A Fast Parallel Algorithm to Determine Edit Distance

Thomas R. Mathies

April 1988
CMU-CS-88-130(?)

This research was sponsored by a National Science Foundation fellowship under grant number RCD 8758040.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

A Fast Parallel Algorithm to Determine Edit Distance

Thomas R. Mathies

April 1988

Abstract

We consider the problem of determining in parallel the cost of converting a source string to a destination string by a sequence of insert, delete and transform operations. Each operation has an integer cost in some fixed range. We present an algorithm that runs in $O(\log m \log n)$ time and uses mn processors on a CRCW PRAM, where m and n are the lengths of the strings. The best known sequential algorithm [MP83] runs in time $O(n^2/\log n)$ for strings of length n, indicating that our parallel algorithm (with time-processor product equal to $O(mn \log m \log n)$) is nearly optimal. An instance of the edit distance problem is represented as a graph. The algorithm finds the shortest path in the graph using a path doubling method with efficient pruning due to the structure of the problem.

Extensions of the algorithm solve approximate string matching and local best fit problems. The problem of finding the largest common submatrix of two matrices is considered and shown to be NP-hard. Finally we present an algorithm for exact two-dimensional pattern matching that runs in $O(\log^2 n)$ time using n^2 processors for a $n \times n$ search matrix.

1 Introduction

The edit distance problem is to determine the cost of transforming a source string of characters into a destination string. Given two strings, $\mathbf{a} = a_1 a_2 \ldots a_m$ and $\mathbf{b} = b_1 b_2 \ldots b_n$, over an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_l\}$, three different operations are used to edit string **a** into string **b**. There is a cost associated with each operation:

- Deletion. Delete character a_i appearing in string **a**. The cost is denoted D_{a_i}.
- Insertion. Insert character b_j appearing in string **b**. The cost is denoted I_{b_j}.
- Transformation. Transform character a_i appearing in string **a** to character b_j appearing in string **b**. The cost is denoted T_{a_i,b_j}.

Solving this problem has a number of applications including redisplay algorithms for video editors [Gos81], the study of bird songs, speech recognition and comparing genetic sequences. See [SK83] for an extensive treatment of this problem.

Sequentially this problem can be solved using a dynamic programming technique in $O(mn)$ time for input strings of length m and n. (Sankoff and Kruskal [SK83] cite nine papers in several fields that present this algorithm. Foulser [Fou86] offers two more.) Masek and Paterson [MP83] solve a slightly restricted problem in $O(n^2/\log n)$ time for strings each of length n and indicate that (using one method of determining running time) their algorithm is more efficient for strings with 262,419 characters or more.

A special case of the edit distance problem is to find the length of the *longest common subsequence* (LCS) of two strings. The LCS consists of those characters that are preserved by the least cost sequence of edit operations using the following costs: all deletions and insertions cost one; $T_{\sigma_i,\sigma_i} = 0\ \forall i$; and $T_{\sigma_i,\sigma_j} > 2\ \forall i \neq j$ (in effect disallowing transforming one character into a different character). For example, an LCS of "acbacbba" and "bcabbacc" is "bcbba."[1]

We will compute the *length* of the LCS (also showing that an LCS could simultaneously be determined) and then show how the edit distance can similarly be computed.

An instance of the LCS problem can be depicted as a graph. Given strings $\mathbf{a} = a_1 a_2 \ldots a_m$ [2] and $\mathbf{b} = b_1 b_2 \ldots b_n$ we construct a directed graph $G(V, E)$ as follows:

$$V = \{0, 1, \ldots, m\} \times \{0, 1, \ldots, n\}$$
$$E = \{\langle (i,j), (i,j+1) \rangle \mid i \in \{0, 1, \ldots, m\} \wedge j \in \{0, 1, \ldots, n-1\}\} \bigcup$$

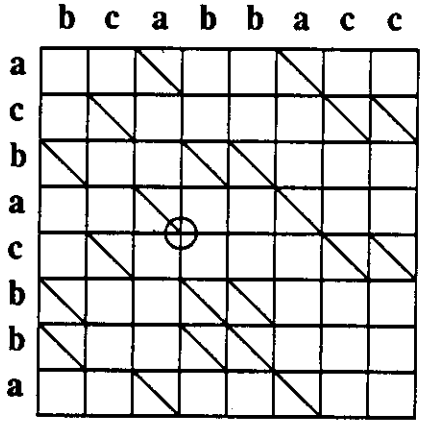
[1] The LCS is not necessarily unique: "cabba" and "babba" are also solutions.

[2] For simplicity of presentation, we assume m is a power of 2.

$$\{\langle (i,j),(i+1,j)\rangle \mid i \in \{0,1,\ldots,m-1\} \wedge j \in \{0,1,\ldots,n\}\} \bigcup$$
$$\{\langle (i-1,j-1),(i,j)\rangle \mid a_i = b_j \wedge i \in \{1,2,\ldots,m\} \wedge j \in \{1,2,\ldots,n\}\}$$

An edge in the graph of the form $\langle (i-1,j-1),(i,j)\rangle$ has a cost of zero. Every other edge has a cost of one.

The graph is basically a grid with some diagonal edges:



Intuitively, horizontal edges (directed to the right) correspond to insertions. Vertical edges (directed downward) correspond to deletions. Diagonal edges (directed downward and to the right) correspond to preserving a character (transforming a character into the same character). Transformations of one character to a different character are not represented as the same result can be achieved by an insertion followed by a deletion. Vertices represent positions in the strings and are referred to by ordered pairs of integers. The upper left vertex is $(0,0)$; the circled vertex above is $(4,3)$.

A path from the origin (upper left vertex) to a vertex in the graph represents a sequence of edit operations that edits a prefix of string **a** to a prefix of string **b**. For example, any path from the origin to the circled vertex represents edit operations that take "acba" to "bca." It should be clear that the least cost path corresponds to the least cost sequence of edit operations. It should also be clear that such a path contains as many diagonal edges as possible. Further, the number of diagonal edges on this path is the length of the LCS.

In this paper, a parallel algorithm to compute to length of the LCS is presented. We then show how the algorithm can be modified to solve the edit distance problem and present two other problems that can be solved by modifying the algorithm: approximate pattern matching [LV86] and local best fit [ES83],

which are defined later. We then investigate extending the problem to two dimensions and show that finding the largest common submatrix of two matrices is NP-hard. Finally we outline a solution to the problem of two-dimensional (exact) pattern matching.

The model we use for parallel computation is the concurrent read, concurrent write (CRCW) parallel random access machine (PRAM). Such a machine has p synchronized processors each with access to a common memory. Simultaneous read and write access to the same memory location is allowed with the stipulation that two processors writing to the same memory location must write the same value.

2 Algorithm to compute the length of the LCS

To rephrase, the problem is: given a directed grid graph with some diagonal edges and all edges directed downward and/or to the right, find the path from the upper left corner to the lower right corner containing the most diagonal edges.

Consider a path P from vertex (i,j) to vertex (p,q) and suppose there are d diagonal edges in this path. This path is defined to be a *critical path* if there is no path from vertex (i,j) to vertex $(p,q-1)$ that contains d diagonal edges. If P is a critical path, we call vertex (p,q) a *critical point* for vertex (i,j).

We use a "row hopping" technique. Initially all rows of vertices are "active"—considered still useful for the computation. The algorithm proceeds in $O(\log m)$ stages. After the kth stage, rows $0, 2^k, 2\cdot 2^k, 3\cdot 2^k, \ldots, m-2^k$ are active and all other rows are inactive. At each vertex (i,j) of an active row i, the critical points to row $i+2^k$ are remembered. This is done by defining $f_k(i,j,d)$ (for $d = 0, 1, \ldots, 2^k$) to be the critical point for the critical path from vertex (i,j) to row $i+2^k$ that contains d diagonal edges, i.e., vertex $(i+2^k, q)$ for some q.[3]

The use of the critical points is indicated by the following lemmas:

lemma 1 *Suppose P is any path in G from vertex (i,j) to vertex $(i+2^k,q)$ and contains d diagonals. Then there exists a path $\hat{P}$ in G from vertex (i,j) to vertex $(i+2^k,q)$ containing d diagonals that passes through $f_{k-1}(i,j,0)$ or $f_{k-1}(i,j,1)$ or ... or $f_{k-1}(i,j,2^{k-1})$.*

[3]The critical paths could be remembered as well in order to produce a longest common subsequence rather than its length. See remark at the end of this section.

Proof. Path P passes through at least one vertex in row $i + 2^{k-1}$. Let this be vertex $(i+2^{k-1}, r)$. Let P' be the portion of P from vertex (i,j) to this vertex and let P'' be the remainder of P. Suppose P' contains t diagonal edges. Let $\hat{P}'$ be the path from vertex (i,j) through critical point $f_{k-1}(i,j,t)$ and then (horizontally) to vertex $(i+2^{k-1}, r)$. This is possible by the definition of critical point and the construction of the graph. Then $\hat{P} = \hat{P}'P''$ satisfies the lemma.

∎

lemma 2 *Suppose P is a critical path from vertex (i,j) to row $i+2^k$ and contains d diagonal edges and passes through vertex $(i+2^{k-1}, p) = f_{k-1}(i,j,s)$ for some s. Then $f_k(i,j,d) = f_{k-1}(i+2^{k-1}, p, d-s)$.*

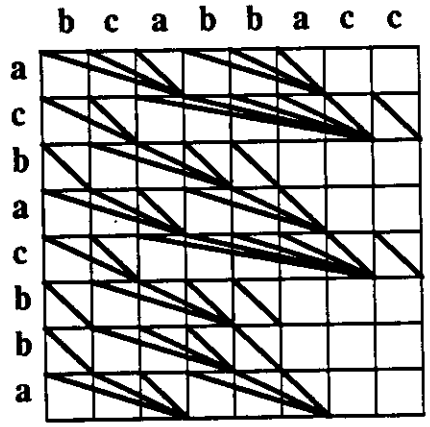
The lemma follows from the definition of critical paths.

lemma 3 *Suppose Q is a critical path from vertex (i,j) to row $i+2^k$ and contains $d+1$ diagonal edges and passes through vertex $f_{k-1}(i,j,t)$ for some t. Then there exists a path P from vertex (i,j) containing d diagonal edges that ends at $f_k(i,j,d)$ and passes through $f_{k-1}(i,j,s)$ for some $s \leq t$.*

Proof. Suppose not. Suppose $s > t$ for any suitable s. Then the paths Q and P must intersect at some vertex since $f_k(i,j,d)$ is to the left of $f_k(i,j,d+1)$. Call this vertex (i',j'). Let Q' be the portion of path Q from vertex (i,j) to (i',j') and let Q'' be the remainder of Q. Define P' and P'' similarly. Then the number of diagonal edges in Q' must be less than the number of such edges in P' otherwise Q' could be used to reach $f_k(i,j,d)$. Then the path $P'Q''$ contains more diagonal edges than path Q. Hence Q is not a critical path containing $d+1$ edges, a contradiction.

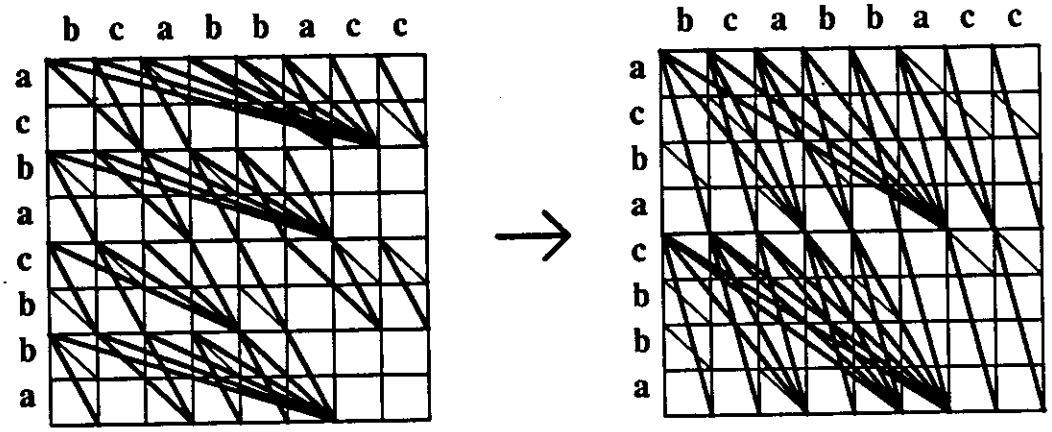
∎

Note $f_k(i,j,0) = (i+2^k, j)$ for any k. Initially each vertex (i,j) has one processor, $p_{i,j}$, assigned to it. It is easy to see that we can find $f_0(i,j,1)$ in $O(\log n)$ time by using recursive doubling to find in each row the first diagonal edge to the right of each vertex. In the diagrams, critical points for vertex (i,j) are indicated by heavy lines from (i,j) to each critical point:



Critical points after initialization

For example, $f_1(3, 0, 1) = f_1(3, 1, 1) = f_1(3, 2, 1) = (4, 3)$.



"Row hopping" from stage k=1 to stage k=2

In each stage of the algorithm, the number of active rows is halved. Critical points are found for vertices in the remaining active rows. These newly found critical points are associated with critical paths that span twice as many rows as those paths associated with critical points found in the previous stage.

After $\log n$ stages, the algorithm terminates having computed the endpoints of the critical paths from vertices in the top row to the bottom row. The length of the LCS is the number of diagonals on the highest critical path (i.e., the one

with the most diagonal edges) from vertex $(0, 0)$ to the bottom row.

We now describe stage k+1 of the algorithm. Let $h = 2^k$. Rows $0, h, 2h, \ldots, m-h$ are active. In this stage, processors $p_{i,j}, p_{i+1,j}, \ldots, p_{i+2h-1,j}$ are assigned to each vertex (i, j) in rows $0, 2h, 4h, \ldots, m-2h$. For each of these vertices (i, j) we do in parallel:

Step 1. Find the highest critical path from vertex (i, j) to row $i+2h$: compute for each $f_k(i, j, s)$, $0 \leq s \leq h$, the sum of

- s (the number of diagonals in a critical path from vertex (i, j) to $f_k(i, j, s)$), and
- the number of diagonals in the highest critical path from $f_k(i, j, s)$ to row $i + 2h$.

The maximum of these sums yields the number of diagonals in the highest critical path from vertex (i, j) to row $i+2h$ by Lemmas 1 and 2. (This maximum can be computed by processors $p_{i,j}, p_{i+1,j}, \ldots, p_{i+2h-1,j}$ in constant time using a bucket sort technique (Rudolph86).)

Step 2. Suppose the highest critical path found in Step 1 from vertex (i, j) to row $i+2h$ contains d diagonal edges and passes through vertex $f_k(i, j, q)$. We wish to compute critical points $f_{k+1}(i, j, t)$, $1 \leq t < d$. First these critical points are partitioned into intervals (sets of consecutive critical points), $I_0, I_1, \ldots, I_q$, such that a critical path to a critical point in I_s passes through critical point $f_k(i, j, s)$. Lemma 3 assures us that such intervals exist. Once these intervals are found, the critical points can be quickly computed.

Let $I_{x\ldots z}$ denote an interval such that a critical path from (i, j) to a critical point in this interval passes through one of the critical points $f_k(i, j, y)$, $x \leq y \leq z$. Let $I_{x\ldots z}.first$ and $I_{x\ldots z}.last$ denote the first and last critical points in interval $I_{x\ldots z}$.

We find the intervals using a divide-and-conquer approach in $O(\log n)$ substages. The initial interval is $I_{0\ldots q}$.

Given an interval $I_{x\ldots z}$ the algorithm proceeds as follows. If $x = z$, then no computation is done on this interval. If interval $I_{x\ldots z}$ is empty, no computation is done. Otherwise, $z - x + 1$ processors [4] are used to find critical point $f_{k+1}(i, j, d')$

[4] Since processors $p_{i,j}, p_{i+1,j}, \ldots, p_{i+2h-1,j}$ are available, we can assign two to each $f_k(i, j, s)$, $0 \leq s \leq q$. Each $f_k(i, j, s)$ might be used to find a path from (i, j) to at most two intervals where computation is needed, namely $I_{x\ldots s}$ and $I_{s\ldots z}$ for some x and z since no computation is done on an interval of the form $I_{s\ldots s}$. Otherwise $f_k(i, j, s)$ might be used to find a path from (i, j) to only one interval, $I_{x\ldots z}$, where $x < s < z$.

(where $d' = \lfloor (I_{x\ldots z}.first + I_{x\ldots z}.last)/2 \rfloor$) using a technique similar to that employed in step 1. (For each $f_k(i,j,s)$, $x \leq s \leq z$, find where in row $i+2h$ a path from (i,j) passing through $f_k(i,j,s)$ and containing d' diagonals ends. The left-most of these endpoints is critical point $f_{k+1}(i,j,d')$.)

Suppose a critical path to $f_{k+1}(i,j,d')$ is found to pass through $f_k(i,j,y)$. Then interval $I_{x\ldots z}$ is split (at critical point $f_{k+1}(i,j,d')$) into intervals $I_{x\ldots y}$ and $I_{y\ldots z}$.

As these intervals are found, all intervals of the form $I_{x\ldots x}$ are merged to form interval I_x. At most two intervals named $I_{x\ldots x}$ are formed in each substage so the time bound is not changed by this merging.

The size of an interval is halved during each substage, thus there are $O(\log n)$ substages. Each substage takes constant time.

To find the remaining critical points in row $i+2h$, we assign one processor to find each $f_{k+1}(i,j,t)$, $1 \leq t < d$. Using binary search and $O(\log n)$ time we find in which interval each of these critical points lies. By Lemma 2, once we know that a critical path to critical point $f_{k+1}(i,j,t)$ passes through $f_k(i,j,s)$ we can look up the value for $f_{k+1}(i,j,t)$.

Thus we halve the number of active rows in $O(\log n)$ time yielding an $O(\log m \log n)$ time algorithm using mn processors.

As indicated earlier, an LCS of the two strings can be determined by remembering diagonal edges of the critical paths. This is achieved by storing $L_{k+1}(i,j,t)$ along with $f_{k+1}(i,j,t)$, where $L_{k+1}(i,j,t)$ is a critical point $f_k(i,j,s)$ found above in Step 2 to lie on a critical path to $f_{k+1}(i,j,t)$.

After the length of the LCS has been found, we can backtrack using L to determine the intermediate critical points lying on the highest critical path from the origin to row m. Among these intermediate critical points (there will be m of them), those at which a diagonal edge ends yield an LCS of the two strings. This follows from Lemma 2.

3 Extending the algorithm to compute edit distance

Suppose that horizontal and vertical edges have costs of zero and diagonal edges have integer costs between -1 and some constant, $-C$, less than zero. Then the algorithm just described can be readily modified to compute the least cost path from vertex $(0,0)$ to vertex (m,n). (Calculate up to $C \cdot h$ critical paths from vertex (i,j) to row $i+h$.) This observation is used to solve the general edit

distance problem.

The edit distance problem can be depicted as a graph in the same manner as the LCS problem. Costs are assigned to the edges as follows: an edge of the form $\langle (i,j),(i,j+1)\rangle$ has cost $I_{b_{j+1}}$; an edge of the form $\langle (i,j),(i+1,j)\rangle$ has cost $D_{a_{i+1}}$; and an edge of the form $\langle (i,j),(i+1,j+1)\rangle$ has cost $T_{a_{i+1},b_{j+1}}$.

A path through the graph will contain exactly one of the (horizontal or diagonal) edges in a given column of edges. Thus we can change the cost of each edge in a given column by the same amount without altering the critical paths. Since all of the horizontal edges in a given column have the same cost, we subtract this cost from each (horizontal and diagonal) edge in this column yielding horizontal edges of cost zero. We do this for each column and likewise for each row and the vertical edges.

This leaves a grid graph with horizontal and vertical edges all of cost zero and diagonal edges with negative costs. (Any diagonals with positive costs can be discarded as they will never be in the least cost path.) This is the case discussed in the above observation[5] and a least cost path through the graph can be quickly found. The edit distance is determined by summing the cost of this path and the sum of the horizontal and vertical edges.[6]

4 Other extensions of the algorithm

Global best fit. String **a** can be viewed as pattern (of length m) and string **b** as text (of length n). We may ask, "what (contiguous) substring of **b** is closest to **a**?" Landau and Vishkin [LV86] present a parallel algorithm that finds all substrings in the text with up to k differences from the pattern. Their algorithm uses m^2+n processors and runs in $O(\log m+k)$ time. The edit distance algorithm presented above yields sufficient information to determine a substring of **b** that is globally closest to pattern **a**. "Globally closest" means at least as close as any other substring of **b** [Sel80].

In the process of computing the edit distance, we have also found the endpoints for all critical paths to the bottom row from each vertex in the top row of the grid graph. Each such path corresponds (by its endpoints) to a substring

[5]This requires that edit costs be integers and bounded by some constant.

[6]Namely, $\sum_{i=1}^{m} D_{a_i} + \sum_{j=1}^{n} I_{b_j}$—just those costs that were subtracted to achieve costs of zero for horizontal and vertical edges.

of **b**. We claim that one of thes substrings is the global best fit to **a**.

To see this, assume the global best fit starts at character j in string **b**. Consider a noncritical point (m, j') and the nearest critical point (m, j'') to the left of (m, j') in the graph. The least cost path to each of these has the same cost (by definition). Let this cost be $-c$. The cost of editing string **a** to substring $b_j \ldots b_{j'}$ is $\sum_{i=1}^{m} D_{a_i} + \sum_{i=j}^{j'} I_{b_i}$. The cost of editing string **a** to substring $b_j \ldots b_{j''}$ is $\sum_{i=1}^{m} D_{a_i} + \sum_{i=j}^{j''} I_{b_i}$, which must be less as $j'' < j'$.

To find the global best fit then, the edit distance is calculated for each critical path from the top row to the bottom row. (This can be readily done in parallel since the $\sum I_{b_i}$ terms can be computed by a prefix sum algorithm.) The critical paths with the least edit distance among all of these yields (by their endpoints) the substrings of **b** globally closest to **a**.

Local best fit. We write $b' \subset \mathbf{b}$ if b' is a substring of **b**. Let $d(s_1, s_2)$ denote the edit distance from string s_1 to string s_2. String $b' \subset \mathbf{b}$ *most resembles* **a** *locally* [ES83] if and only if $d(\ \mathbf{a}\ , b_1) \leq d(\ \mathbf{a}\ , b')$ for all $b_1 \subset b'$ and $d(\ \mathbf{a}\ , b_2) \leq d(\ \mathbf{a}\ , b')$ for all $b' \subset b_2 \subset\ \mathbf{b}$.

To find all such b', again compute the edit distance for each critical path from the top row to the bottom row of the graph. Then for each vertex in the top row, retain only those critical paths from that vertex having the least edit cost. Then do the computation "bottom up": compute the endpoints for critical paths from the *bottom* row to the top row, compute the edit costs for these paths, and for each vertex in the bottom row retain those critical paths having the least edit cost. We claim $b_j \ldots b_{j'}$ most resembles **a** locally if and only if vertex $(0, j)$ has a least cost critical path to (m, j') and vertex (m, j') has a least cost critical path to $(0, j)$.

Clearly the condition is necessary. To see that it is sufficient, suppose $b_j \ldots b_{j'} \subset b_i \ldots b_{i'} \subset\ \mathbf{b}$ with $i < j$ and $j' < i'$. Then a critical path from $(0, i)$ to (m, i') must intersect a critical path from $(0, j)$ to (m, j'). Suppose they intersect at vertex (p, q). Then $d(a_{p+1} \ldots a_m, b_{q+1} \ldots b_{j'}) \leq d(a_{p+1} \ldots a_m, b_{q+1} \ldots b_{i'})$ since the path from $(0, j)$ to (m, j') is a least cost path from vertex $(0, j)$ to row m. A similar argument (taking the "bottom up" view) yields $d(a_1 \ldots a_p, b_j \ldots b_q) \leq d(a_1 \ldots a_p, b_i \ldots b_q)$. Thus no substring containing $b_j \ldots b_{j'}$ is closer to **a**. Likewise we can argue that no substring contained in $b_j \ldots b_{j'}$ is closer to **a**.

5 Largest common submatrix problem

The LCS problem may be extended to two dimensions. Given two matrices and allowing deletions of rows and columns from each matrix, find their largest common submatrix.

This problem is NP-hard and we show this with a reduction from the k-clique problem.

Suppose the graph $G = (V, E)$ is given as an adjacency matrix, M, with 2's on the main diagonal. ($M_{i,i} = 2\ \forall i$; $M_{i,j} = 1$ if $(i, j) \in E$; and $M_{i,j} = 0$ otherwise.) To answer the question, "does G have a clique of size k?", find the largest common submatrix of M and a $k \times k$ matrix with 2s on its main diagonal and 1s everywhere else. The largest common submatrix is of size $k \times k$ iff G has a clique of size k.

6 Two-dimensional pattern matching

The problem of finding an exact match of a pattern in a text string (of length n) has been considered by Galil [Gal84] and Vishkin [Vis85], each of whom present optimal parallel algorithms using $n/\log n$ processors and running in $O(\log n)$ time. The problem of finding in parallel an exact match of a pattern matrix (of size $m \times m$) in a search matrix (of size $n \times n$) can readily be solved using $m^2 n^2$ processors in $O(1)$ time. While considering the largest common submatrix problem, we discovered a more efficient algorithm for this problem.

Baker [Bak78] shows that two-dimensional pattern matching can be reduced to string matching. First the rows of the pattern matrix are matched against the rows of the search matrix The pattern matrix is then represented as a single column of integers and the search matrix is represented as a matrix of integers. The pattern column is then matched against the integer search matrix. Thus the algorithm has two phases.

In the first phase, the two matrices are viewed as a single string: rows of the matrices are catenated and separated by some special symbol to prevent matches from "wrapping around rows." Galil [Gal84] observes that the string matching algorithm in [KMR72] can be parallelized with a time-processor product of $O(n \log^2 n)$. Specifically this can be done with n processors in $O(\log^2 n)$ time for a search string of length n. This yields a set of equivalence classes: each distinct

row of the pattern is identified with an integer. If a substring of the search matrix matches a row of the pattern matrix, then the corresponding integer is stored in a new search matrix (in the position where the matched substring ends in the original search matrix).

In the second phase of the algorithm, the pattern consists of a column of integers associated with the rows of the pattern matrix and is essentially a pattern string. The columns of the new integral search matrix are catenated (with separating symbols) and viewed as text string. One of the above optimal string matching algorithms is then employed to solve the problem.

The first phase uses n^2 processors and $O(\log^2 n)$ time (assuming $n > m$). These resources dominate those used during the second phase ($n^2/\log n$ processors and $O(\log n)$ time).

References

[Bak78] Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, November 1978.

[ES83] Bruce W. Erickson and Peter H. Sellers. Recognition of patterns in genetic sequences. In David Sankoff and Joseph B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, chapter 2, pages 55–91, Addison-Wesley Publishing Company, Inc., 1983.

[Fou86] David E. Foulser. *On Random Strings and Sequence Comparisons*. Technical Report STAN-CS-86-1101, Dept. of Computer Science, Stanford University, February 1986.

[Gal84] Zvi Galil. Optimal parallel algorithms for string matching. In *Sixteenth Annual Symp. on Theory of Computing*, pages 240–247, 1984.

[Gos81] James Gosling. A redisplay algorithm. *SIGPLAN Notices*, 16(6):123–129, 1981.

[KMR72] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays.

In *Fourth Annual Symp. on Theory of Computing*, pages 125–136, 1972.

[LV86] Gad M. Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Eighteenth Annual Symp. on Theory of Computing*, pages 220–230, 1986.

[MP83] William J. Masek and Michael S. Paterson. How to compute string-edit distances quickly. In David Sankoff and Joseph B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, chapter 14, pages 337–349, Addison-Wesley Publishing Company, Inc., 1983.

[Sel80] Peter H. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[SK83] David Sankoff and Joseph B. Kruskal. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison.* Addison-Wesley Publishing Company, Inc., 1983.

[Vis85] Uzi Vishkin. Optimal parallel pattern matching in strings. In Wilfried Brauer, editor, *Automata, Languages and Programming, 12th Colloquium*, pages 497–508, Springer-Verlag, Berlin, 1985.