Parallel Dynamic Programming

Zvi Galil

Department of Computer Science
Columbia University, New York, 10027, USA and
Tel-Aviv University, Tel-Aviv, Israel

Kunsoo Park

Department of Computing
King's College London
Strand, London WC2R 2LS, UK

ABSTRACT

We study the parallel computation of dynamic programming. We consider four important dynamic programming problems which have wide application, and that have been studied extensively in sequential computation: (1) the 1D problem, (2) the gap problem, (3) the parenthesis problem, and (4) the RNA problem. The parenthesis problem has fast parallel algorithms; almost no work has been done for parallelizing the other three.

We present a unifying framework for the parallel computation of dynamic programming. We use two well-known methods, the closure method and the matrix product method, as general paradigms for developing parallel algorithms. Combined with various techniques, they lead to a number of new results. Our main results are optimal sublinear-time algorithms for the 1D, parenthesis, and RNA problems.

Correspondence: Kunsoo Park
Department of Computing
King's College London
Strand, London WC2R 2LS
United Kingdom
Tel: (071) 873-2841
Email: k.park@oak.cc.kcl.ac.uk

1. Introduction

Dynamic programming imposes some order in which the entries of its table are to be computed. The order is of sequential nature, which seems to prevent the development of efficient parallel algorithms. A straightforward parallelization of computing entries requires $O(n)$ time at best.

Dynamic programming is a general problem-solving technique that has been widely used in various fields such as control theory, operations research, biology, and computer science. Sequential computation for dynamic programming has been studied extensively. Due to the difficulty described above, however, little work has been done for parallel dynamic programming. We consider the following four problems.

Problem 1. Given a real-valued function w and $D[0]$, compute

$$D[j] = \min_{0 \le i < j} \{D[i] + w(i,j)\} \quad \text{for } 1 \le j \le n. \tag{1}$$

This problem was called the least weight subsequence problem by Hirschberg and Larmore [12]. It will be called the *1D* problem. Its applications include an optimum paragraph formation problem and the problem of finding a minimum height B-tree.

Problem 2. Given w, w', s_{ij}, and $D[0,0] = 0$, compute

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + s_{ij} \\ \min_{0 \le q < j}\{D[i,q] + w(q,j)\} \\ \min_{0 \le p < i}\{D[p,j] + w'(p,i)\} \end{cases} \quad \text{for } \begin{matrix} 0 \le i \le m \\ 0 \le j \le n \end{matrix}. \tag{2}$$

We assume that m and n are of the same order of magnitude. This is the problem of computing the edit distance when allowing gaps of insertions and deletions [8]. It will be called the *gap* problem. The gap problem arises in molecular biology, geology, and speech recognition.

Problem 3. Given w and $D[i, i+1]$ for $0 \le i < n$, compute

$$D[i,j] = \min_{i < r < j} \{D[i,r] + D[r,j] + w(i,r,j)\} \quad \text{for } 0 \le i < j \le n. \tag{3}$$

We will call this the *parenthesis* problem since it computes the minimum cost of parenthesizing n elements. Its applications include the matrix chain product, the construction of optimal binary search trees, and the maximum perimeter inscribed polygon problem [20].

Problem 4. Given w and $D[i,0]$ and $D[0,j]$ for $0 \le i,j \le n$, compute

$$D[i,j] = \min_{\substack{0 \le p < i \\ 0 \le q < j}} \{D[p,q] + w(p,q,i,j)\} \quad \text{for } 1 \le i,j \le n. \tag{4}$$

This problem has been used to compute the secondary structure of RNA without multiple loops [19]. It will be called the *RNA* problem.

A fifth important dynamic programming problem is the edit distance problem [1,3,9]. In the edit distance problem an entry of its dynamic programming table depends on $O(1)$ entries. Note that an entry of D depends on $O(n)$ entries in Problems 1, 2, 3, and on $O(n^2)$ entries in Problem 4. It is this increased dependency that makes efficient parallel algorithms hard to find for the four problems we consider. Sequential algorithms for the five problems are surveyed in [9].

For the edit distance problem Apostolico et al. [3] and Aggarwal and Park [1] gave $O(\log n \log m)$ time algorithms with $mn/\log m$ CREW processors. The processor bounds of their algorithms differ slightly in the case of CRCW processors. For the parenthesis problem Rytter gave an $O(\log^2 n)$ time algorithm with $n^6/\log n$ CREW processors, which was later improved to $n^6/\log^5 n$ by Viswanathan et al. [18]. Huang et al. [13] also gave an $O(\sqrt{n}\log n)$ time algorithm with $n^{3.5}/\log n$ CREW processors. To the best of our knowledge, no work has explicitly dealt with Problems 1, 2, 4. Greenberg et al. [10] solved a linear recurrence in $O(\log^2 n)$ time with $n^3/\log^2 n$ CREW processors, which solves the 1D problem as a special case in the same complexity.

In this paper we present a unifying framework for the parallel computation of dynamic programming. We use two well-know methods, the *closure* method and the *matrix product* method, as general paradigms for developing parallel algorithms. Combined with various techniques, they lead to a number of new results. We say that a parallel algorithm for a problem is *optimal* if the total number of its operations is asymptotically the same as that of the best known sequential algorithm for the problem. Our main results are optimal sublinear-time algorithms for Problems 1, 3, 4. All our algorithms can be run in $O(\log^2 n)$ time with more processors. Following is the list of our results.

1. The 1D problem:
   1.1. For general $w(i,j)$, $O(\sqrt{n}\log n)$ time in optimal $O(n^2)$ operations.
   1.2. For $w(i,j) = g(j-i)$, $O(\log^2 n)$ time in optimal $O(n^2)$ operations.
2. The gap problem:
   2.1. For general $w(i,j)$, $O(\sqrt{n}\log n)$ time in $O(n^4)$ operations.
   2.2. For $w(i,j)$ and $s_{ij} = +\infty$, $O(\sqrt{n}\log n)$ time in $O(n^2)$ operations.
   2.3. For $w(i,j) = g(j-i)$ and $s_{ij} = s$, $O(\log^2 n)$ time in $O(n^2)$ operations.
   2.4. For $w(i,j) = g(j-i)$ and $s_{ij} = s_i$ or s_j, $O(\sqrt{n}\log n)$ time in $O(n^3)$ operations.
3. The parenthesis problem: $O(n^{3/4}\log n)$ time in optimal $O(n^3)$ operations.
4. The RNA problem: $O(\sqrt{n}\log n)$ time in optimal $O(n^4)$ operations.

Our model of parallel computation is the CREW PRAM [6,14]. The PRAM has a collection of identical processors and a separate collection of memory cells, and any processor can access any memory cell in unit time. The CREW (concurrent read exclusive write) PRAM allows several processors to read the same memory cell at once, but disallows concurrent writes to a cell. The following theorem by Brent [4] is useful in analyzing parallel algorithms, since it allows us to count only the time and total number of operations.

Theorem 1. [4] If a parallel computation can be performed in time t using q operations, then it can be performed in time $t + (q - t)/p$ using p processors.

For example, if a parallel algorithm runs in $O(\sqrt{n} \log n)$ time using $O(n^2)$ operations, then it can be performed in the same $O(\sqrt{n} \log n)$ time using $n^{1.5}/\log n$ processors. Theorem 1 requires the assignment of processors to their tasks, which can be easily done in our algorithms. In our algorithms we also use the routine of finding the minimum of n elements, which takes $O(\log n)$ time with $n/\log n$ CREW processors. Faster CRCW algorithms for the four problems can be obtained by replacing the complexity of the minimum finding by $O(\log \log n)$ time with $n/\log \log n$ CRCW processors [17].

In the following two sections we give two general approaches, the closure method and the matrix product method, for the 1D problem. These approaches are based on well-known methods, but go further to improve the total number of operations. In Sections 4, 5, 6 we use the closure method for Problems 2, 3, 4.

2. The Closure Method

2.1. Closed semirings

Aho et al. [2] introduced a closed semiring $SR = (S, +, \cdot, 0, 1)$, where S is a set of elements, $+$ and $\cdot$ are binary operations on S, and $0, 1$ are additive and multiplicative identities, respectively. A noteworthy property of a closed semiring is that $+$ is idempotent (i.e., $a + a = a$ for $a \in S$). In particular, $SP = (R, \min, +, +\infty, 0)$ is a closed semiring, where R is the set of nonnegative reals including $+\infty$. The semiring SP has an additional property that for all $a \in R$, $\min(0, a) = 0$ ($1 + a = 1$ in the SR notation).

The square matrices of a fixed size over a closed semiring form a closed semiring under the usual definitions of matrix operations. Thus the closure M^* of a matrix M is defined by

$$M^* = I + M + M^2 + M^3 + \cdots$$

Throughout the paper we will use the closed semiring SP. Whenever it is convenient (especially for matrices), we will use the notation $(R, +, \cdot, 0, 1)$ instead of $(R, \min, +, +\infty, 0)$.

2.2. The basic closure method

We build a graph G from Recurrence 1. The vertices of the graph G are $0, 1, \ldots, n$. There are edges (i, j) for all $i < j$, and edge (i, j) has length $w(i, j)$. Let $f(j)$ be the length of the shortest path from 0 to j in graph G.

Lemma 1. Solving Recurrence 1 reduces to the problem of finding the shortest paths in G from 0 to all vertices.

Proof. We prove by induction that $D[j] = D[0] + f(j)$ for $1 \le j \le n$. It is obvious that $D[1] = D[0] + f(1)$, since $f(1) = w(0, 1)$. Suppose that $D[i] = D[0] + f(i)$ for all $i < j$. Since $f(j)$ is the length of the shortest path from 0 to j, $f(j) = \min_{0 \le i < j}\{f(i) + w(i, j)\}$. $D[j] = \min_{0 \le i < j}\{D[i] + w(i, j)\} = D[0] + \min_{0 \le i < j}\{f(i) + w(i, j)\} = D[0] + f(j)$. Therefore, once all $f(j)$ have been computed, we can obtain all $D[j]$ in constant time with n processors. $\square$

It is well known that the shortest path problem is solved by a matrix closure. Let H be a matrix such that

$$\begin{aligned} H(i,i) &= 0 \quad \text{for } 0 \le i \le n, \\ H(i,j) &= w(i,j) \quad \text{for } 0 \le i < j \le n, \\ H(i,j) &= +\infty \quad \text{for } i > j. \end{aligned}$$

We define the squaring of H:

$$H^2(i,j) = \min_{i \le r \le j} \{H(i,r) + H(r,j)\} \quad \text{for } 0 \le i < j \le n.$$

This is the usual definition of matrix multiplication in $(\min, +, +\infty, 0)$ notation. Then $H^k(i,j)$ is the length of the shortest path from i to j with at most k edges, and H^* contains the lengths of shortest paths between all pairs.

Lemma 2. $H^* = H^n$.

Proof. We use $(+, \cdot, 0, 1)$ notation.

$$\begin{aligned} H^* &= I + H + H^2 + \cdots + H^n \text{ (since there is no negative cycle)} \\ &= (I + H)^n \text{ (since the additive operation is idempotent)} \\ &= H^n. \end{aligned}$$

$\square$

Furthermore, $H^* = H^m$ for any $m \ge n$. Thus H^* can be computed in $\lceil \log n \rceil$ multiplications by repeated squaring. Since matrix multiplication in a semiring takes $O(n^3)$ operations, each squaring takes $O(\log n)$ time using $O(n^3)$ operations. Thus H^* takes $O(\log^2 n)$ time using $O(n^3 \log n)$ operations. Note that we solved the all-pair shortest path problem which is harder than the 1D problem (single-source shortest path).

The basic closure method works for computing shortest paths of *general* graphs. For general graphs no better algorithms have been known [14]. For the special graph G we can reduce the total number of operations by the following technique, which will be called the reduction technique.

2.3. Reducing the number of operations

Part 1: In the basic closure method there are redundant computations. That is, for small $j - i$, $H^*(i,j)$ is computed many times. For example, in general graphs the shortest path from 3 to 5 may go through all other vertices, but in graph G it is either $w(3,5)$

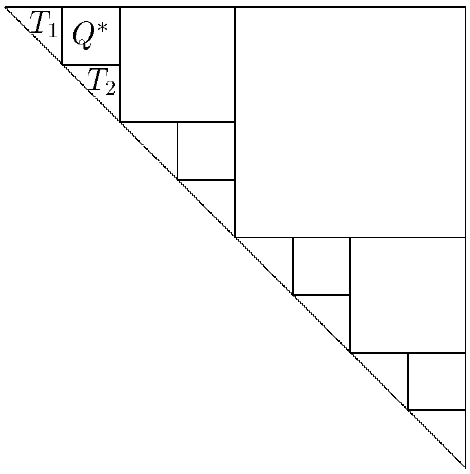

Fig. 1. The square decomposition of H^*

or $w(3,4) + w(4,5)$; $H^*(3,5)$ is computed by one squaring. We can avoid redundant computations by computing H^* from the main diagonal to $(0,n)$.

We use the square decomposition of the upper triangle H^* as shown in Fig. 1, and compute the squares from smallest to largest. We will refer to a $2^k \times 2^k$ square as a square of size 2^k. Since the squares of the same size can be computed simultaneously, all H^* is computed in $\log n$ levels. At level k $(1 \le k \le \log n)$, $n/2^k$ squares of size 2^{k-1} are computed.

Now we show how to compute a square Q^* from two triangles T_1, T_2 adjacent to it (see Fig. 1). Assume that the indices for the new triangle containing Q^*, T_1, T_2 are from 1 to m. Thus

$$\begin{aligned} T_1 &= H^*(i,j) \quad \text{for } 1 \le i < j \le m/2, \\ T_2 &= H^*(i,j) \quad \text{for } m/2 < i < j \le m, \\ Q^* &= H^*(i,j) \quad \text{for } i \le m/2 \text{ and } m/2 < j. \end{aligned}$$

Let

$$Q = H(i,j) \quad \text{for } i \le m/2 \text{ and } m/2 < j.$$

The square Q^* is computed from T_1, T_2, and Q.

Lemma 3. $Q^* = T_1 Q T_2$.

Proof. Let $1 \le i \le m/2$ and $m/2 < j \le m$. Since graph G does not have backward edges, the shortest path from i to j consists of

1. the shortest path from i to p for $i \le p \le m/2$,
2. edge (p,q) for $m/2 < q \le j$, and
3. the shortest path from q to j.

Thus $Q^*(i,j)$ is the minimum of the paths $[(i,p),(p,q),(q,j)]$ for $(i,p) \in T_1$, $(p,q) \in Q$, and $(q,j) \in T_2$. □

By Lemma 3 a square of size 2^k is computed in $O(2^{3k})$ operations. Since there are $n/2^k$ squares at level k, level k requires $O(n2^{2k})$ operations. Since level $k = \log n$ requires $O(n^3)$

operations and the number of operations decreases by a constant factor as k decreases, the total number of operations is $O(n^3)$. Thus we get $O(\log^2 n)$ time using $O(n^3)$ operations.

Part 2: The number of operations can be further reduced by the observation that we do not have to compute H^* as long as we can solve the 1D problem. We compute f in two stages. Note that f is the first row of H^*.

(1) Compute the squares as above, but until level k such that $2^k = \nu$ $(\nu \leq n)$. This computation takes $O(n\nu^2)$ operations since there are n/ν triangles and each triangle takes $O(\nu^3)$ operations.
(2) Divide the first row of H^* into n/ν intervals of length ν, and compute f from leftmost interval to rightmost. For $j \leq \nu$, $f(j) = H^*(0,j)$, which is given from the first triangle. For $l \geq 2$, we compute the l-th interval (i.e., $f(l\nu+1), \ldots, f((l+1)\nu)$) from all previous intervals (i.e., $f(0), \ldots, f(l\nu)$). This is a variant of computing a square from two triangles, but this time we compute a row instead of a square. That is, $f(l\nu+1), \ldots, f((l+1)\nu)$ are computed from $f(0), \ldots, f(l\nu)$, a triangle with indices $l\nu+1$ to $(l+1)\nu$, and H as in Lemma 3. Thus f is computed in n/ν levels, and each level takes $O(n\nu)$ operations; $O(n \log n/\nu)$ time in $O(n^2)$ operations.

Overall, $O(\log^2 \nu + n \log n/\nu))$ time and $O(n\nu^2 + n^2)$ operations are required. The 1D problem requires $\Theta(n^2)$ operations since all $w(i,j)$ for $i < j$ have to be considered. By choosing $\nu = \sqrt{n}$, we have an $O(\sqrt{n} \log n)$ time algorithm in optimal $O(n^2)$ operations. Summarizing all discussions so far, we have the following theorem.

Theorem 2. The 1D problem is solved in $O(\sqrt{n} \log n)$ time using optimal $O(n^2)$ operations.

2.4. Case $w(i,j) = g(j-i)$

When the cost function $w(i,j)$ is a function of the difference $j - i$ [5], we can solve the 1D problem more efficiently. In this case H becomes an upper Toeplitz matrix (i.e., $H(i,j) = H(0, j-i)$ for all $i \leq j$).

Lemma 4. For any $k \geq 1$, H^k is upper Toeplitz.

Proof. By induction on k. It is obvious that H is upper Toeplitz: $H(i,j) = g(j-i)$ for all $i < j$, and $H(i,i) = 0$ for all i. Suppose that H^k for $k \geq 1$ is upper Toeplitz: $H^k(i,j) = g'(j-i)$ for all $i < j$, and $H^k(i,i) = 0$ for all i. Then for $i < j$, $H^{k+1}(i,j) = \min_{i \leq r \leq j}\{g'(j-r) + g(r-i)\}$, which is the same for fixed $j - i$. Since $H^{k+1}(i,i) = 0$ for all i, H^{k+1} is upper Toeplitz. □

By Lemma 4 we need to compute only n elements when squaring H. By the basic closure method, the total number of operations is $O(n^2 \log n)$. Using the square decomposition, H^* can be computed in $O(\log^2 n)$ time using $O(n^2)$ operations. Part 2 of the reduction technique does not help because its second stage takes $O(n^2)$ operations anyway.

In sequential computation Eppstein [5] solved this case by dividing g into piecewise convex and concave functions, and gave an $O(ns\alpha(n/s))$ algorithm (s is the number of piecewise functions), which is $O(n^2)$ in the worst case. Thus our parallel algorithm is optimal.

Theorem 3. When $w(i,j) = g(j-i)$, the 1D problem is solved in $O(\log^2 n)$ time using optimal $O(n^2)$ operations.

3. The Matrix Product Method

Consider the mth order linear homogeneous recurrence

$$x_i = x_{i-1}a_{1,i} + \cdots + x_{i-m}a_{m,i} \quad \text{for } 1 \le i \le n, \tag{5}$$

given $a_{r,s}$ and $x_0, x_{-1}, \ldots, x_{1-m}$. Recurrence 5 can be put in the form

$$u_i = u_{i-1} \cdot A_i,$$

where

$$\begin{aligned} u_0 &= (x_0, x_{-1}, \ldots, x_{1-m}) \\ u_1 &= (x_1, x_0, \ldots, x_{2-m}) \\ &\cdots \\ u_n &= (x_n, \ldots, x_{n-m+1}) \end{aligned}$$

and

$$A_i = \begin{pmatrix} a_{1,i} & 1 & 0 & \cdots & 0 \\ a_{2,i} & 0 & 1 & & 0 \\ \vdots & & & \ddots & \\ a_{m-1,i} & 0 & 0 & & 1 \\ a_{m,i} & 0 & 0 & \cdots & 0 \end{pmatrix}.$$

Observe that in the semiring SP Recurrence 5 solves the 1D problem; $m = n$, $x_i = D[i]$, and $a_{r,i} = w(i-r, i)$. For simplicity we assume n is a power of two. In this section we use $(+, \cdot, 0, 1)$ notation unless otherwise specified.

As far as the results we obtained are concerned, this method is weaker than or equivalent to the closure method. However, the matrix product method itself is an interesting way of developing parallel algorithms for dynamic programming as shown below, and might warrant further study.

Let $B = A_1 \cdots A_n$. Then $u_n = u_0 \cdot B$. We compute B by a complete binary tree. As we multiply A_i's, the number of columns which have nontrivial values is increasing; a matrix at level k ($1 \le k \le \log n$) has 2^k such columns. The number of matrix products at level k is $n/2^k$, and the number of operations for each matrix product at level k is $O(n2^{2k})$. Thus the number of operations at level k is $O(n^2 2^k)$. Since the number of operations decreases by a constant factor as level k goes down, computing B requires $O(n^3)$ operations. Thus we get $O(\log^2 n)$ time using $O(n^3)$ operations [10].

We can also improve the total number of operations. Since $a_{r,i} = 0$ for $i - r < 0$ in the 1D problem, the leftmost matrix at each level has only one row of nontrivial values. Compute B from the bottom until the level k such that $2^k = \nu$. Then there remain n/ν matrices. The number of operations up to that level is $O(n^2\nu)$. The remaining computation is done from left to right in $O(n \log n/\nu)$ time with $O(n^3/\nu)$ operations. Thus $O(\log^2 \nu + n \log n/\nu)$ time and $O(n^2\nu + n^3/\nu)$ operations are required. By choosing $\nu = \sqrt{n}$, we have an $O(\sqrt{n} \log n)$ time algorithm in $O(n^{2.5})$ operations.

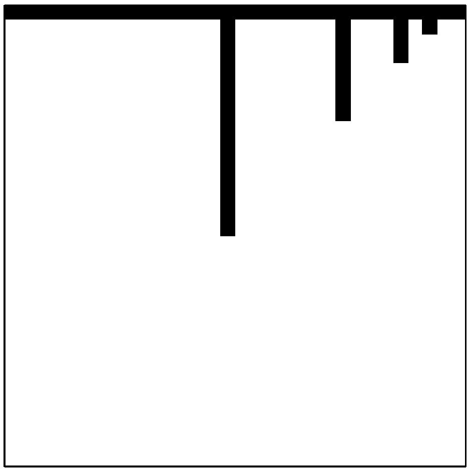

Fig. 2. Row 1 and icicles of B

3.1. Case $w(i,j) = g(j-i)$

If $w(i,j) = g(j-i)$, then $a_{r,1} = a_{r,2} = \cdots = a_{r,n}$ unless it is 0. We can make $a_{r,i} = a_r$ for all r and i by setting $x_{-1} = \cdots = x_{1-m} = 0$ (additive identity). Thus $A_i = A$ for all i. Now we can compute $B = A^n$ by successive squarings, but we still have $O(n^3)$ operations because the last squaring itself requires $O(n^3)$ operations.

Lemma 5. [7] Let $v_{1-n} = (0, \ldots, 0, 1)$ and $v_{i+1} = A \cdot v_i$ for $i \geq 1-n$.
Then $A^n = [v_n, v_{n-1}, \ldots, v_1]$.

Thus the last column of A^n is $v_1 = (a_1, \ldots, a_n)$, and A^n is computed by the following rules (also in [11]):

1. For $i < n$ and $j < n$, $A^n(i,j) = a_i \cdot A^n(1, j+1) + A^n(i+1, j+1)$.
2. For $j < n$, $A^n(n,j) = a_n \cdot A^n(1, j+1)$.

These rules give a dynamic programming recurrence for computing $A^n = B$; B can be computed from the last column to the first column in $O(n^2)$ operations.

We show that the recurrence can be solved optimally in $O(\log^2 n)$ time. Now we think in terms of $(\min, +, +\infty, 0)$ notation; i.e., shortest paths. Rules 1 and 2 imply that an element $(1,j)$ in row 1 has edges to all elements in column $j-1$, but an element (i,j) in other rows ($j > 1$) has only one edge to $(i-1, j-1)$. Note that the solution of the 1D problem is the first row of B. We compute the following elements of B: row 1 and *icicles* (Fig. 2). An icicle is a subset of a column, and icicles are at columns $n - 2^k$ ($0 \leq k < \log n$). The icicle at column $n - 2^k$ consists of 2^k elements $B(1, n-2^k), \ldots, B(2^k, n-2^k)$.

Lemma 6. Let $f(p,q)$ be the length of the shortest path from $(1,q)$ to $(1,p)$, $p < q$. Then $f(p,q) = f(n-(q-p), n)$.

Proof. Since the length of an edge to (i,j) is either a_i or 0 by Rules 1 and 2, the edge length does not depend on column number. Therefore, two paths such that one is shifted horizontally from the other have the same length. □

Now we get Theorem 3 by the matrix product method as follows. We compute the icicles and the parts of row 1 between icicles alternately from smallest to largest.

1. Computing $B(i,j)$ in the icicle at column j: $B(i,j)$ is the minimum of $\min_{j<q\leq n} \{B(1,q)+a_{(q-1)-(j-i)}\}$ and $B(n-(j-i),n)$. Thus the icicle of length 2^k is computed in $O(n2^k)$ operations; $O(n^2)$ operations for all icicles.
2. Computing $B(1,p)$: Let q be the column of the first icicle to the right of $(1,p)$. Then $B(1,p) = \min_{1\leq i\leq q-p+1}\{B(i,q) + f(p,q-i+1)\}$. The part of row 1 between icicles at columns $n-2^k$ and $n-2^{k+1}$ requires $O(2^{2k})$ operations; $O(n^2)$ operations for the entire row 1.

4. The RNA Problem

In the RNA problem the cost function $w(p,q,i,j)$ involves three different functions depending on p,q,i,j [19]. In sequential computation we can divide Recurrence 4 into three recurrences by the functions, and compute $D[i,j]$ as the minimum of the three recurrences. In parallel computation, however, such a division imposes an order on $D[i,j]$ so that linear time cannot be avoided. Thus the three functions should be considered together in w.

The following method for the RNA problem is essentially a two-dimensional version of the closure method for the 1D problem in Section 2. Let

$$\begin{aligned} H(i,j,i,j) &= 0 \quad \text{for } 0 \leq i,j \leq n, \\ H(p,q,i,j) &= w(p,q,i,j) \quad \text{for } p<i \text{ and } q<j, \\ H(p,q,i,j) &= +\infty \quad \text{for all others.} \end{aligned}$$

We define the squaring of H: for $p \leq i$ and $q \leq j$

$$H^2(p,q,i,j) = \min_{\substack{p\leq r\leq i \\ q\leq s\leq j}} \{H(p,q,r,s) + H(r,s,i,j)\}.$$

Since a squaring takes $O(n^6)$ operations, H^* computed by $\log n$ squarings takes $O(\log^2 n)$ time using $O(n^6 \log n)$ operations. We give a two-dimensional version of the reduction technique in Subsection 2.3. We use the decomposition of a square into four subsquares in the D matrix (see Fig. 3), and compute H^* from the smallest squares to the largest. Since H^* of the squares of the same size can be computed simultaneously, all H^* is computed in $\log n$ levels. At level k, H^* of $(n/2^k)^2$ squares of size 2^k are computed.

We compute H^* of a square Q from H^* of four subsquares Q_1, Q_2, Q_3, Q_4. We first compute H^* of $Q_1 \cup Q_2$ and H^* of $Q_3 \cup Q_4$, and then compute H^* of Q. We show the computation of H^* of $Q_1 \cup Q_2$; other computations are similar. Let

$$\begin{aligned} G_1 &= H^*(p,q,i,j) \quad \text{for } (p,q),(i,j) \in Q_1, \\ G_2 &= H^*(p,q,i,j) \quad \text{for } (p,q),(i,j) \in Q_2, \\ G_{12} &= H^*(p,q,i,j) \quad \text{for } (p,q) \in Q_1 \text{ and } (i,j) \in Q_2, \\ H_{12} &= H(p,q,i,j) \quad \text{for } (p,q) \in Q_1 \text{ and } (i,j) \in Q_2, \end{aligned}$$

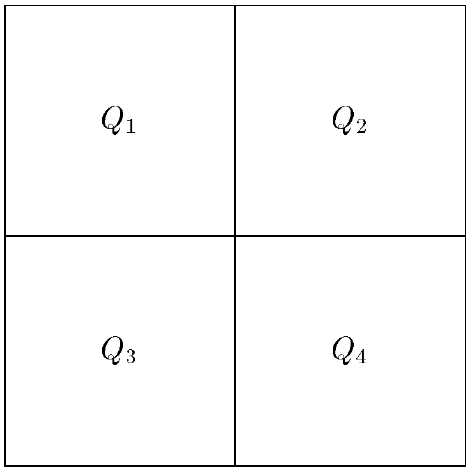

Fig. 3. The decomposition of square Q

Lemma 7. $G_{12} = G_1 H_{12} G_2$.

Proof. Let $(p, q) \in Q_1$ and $(i, j) \in Q_2$. The shortest path from (p, q) to (i, j) consists of

1. the shortest path from (p, q) to (p', q') for $(p', q') \in Q_1$,
2. an edge from (p', q') to (i', j') for $(i', j') \in Q_2$, and
3. the shortest path from (i', j') to (i, j).

Thus $G_{12}(p, q, i, j)$ is the minimum of $G_1(p, q, p', q') + H_{12}(p', q', i', j') + G_2(i', j', i, j)$ for all $(p', q') \in Q_1$ and $(i', j') \in Q_2$. $\square$

Thus H^* of a square of size 2^k is computed in $O(2^{6k})$ operations. H^k of squares at level k requires $O(n^2 2^{4k})$ operations. The total number of operations is $O(n^6)$. We further reduce the number of operations as in Section 2. Let $f(i, j)$ be the length of the shortest path from $(0, 0)$ to (i, j).

(1) Compute H^* of the squares until level k such that $2^k = \nu$. This computation takes $O(n^2 \nu^4)$ operations since there are n^2/ν^2 squares and H^* of each square takes $O(\nu^6)$ operations.

(2) Compute f of squares of size ν by backward diagonals ($i + j$ is constant). That is, f of squares with the same backward diagonal is computed simultaneously. Computing f of each square takes $O(n^2 \nu^2)$ operations. Since there are n^2/ν^2 squares, the total number of operations is $O(n^4)$. Time is $O(n \log n/\nu)$, because there are $O(n/\nu)$ diagonals.

Overall, $O(\log^2 \nu + n \log n/\nu))$ time and $O(n^2 \nu^4 + n^4)$ operations are required. The RNA problem requires $\Theta(n^4)$ operations since all $w(p, q, i, j)$ for $p < i$ and $q < j$ have to be considered. For $\nu = \sqrt{n}$, we have an $O(\sqrt{n} \log n)$ time algorithm in optimal $O(n^4)$ operations.

Theorem 4. The RNA problem is solved in $O(\sqrt{n} \log n)$ time using optimal $O(n^4)$ operations.

5. The Parenthesis Problem

The parenthesis problem is different from Problems 1, 2, 4 because this problem is to find a *binary tree* of minimum cost, while the three problems are to find a *path* of minimum length. First, we give a clean version of Rytter's algorithm [16], and then improve the total number of operations.

Let $f(i,j)$ be the cost of the optimal binary tree rooted at (i,j) (i.e., $f(i,j) = D[i,j]$). We define a *partial tree* T to be a tree rooted at some vertex (i,j) with one of its non-leaf nodes (p,q) treated as a leaf (i.e., the subtree rooted at (p,q) is deleted). We say that (p,q) is the gap of T (nothing to do with the gap problem). Let $H(p,j,i,j)$, $p > i$, be the cost of the partial tree rooted at (i,j) with gap (p,j) such that (p,j) is a child of (i,j). Then $H(p,j,i,j) = f(i,p) + w(i,p,j)$. Define $H(i,q,i,j)$, $q < j$, similarly. Then $H(i,q,i,j) = f(q,j) + w(i,q,j)$. $H(p,q,i,j) = +\infty$ for all others. Let $H^*(p,q,i,j)$ be the cost of the optimal partial tree rooted at (i,j) with gap (p,q).

A main difference from the previous problems (path problems) is that H is not available initially because of f in its definition. In path problems we just compute H^* from H, then f can be read off from H^*. In the parenthesis problem we must compute f together with H^*. Let H' and f' be intermediate values of H^* and f, respectively. Initially, $f(i,i+1)$ are given for all i, $f(i,j) = +\infty$ for all others, and $H'(p,q,i,j) = +\infty$ for all p,q,i,j.

Since computing H' requires f by the definition of H above, H' requires H' (squaring), and f requires H' and f, we need the following three steps.

Activate: for all $0 \le i < p < j \le n$,

$$H'(i,p,i,j) = f'(p,j) + w(i,p,j),$$
$$H'(p,j,i,j) = f'(i,p) + w(i,p,j).$$

Square: for all $0 \le i \le p < q \le j \le n$,

$$H'(p,q,i,j) = \min_{\substack{i \le r \le p \\ q \le s \le j}} \{H'(p,q,r,s) + H'(r,s,i,j)\}.$$

Pebble: for all $0 \le i \le p < q \le j \le n$,

$$f'(i,j) = \min_{i \le p < q \le j} \{f'(p,q) + H'(p,q,i,j)\}.$$

Rytter showed that $f' = f$ after $\log n$ iterations of (activate, square, square, pebble). Since the square step takes $O(n^6)$ operations, computing f takes $O(\log^2 n)$ time using $O(n^6 \log n)$ operations.

Now we reduce the number of operations by a variation of the reduction technique. We use the decomposition of the upper triangular matrix D into (forward) diagonal strips (Fig. 4).

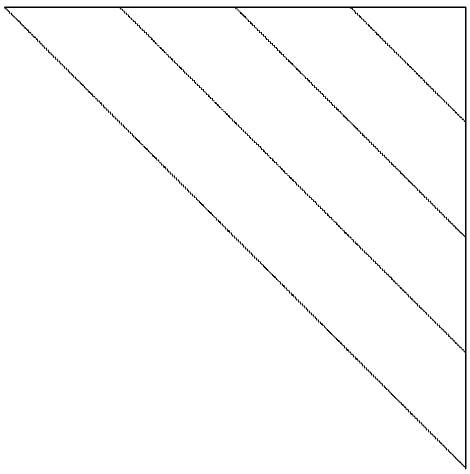

Fig. 4. Diagonal strips of D

Step 1: Run Rytter's algorithm to compute f of the first strip of width ν. In the square step there are $O(n\nu)$ (i,j)'s and for each (i,j) there are $O(\nu^2)$ (p,q)'s, so the square step takes $O(n\nu^5)$ operations. Overall $O(n\nu^5 \log \nu)$ operations are required for the first strip.

Step 2: Compute H^* of strips from narrowest to widest until the width is ν. In this computation $H(p,j,i,j) = f(i,p) + w(i,p,j)$ and $H(i,q,i,j) = f(q,j) + w(i,q,j)$ are available since $p - i \leq \nu$ and $j - q \leq \nu$. Thus computing H^* is similar to that of the RNA problem. Let S be a strip of width ν, and let S_1 and S_2 be the strips of width $\nu/2$ such that they compose S, and the diagonals of S_1 are less than the diagonals of S_2. Let G_S, G_1, and G_2 be H^* of S, S_1, and S_2, respectively, and H_S be H in strip S. We define the multiplication of matrices as defined in the square step.

Lemma 8. $G_S = G_1 H_S G_2$.

Proof. Let $(p,q) \in S_1$ and $(i,j) \in S_2$. The optimal binary tree rooted at (i,j) with gap (p,q) consists of

1. the optimal binary tree rooted at (i,j) with gap (i',j') for $(i',j') \in S_2$,
2. the tree rooted at (i',j') with gap (p',j') for $(p',j') \in S_1$ or the tree rooted at (i',j') with gap (i',q') for $(i',q') \in S_1$,
3. the optimal binary tree rooted at (p',j') (or (i',q')) with gap (p,q).

Thus $G_S(p,q,i,j)$ is the minimum of $G_1(p,q,p',j') + H_S(p',j',i',j') + G_2(i',j',i,j)$ and $G_1(p,q,i',q') + H_S(i',q',i',j') + G_2(i',j',i,j)$ for all $(p',j'),(i',q') \in S_1$ and $(i',j') \in S_2$. □

Computing H^* of a strip of width ν takes $O(n\nu^5)$ operations. Since there are n/ν strips of width ν, $O(n^2\nu^4)$ operations are required for all strips of width ν. Since the number of operations decreases by a constant factor as width decreases, the total number of operations is $O(n^2\nu^4)$.

Step 3: Compute f by strips of width ν. This is not a special case of Step 2; if it were, we would have needed H, which is not available for strips of width larger than ν. We compute f by the following.

Lemma 9. Let S be the strip of width ν which we compute f of, and S' be the strip of width $l\nu$ of which f is already computed (i.e., all previous strips). Then f of S is computed by the following steps. For $(i,j) \in S$,

$$f'(i,j) = \min_{j-l\nu \le r \le i+l\nu} \{f(i,r) + f(r,j) + w(i,r,j)\}.$$

And for $(i,j),(p,q) \in S$,

$$f(i,j) = \min_{i \le p,q \le j} \{f'(p,q) + H^*(p,q,i,j)\}.$$

Proof. If the optimal tree rooted at (i,j) has both children in S', then $f(i,j) = f'(i,j)$. Otherwise, we follow the child x that is in S successively until x has both children in S'. Let $x = (p,q)$ be the descendant of (i,j) that is in S, and that has both children in S'. Then $f(i,j) = f(p,q) + H^*(p,q,i,j)$. □

Computing f in Step 3 takes $O(n \log n/\nu)$ time in $O(n^2\nu + n\nu^3)$ operations. Overall, $O(\log^2 \nu + n \log n/\nu))$ time and $O(n^2\nu^4 + n\nu^5 \log \nu)$ operations are required. By choosing $\nu = n^{1/4}$, we have an $O(n^{3/4} \log n)$ time algorithm in optimal $O(n^3)$ operations. Note that for $\nu = \sqrt{n}$ we get $O(\sqrt{n} \log n)$ time in $O(n^4)$ operations, which is the same complexity as [13].

Theorem 5. The parenthesis problem is solved in $O(n^{3/4} \log n)$ time using optimal $O(n^3)$ operations.

6. The Gap Problem

The gap problem is solved by the result of Section 4 in $O(\sqrt{n} \log n)$ time using $O(n^4)$ operations. Though the gap problem is easier than the RNA problem, we were unable to get a better algorithm. We consider the following special cases, and show how improved bounds are obtained. (These cases of the gap problem are unrealistic in its current applications because $s_{ij} = 0$ if the ith character of a string matches the jth character of the other string, and $s_{ij} > 0$ otherwise.)

In the gap problem the edge length from (i,q) to (i,j) for $q < j$ is the same for all i, which is $w(q,j)$. Similarly, the edge length from (p,j) to (i,j) for $p < i$ is $w'(p,i)$ for all j. Let $f(j)$ be the length of the shortest path from $(i,0)$ to (i,j), and $f'(i)$ be the length of the shortest path from $(0,j)$ to (i,j).

6.1. Case $s_{ij} = +\infty$

Lemma 10. $D[i,j] = f'(i) + f(j)$.

Proof. Since there is no diagonal edge, all paths consist of vertical and horizontal edges. Since the length of a vertical edge does not depend on column number, we can move all vertical edges in the optimal path to column j. Similarly, we can move all horizontal edges to row 0. Thus the shortest path from $(0,0)$ to (i,j) consists of the shortest path from $(0,0)$ to $(0,j)$ and the shortest path from $(0,j)$ to (i,j). □

We solve two instances of the 1D problem and then compute D by Lemma 10. The time/processor complexity is the same as that of the 1D problem.

6.2. Case $w(i,j) = g(j-i), w'(i,j) = g'(j-i)$ and $s_{ij} = s$

Lemma 11. For $i \le j$, $D[i,j] = \min_{0 \le r \le i}\{rs + f'(i-r) + f(j-r)\}$. For $i > j$, $D[i,j] = \min_{0 \le r \le j}\{rs + f'(i-r) + f(j-r)\}$.

Proof. Since $w(i,j) = g(j-i)$, horizontal edges can shift horizontally without changing their lengths, in addition to vertical shifts. Similarly, vertical edges can shift vertically. Since the substitution cost is a constant, the places where substitutions occur do not affect the total length; what matters is the number of substitutions. Once the number of substitutions r is fixed for $D[i,j]$, we can move the substitutions to the places from $(0,0)$ to (r,r). Then the shortest path for the rest is $f'(i-r) + f(j-r)$ as in Lemma 10. □

We again solve two instances of the 1D problem with $w(i,j) = g(j-i)$. Now we compute each diagonal of D separately. Note that $D[i,j] = \min\{s + D[i-1,j-1], f'(i) + f(j)\}$, and this is the prefix computation [6,14]. Suppose we compute $D[0,0], D[1,1], \ldots, D[n,n]$ and $n+1$ is a power of 2 for simplicity. Let $v_i = f'(i) + f(i)$ for $0 \le i \le n$ and $k = 1$ initially.

1. For each odd i, compute $x_{(i-1)/2} = \min\{ks + v_{i-1}, v_i\}$.
2. $k \leftarrow 2k$. Recursively do the prefix computation with array x and store the output into array y.
3. For each odd i, output $y_{(i-1)/2}$. For each even $i \ge 2$, output $\min\{ks + y_{i/2-1}, v_i\}$ (for $i = 0$, output v_0).

The computation above takes $O(\log n)$ time in $O(n)$ operations. Thus the computation of f, f' dominates; $O(\log^2 n)$ time in $O(n^2)$ operations.

6.3. Case $w(i,j) = g(j-i), w'(i,j) = g'(j-i)$ and $s_{ij} = s_i$ or s_j

We compute $H^*(p,q,i,j)$ as in the RNA problem. If $s_{ij} = s_i$, then $H^*(p,q,i,j) = H^*(p,q',i,j')$ for $q'-q = j'-j$. Therefore, we have only to compute $H^*(p,0,i,j)$. Computing $H^*(p,0,i,j)$ by $\log n$ squarings takes $O(\log^2 n)$ time using $O(n^5 \log n)$ operations. By Part 1 of the operation reduction technique, $H^*(p,0,i,j)$ requires $O(n^5)$ operations. In Part 2 we compute $H^*(p,0,i,j)$ for squares until their size is ν. Since there are only n/ν squares, this computation takes $O(\log^2 \nu + n \log n/\nu)$ time and $O(n\nu^4 + n^2\nu^2)$ operations. Thus we get $O(\sqrt{n}\log n)$ time in $O(n^3)$ operations.

7. Conclusions

We have presented two general methods for parallel dynamic programming, and a number of new results. There is yet another method called *program transformation*: If we transform Recurrence 1 so that $D[2j]$ depends on $D[j], D[j-1], \ldots, D[0]$, then we can compute D in $O(\log^2 n)$ time. Pettorossi and Burstall [15] derived such transformation, but for the 1D problem the computation based on the transformation turns out to be equivalent to that of the matrix product method.

There are many open problems. Among them are

1. All our algorithms can be run in $O(\log^2 n)$ time with more processors. Can we reduce the processor bounds while staying in NC?
2. Is there a better algorithm for the gap problem?
3. In sequential computation the convexity and concavity of the cost function w were used to develop improved algorithms. Can we take advantage of convexity or concavity in parallel computation?

References

1. Aggarwal, A., and Park, J. Parallel searching in multidimensional monotone arrays. Manuscript.
2. Aho. A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
3. Apostolico, A., Atallah, M. J., Larmore, L. L., and McFaddin, S. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.* **19** (1990), 968–988.
4. Brent, R. P. The parallel evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.* **21** (1974), 201–206.
5. Eppstein, D. Sequence comparison with mixed convex and concave costs. *J. Algorithms* **11** (1990), 85–101.
6. Eppstein, D., and Galil, Z. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.* **3** (1988), 233–283.
7. Fiduccia, C. M. An efficient formula for linear recurrence. *SIAM J. Comput.* **14** (1985), 106–112.
8. Galil, Z., and Giancarlo, R. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science* **64** (1989), 107–118.
9. Galil, Z., and Park, K. Dynamic programming with convexity, concavity, and sparsity. *Theoretical Computer Science* to appear.
10. Greenberg, A. C., Ladner, R. E., Paterson, M. S., Galil, Z. Efficient parallel algorithms for linear recurrence computation. *Inform. Process. Lett.* **15** (1982), 31–35.
11. Gries, D., and Levin, G. Computing fibonacci numbers (and similarly defined functions) in log time. *Inform. Process. Lett.* **11** (1980), 68–69.

12. Hirschberg, D. S., and Larmore, L. L. The least weight subsequence problem. *SIAM J. Comput.* **16**, 4 (1987), 628–638.

13. Huang, S., Liu, H., and Viswanathan, V. A sublinear parallel algorithm for some dynamic programming problems. *Proc. 1990 International Conf. Parallel Processing* Vol. 3, 1990, pp. 261–264.

14. Karp, R. M., and Ramachandran, V. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, North-Holland, 1990.

15. Pettorossi, A., and Burstall, R. M. Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica* **18** (1982), 181–206.

16. Rytter, W. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science* **59** (1988), 297–307.

17. Shiloach, Y., and Vishkin, U. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms* **2** (1981), 88–102.

18. Viswanathan, V., Huang, S., and Liu, H. Parallel dynamic programming. *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, 1990, pp. 497–500.

19. Waterman, M. S., and Smith, T. F. RNA secondary structure: a complete mathematical analysis. *Math. Biosciences* **42** (1978), 257–266.

20. Yao, F. F. Efficient dynamic programming using quadrangle inequalities. *Proc. 12th ACM Symp. Theory of Computing*, 1980, pp. 429–435.