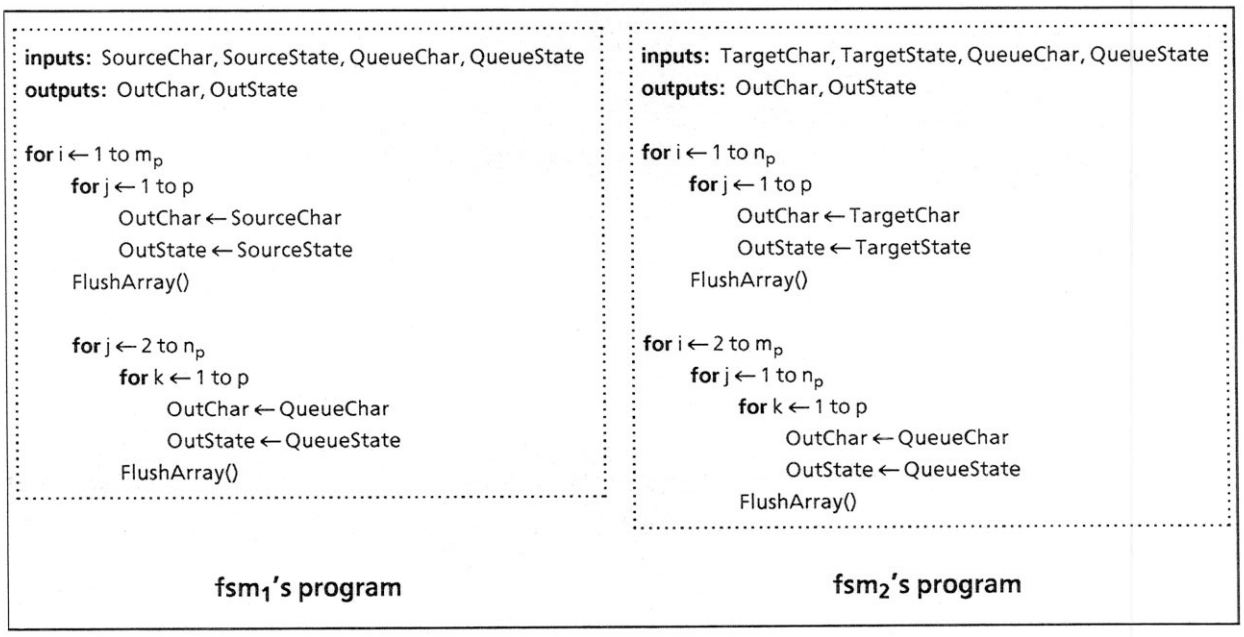

figure 2.3 the finite state machine programs

The time needed for the entire computation using this method is

$$(number\ of\ blocks)(time\ per\ block) = (\frac{mn}{p^2})(2(2p-1))$$

The number of processors is, of course,

$$2p-1$$

Hence, the processor utilization is

$$PU = \frac{1}{8 - \frac{8}{p} + \frac{2}{p^2}}$$

which, as before, is effectively 1/8 for p's of interest.

3 Systolic Implementation of a String Matching Heuristic

Partitioning the computation, as described in the previous section, has the distinct disadvantage of requiring external queues (of programmable size if strings of differing lengths are to be compared) and a rather complicated data recirculation. Fortunately, in some situations long strings can be compared on an array of fixed length by using a heuristic modification of the original dynamic programming algorithm, with no such increase in hardware complexity.

3.1 The Heuristic

It is frequently the case that we are interested in the precise edit distance between two strings only if they are quite similar, for example, when searching databases for near matches. The heuristic is based on the observation that if two strings are close, then the path of optimal editing decisions must not stray far from the main diagonal [Fick84] (a similar remark applies to the dynamic time warping of speech [Rabi78]). Hence the modified algorithm only calculates edit distances for index points within some δ of the diagonal.

Define $d^*_{i,j}$ to be the edit distance calculated under the heuristic. Then, for δ greater than or equal to two, the initial conditions are

$$d^*_{i,0} = i \quad 0 \le i < \delta \qquad and \qquad d^*_{0,j} = j \quad 0 \le j < \delta$$

and the recurrence breaks into three cases

$$d^*_{i,j} = min\left[d^*_{i,j-1} + 1, d^*_{i-1,j} + 1, \left\{ \begin{array}{ll} d^*_{i-1,j-1} & if s_i = t_j \\ d^*_{i-1,j-1} + 2 & otherwise \end{array} \right] \right] \qquad if |i-j| < \delta - 1$$

and

$$d^*_{i,j} = min\left[d^*_{i-1,j} + 1, \left\{ \begin{array}{ll} d^*_{i-1,j-1} & if s_i = t_j \\ d^*_{i-1,j-1} + 2 & otherwise \end{array} \right] \right] \qquad if i - j = \delta - 1$$

and

$$d^*_{i,j} = min\left[d^*_{i,j-1} + 1, \left\{ \begin{array}{ll} d^*_{i-1,j-1} & if s_i = t_j \\ d^*_{i-1,j-1} + 2 & otherwise \end{array} \right] \right] \qquad if j - i = \delta - 1$$

That is, we only calculate those $d^*_{i,j}$ which are within δ of the main diagonal.

Figure 3.1.1 shows the effect of the heuristic on the computation of the earlier example, with δ equal to four. In this case, we get precisely the same answer as originally; this is because optimal

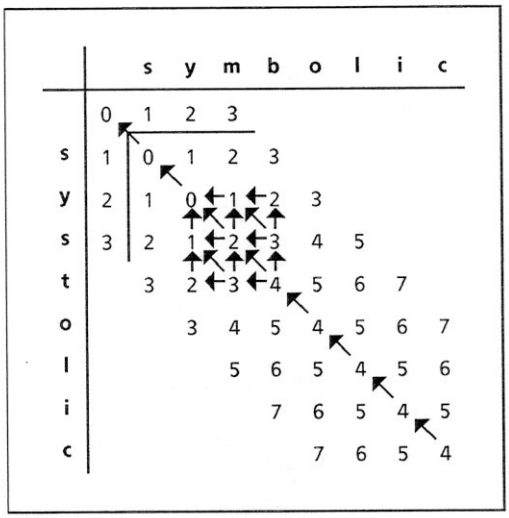

figure 3.1.1 the heuristic with δ = 4

paths exist close to the diagonal. We may not always be so lucky, as is demonstrated in figure 3.1.2.

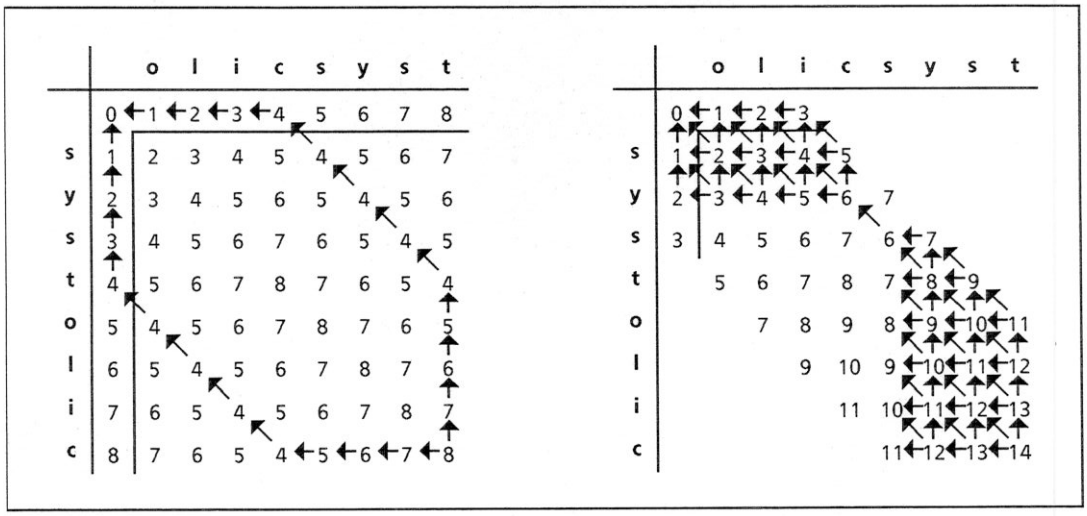

figure 3.1.2 a case where the heuristic fails

Here the optimal paths stray far from the center diagonal. When the search is limited we obtain a poor result.

We now characterize the quality of this heuristic. First we show that it always returns edit distances at least as great as those calculated by the original algorithm.

Theorem 1:

$$d^*_{i,j} \geq d_{i,j} \qquad \textit{for any } i \textit{ and } j \textit{ for which } d^*_{i,j} \textit{ is defined}$$

Proof: by induction. Clearly $d^*_{i,0}$ equals $d_{i,0}$ and $d^*_{0,j}$ equals $d_{0,j}$, so the basis holds. Any index that appears in the minimization determining $d^*_{i,j}$ also appears in that determining $d_{i,j}$. Hence, by the inductive hypothesis, the theorem holds.

Now we show that if the original algorithm returns a final edit distance less than or equal to some value, the heuristic is guaranteed to give us the same result. Without loss of generality, say that the source string is the longer of the two and the difference between the lengths, $m-n$, is α. Then we have

Theorem 2: if

$$d_{m,n} \leq 2(\delta - \alpha - 1)$$

then

$$d^*_{m,n} = d_{m,n}$$

Proof: let us view the transformation as performing the necessary editing operations solely on the source string. Consider any path of optimal decisions which determines $d_{m,n}$. First we observe that it must include α deletions to equalize lengths. After these operations are accounted for, each of β deletions in the source string must be matched by an insertion. Now, clearly

$$\alpha + 2\beta \leq d_{m,n} \leq 2(\delta - \alpha - 1)$$

so

$$2(\alpha + \beta) \leq 2\delta - \alpha - 1$$

or, since α is greater than or equal to zero

$$\alpha + \beta < \delta$$

This last result implies that at no point does the optimal path stray more than δ from the main diagonal. Hence, the computation determining $d^*_{m,n}$ has access to the same path in its entirety. The proof that $d^*_{m,n}$ equals $d_{m,n}$ now follows by a simple induction.

These two theorems demonstrate that strings which are found to be distant under the original algorithm remain at least as distant under the heuristic, while strings which are close remain just as close. The choice of δ determines when strings are considered "close."

If, as before, we assume that the source string is the longer the heuristic requires at most

$$(2\delta - 1)(m - 2\delta) + 2 \sum_{i=\delta}^{2\delta - 1} i = 2m\delta - m - \delta^2 - \delta$$

steps, which, for δ a constant, is $O(m)$. The space required is still $O(m)$ since eventually the full source and target strings are needed. However, the state of the computation at any point in time uses only constant storage.

3.2 A Systolic Implementation

Fortunately, the structure of the heuristic is nearly identical to that of the original algorithm; we can easily map it onto our existing systolic array. The index set I^* for the recurrence is

$$I^* = \left\{ \begin{bmatrix} i \\ j \end{bmatrix} : 1 \leq i \leq m, (i - \delta + 1) \leq j \leq (i + \delta - 1) \right\}$$

and the dependency vectors are, as before,

$$D^* = \left\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$$

Hence we can use precisely the same transformation

$$\Phi^* = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

Figure 3.2.1 depicts the corresponding systolic implementation of the heuristic. The only difference between this array and the one shown in figure 1.3.2 is that in this case end processors minimize two values instead of three. Our current chip can be "fooled" into doing this by always inputing values which are one greater than these processors' stored states. If the simplifying

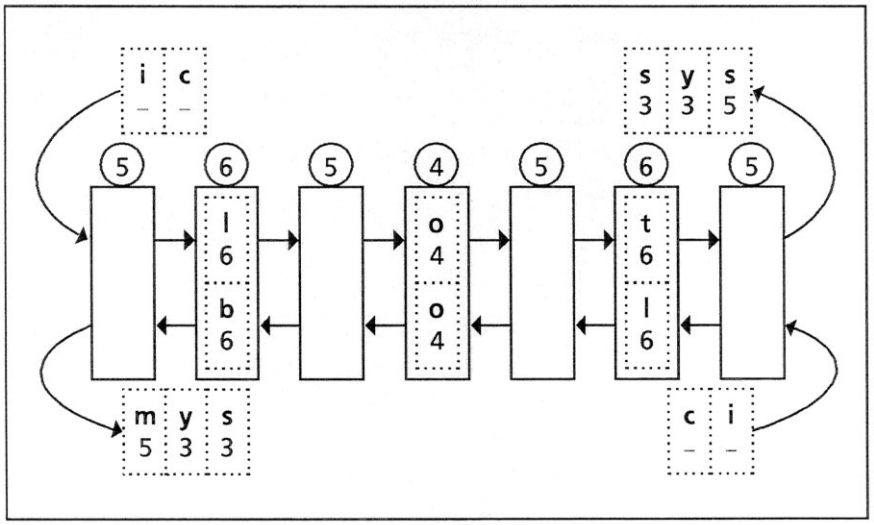

figure 3.2.1 a systolic array for the string matching heuristic

technique of section 1.4 is to be used the finite state machine must be changed slightly as now edit distances shifting out of the array can differ by two instead of one.

Since the first index point to be processed is $[1 \;\; 1]^t$ and the last is $[m \;\; n]^t$ the computation requires $2m-1$ clock cycles. To this must be added $2\delta-1$ cycles for shifting the initial values in and the final values out of the array. Under the chosen allocation P^*, the point $[i \;\; i-\delta+1]^t$ will be processed in the leftmost processor and $[i \;\; i+\delta-1]^t$ will be processed in the rightmost, so the heuristic requires $2\delta-1$ processing elements. Hence the processor utilization is

$$PU = \frac{2m\delta - m - \delta^2 - \delta}{(2m-1)(2\delta-1)+(2\delta-1)^2} = \frac{2\delta - 1 - \dfrac{\delta^2}{m} - \dfrac{\delta}{m}}{4\delta - 2 + \dfrac{4\delta^2}{m} - \dfrac{6\delta}{m} + \dfrac{2}{m}}$$

as m gets large this approaches

$$PU = \frac{2\delta-1}{4\delta-2} = \frac{1}{2}$$

which, under this alternating scheme, is perfect utilization of the processors.

4 Conclusions

In this paper we have presented two techniques for comparing long strings on a short systolic array. The first always returns a correct edit distance, the second does so only when the strings are similar. Although the discussion was illustrated by the specific implementation we have realized in silicon, both methods can be generalized. The relative merits of alternative architectures, in light of the partitioning problem, as well as the usefulness of the string matching heuristic deserve further investigation.

5 References

[Capp84] Peter R. Cappello and Kenneth Steiglitz, "Unifying VLSI Array Design with Linear Transformations of Space-Time," *Advances in Computing Research*, volume 2, 1984, pp. 23-65.

[Chen85a] H. D. Cheng and K. S. Fu, "Algorithm Partition for a Fixed-Size VLSI Architecture Using Space-Time Domain Expansion," *Proceedings of the Seventh Symposium on Computer Arithmetic*, June 1985, pp. 126-132.

[Chen85b] H. D. Cheng and K. S. Fu, "VLSI Architectures for Pattern Matching Using Space-Time Domain Expansion Approach," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, 1985, pp. 181-184.

[Fick84] James W. Fickett, "Fast Optimal Alignment," *Nucleic Acids Research*, volume 12, number 1, 1984, pp. 175-179.

[Fost80] M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *IEEE Computer*, volume 13, January 1980, pp. 26-40.

[Hall80] Patrick A. V. Hall and Geoff Dowling, "Approximate String Matching," *ACM Computing Surveys*, volume 12, number 4, December 1980, pp. 381-402.

[Hwan81] Kai Hwang and Yen-Heng Cheng, "Partitioned Algorithms and VLSI Structures for Large-Scale Matrix Computations," *Proceedings of the Fifth Symposium on Computer Arithmetic*, 1981, pp. 222-232.

[Lipt85] Richard J. Lipton and Daniel Lopresti, "A Systolic Array for Rapid String Comparison," *1985 Chapel Hill Conference on Very Large Scale Integration*, Henry Fuchs, ed., Rockville, MD: Computer Science Press, 1985, pp. 363-376.

[Lipt86] Richard J. Lipton and Daniel Lopresti, "Using Residue Arithmetic to Simplify VLSI Processor Arrays for Dynamic Programming," TR CS-022, Princeton University, January 1986.

[Liu84] Hsi-Ho Liu and King-Sun Fu, "VLSI Arrays for Minimum-Distance Classifications," *VLSI for Pattern Recognition and Image Processing*, King-Sun Fu, ed., Berlin: Springer-Verlag, 1984, pp. 53-59.

[Mold84] D. I. Moldovan, C. I. Wu, and J. A. B. Fortes, "Mapping an arbitrarily large QR algorithm into a fixed size VLSI array," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984.

[Mold86] Dan I. Moldovan and Jose A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Transactions on Computers*, volume C-35, number 1, January 1986, pp. 1-12.

[Rabi78] Lawrence R. Rabinev, Aaron E. Rosenberg, and Stephen E. Levinson, "Considerations in Dynamic Time Warping Algorithms for Discrete Word Recognition," *IEEE Transactions*

on Acoustics, Speech, and Signal Processing, volume ASSP-26, number 6, December 1978, pp. 575-582.

[Wagn74] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the Association for Computing Machinery*, volume 1, 1974, pp. 168-173.