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 $n$, $m \leq n$, where some errors are allowed in these occurrences. A classical sequential algorithm based on dynamic programming solves the problem in time $O(mn)$. 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 $NT = O(mn)$, 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, 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 $\Sigma$. 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, 11, 13, 14). Other studies have treated the presence of don't cares, that is special characters matching any other character (11, 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.)

1. **Mismatch between $x_i$ and $y_j$**. We define a function $w_1 : \Sigma \to \mathbb{Z}_+$, where $w_1(x_i, y_j)$ is the cost of mismatch, with $w_1(y_j, x_i) = 0$ for $x_i = y_j$ (match).

   Note that 1 includes the don't care conditions in a general form. In fact, if $w_1(x_i, y_j) = 0$ for $x_i = y_j$, then $x_i$ (or $y_j$) is a don't care character for $y$ (or $x$).

2. **Extra character in $Y$ (or missing character in $X$)**. We define a function $w_2 : \Sigma \times \Sigma \to \mathbb{Z}_+$, where $w_2(x_i, y_j)$ is the cost of an occurrence of $x_i$ in $Y$ without a corresponding character in $X$.

3. **Extra character in $X$ (or missing character in $Y$)**. We define a function $w_3 : \Sigma \to \mathbb{Z}_+$, where $w_3(x_i)$ is the cost of an occurrence of $x_i$ in $X$ without a corresponding character in $Y$.

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

The approximate string matching problem (ASMP) under errors 1 to 4 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 particular we say that $X$ occurs in position $h$ if an occurrence of $X$ in $Y$ ends in $y_h$.

ASMP has been solved under various subsets and/or restrictions of the errors 1 to 4, with algorithms of different complexities. In particular, 'fast' algorithms have been devised for particular subsets of errors, 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 parallel versions of parallel algorithms have been devised for the general case.

In this paper we consider ASMP under errors $e_1$ to $e_4$, 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 $NT = O(mn)$, thus achieving optimal speedup.

2. The parallelisation scheme

As already stated, we treat ASMP under errors $e_1$ to $e_4$. 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 costs between $x_1 \ldots x_i$ and $y_1 \ldots y_j$, and any subsequence of $Y$ ending in $y_1 \ldots y_j$. $D$ is bordered by a row 0, with $D[i, 0] = 0$ for $0 < i < n$, and a column 0 with $D[0, j] = w_2(x_i, y_j)$ for $1 < i < m$. 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 $w_3(y_j)$ is the total error induced by the extra characters $x_1, \ldots, x_m$ of the pattern $X$.

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

$$D[i, j] = \min\{D[i, j-1] + w_1(x_i, y_j),$$

$$D[i, j-1] + w_2(x_i, y_j),$$

$$D[i-1, j] + w_3(y_j),$$

$$D[i, j-2] + w_4(x_{i+1}, x_i),$$

$$D[i-1, j-1] + w_3(y_j),$$

$$D[i, j-2] + w_4(x_{i+1}, x_i),$$

$$(1)$$

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 of 0 is assigned to $D[i, j]$), the solution of ASMP is given by the values of the last row of $D$ for which $D[i, j] \leq k$.

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 $e_1$ to $e_4$ (or any subset or restrictions of such errors).

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

$$D[j-1, j] = \begin{cases} j \cdot tw_m & \text{for } i < k, \\
D[j-1, j-1] - 1 & \text{otherwise.} \end{cases}$$

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 $i-j$. Consider the diagonal

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

where

$$d = \begin{cases} j \cdot tw_m & \text{for } i < k \\
D[j-1, j-1] - 1 & \text{otherwise.} \end{cases}$$

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 $D[i, j]$ belonging to $d$, with $s = r = d, r < i, j$. 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 $r', j'$, hence $D[i', j']$ would also depend on $D[r', j']$.

We consider two cases.

(1) $i < k \cdot w_m$. We have from (a):

$$d = j - i \cdot tw_m = -i.$$**

An optimal path leading from $D[r, j]$ to $D[i, j]$ would include exactly $i-r$ diagonal steps with $w_1(x_i, y_j) = 0$ (matchings or don't cares), and $j-r$ horizontal steps with $w_2(x_i, y_j)$ yielding a limit value $D[i, j] = (j-r) \cdot w_m + D[r, j]$ (see the next section).
On the other hand $D[i,j]$ can be computed from $D[i,j]$ through a vertical path, that is $D[i,j]$ is computed from $D[i,j]$ only if $(j-s-i+r) \leq w_{2m}$, that is $j-i \leq w_{2m}$, a contradiction with (b).

(2) $s > k = w_{2m}$. We have from (a):

$$d = j - k = w_{2m} = i - j$$

The limit value $D[i,j] \geq (j-s-i+r)w_{tm}$ is computed from $D[r,s]$ as in Case 1. Combining this result with (c) and recalling $d = r-s$, we have: $D[i,j] \geq (k + w_{2m})/2$, hence the default value $D[i,j]$ should be given to $D[i,j]$.

- Case 1:
  - $D[r,s]$ yields a value $D[i,j] \leq w_{2m}$. This value could be $\leq k$, but 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 $k = 10$, $w_{2m} = 2$, $w_{2m} = 3$. Let $h = m + k = w_{2m}$, and consider an $m \times m$ matrix $D'$, 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. 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:

$$N = 2m, \quad R = n + m, \quad T \in O(n);$$

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

Case 3. $N = cm$, with $c$ integer, $2 \leq c < n/2$. Matrix $D$ is subdivided in $c$ consecutive submatrices $D_0, \ldots, D_{c-1}$ of $m$ rows and $n/c$ columns, with $D_0$ starting at column $1 + j/n(c)$, $0 \leq j \leq c - 1$ (Fig. 3a). $D_0, \ldots, D_{c-1}$ are bordered by a row of zeroes, and a column 0 containing the correct border conditions $w_1', \ldots, w_c'$ for $D_0$ and all zeroes for $D_1, \ldots, D_{c-1}$. We assign $c$ processors $P_0, P_1, \ldots, P_{c-1}$ to each submatrix $D_i$. Each processor $P_i$ computes the values on the row $i$ of $D_i$ from left to right, starting at instant $i$. After $c$ steps the values on segment $a_i$ have been computed (Fig. 3b), that is the entries in the grey zones of Fig. 3a 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 $a_i, b_i$.

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

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

$$NR = mn + m^2n/c, \quad NT \in O(mn).$$

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 3 (essentially, we do not consider any value $N > n$), but instead 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 4. $N = mn/(2h)$ (for simplicity assume that $n/(2h)$ is integer). As for Case 3, except that the submatrices $D_0, \ldots, D_{mn/(2h)}$ are composed of exactly two blocks. We have:

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

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

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

$$R < 7m, \quad T \in O(m).$$

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

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

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

In particular, under constraint (2) we have:

$$R < 5m, \quad T \in O(m).$$

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

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 patterns, have been preloaded in a common 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:

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

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

**Case 2.** $N = m$. As before $D$ is bordered with row 0 containing all zeroes and column 0 containing $w_1', \ldots, w_c'$. Each processor $P_i$, $1 \leq i \leq m$, 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:

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

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

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

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

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

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

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

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

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

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

In particular, under constraint (2) we have:

$$R < 5m, \quad T \in O(m).$$

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 3 (essentially, we do not consider any value $N > n$), but instead 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,
where, for \( 0 \leq i \leq N - 2 \), \( D_i \) starts at columns \( 1 + j/N \) and ends at column \( (j+1)n/N + h \) (i.e. \( D_i \) and \( D_{i+1} \) 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 \( \mathcal{O}(nm) \).

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, and at least \( \log^2 N / \log \log N \) for deterministic simulation. 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 subprocessor algorithm on feasible computing structures. Cases 2–5 will be later used.

Cases 3, 4 and 5 are quite similar to Case 2. The processors assigned to each submatrix \( D_i \) work independently of the ones assigned to the other submatrices. Therefore, the processors of each \( D_i \) are connected, and operate, as the processors of Case 2, with the only difference that the submatrices \( D_i \) having \( j \geq 1 \), have default border conditions.

Note that the text must now be divided in substeps, to be communicated in parallel to the different processors assigned to the first rows of all \( D_i \). The substeps 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 \( D_i \). 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 \( w_2, w_3 \) 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 refs. 5, 11), namely: (1) \( w_0 = 0 \) 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 processor scheme for the cost functions makes no particular use of constant time and area. More general functions can be typically computed in constant time by table lookup, but the tables must reside inside the processor, 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 (ASM) 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, but these algorithms have been devised 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 ASM involves context dependency. In this case the cost of an error depends on the substrings surrounding the characters under consideration. 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.

References