Short Notes

A Parallel Solution to the Approximate String Matching Problem

The approximate string matching problem (ASMP) consists of finding all the occurrences of a string of characters X of length m in another string Y of length n, $m \ll n$, where some errors are allowed in these occurrences. A classical sequential algorithm based on dynamic programming solves the problem in time of $O(mn)$. A parallelisation scheme for this algorithm is proposed, which applies to a very general set of errors, and allows to solve ASMP in time T with N processors, with NT of $O(mn)$, thereby achieving optimal speedup. The scheme is suitable for VLSI implementation on a bounded degree network.

Received October 1988

1. Introduction

The problem of approximate string matching consists of finding all the occurrences of a string $X = x_1 \ldots x_m$, called the *pattern*, in another string $Y = y_1 \ldots y_n$, called the *text*, $n \gg m$, where each occurrence may be affected by a certain number of errors (see, for example refs 5, 9, 13).

The elements (*characters*) of X and Y belong to a fixed alphabet Σ. Various errors have been considered in the literature, such as mismatches of characters between X and Y, missing characters, extra characters or transposed characters in X and Y.[10, 13, 15] Other studies have treated the presence of *don't cares*, that is special characters matching any other character.[1, 11] The following set of errors is very general, and contains all cases considered in the literature. (In particular, errors and don't cares will be treated together.)

e1. *Mismatch* between x_i and y_j. We define a function $w_1: \Sigma^2 \rightarrow Z_0^+$, where $w_1(x_i, y_j)$ is the cost of mismatch, with $w_1(x_i, y_j) = 0$ for $x_i = y_j$ (match).

Note that **e1** includes the don't care conditions in a general form. In fact, if $w_1(x, y) = 0$ for $x \neq y$, then x (or y) is a don't care character for y (or x).

e2. *Extra character in Y* (or missing character in X). We define a function $w_2: \Sigma \rightarrow Z^+$, where $w_2(y_j)$ is the cost of an occurrence of y_j in Y without a corresponding character in X.

e3. *Extra character in X* (or missing character in Y). We define a function $w_3: \Sigma \rightarrow Z^+$, where $w_3(x_i)$ is the cost of an occurrence of x_i in X without a corresponding character in Y.

e4. *Transposition* of two consecutive characters. This error occurs for two index values i, j such that $x_{i-1} = y_j$, $x_i = y_{j-1}$, $x_{i-1} \neq x_i$. We define a function $w_4: \Sigma^2 \rightarrow Z^+$, where $w_4(x_{i-1}, x_i)$ is the cost of transposition.

The approximate string matching problem (ASMP) under errors **e1** to **e4** is then one of finding all the occurrences of X in Y, such that the sum of costs of errors in each occurrence does not exceed a given value $k \in Z_0^+$. In particular we say that *X occurs in position h* if an occurrence of X in Y ends in y_h.

ASPM has been solved under various subsets and/or restrictions of the errors **e1** to **e4**, with algorithms of different complexities. In particular, 'fast' algorithms have been devised for particular subsets of errors,[3, 9] requiring $O(kn)$ sequential time, and $O(k + \log m)$ parallel time on a PRAM model. However, the only sequential algorithm valid for general choices of errors, and for don't care conditions, requires time of $O(mn)$ (see the next section), while no interesting parallel algorithm has been devised for the general case.

In this paper we consider ASPM under errors **e1** to **e4**, and propose a parallelisation scheme for the $O(mn)$ sequential algorithm, suitable for VLSI implementation on a bounded degree network. For any number N of processors ranging from 1 to n, the required processing time T is such that $NT \in O(mn)$, thus achieving optimal speedup.

2. The parallelisation scheme

As already stated, we treat ASMP under errors **e1** to **e4**. Following a classical approach, let us define the $m \times n$ matrix D, where, for all i, j, $D[i,j]$ is the minimum value of the sum of errors between $x_1, \ldots, x_i$ and any subsequence of Y ending in y_j. D is bordered by a row 0, with $D[0,j] = 0$ for $0 \leqslant j \leqslant n$, and a column 0 with $D[i,0] = w_3^i$, where $w_3^i = \Sigma_{h=1,i} w_3(x_h)$ for $1 \leqslant i \leqslant m$. Row 0 corresponds to an empty pattern $X = \varnothing$, where the total cost of errors is zero because X can be recognised in any position of Y. Column 0 corresponds to an empty text $Y = \varnothing$, where w_3^i is the total error induced by the extra characters $x_1, \ldots, x_i$ of X.

The sequential computation of D for $i > 0$, $j > 0$, is based on the dynamic programming relation:

$$
\begin{aligned}
D[i,j] = \min\,(&D[i-1,j-1] + w_1(x_i, y_j),\\
&D[i,j-1] + w_2(y_j)\\
&D[i-1,j] + w_3(x_i),\\
&\text{if } (i > 1, j > 1, x_{i-1} \neq x_i, x_{i-1} = y_j,\\
&\quad x_i = y_{j-1})\\
&\text{then } D[i-2,j-2] + w_4(x_{i-1}, x_i)). \qquad (1)
\end{aligned}
$$

The application of relation (1) iterated for mn times, starting from the border values, and proceeding for increasing values of i and j, leads to the computation of the whole matrix D. If we have $D[i,j] > k$ for some i, j a default value ∞ is assigned to $D[i,j]$. The solution of ASMP is given by the values of the last row of D for which $D[m,j] \leqslant k$.

The purpose of this work is to study the parallel computation of D, when several processors are available. A non trivial use of such processors is based on the following Theorem 1, which applies to ASMP with errors **e1** to **e4** (or any subset or restrictions of such errors). The theorem is an extension of a result of ref. 4.

Referring to errors **e2** and **e3**, let $w_{2m} = \min\,(w_2(y) : y \in \Sigma)$, and $w_{3M} = \max\,(w_3(x) : x \in \Sigma)$.

Theorem 1. $D[i,j]$ can be computed independently of the values $D[i,j']$, $D[i-1, j'-1], \ldots, D[1, j'-i+1]$ (wherever defined), where:

$$
j' \begin{cases} j - iw_{3M}/w_{2M} & \text{for } i \leqslant k/w_{3M} \\ j - \lfloor k/w_{2m} \rfloor - 1 & \text{otherwise.} \end{cases}
$$

Proof. For any pair i, j, denote by the integer value $d = j - i$ the diagonal of D containing all the entries $D[i,j]$ with the same values of $j - i$. Consider the diagonal

$$d = j^* - i,$$

where

$$
j^* = \left\{ \begin{array}{ll} j - \lceil iw_{3M}/w_{2m} \rceil & \text{for } i \leqslant k/w_{3M} \\ j - \lfloor k/w_{2m} \rfloor - 1 & \text{otherwise.} \end{array} \right\} \quad (a)
$$

It is sufficient to show that $D[i,j]$ can be generated through a computational path (i.e. a sequence of values in D) which does not include any entry $D[r,s]$ belonging to d, with $s - r = d$, $r \leqslant i$, $s < j$. In fact, if $D[i,j]$ depended on an element $D[i',j']$ lying on the left of d, then the computational path from $D[i',j']$ to $D[i,j]$ should traverse d in a position i'', j'', hence $D[i,j]$ would also depend on $D[i'',j'']$. We consider two cases.

(1) $i \leqslant k/w_{3M}$. We have from (a):

$$d = j - \lceil iw_{3M}/w_{2m} \rceil - i. \qquad (b)$$

An optimal path leading from $D[r,s]$ to $D[i,j]$ would include exactly $i - r$ diagonal steps with $w_1(x,y) = 0$ (matchings or don't cares), and $j - s - (i - r)$ horizontal steps with $w_2(y) = w_{2m}$, yielding a limit value $D[i,j] = (j - s - i + r)w_{2m} + D[r,s] \geqslant (j - s - i + r)w_{2m}$.

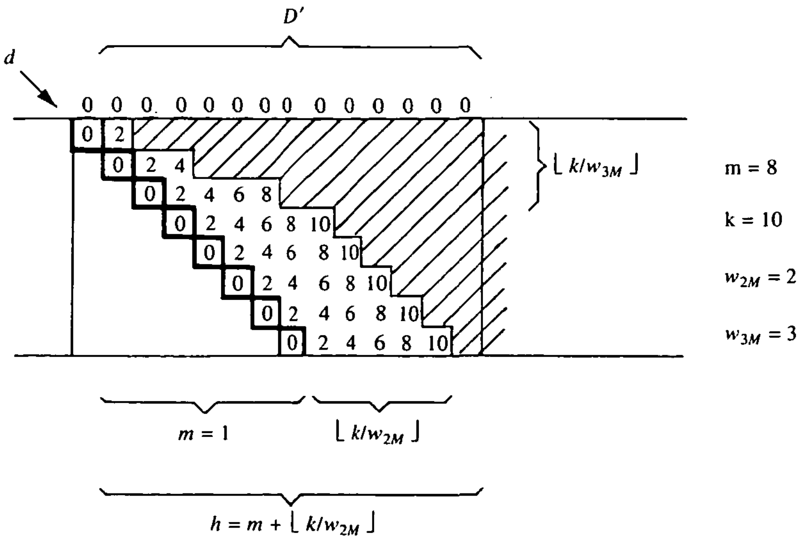

Figure 1. Propagation of the values of a diagonal *d*, in a block *D'*.

On the other hand $D[i,j]$ can be computed from $D[0,j]$ through a vertical path, that is $D[i,j] \leqslant iw_{3M}$. Combining these results, we conclude that $D[i,j]$ must be computed from $D[r,s]$ only if $(j-s-i+r)\,w_{2m} < iw_{3M}$, that is $j-iw_{3M}/w_{2m}-i < s-r = d$, a contradiction with (*b*).

(2) $i > k/w_{3M}$. We have from (*a*):

$$d = j - \lfloor k/w_{2m} \rfloor - 1 - i. \qquad (c)$$

The limit value $D[i,j] \geqslant (j-s-i+r)\,w_{2m}$ is computed from $D[r,s]$ as in Case 1. Combining this result with (*c*) and recalling $d = r-s$, we have: $D[i,j] \geqslant (\lfloor k/w_{2m} \rfloor + 1)\,w_{2m} > k$, hence the default value ∞ should be given to $D[i,j]$. On the other hand, a vertical path from $D[0, j]$ yields a value $D[i,j] \leqslant iw_{3M}$. This value could be $\leqslant k$, thus beating the one deriving from $D[r,s]$, or $>k$, thus generating the same default value ∞. ∎

For example, consider the matrix D shown in Fig. 1, with $m = 8$. Assume that $k = 10$, $w_{2m} = 2$, $w_{3M} = 3$. Let $h = m + \lfloor k/w_{2m} \rfloor$, and consider an $m \times h$ submatrix D', called a block of D. By Theorem 1, the entries on the diagonal d (block framed squares), as well as those to the left of d, do not influence the computation of the entries in the grey zone. The numerical values reported in D' are the minimal values which can be originated from d (in fact, when all values in d are zero).

The main consequence of Theorem 1 is that one could apply the dynamic programming approach to compute D', bordered with an uppermost row and a leftmost column of zeroes, to obtain the correct values for D in the grey zone. If this computation proceeds on D to the right of D', all correct values are found. Therefore, one could perform a computation of D by calculating several blocks D' in parallel, and proceeding to the right until the grey zones have covered the whole matrix. In general, such an approach would be more efficient for small values of h, that is when the bottom left white triangles in D' are small. A common hypothesis[8, 9, 14] is that $k < m$ (obviously, this hypothesis must be verified if all errors have cost 1), leading to the constraint:

$$h < 2m. \qquad (2)$$

Let N be the number of processors, T be the processing time, and R be the number of successive times in which relation (1) is computed in one or more parallel instances. In our analysis we momentarily adopt a PRAM model, where all the processors have access to a common memory, and may read the same data concurrently. (Concurrently write in the same memory location is instead forbidden.) Assuming that the border values of D, the text and the pattern, have been preloaded in memory, and that a computation of relation (1) can be performed in constant time, we have in all cases $T \in O(R)$. The use of a more realistic computational model, based on a bounded degree network, will be discussed in the next section.

We consider several cases for increasing values of N.

Case 1. $N = 1$. This is the sequential case already discussed. We have:

$$N = 1, \quad R = mn, \quad T \in O(mn); \qquad (3.1)$$

$$NR = mn, \quad NT \in O(mn). \qquad (3.2)$$

Case 2. $N = m$. As before D is bordered with row 0 containing all zeroes and column 0 containing $w_3^1, \ldots, w_3^m$. Each processor P_i, $1 \leqslant i \leqslant m$, is used to compute row i of D from left to right, starting at instant i. After step s, the values in the grey zone of Fig. 2 have been computed. The values one position to the right of segment a can be computed, at the next step, as a function of the values on a. We have:

$$N = m, \quad R = n + m, \quad T \in O(n); \qquad (4.1)$$

$$NR = mn + m^2, \quad NT \in O(mn). \qquad (4.2)$$

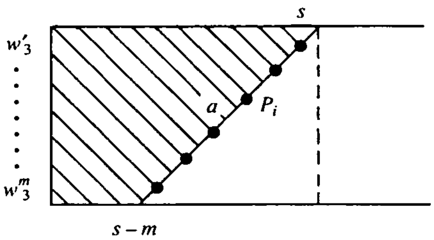

Figure 2. Parallel computation of D with m processors (dots).

Case 3. $N = cm$, with c integer, $2 \leqslant c < n/(2h)$. Matrix D is subdivided in c consecutive submatrices $D_0, \ldots, D_{c-1}$, of m rows and n/c columns, with D_j starting at column $1 + jn/c$, $0 \leqslant j \leqslant c-1$ (Fig. 3*a*). $D_0, \ldots, D_{c-1}$ are bordered by a row 0 of zeroes, and a column 0 containing the correct border conditions $w_3^1, \ldots, w_3^m$ for D_0, and all zeroes for $D_1, \ldots, D_{c-1}$. We assign m processors $P_1^j, \ldots, P_m^j$ to each submatrix D_j. Each processor P_i^j computes the values on the row i of D_j, from left to right, starting at instant i. After n/c steps the values on segment a_j have been computed (Fig. 3*b*), that is the elements in the grey zones of Fig. 3*a* have the correct values by Theorem 1. The computation proceeds for other $m+h$ steps, to cover the remaining white zones bordered by the pairs of segments a_j, b_{j+1}.

Since $2 \leqslant c \leqslant n/(2h)$ we have:

$$N = cm < n/2, \quad R = n/c + m + h < 2n/c, \quad T \in O(n/c). \qquad (5.1)$$

Noting that $NR = (n/c + m + h)\,mc = nm + cm^2 + cmh$, we have:

$$mn + 4m^2 < NR < 3/2mn + nm^2/(2h) < 2mn, \quad NT \in O(mn). \qquad (5.2)$$

Case 4. $N = mn/(2h)$ (for simplicity assume that $n/(2h)$ is integer). As for Case 3, except that the submatrices $D_0, \ldots, D_{n/(2h)}$ are composed of exactly two blocks. We have:

$$N = mn/(2h) < n/2, \quad R = 3h + m, \quad T \in O(h); \qquad (6.1)$$

$$NR = 3/2mn + m^2n/(2h) < 2mn, \quad NT \in O(mn). \qquad (6.2)$$

Under constraint (2), we have the following refinement of relation (6.1):

$$R < 7m, \quad T \in O(m). \qquad (6.1')$$

Case 5. $N = mn/h$. As for Case 4, with the submatrices D_j composed of one block. We have:

$$N = mn/h < n, \quad R = 2h + m, \quad T \in O(h); \qquad (7.1)$$

$$NR = 2mn + m^2n/h < 3mn, \quad NT \in O(mn). \qquad (7.2)$$

In particular, under constraint (2) we have:

$$R < 5m, \quad T \in O(m). \qquad (7.1')$$

Note that, in all cases we have obtained $NT \in O(mn)$ as for the sequential case. This result is generally referred to as achieving *optimal parallel speedup*.

We do not increase the number of processors over the one considered in Case 5 (essentially, we do not consider any value $N > n$), and confine our attention to values of N increasing as multiples of m. A different case of interest, however, arises when a fixed number of processors $N < m$ is available. In this case the following naive method can be established.

Case 6. $1 < N < m$. Matrix D is subdivided in N submatrices $D_0, \ldots, D_{N-1}$ of m rows,

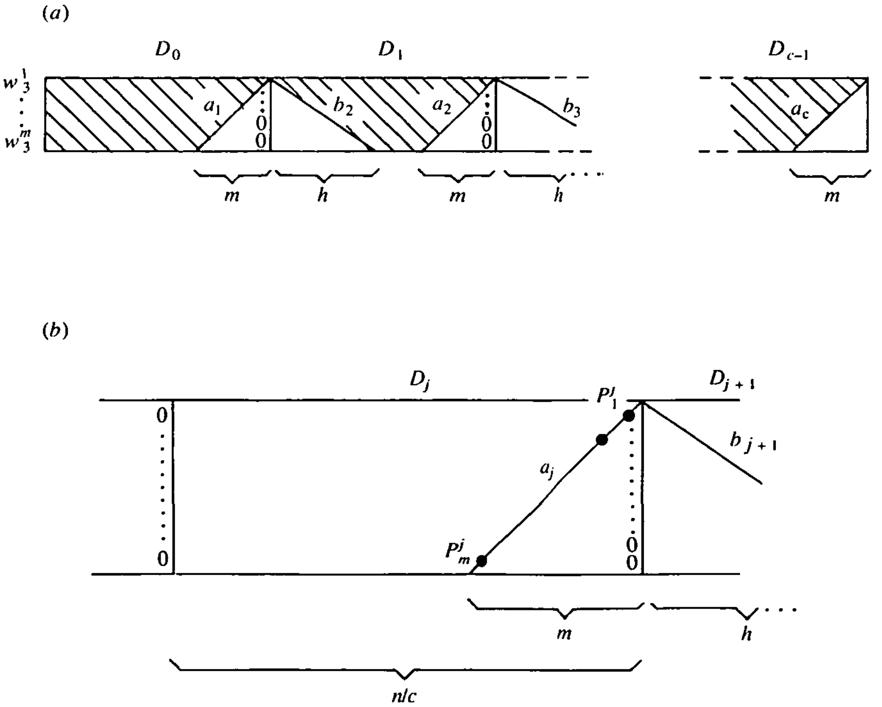

Figure 3. (*a*) Subdivision of D in c submatrices. (*b*) Parallel computation of D_j by the processors (dots) $P_1^j, \ldots, P_m^j$.

where, for $0 \leqslant j \leqslant N-2$, D_j starts at columns $1+jn/N$ and ends at column $(j+1)n/N+h$ (i.e. D_j and D_{j+1} overlap on h columns); and D_{N-1} starts at column $1+(N-1)n/N$ and ends at column n. $D_0, \ldots, D_{N-1}$ are bordered as in Case 3. One processor is assigned to each submatrix, and these are computed in parallel with the sequential algorithm of Case 1, thus requiring $(n/N+h)m$ steps. For fixed N we still have

$$NT \in O(mn). \tag{8.1}$$

In the next section we shall consider the implementation of our parallel algorithms on feasible computing structures. Cases 2–5 will lead to significant considerations, while no comments seems to be necessary for Case 6.

3. Working on a bounded degree network

The parallel scheme of the previous section has been devised for the PRAM model. As well known, this model is technologically unfeasible unless simulated on a bounded degree network, with a multiplicative slowdown of $\log N$ for probabilistic simulation,[6, 12] and at least $\log^2 N/\log\log N$ for deterministic simulation.[2, 7, 16] Moreover, in view of a VLSI implementation we should refer to elementary processors of fixed size. We shall see now that the parallel scheme described for Cases 2–5 can be directly embedded on a bounded degree network, thereby avoiding the simulation slowdown; and that, under proper conditions, the required processors have fixed size.

First consider Case 2. The interconnection network simply consists of one link from each processor P_i to the subsequent processor P_{i+1}, $1 \leqslant i \leqslant m-1$. The algorithm starts at instant 0, when processor P_1 receives the pattern character x_1 from an input channel, and computes the border value $w_3^1 = w_3(x_1)$ (see below on how function w_3 is computed). Then, at each instant i, $1 \leqslant i \leqslant m-1$, processor P_{i+1} receives x_{i+1} from outside and w_3^i from P_i, and computes $w_3^{i+1} = w_3^i + w_3(x_{i+1})$. The text characters $y_1, \ldots, y_n$ are communicated to processor P_1 at instants $1, \ldots, n$, and the output values (in the last row of D) are output by processor P_m.

The computation of all the elements $D[i,j]$, with $i+j = s$, takes place at instant s (see Fig. 2). To compute $D[i,j]$, processor P_i makes use of the values $D[i-1,j-1]$, $D[i-1,j]$, $D[i,j-1]$ and $D[i-2,j-2]$, and must know the characters x_i, y_j, x_{i-1} and y_{j-1} [see relation (1)]. $D[i-1,j-1]$ and $D[i-1,j]$ are passed to P_i by P_{i-1}, as the values computed by P_{i-1} in the previous two steps and stored in this processor up to instant s. $D[i,j-1]$ is known to P_i itself, as the value locally computed in the previous step. $D[i-1,j-2]$ was transferred from P_{i-2} to P_{i-1} at instant $s-2$, for the computation of $D[i-1,j-1]$, and it is stored in P_{i-1} up to instant s, to be communicated to P_i. As far as characters are concerned, x_i is known to P_i, and x_{i-1} is passed to P_i by P_{i-1}. The text characters are input in P_1 and passed down, step by step, from one processor to the next. Each processor stores the last two characters received, such that, when the computation of $D[i,j]$ is to be performed, y_{j-1} and y_j are known to P_i. In total, each processor needs a fixed number registers to store data that will be later used.

Cases 3, 4 and 5 are quite similar to Case 2. The processors assigned to each submartrix D_j work independently of the ones assigned to the other submatrices. Therefore, the processors of each D_j are connected, and operate, as the processors of Case 2, with the only difference that the submatrices $D_j, j \geqslant 1$, have default border conditions.

Note that the text must now be divided in substrings, to be communicated in parallel to the different processors assigned to the first rows of all D_j. The substrings overlap partially, but no character must be sent to more than two processors. The situation is more serious for the pattern, where each character x_i must be initially sent to the N/m processors working on the row of i of all submatrices D_j. On a bounded degree network, this implies an additive initial delay of $\log(N/m)$ using a broadcasting tree for each row of D. The maximum value of this delay is $\log(n/h)$ for Case 5. If the string matching problem must be solved for different instances of the pattern, the broadcasting delay will affect only the first instance, while the successive patterns will be pipelined through the broadcasting tree.

The main problem still to discuss is the computation of the cost functions w_1, w_2, w_3 and w_4 specified in relation (1). In fact in our bounded degree model each processor must compute these functions locally. Some simplifying assumptions are usually made in the literature (see, for example ref. 5), namely: (1) $w_1 = 0$ for match (or don't care with a fixed character), and $w_1 = c$, with c fixed constant, for mismatch. (2) w_2, w_3 and w_4 are fixed constants. Under assumptions (1) and (2) the processors can compute the cost functions in constant time and area. More general functions can be typically computed in constant time by table lookup, but the tables must reside into the processors, and may require a large area if the alphabet is large. In general time and area requirements must be determined case by case.

4. Concluding remarks

In this note we have shown how the approximate string matching problem (ASMP) can be solved in parallel on a bounded degree network of elementary processors. The proposed parallelisation scheme is very simple. It is based on a standard sequential method of dynamic programming, and attains optimal speedup. Faster algorithms have been proposed in the literature,[9] however, these algorithms have been defined for the idealistic PRAM model, and apply only to restricted classes of errors. Our scheme is instead suitable for VLSI implementation, and takes into account a very general set of errors, for which no 'fast' algorithm is known.

A different approach to ASMP involves context dependency. In this case the cost of an error depends on the symbols surrounding the characters under considerations. Particular instances of these errors occur in applications of text editing, pattern matching and speech recognition. (For a general review of context dependent errors and related algorithms see ref. 3.) The techniques proposed here could probably be extended to the parallel solution of some instances of the context dependent problem.

A. A. BERTOSSI, F. LUCCIO, L. PAGLI

Dipartimento di Informatica, Università di Pisa, 56100 Pisa, Italy

and E. LODI

Dipartimento di Matematica, Università di Siena, 53100 Siena, Italy

References

1. K. Abramson, Generalized string matching. *SIAM J. Comp.* **16**, 1039–1051 (1987).
2. H. Alt, T. Hagerup, K. Mehlhorn and F. P. Preparata, Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comp.* **16**, 808–835 (1987).
3. A. A. Bertossi, E. Lodi, F. Luccio and L. Pagli, String matching with context-dependent errors. Int. Workshop on 'Sequences', Positano (1988).
4. A. A. Bertossi, E. Lodi, F. Luccio and L. Pagli, Approximate string matching with context-dependency. *Proc. 25th Allerton Conference*, pp. 46–53 (1987).
5. Z. Galil and R. Giancarlo, Data structures and algorithms for approximate string matching. Research Report, Columbia University, Computer Science Department (1987).
6. A. R. Karlin and E. Upfal, Parallel hashing – an efficient implementation of shared memory. *Proc. 18th Annual ACM Symp. on Theory of Comput.*, pp. 160–168. Berkeley, CA (1986).
7. F. Luccio, A. Pietracaprina and G. Pucci, A new scheme for the deterministic simulation of PRAM's in VLSI. *Algorithmica* (to appear).
8. G. M. Landau and U. Vishkin, Efficient string matching in the presence of errors. *Proc. 26th IEEE FOCS*, pp. 126–136 (1985).
9. G. M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm. *Proc. 18th Annual ACM Symp. on Theory of Comput.*, pp. 220–230. Berkeley, CA (1986).
10. W. J. Masek and M. S. Paterson, A faster algorithm computing string edit distances. *J. of Comput. System Sci.* **20**, 18–31 (1980).
11. R. Pinter, Efficient string matching with don't care patterns. In *Combinatorial Algorithms on Words*, edited A. Apostolico and Z. Galil, pp. 11–29. NATO ASI Series F-12. Springer-Verlag (1985).
12. A. G. Ranade, How to emulate shared memory. *Proc. 28th IEEE FOCS*, pp. 185–194 (1987).
13. D. Sankoff and J. B. Kruskal (eds), *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA (1983).
14. E. Ukkonen, On approximate string matching. *Lecture Notes in Comp. Sci.*, 158, pp. 487–495. Springer-Verlag (1983).
15. E. Ukkonen, Finding approximate patterns in strings. *J. of Algorithms* **6**, 132–137 (1985).
16. E. Upfal and A. Widgerson, How to share memory in a distributed system. *JACM* **34** (1), 116–127 (1987).