Research Report 225

Parallel Algorithm for the Matrix Chain Product Problem*

Czumaj A

RR225

This paper considers the problem of finding an optimal order of the multiplication chain of matrices. All parallel algorithms known use the dynamic programming approach and run in a polylogarithmic time using, in the best case, $n^6/\log^6 n$ processors. Our algorithm uses a different approach and reduces the problem to computing some recurrence on a tree. We show that this recurrence can be optimally solved which enables us to improve the parallel bound by a few factors. Our algorithm runs in $O(\log^3 n)$ time using $n^2/\log^3 n$ processors on a CREW PRAM and in $O(\log^2 n \log\log n)$ time using n 2/(log2 n log log n) processors on a CRCW PRAM. This algorithm solves also the problem of finding an optimal triangulation in a convex polygon. We show that for a monotone polygon this result can be even improved to get an $O(\log^2 n)$ time and n processor algorithm on a CREW PRAM.

* This work was supported by the grant KBN 2-11-90-91-01.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

June 1992

Parallel Algorithm for the Matrix Chain Product Problem*

Artur Czumaj

Warsaw University
and
University of Warwick

June, 1992

Abstract

This paper considers the problem of finding an optimal order of the multiplication chain of matrices. All parallel algorithms known use the dynamic programming approach and run in a polylogarithmic time using, in the best case, $n^6/\log^6 n$ processors. Our algorithm uses a different approach and reduces the problem to computing some recurrence on a tree. We show that this recurrence can be optimally solved which enables us to improve the parallel bound by a few factors. Our algorithm runs in $O(\log^3 n)$ time using $n^2/\log^3 n$ processors on a CREW PRAM and in $O(\log^2 n \log\log n)$ time using $\frac{n^2}{\log^2 n \log\log n}$ processors on a CRCW PRAM. This algorithm solves also the problem of finding an optimal triangulation in a convex polygon. We show that for a monotone polygon this result can be even improved to get an $O(\log^2 n)$ time and n processor algorithm on a CREW PRAM.

1 Introduction

The problem of computing an *optimal order of matrix multiplication* (*the matrix chain product problem*) is defined as follows (see also e.g. [AHU-74]).

Consider the evaluation of the product of n matrices

$$M = M_1 \times M_2 \times \cdots \times M_n$$

where M_i is a $d_{i-1} \times d_i$ ($d_i \geq 1$) matrix. Since matrix multiplication satisfies the associative law, the final result is the same for all orders of multiplying. However, the order of multiplication greatly affects the total number of operations to evaluate M. The problem is to find an optimal order of multiplying the matrices, such that the total number of

*This work was supported by the grant KBN 2-11-90-91-01.

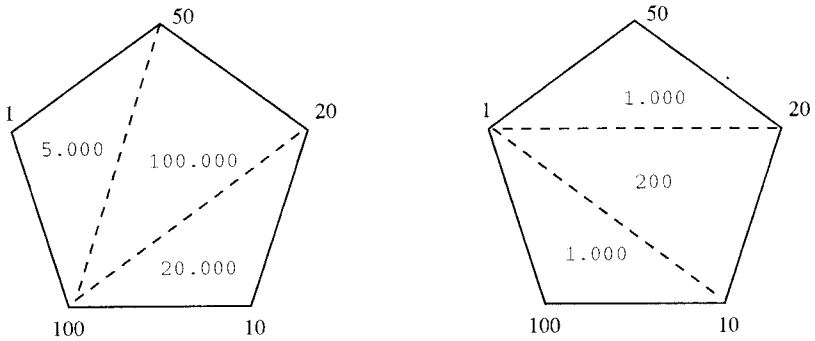

Figure 1: Geometric representation of the evaluation of a matrix chain. Above triangles correspond to the chain $M_1 \times M_2 \times M_3 \times M_4$, where dimensions are as follows $50 \times 20 \times 10 \times 100 \times 1$. Left triangualtion corresponds to the order $(M_1 \times (M_2 \times M_3)) \times M_4$ while the right one to $M_1 \times (M_2 \times ((M_3 \times M_4))$. The second order is optimal.

operations is minimized. Here, we assume that the number of operations to multiply a $p \times q$ matrix by a $q \times r$ matrix is pqr.

One can show that this problem is equivalent to the problem of finding an *optimal triangulation of a convex polygon* (see [HS-80]). Given a convex polygon $(v_0, v_1, \ldots, v_n)$. Divide it into triangles, such that the total cost of partitioning is the smallest possible. By the total cost of a triangulation we mean the sum of costs of all triangles in this partition. The cost of a triangle is the product of weights at each vertex of the triangle (see also figure 1).

Transformation from one problem to another one can be done in linear sequential time [HS-80] [HS-82] and also in $O(1)$ parallel time using n processors on a CREW PRAM [Cz-92]. Thus we will consider only the latter problem. In figure 1 there is an example of ordering of matrices and the corresponding triangulation of a polygon.

Both above problems can be solved in $O(n \log n)$ serial time [HS-80]. This and all other known algorithms seem to be highly sequential. The best previous known approach to design parallel algorithms is based on dynamic programming. It gives us NC algorithms which run in $O(\log^2 n)$ time using $n^6/\log^k n$ processors on a CREW PRAM for some constants k ([Ry-88] and [GP-92]).

So there was a big gap between the best sequential and parallel algorithms. A similar situation holds in general, for all tree problems which can be solved by dynamic programming. Such problems like the optimal binary search trees, the alphabetic binary trees, the problem of finding the optimal triangulation in a polygon and the recognizing of context free languages can, to the best of our knowledge, be solved in $O(\log^2 n)$ using almost n^6 processors. But the best sequential algorithms for these problems run in $O(n^3)$, $O(n^2)$ or even in $O(n \log n)$ time. The only exception is the Huffman coding problem where the best parallel NC algorithm performs $O(n^2 \log n)$ operations [AKLMT-89] compared with optimal $O(n \log n)$ sequential time.

Since all these problems are highly-sequential, recently there was discovered only approximate algorithms. For almost optimal binary search trees it runs in $O(\log^2 n)$ time with $O(n^2)$ total work on a CREW PRAM [AKLMT-89], for almost optimal coding trees in $O(\log n)$ time with $O(n)$ total work [AKLMT-89], and for a near-optimal order

of matrix multiplication in $O(\log n)$ time on a CREW PRAM and in $O(\log\log n)$ time on a CRCW PRAM, in both cases with linear number of operations [Cz-92]. These algorithms partially fill the gap between the total work in the sequential and parallel approaches.

In this paper we present parallel algorithm for the matrix chain product problem and for the problem of an optimal triangulation of a convex polygon. This algorithm improves the best previous parallel bound by a few factors. It runs in $O(\log^3 n)$ time using only $n^2/\log^3 n$ processors on a CREW PRAM. It can be also implemented on a CRCW PRAM model to run in $O(\log^2 n \log\log n)$ time with $O(n^3)$ total work.

This paper is organized as follows. In Section 2 we introduce some basic concepts and notations. We also describe an $O(n^2)$ time sequential algorithm for the problem of finding an optimal triangulation of a convex polygon. Then in Section 3 we show the main idea of our parallel algorithm for the matrix chain product problem and divide it into a sequence of operations *PEBBLE* and *COMPRESS*. Section 4 gives an $O(n^2)$ work NC parallel algorithm for solving the recurrence for computing the cost of an optimal triangulation of a convex polygon. In Section 5 we show how to find an optimal triangulation using computed recurrence. Then summarize all results we obtain an NC algorithm for the matrix chain product problem with $O(n^2)$ total work. In Section 6 we describe extension of our algorithm to optimal triangulation of a monotone polygon.

2 Basic notations and definitions

The following lemma is useful in analyzing parallel algorithms, since it allows us to count only the time and the total number of operations.

Lemma 2.1 *[Br-74] Let A be a given algorithm with a parallel computation time of t. Suppose that A involves a total number of m computational operations. Then A can be implemented using p processors in $O(t + m/p)$ parallel time.*

This lemma requires two qualifications before one can apply it to a PRAM. At the beginning of the i-th parallel step we must be able to compute the amount of the work W_i done by that step in $O(W_i/p)$ time using p processors, and we must know hot to assign processors to their tasks. Both these conditions will be easily satisfied by our algorithms.

2.1 The single-source minimum path problem

In this paper we consider a particular single-source minimum path problem. We are given a directed acyclic graph (DAG) whose vertices are $\{1, \ldots, n\}$. Let M be the $n \times n$ matrix giving the weights of the edges of the graph. Since our digraph is acyclic we assume that for $i \geq j$, $M(i,j) = +\infty$. For all others entries (i.e., for $i < j$) define $M(i,j) = w(i,j)$, where w is some real-valued function.

The single-source minimum path problem is to find in a graph a shortest path from 1 to i, for every $1 \leq i \leq n$. One can show (see e.g. [GP-92]), that in a DAG this problem is

equivalent to the least weight subsequence problem [HL-87]. Given a real-valued weight function $w(i,j)$ and $d(1)$. Compute

$$d(j) = \min_{1 \le i < j} \{d(i) + w(i,j)\}, \quad \text{for all } 1 < j \le n$$

This problem was recently analysed in many papers, since it has a long list of applications.

The weight function w is said to be *convex* if it satisfies the inverse quadrangle inequality

$$w(i,j) + w(i+1,j+1) \ge w(i,j+1) + w(i+1,j), \quad \text{for all } 1 \le i < j-1 < n$$

We will also said the function w to be *concave* if it satisfies the quadrangle inequality

$$w(i,j) + w(i+1,j+1) \le w(i,j+1) + w(i+1,j), \quad \text{for all } 1 \le i < j-1 < n$$

In the general case the least weight subsequence problem can be solved in $O(n^2)$ optimal sequential time and in $O(\log^2 n)$ time using $O(n^3/\log^4 n)$ processors on a CREW PRAM [GP-92]. But when the weight functions are either concave or convex we can do it much better. The best sequential algorithm runs in $O(n)$ time when the weight functions are concave [Wil-88] and in $O(n\alpha(n))$ time for the convex weights [KK-90]. Recently there was discovered also parallel algorithms. The best NC algorithm for the concave weight runs in $O(\log^2 n)$ time using $n^2/\log n$ processors. For the convex weight function we can do it more efficiently.

Fact 2.2 *[CL-90] The convex least weight subsequence problem can be solved in $O(\log^2 n)$ time using n processors on a CREW PRAM*[1].

2.2 Notation concerning the triangulation problem

Throughout this paper we will use $v_0, v_1, \ldots, v_n$ to denote vertices as well as their weights in a convex polygon. For simplicity we assume that all weights are distinct. If there are some vertices with the same weights then we assume that a particular ordering is chosen and remains fixed.

Define a vertex v_i to be the *smallest* (minimum) one if for each other vertex v_j we have $v_i < v_j$. Similarly we define the kth *smallest* vertex v_i if there are exactly $k-1$ vertices smaller than v_i.

Define a *basic polygon* to be a polygon where the second and the third smallest vertices are neighbours of the smallest vertex. The following fact reduces our problem to the triangulation of *basic polygons*.

Fact 2.3 *[HS-80] There exists an optimal triangulation of a convex polygon containing arcs or sides between the smallest vertex and both the second and the third smallest ones*

[1]Chan and Lam showed in his paper an algorithm which runs in $O(\log^2 n \log\log n)$ time with $O(n \log^2 n)$ total work on a CREW PRAM. But using result for finding the all row minima in a totally monotone 2-dimensional array [AK-90], we can simply improve it to the presented form.

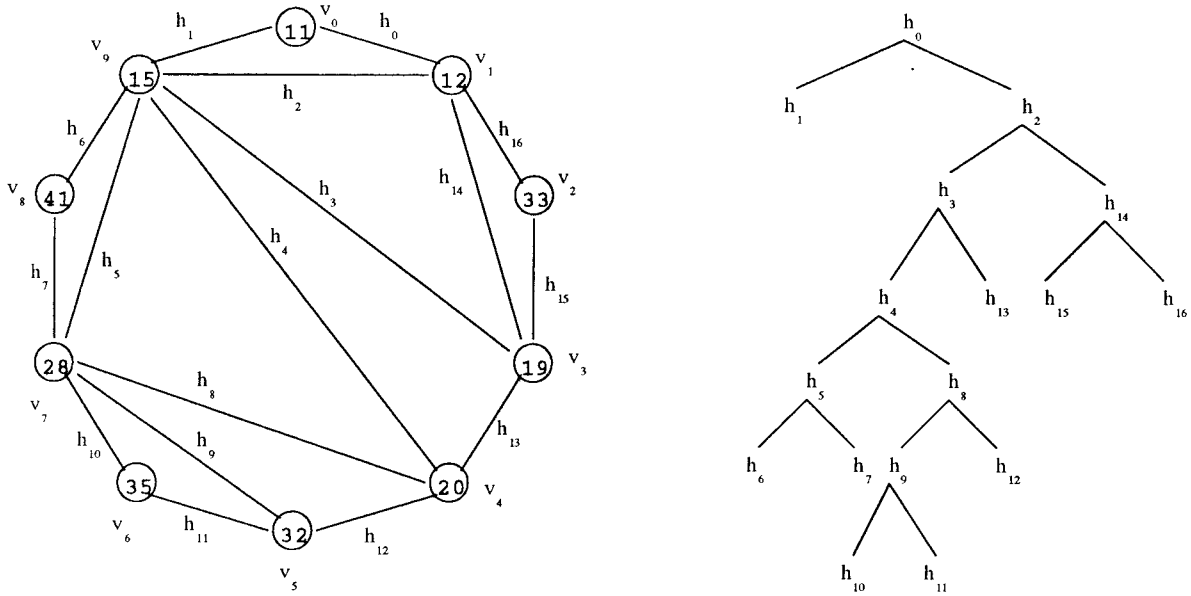

Figure 2: Candidates in a polygon and corresponding tree of candidates.

Fact 2.3 implies a partition of a convex polygon into smaller nonintersecting basic subpolygons which are in an optimal triangulation. In [Cz-92] was shown that such partitioning can be found in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM.

From now on, we will find an optimal triangulation in each basic subpolygon independently. We will consider only basic polygons $(v_0, v_1, \ldots, v_n)$, where $v_0 < v_1 < v_n < v_i$, for each $1 < i < n$.

Also the following fact holds.

Fact 2.4 *[HS-80] There exists an optimal triangulation of a convex polygon containing either the arc joining the smallest vertex with the fourth smallest one or the arc joining the second smallest vertex with the third smallest one.*

This fact allows us to design a sequential $O(n^2)$ time algorithm.

2.3 An sequential $O(n^2)$ time algorithm for the matrix chain product problem

Yao uses tabulation methods (dynamic programming) to find an optimal triangulation in $O(n^2)$ sequential time [Yao-82]. We briefly describe this algorithm.

Define a *candidate* to be an arc or side (v_i, v_j) such that for each k, $i < k < j$, the inequalities $v_i < v_k$, $v_j < v_k$ hold. One can show that no candidates intersect (except possibly at the endpoint), thus the number of candidates is linear (in an n-gon there are exactly $2n - 3$ candidates).

Define the *tree of candidates.* Candidate (v_i, v_j) is an ancestor of candidate (v_k, v_l) if and only if $i \leq k < l \leq j$ and $(v_i, v_j) \neq (v_k, v_l)$. It is easy to see that such defined tree is binary. In [Cz-92] was shown how to find the tree of candidates in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM. In this tree, the sides of the polygon are leaves,

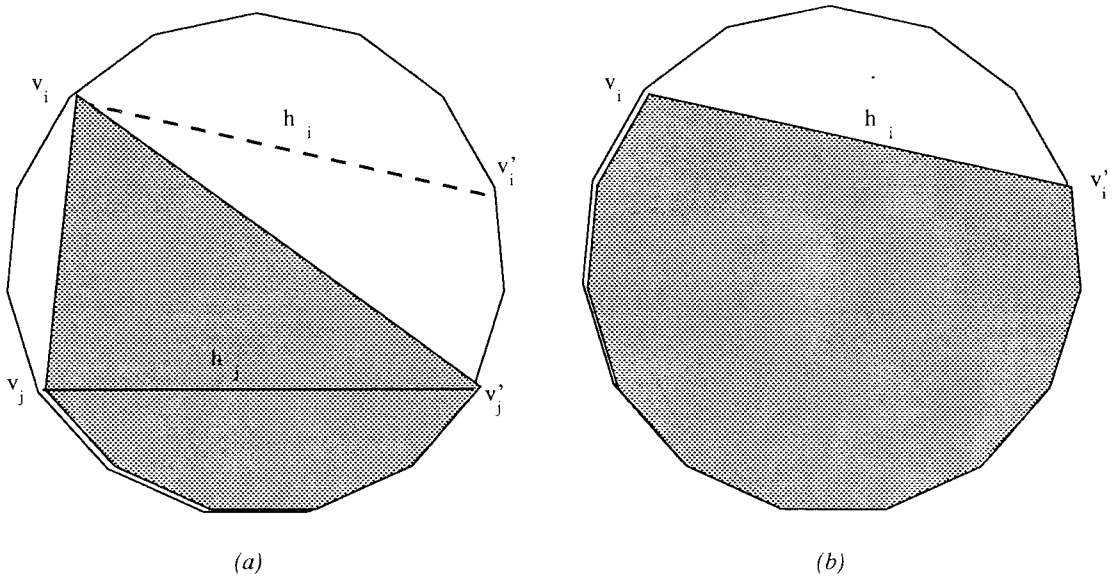

Figure 3: Cones - *(a)* cone $Q(h_i, h_j)$; *(b)* cone $Q(h_i, h_i)$

except the side (v_0, v_1) which is the root of the tree. We will also use the notation h_i to denote a candidate (v_i, v_i') and in such case we will always assume that $v_i < v_i'$. An example of candidates and of a tree of candidates is shown in figure 2.

We will say that a polygon P is *below* an arc (v_i, v_j) where $i < j$, if $P = (v_i, \ldots, v_j)$. In figure 2 the polygon below candidate (v_3, v_9) is $P = (v_3, v_4, v_5, v_6, v_7, v_8, v_9)$.

Define also the *cone* to be a subpolygon Q of the input polygon, such that Q is equal to the sum of the polygon below some candidate $h_j = (v_j, v_j')$ and of the triangle (v_i, v_j, v_j') where v_i is on a candidate h_i which is an ancestor of h_j or $h_i = h_j$. We will denote such a cone as $Q(h_i, h_j)$ or $Q(i, j)$. In figure 3 is shown the cone $Q(h_i, h_j)$ - with assumption (a) that h_i is an ancestor of h_j or (b) $h_i = h_j$. Also, in figure 2 the cone $Q(h_2, h_8)$ is the polygon $Q = (v_1, v_4, v_5, v_6, v_7)$.

Define $l(i)$ $(r(i))$ to be the left (right) son of candidate h_i in the tree of candidates. Let us also define $s(i)$ $(g(i))$ to be the son of h_i in the tree of candidates which is joining with smaller (greater) vertex on a candidate h_i, that is with v_i (respectively v_i'). Denote by $\triangle(i, j)$ the cost of the triangle (v_i, v_j, v_j'). Define also by $c(i, j)$ the cost of an optimal triangulation of a cone $Q(i, j)$.

Let us assume that we want to compute value $c(i, i)$ (see figure 4 (a)). Since $c(i, i)$ denotes the cost of the polygon below h_i where v_i is the smallest vertex and v_i' is the second smallest one and moreover $v_{l(i)}'$ is the third smallest one, we can join vertex v_i with $v_{l(i)}'$ using Fact 2.3.

Let us assume that we want to compute value $c(i, j)$ where h_i is a ancestor of h_j (see figure 4 (b)(c)). In $Q(i, j)$, v_i is the smallest vertex, v_j, v_j' are the second and the third smallest ones and $v_{l(j)}'$ is the fourth smallest one. Thus using Fact 2.4 we have to choose the smallest of the partitions either after joining v_i with $v_{l(j)}'$, or after joining v_j with v_j'.

These observations reduce our algorithm to the problem of solving the following

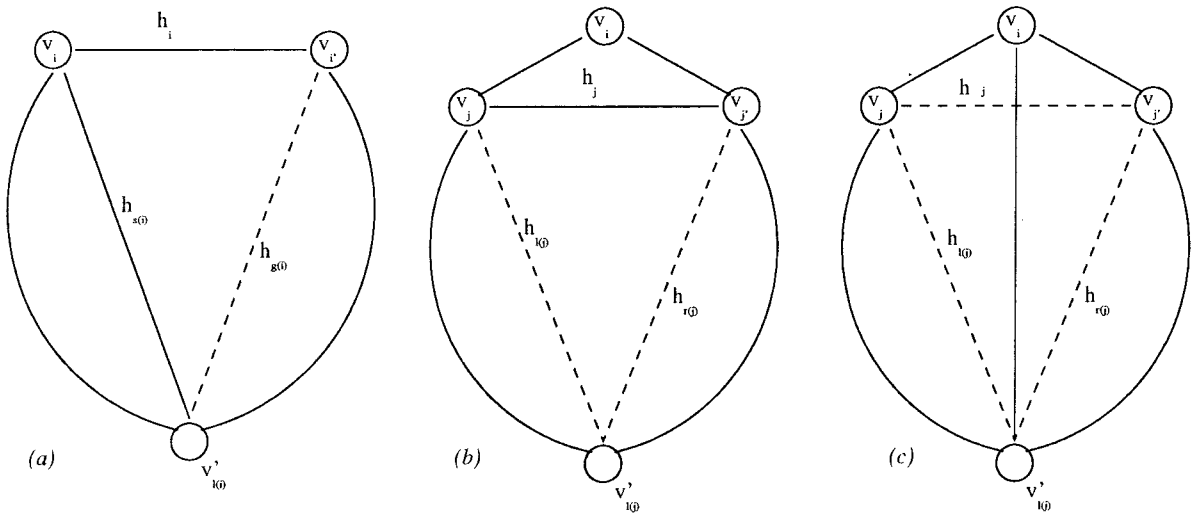

Figure 4: Possible partitioning of a cone and possible computations of values c.

$$\begin{array}{ll} \text{(a)} & c\,(i,i) = c\,(i,g\,(i)) + c\,(s\,(i),s\,(i)) \\ \text{(b)} & c\,(i,j) = \triangle(i,j) + c\,(j,j) \\ \text{(c)} & c\,(i,j) = c\,(i,l\,(j)) + c\,(i,r\,(j)) \end{array}$$

recurrences in the tree.

$$c(i,j) = \begin{cases} 0 & \text{if } i = j \text{ and } h_i \text{ is a leaf} \\ \triangle(i,j) & \text{if } i \neq j \text{ and } h_j \text{ is a leaf} \\ c(i,g(i)) + c(s(i),s(i)) & \text{if } i = j \text{ and } h_j \text{ is not a leaf} \\ \min \begin{cases} \triangle(i,j) + c(j,j) \\ c(i,r(j)) + c(i,l(j)) \end{cases} & \text{if } h_i \text{ is an ancestor of } h_j \text{ and } h_j \text{ is not a leaf} \end{cases}$$

Our goal is to compute the value $c((v_0,v_1),(v_0,v_1))$, i.e., $c(\text{ROOT},\text{ROOT})$. And it is clear that the reconstruction of an optimal triangulation from computed values $c(i,j)$ can be done in $O(n)$ sequential time.

Fact 2.5 *[Yao-82] There exists an algorithm for computing an optimal triangulation of a convex polygon which runs in $O(n^2)$ time.*

Proof: Correctness of the algorithm follows from the previous comments (see also [Yao-82]). We compute function $c(i,j)$ in a bottom-up manner. Before we start to compute $c(i,j)$ we have already computed all values $c(i,k)$ and $c(s,j)$ for all h_k - descendants of h_j and h_s - descendants of h_i. Thus an $O(n^2)$ running time is clear. □

3 Outline of a parallel algorithm for the triangulation problem

In Section 2 we showed how to reduce our problem to the problem of computing some recurrence on trees. Using standard methods [GR-88] we can solve this recurrence in $O(\log^2 n)$ time using n^6 processors. In this section we show how to reduce the number

of processors needed.
Our algorithm runs in the following four steps:

1. Divides the polygon into basic polygons.
2. Computes the tree of candidates for basic polygons.
3. Computes $c(i,j)$ for all pairs i,j.
4. Finds an optimal triangulation using the values $c(i,j)$.

Steps (1) and (2) can be done in $O(\log n)$ using $n/\log n$ processors on a CREW PRAM [Cz-92]. So we only show how to implement steps (3) and (4) in an efficient way. In this section we give an outline of step (3) which will be analysed in detail in the following section. Section 5 gives algorithm for reconstruction of an optimal polygon from recurrence for the array c.

Let us define a vertex h_j in the tree of candidates to be *pebbled* if all values $c(i,j)$ and $c(j,i)$ have been computed.

At the beginning of the algorithm we can easily pebble all leaves (corresponding to the sides of a basic polygon) in $O(1)$ time with n^2 processors. And at the end of the algorithm we want to pebble the root of the tree.

We will use the idea of *tree contraction* [MR-85] [Ry-85]. Define two operations on a tree. Operation *PEBBLE* pebbles all vertices for which both sons are already pebbled. Operation *COMPRESS* operates on a chain of vertices (see figure 5).

Suppose we have a sequence of vertices $h_1, \ldots, h_k$ such that

- h_i is a father of h_{i+1} and
- each h_i is not pebbled and
- h_k has two pebbled sons and
- each h_i (except h_k) has got exactly one pebbled son

We will call such a sequence the *chain*. The operation *COMPRESS* pebble all vertices on all chains in the tree.

It is well known that in a binary tree with m vertices the following algorithm will pebble the root of the tree.

repeat $\lceil \log_2 m \rceil$ **times**
  PEBBLE; COMPRESS;

Thus it is enough to show how the operations *PEBBLE* and *COMPRESS* may be executed in an efficient way. We will also ensure the following invariant after each operation. If vertex h_i is pebbled then all its descendants are also pebbled.

The operation *PEBBLE* can be easily performed in constant time with $O(n^2)$ operations on a CREW PRAM. This is because to pebble vertex h_i we need only to have already computed all values c for its sons. And we know that these values are computed because both sons are pebbled. In the following section we show how to execute the operation *COMPRESS* with the same work.

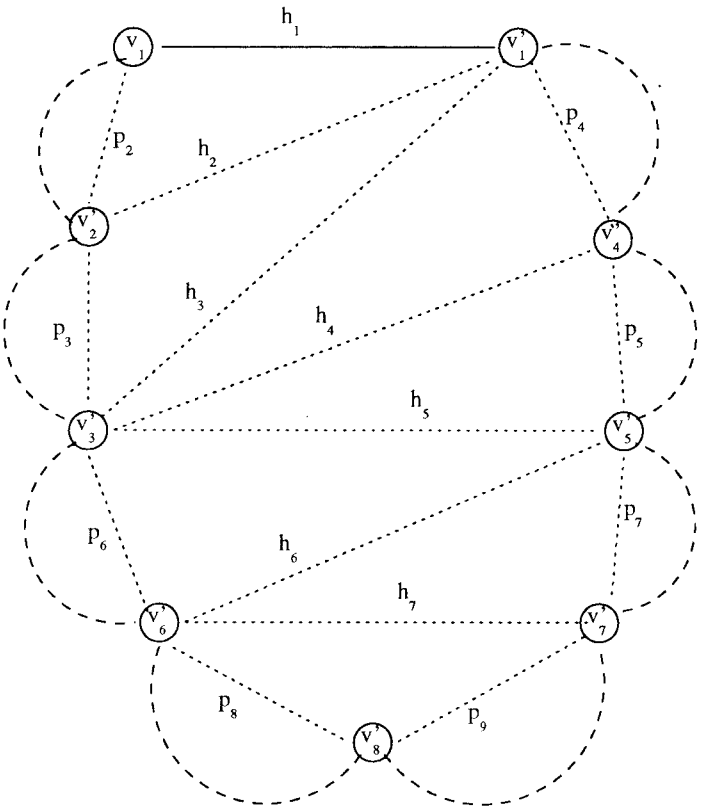

Figure 5: Chain $h_1, h_2, \ldots, h_7$. Candidates $p_2, \ldots, p_9$ are pebbled and h_i is a father of h_{i+1}.

4 Computing the cost of an optimal triangualtion of a polygon

Since the operation *PEBBLE* can be easily executed in constant time with $O(n^2)$ number of processors, we have only to show how to perform with the same bound the operation *COMPRESS*.

We are given a chain of candidates $h_1, h_2, \ldots, h_k$ (see figure 5). h_i is the father of h_{i+1} and p_{i+1} in the tree of candiadtes, and p_{i+1} has already been pebbled. Both sons of h_k (p_{k+1} and p_{k+2}) have already been pebbled. One property of such a chain is that $v_i < v'_i < v'_{i+1}$ and that either $v_i = v_{i+1}$, or $v'_i = v_{i+1}$.

We start by computing values $c(i,i)$ for all $1 \leq i \leq k$ and then we compute values $c(i,j)$ for all h_j on the chain.

4.1 Computing values $c(i,i)$

Let $bottom(i)$ denotes the cost of an optimal triangulation of the polygon below candidate h_i without candidates from the chain. Let also $fan(i,j)$ denotes the cost of an optimal triangulation of the polygon between two candidates from the chain h_i and h_j, where there is no other candidate from the chain. We will always assume that h_i is an ancestor of h_j. For example in figure 5, $fan(2,6)$ denotes the cost of an optimal partitioning of

polygon $P = (v'_1 \ldots v'_4 \ldots v'_5, v'_6 \ldots v'_3 \ldots v'_2)$.

In an optimal triangulation of the polygon below h_i we have two cases - either below h_i there exists at least one candidate from the chain $h_1, h_2, \ldots, h_k$, or there does not.

If below h_i in an optimal triangulation there is no candidate from the chain then $c(i,i) = bottom(i)$. If there are, then let h_j be the highest candidate from this partition (i.e., with the smallest index). In this case we get $c(i,i) = fan(i,j) + c(j,j)$. Thus, since we are interested in the best partitioning, we obtain the following formula for computing values $c(i,i)$.

$$c(i,i) = \min \begin{cases} bottom(i) \\ \min_{i<j\leq k}\{fan(i,j) + c(j,j)\} \end{cases}$$

This recurrence is equivalent to the single-source minimum path problem in a DAG. We are given the weight of the edge from the source - $bottom(i)$. For each vertex i, a minimum path is either the edge directly from the source or is a minimum path to one of precedes vertices and then the edge from this vertex to i. To reduce our problem to the above one we have to compute in advance values $bottom(i)$ and $fan(i,j)$. We can do it using the following lemma.

Lemma 4.1 *Let h_i and h_j be two candidates, where h_i is an ancestor of h_j. If in an optimal partitioning of the polygon below h_i there is no candidate which lies between h_i and h_j then there exists an optimal triangulation where v_i is joined to both v_j and v'_j.*

Proof: Our proof is by induction. If $i+1 = j$ then in the polygon below h_i, v_i is the smallest vertex, v'_j is the third smallest one and v_j is either equal to v_i or v_j is the second smallest one. Thus using Fact 2.3 the result follows.

So, assume that $i < j-1$. From the induction assumption we get that v_i is connected both with v_{j-1} and v'_{j-1}. This implies that the cone $Q(i, j-1)$ is in an optimal partition of the polygon (see figure 6). Now we consider only a triangulation of this cone. Because either $v_j = v_{j-1}$ or $v_j = v'_{j-1}$, it is enough to prove only that v_i is joined with v'_j. Since v_{j-1} is not joined with v'_{j-1} we get $v_i \neq v_{j-1}$. Thus in the cone $Q(i, j-1)$, v_i is the smallest vertex, v_{j-1} - the second, v'_{j-1} - the third and v'_j - the fourth smallest vertex. Since v_{j-1} is not joined with v'_{j-1}, Fact 2.4 implies that v_i is connected with v'_j. □

Using this lemma we can compute the functions *bottom* and *fan*.

$$bottom(i) = \sum_{r=i+1}^{k+2} c(h_i, p_r)$$

$$fan(i,j) = \triangle(h_i, h_j) + \sum_{r=i+1}^{j} c(h_i, p_r)$$

Here $c(h_i, p_r)$ denotes the cost of an optimal triangulation of the cone $Q(h_i, p_r)$. And since all candidates p_r have been pebbled, all values $c(h_i, p_r)$ are already computed. Hence we can compute the values $bottom(i)$ and $fan(i,j)$ in $O(\log n)$ time using $n^2/\log n$ processors on a CREW PRAM.

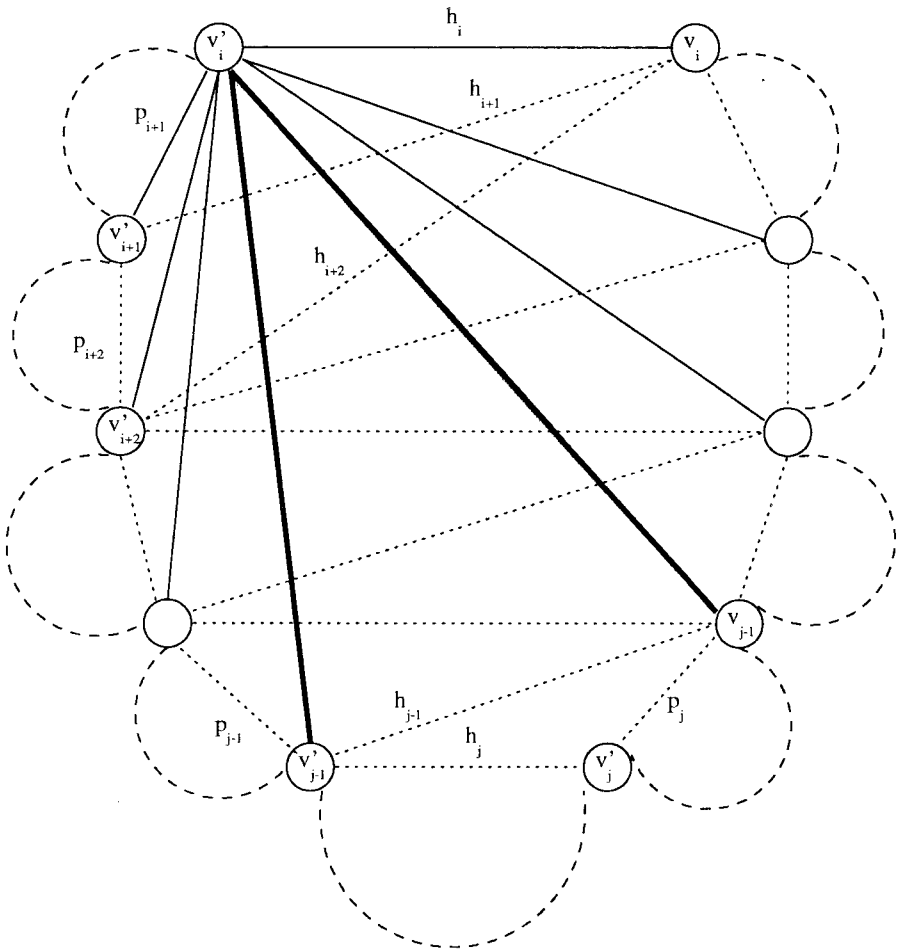

Figure 6: Chain $h_i, h_{i+1}, \ldots, h_{j-1}, h_j$. In an optimal partition of the polygon below h_i there is no candidates between h_i and h_j.

To reduce our problem to the single-source minimum path one we define the weight matrix of the graph as follows.

$$M(i,j) = \begin{cases} +\infty & i \geq j \\ fan(j,i) & i < j \leq k \\ bottom(i) & i < j = k+1 \end{cases}$$

and we are looking for the minimum path from the source $k+1$.

Thus using standard methods we can compute all values $c(i,i)$ with almost $O(n^3)$ work, but we can improve this bound because the following lemma holds.

Lemma 4.2 *Matrix M is convex.*

Proof: To prove that M is a convex matrix we must check whether below inequality holds.

$$M(i,j) + M(i+1,j+1) - M(i,j+1) - M(i+1,j) \geq 0 \qquad \text{for all } i < j-1$$

First we consider the case when $j+1 < k+1$. From the definition we get

$$M(i,j) = \text{fan}(i,j) = \triangle(h_i, h_j) + \sum_{r=i+1}^{j} c(h_i, p_r)$$

$$M(i+1,j+1) = \text{fan}(i+1,j+1) = \triangle(h_{i+1}, h_{j+1}) + \sum_{r=i+2}^{j+1} c(h_{i+1}, p_r)$$

$$M(i,j+1) = \text{fan}(i,j+1) = \triangle(h_i, h_{j+1}) + \sum_{r=i+1}^{j+1} c(h_i, p_r)$$

$$M(i+1,j) = \text{fan}(i+1,j) = \triangle(i+1,j) + \sum_{r=i+2}^{j} c(h_{i+1}, p_r)$$

Thus

$$\begin{aligned}
&M(i,j) + M(i+1,j+1) - M(i,j+1) - M(i+1,j) \\
&= \text{fan}(i,j) + \text{fan}(i+1,j+1) - \text{fan}(i,j+1) - \text{fan}(i+1,j) \\
&= \triangle(h_i,h_j) + \triangle(h_{i+1},h_{j+1}) - \triangle(h_i,h_{j+1}) - \triangle(h_{i+1},h_j) + c(h_{i+1},p_{j+1}) - c(h_i,p_{j+1}) \\
&= v_i v_j v'_j + v_{i+1} v_{j+1} v'_{j+1} - v_i v_{j+1} v'_{j+1} - v_{i+1} v_j v'_j + c(h_{i+1},p_{j+1}) - c(h_i,p_{j+1}) \\
&= (v_{i+1} - v_i)(v_{j+1} v'_{j+1} - v_j v'_j) + c(h_{i+1},p_{j+1}) - c(h_i,p_{j+1})
\end{aligned}$$

Now we can use some properties of the chain

- $v_i \le v'_i$, $v_i \le v_{i+1}$, $v'_i \le v'_{i+1}$, for all i
- if P and P' are both m-gons where the corresponding weights satisfies $w_i \le w'_i$, then the cost of an optimum partition of P is less than or equal to the cost of an optimum partition of P'. This natural observation was shown first in [HS-82]. From the above follows that $c(h_{i+1}, p_{j+1}) \ge c(h_i, p_{j+1})$.

From these properties we get the result.

Now we consider the case when $j+1 = k+1$. From the definition we get.

$$M(i,j) = \text{fan}(i,k) = \triangle(h_i, h_k) + \sum_{r=i+1}^{k} c(h_i, p_r)$$

$$M(i+1,j+1) = \text{bottom}(i+1) = \sum_{r=i+2}^{k+2} c(h_{i+1}, p_r)$$

$$M(i,j+1) = \text{bottom}(i) = \sum_{r=i+1}^{k+2} c(h_i, p_r)$$

$$M(i+1,j) = \text{fan}(i+1,k) = \triangle(h_{i+1}, h_k) + \sum_{r=i+2}^{k} c(h_{i+1}, p_r)$$

Thus

$$\begin{aligned}
&M(i,j) + M(i+1,j+1) - M(i,j+1) - M(i+1,j) \\
&= (v_i - v_{i+1}) v_k v'_k + c(h_{i+1}, p_{k+1}) - c(h_i, p_{k+1}) + c(h_{i+1}, p_{k+2}) - c(h_i, p_{k+2})
\end{aligned}$$

Now we must look more carefully at the definition of $c(h_{i+1}, p_{k+1})$. This function denotes the cost of a minimal triangulation of the polygon below p_{k+1} with a triangle $\triangle(h_{i+1}, p_{k+1})$. In this polygon all vertices v_t, v_s (except v_{i+1}) satisfy $v_t v_s \geq v_k v'_k$. Let us consider the same triangualation with weight v_i instead of v_{i+1}. Denote its cost as $d(h_i, p_{k+1})$. It is clear that $c(h_i, p_{k+1}) \leq d(h_i, p_{k+1})$. In both polygons corresponding to $c(h_{i+1}, p_{k+1})$ and $d(h_i, p_{k+1})$ there must be some triangle with vertex v_{i+1} (respectively v_i). Let other vertices of this triangle be v_t and v_s. Thus we get

$$c(h_{i+1}, p_{k+1}) - d(h_i, p_{k+1}) \geq (v_{i+1} - v_i) v_t v_s$$

And since $v_t v_s \geq v_k v'_k$ and $c(h_i, p_{k+1}) \leq d(h_i, p_{k+1})$, we obtain

$$(v_i - v_{i+1}) v_k v'_k + c(h_{i+1}, p_{k+1}) - c(h_i, p_{k+1}) \geq 0$$

Since there is also $c(h_{i+1}, p_{k+2}) - c(h_i, p_{k+2}) \geq 0$ we get the result. □

Because we have to compute in advance the arrays *bottom* and $fan(i,j)$ with $O(n^2)$ total work, we need $O(\log n)$ time with $n^2/\log n$ processors on a CREW PRAM for preprocessing. And then, since our weights are convex, we compute all values $c(i,i)$ using Fact 2.2 in $O(\log^2 n)$ time with n processors. This gives us $O(\log^2 n)$ time and $O(n^2)$ operations for this step.

4.2 Computing entries $c(i,j)$

Now we describe how to compute the values $c(i,j)$ for either h_i or h_j from the chain. We may assume that h_i is an ancestor of h_j, since when $h_i = h_j$ we have computed these values in the previous section. And because we have already computed such values for all h_j which are not in the chain and are below the candidate's root on the chain, we will only consider h_j from the chain.

From Lemma 2.4 we obtain the following recurrence for $c(i,j)$

$$c(i,j) = \min \begin{cases} \triangle(i,j) + c(j,j) \\ c(i, r(j)) + c(i, l(j)) \end{cases}$$

Since the values $\triangle(i,j)$, $c(j,j)$ and either $c(i,r(j))$ or $c(i,l(j))$ (because either $h_{l(j)}$ or $h_{r(j)}$ is pebbled) are already computed, we may assume that are computed in advance. For fixed index i this recurrence can be solved using standard algorithms for expression evaluation problem [GiRy-86]. This gives us $O(\log n)$ time with $O(n)$ work for fixed i. Thus we can compute values $c(i,j)$ for all $i \neq j$, such that h_j lies on a chain, in $O(\log n)$ time with $n^2/\log n$ processors on a CREW PRAM.

4.3 Computing all entries of the array c

Now we can count the total work of the algorithm. The operation *PEBBLE* can be done in $O(1)$ time with n^2 processors on a CREW PRAM. To compute values $c(i,i)$ we need $O(\log^2 n)$ time with $n^2/\log^2 n$ processors and to compute all other values $c(i,j)$ we need $O(\log n)$ time with $n^2/\log n$ processors on a CREW PRAM. Hence we obtain an

algorithm for solving the recurrence for the array c, which runs in $O(\log^3 n)$ time with $n^2/\log^2 n$ processors on a CREW PRAM.

But we can look more precisely at the needed number of operations. Let m_t be the number of vertices which are pebbled in the t-th step of the main loop. The operation *PEBBLE* can be done in constant time with $O(m_t^2)$ operations. The operation *COMPRESS* need $O(\log^2 n)$ time and only $O(nm_t)$ operations on a CREW PRAM. Hence in the t-th step both the operations *PEBBLE* and *COMPRESS* can be executed in $O(\log^2 n)$ time with $O(nm_t)$ total work. Since $\sum_t m_t = O(n)$, we get $O(n^2)$ number of operations in the whole algorithm. Using Brent's Lemma 2.1 we can decrease the number of needed processors to $n^2/\log^3 n$ for a CREW PRAM and to $\frac{n^2}{\log^2 n \log\log n}$ for a CRCW PRAM. This lemma requires the assignment of processors to their tasks, which can be easily done in our algorithm. Hence we obtain the following lemma.

Lemma 4.3 *We can compute the array c in $O(\log^3 n)$ time using $n^2/\log^3 n$ processors on a CREW PRAM and in $O(\log^2 n \log\log n)$ time using $\frac{n^2}{\log^2 n \log\log n}$ processors on a CRCW PRAM.*

5 Reconstruction of an optimal triangulation

Now we are given correctly computed the array c. Thus to solve the whole triangulation problem we must only show how to find an optimal partition of a basic convex polygon using the array c. There exists simple sequential linear time algorithm for reconstruction but is harder to find an $O(n^2)$ work NC parallel algorithm for this problem.

Our algorithm runs in three steps. First, it finds for each candidate its *ceil*. Then it computes for each candidate the set of descendants which are in an optimal triangulation of the polygon below this candidate. In the last step algorithm finds all arcs which are in an optimal triangulation.

One can show that the reconstruction can be done during the executing of algorithm which computes the array c. But for better presentation we describe these operations independently.

5.1 Finding ceils

For each candidate h_i define its *ceil* to be the set of candidates $\{h_{j_1}, \ldots, h_{j_{t_i}}\}$ such that

1. every h_{j_s} is a descendant of h_i
2. every h_{j_s} exists in an optimal triangulation in the polygon below h_i
3. all candidates from the ceil are the highest ones which satisfy (1) and (2), that is if h_k lies between h_i and h_{j_s} then h_k does not belong to the ceil of h_i

Such defined set we will denote as $Ceil(h_i)$.

We compute the sets $Ceil(h_i)$ for each h_i independently. From recurrence for the array c follows that one of sons of h_i is in its ceil. Thus to compute $Ceil(h_i)$ we consider the subtree of the tree of candidates which is rooted at the second son of h_i - $h_{g(i)}$. It is easy to

see that if $c(i, g(i)) = \triangle(i, g(i)) + c(g(i), g(i))$ then $h_{g(i)}$ exists in an optimal triangulation. Otherwise, $c(i, g(i)) < \triangle(i, g(i)) + c(g(i), g(i))$ and in this case a son h_k of $h_{g(i)}$ exists in an optimal triangulation of the polygon below h_i only if $c(i, k) = \triangle(i, k) + c(k, k)$. This observation gives us the following condition for candidates from the ceil of h_i.

$h_k \in Ceil(h_i)$ if and only if either $h_k = h_{l(i)}$ or

- $h_k = h_{g(i)}$ or h_k is a descendant of $h_{g(i)}$ and
- $c(h_i, h_k) = \triangle(h_i, h_k) + c(h_k, h_k)$ and
- if h_m lies between h_i and h_k then $h_m \notin Ceil(h_i)$

Hence our problem can be reduced to the following one. We are given a binary tree T ($h_{g(i)}$ is the root of this tree). There are some marked vertices in the tree (h_j is marked iff $c(h_i, h_j) = \triangle(h_i, h_j) + c(h_j, h_j)$). For each marked vertex check whether all its ancestors are not marked. This problem can be solved in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM as follows.

First we create the Euler tour of the tree of candidates [TaVi-85] in constant time with n processors on a CREW PRAM. That is, we create a list of directed edges of the tree in such a way. Let for any vertex v, $f(v)$ denotes its father, $l(v)$ denotes its left son, and $r(v)$ denotes its right son. Then if v is a leaf we take the edge following $(f(v), v)$ to be $(v, f(v))$. Otherwise we follow $(f(v), v)$ by $(v, l(v))$, and $(l(v), v)$ by $(v, r(v))$ and $(r(v), v)$ by $(v, f(v))$. Additionally if v is the root, then the edge $(v, l(v))$ is the first vertex on the list, and $(r(v), v)$ is the last one.

Now we can solve our problem using the prefix computation scheme. If a vertex v is not marked, then assign for the edges $(v, l(v))$, $(l(v), v)$, $(v, r(v))$ and $(r(v), v)$ value 0. Otherwise assign for the edges $(v, l(v))$ and $(v, r(v))$ value 1 and for the edges $(l(v), v)$ and $(r(v), v)$ value -1. From the construction of the Euler tour follows that for every vertex v all its ancestors are not marked if and only if the sum of all edges which precede the edge $(v, f(v))$ is equal[2] to 0. Using an optimal $O(\log n)$ time algorithm for list ranking [CV-86], we can compute this sum in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM. Hence we can find $Ceil(h_i)$ for all candidates in $O(\log n)$ time with $O(n^2)$ total work on a CREW PRAM.

5.2 Finding all candidates which exist in an optimal triangulation of the polygon below h_i

For each two candidates h_i, h_j, where h_i is an ancestor of h_j, define $D(i, j) = 1$ iff in an optimal triangulation of the polygon below h_i exists candidates h_j. Otherwise $D(i, j) = 0$. It is clear that the array D denotes the transitive closure of the function $Ceil$.

Initialy we set $D(i, j) := 0$ for all i, j. We will fill entries of the array D during the operations *PEBBLE* and *COMPRESS* of the algorithm. We will ensure the following

[2]To be more precise, this value is equal to the number of ancestors of v which are marked.

invariant after each operation. If a candidate h_i is pebbled, then we have correctly computed values $D(i,j)$ for all j.

When we pebble some vertex h_i, we can compute values $D(i,j)$ as follows. Let $Ceil(h_i) = \{h_{j_1}, \ldots, h_{j_{t_i}}\}$. We start with setting $D(i,k) := 1$ for all $h_k \in Ceil(h_i)$. All other candidates which exist in an optimal triangulation of the polygon below h_i are descendants of candidates from $Ceil(h_i)$. Thus for every h_d which is a descendant of some vertex $h_k \in Ceil(h_i)$ we may set $D(i,d) := D(k,d)$. It can be done in $O(1)$ time and n processors for each pebbled vertex. Thus we can execute all *PEBBLE* steps in $O(\log n)$ time with $O(n^2)$ time-processor product on a CREW PRAM.

When we perform the operation *COMPRESS* on the chain, we may compute values $D(i,j)$ in a similar way. From the formula for $c(i,i)$, we get $c(i,i) = fan(i,s) + c(s,s)$, where either $i = s$ (then $fan(i,i) = 0$) or h_s is a descendant of h_i from the chain. Using this equality we can easily find for every i independently, all candidates on the "minimum path". That is, we can find all candidates from the chain which exist in an optimal triangulation of the polygon below h_i. Denote them as $\{h_{j_1}, h_{j_2}, \ldots, h_{j_m}\}$ in such a way that h_{j_r} is always an ancestor of $h_{j_{r+1}}$. Let us also denote $h_{j_0} = h_i$. These candidates can be easily found in $O(\log n)$ time with $O(n)$ total work for each h_i independently. It is easy to see that $h_{j_{r+1}} \in Ceil(h_{j_r})$. We begin with setting $D(i,j_r) := 1$, for all $1 \le r \le m$. Now we find the "highest pebbled ceil" of h_i. That is, we find the highest candidates which are not on the chain and exist in an optimal triangulation of the polygon below h_i. We can get them using values $Ceil(h_{j_r})$ for $0 \le r \le m$. The highest pebbled ceil of h_i is the sum over all r, $0 \le r \le m$, of sets $Ceil(h_{j_r}) - \{h_{j_{r+1}}\}$. Denote this ceil as $HPCeil(h_i)$. We start with setting $D(i,k) := 1$ for every vertex $h_k \in HPCeil(h_i)$. Then for every h_d which is a descendant of some vertex $h_k \in HPCeil(h_i)$ we set $D(i,d) := D(k,d)$. Hence we can compute all values $D(i,j)$ for h_i on the chain in $O(\log n)$ time with $O(n)$ total work on a CREW PRAM.

Summarizing the discussion above, we can compute the array D in $O(\log^2 n)$ time using $n^2/\log^2 n$ processors on a CREW PRAM[3].

5.3 Reconstruction of an triangulation triangulation

Now we can easily reconstruct an optimal triangulation from the arrays D and $Ceil$. From values $D(0,j)$, where h_0 denotes the root of the tree of candidates, we get all candidates which exist in optimal triangulation of the whole polygon. Let h_i exists in this one. We know that between h_i and its ceil there does not exist any candidate. Thus we may triangulate the polygon between h_i and its ceil using Lemma 4.1. We must only join v_i with all vertices from $Ceil(h_i)$. Hence we can perform this step in constant time with $O(n)$ work on a CREW PRAM.

Summarizing all discussions so far, we have the following lemma.

Lemma 5.1 *We can reconstruct an optimal triangulation of the convex polygon in $O(\log^2 n)$ time using $n^2/\log^2 n$ processors on a CREW PRAM.*

This lemma implies the main theorem.

[3]One can also improve this result to $O(\log n)$ time with $O(n^2)$ total work on a CREW PRAM.

Theorem 1
The matrix chain product problem and the problem of finding an optimal triangulation of a convex polygon can be solved in $O(\log^3 n)$ time using $n^2/\log^3 n$ processors on a CREW PRAM and in $O(\log^2 n \log\log n)$ time using $n^2/\log^2 n \log\log n$ processors on a CRCW PRAM.

6 Algorithm for an optimal triangulation of a monotone polygon

Define a *monotone polygon* to be a convex basic polygon with weights $(v_0, v_1, \ldots, v_n)$, where $v_0 < v_1 < \ldots < v_k$ and $v_k > v_{k+1} > \ldots > v_n$. One can show that in such a polygon the tree of candidates is almost a chain. Each vertex is either a leaf or has at least one son whose is a leaf. Thus after pebbling all leaves, we obtain exactly one chain. On this chain, all non-chained candidates correspond to sides of a polygon. That is, using the same notation as in Section 4, all p_i are sides (see also figure 5). So, our problem is reduced to finding an optimal triangulation below h_1 (the root of the tree). We can solve it in a similar way as in Section 4 to get an $O(n^2)$ total work algorithm. But since we are interested only in the value $c(h_1, h_1)$, we can reduce our problem to the single-source minimum path problem. And since in our acyclic digraph the weight matrix is convex, we can use algorithm from Fact 2.2 [CL-90]. It gives us an $O(\log^2 n)$ time and n processors CREW PRAM algorithm. But unfortunately, the preprocessing for this problem (i.e., computing weights in the graph) seems to need $O(n^2)$ work.

To compute value *bottom*(i) we have to compute the sum : $\sum_{r=i+1}^{k+2} c(h_i, p_r)$. But in this case each value $c(h_i, p_r)$ denotes the cost of a triangle. In fact we can write this sum in the following way - $\sum_{r=i+1}^{k+2} v_i w_r w'_r$, where w_r, w'_r denote the weights on the side p_r. Moreover we get.

$$\sum_{r=i+1}^{k+2} v_i w_r w'_r = \sum_{r=1}^{k+2} v_i w_r w'_r - \sum_{r=1}^{i} v_i w_r w'_r$$

Thus let us denote $LSum(i) = \sum_{r=1}^{i} w_r w'_r$. Now it is clear that

$$bottom(i) = v_i(LSum(k+2) - LSum(i))$$

And since we can compute all values of the array *LSum* in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM, with the same bound we can compute all values in the array *bottom*.

In a similar manner we can compute the array *fan*. From the definition we get.

$$fan(i,j) = \triangle(h_i, h_j) + \sum_{r=i+1}^{j} c(h_i, p_r)$$

Let us denote $USum(i) = \sum_{r=i+1}^{k+2} w_r w'_r$. This gives us the following formula for *fan*.

$$fan(i,j) = \triangle(h_i, h_j) + v_i(\mathrm{LSum}(k+2) - \mathrm{LSum}(i) - \mathrm{USum}(j))$$

Thus we can in sequential constant time with one processor compute the entry $fan(i,j)$. Hence instead of holding these values in the array we can compute them every time when they are needed.

To summarize the discussion above, we can compute the cost of an optimal triangulation of a monotone polygon in $O(\log^2 n)$ time using n processors on a CREW PRAM. And it is easy to see that we can find this partition in $O(\log n)$ time using $n/\log n$ processors on a CREW PRAM. This gives us the following theorem.

Theorem 2
The problem of finding an optimal triangulation of a monotone polygon can be solved in $O(\log^2 n)$ time using n processors on a CREW PRAM.

References

[AHU-74] A.V. Aho, J.E. Hopcroft, J.D. Ullman, "The design and analysis of computer algorithms", Addison-Wesley, 1974.

[AKLMT-89] M.J. Atallah, S.R. Kosajaru, L.L. Larmore, G.L. Miller, S-H. Teng, "Constructing trees in parallel", *Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 421–431.

[AK-90] M.J. Atallah, S.R. Kosajaru, "An efficient algorithm for the row minima of a totally monotone matrix", *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, pp. 394–403, and also Purdue CS Tech. Rept. 959 (2/28/90).

[Br-74] R.P. Brent, "The parallel evaluation of general arithmetic expressions", Journal of the ACM, Vol. 21, 1974, pp. 201–206.

[CL-90] K.F. Chan, T.W. Lam, "Finding least-weight subsequences with fewer processors", *Proceedings of the 1st SIGAL International Symposium on Algorithms*, 1990, Lecture Notes in Computer Science 450, Springer-Verlag, pp. 318–327.

[CV-86] R. Cole, U. Vishkin, "Approximate and exact parallel scheduling with applications to list, tree and graph problems", *Proceedings of the 27th Annual IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 478–491.

[Cz-92] A. Czumaj, "An optimal parallel algorithm for computing a near-optimal order of matrix multiplications", manuscript, February 1992, also to appear in *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*, 1992.

[GP-92] Z. Galil, K. Park, "Parallel dynamic programming", manuscript, 1992.

[GiRy-86] A.M. Gibbons, W. Rytter, "An optimal parallel algorithm for the dynamic expression evaluation and its applications", *Proceedings of Symposium on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, 1986, pp. 453–469.

[GR-88] A.M. Gibbons, W. Rytter, "Efficient parallel algorithms", Cambridge University Press, 1988.

[HL-87] D.S. Hirschberg, L.L. Larmore, "The least weight subsequence problem", SIAM Journal of Computing, Vol. 16, No. 4, 1987, pp. 628–638.

[HS-80] T.C. Hu, M.T. Shing, "Some theorems about matrix multiplications", *Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science*, 1980, pp. 28–35.

[HS-82] T.C. Hu, M.T. Shing, "Computation of matrix chain products. Part I", SIAM Journal of Computing, Vol. 11, No. 2, 1982, pp. 362–373.

[KK-90] M.M. Klawe, D.J. Kleitman, "An almost linear time algorithm for generalized matrix searching", SIAM Journal of Discrete Mathematics, Vol. 3, No. 1, 1990, pp. 81–97.

[KR-90] R.M. Karp, V. Ramachandran, "A survey of parallel algorithms for shared-memory machines", In *Handbook of Theoretical Computer Science*, North-Holland, 1990, pp. 869–941.

[MR-85] G.L. Miller, J.H. Reif, "Parallel tree contraction and its applications", *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, 1985, pp. 478–489.

[Ry-85] W. Rytter, "Remarks on pebble games on graphs", Conference on Combinatorial Analysis and its Applications 1985, *also in Zastosowania Matematyki, Vol. XIX, No. 3-4, 1987, pp. 569–577.*

[Ry-88] W. Rytter, "On efficient parallel computations for some dynamic programming problems", Theoretical Computer Science, Vol. 59, 1988, pp. 297–307.

[TaVi-85] R.E. Tarjan, U. Vishkin, "An efficient parallel biconnectivity algorithms", SIAM Journal of Computing, Vol. 14, No. 4, 1985, pp. 862–874.

[Wil-88] R. Wilber, "The concave least weight subsequence problem revisited", Journal of Algorithms, Vol. 9, No. 3, pp. 418–425.

[Yao-82] F.F. Yao, "Speed-up in dynamic programming", SIAM Journal on Algebraic and Discrete Methods, Vol. 3, No. 4, 1982, pp. 532–540.