ALGORITHMS FOR PARALLEL AND SEQUENTIAL MATRIX-CHAIN PRODUCT PROBLEM

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by
Ting Wang
November, 1997
ACKNOWLEDGMENTS

First I wish to thank my advisor, David Juedes, for his constant patience and support, and his timely suggestions. The time spent discussing problems together with Dr. Juedes was my most enjoyable time at Ohio University, and it is also where I learned the most.

I also wish to thank my parents, my wife and my sister for their constant support and encouragement.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The matrix chain product problem</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Parallel matrix chain multiplication problem</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Classification of parallel computer models and parallel algorithms</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Parallel random access machines and parallel algorithms</td>
<td>6</td>
</tr>
<tr>
<td>2. PARALLEL ALGORITHMS FOR THE MATRIX CHAIN PRODUCT PROBLEM</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Fundamental concepts and related theory</td>
<td>9</td>
</tr>
<tr>
<td>2.2 A heuristic sequential near-optimal algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.3 A very fast near optimal parallel algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3. PARALLEL MATRIX CHAIN PRODUCT ALGORITHM</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Some new notations for parallel polygon triangulation</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Perfectly-balanced polygon parallel triangulation</td>
<td>31</td>
</tr>
<tr>
<td>3.4 Polynomial algorithm for parallel matrix chain product problem</td>
<td>34</td>
</tr>
<tr>
<td>3.5 Parallel polygon triangulation and GNUPLGOT</td>
<td>36</td>
</tr>
<tr>
<td>4. CONCLUSION AND FUTURE RESEARCH</td>
<td>40</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY .......................................................... 42

APPENDICES

A. THE SOURCE CODE IN C ............................................. 45
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>\textit{A fan}</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>\textit{h-arc and v-arc}</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>\textit{cutoff(v_t)}</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Basic polygon</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Cutting a polygon into basic subpolygons</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Model for computing matrix chain product on parallel machine</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Parse tree for the matrix chain multiplication ordering</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Parallel computing using recursive doubling algorithm</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Example for Polygon triangulation</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>A completely-optimal triangulation and a potentially-optimal triangulation</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Parse tree for a multiplication ordering of a matrix chain</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>The relation between matrix chain multiplication ordering and polygon</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>GNUPLOT for polygon triangulation</td>
<td>39</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Some problems of practical interest can be described in terms of matrix multiplication. In particular, it is natural to describe hierarchical approaches to automatic differentiation in terms of multiplying Jacobian matrices [4]. To see this, assume that a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ can be naturally broken up into $k$ functions $f_1 : \mathbb{R}^n \rightarrow \mathbb{R}^{n_1}$, $f_2 : \mathbb{R}^{n_1} \rightarrow \mathbb{R}^{n_2}$, $\ldots$, $f_k : \mathbb{R}^{n_{k-1}} \rightarrow \mathbb{R}^m$ such that $f$ is the composition of these functions, i.e., $f = f_1 \circ f_2 \circ \ldots \circ f_k$. In this case, we can compute the full Jacobian of $f$ by first computing the individual Jacobians matrices for the functions $f_1, \ldots, f_k$ and then applying the chain rule by multiplying the matrices together. The amount of parallel or sequential time that such a strategy uses may depend greatly on the cost of multiplying these matrices. For this reason, we are interested reducing the cost of matrix multiplication.

Matrix multiplication obeys an associative law, and therefore the order of multiplication does not affect the result. However, the cost of matrix multiplication greatly depends on the order in which the matrices are multiplied. To see this, consider ten matrices of size $100 \times 100$ and a single matrix of size $100 \times 1$. Multiplying these eleven matrices from left to right via the standard algorithm takes $8 \times 100^3 + 100^2$ multiplications. Multiplying the matrices from right to left via the standard algorithm takes only $9 \times 100^2$ multiplications, a one hundred fold decrease.
1.1 The matrix chain product problem

The problem of computing an optimal order of matrix multiplication (the matrix chain product problem) is defined as follows. Consider the evaluation of the product of \( n - 1 \) matrices

\[
M = M_1 \times M_2 \times \cdots \times M_{n-1}
\]

where \( M_i \) is a \( w_i \times w_i \) matrix. Since matrix multiplication satisfies the associative law, the final result is the same for all multiplication orderings. 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 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 \( p \times q \times r \).) It was shown [5] that an arbitrary order of matrix multiplications may be as bad as \( O(T_{opt}^3) \), where \( T_{opt} \) is the minimal number of operations required to compute matrix chain products.

One can show that this problem is equivalent to the problem of finding an optimal triangulation of a convex polygon [17] in the case where the cost of a triangle is the product of the weights at each vertex of triangle. A triangulation of a polygon \((v_0, v_1, \cdots, v_n)\) is a set of chords that divides the polygon into triangles. The optimal triangulation of a convex polygon is one where the total cost of the triangulation is the smallest possible, with the total cost of a triangulation being the sum of the costs of the triangles.

For the matrix chain product and polygon triangulation problems, the best known sequential algorithm takes \( O(n \log n) \) steps [17]. Since it is known that a lower bound for this problem is \( \Omega(n \log n) \) [21], research on the sequential algorithms for these
problems is essentially complete. More recent research has concentrated on parallel algorithms for these problems. Currently, the best known parallel algorithm for the problem takes $O(\log^4 n)$ parallel steps using $n$ processors on a CREW PRAM.

1.2 Parallel matrix chain multiplication problem

A natural extension to the matrix chain multiplication problem is to consider the cost of matrix multiplication on parallel machines, where a single machine can perform a single matrix multiplication. This gives the parallel matrix chain multiplication problem, which we define below.

The parallel cost of a matrix product $(A_i \cdot A_{i+1} \cdots A_j) \cdot (A_{j+1} \cdot A_{j+2} \cdots A_k)$ is the maximum parallel cost of computing the matrix chain multiplication $A_i \cdot A_{i+1} \cdots A_j$ and $A_{j+1} \cdot A_{j+2} \cdots A_k$ plus the cost of computing the outermost multiplication, i.e., $\text{rows}[A_i] \cdot \text{columns}[A_j] \cdot \text{columns}[A_k]$. For example, if $A_1, A_2, A_3, A_4$ are $3 \times 5, 5 \times 4, 4 \times 10, 10 \times 3$ matrices respectively, then the parallel cost of computing $(A_1 A_2) \cdot (A_3 A_4)$ is $\max\{3 \times 5 \times 4, 4 \times 10 \times 3\} + 3 \times 4 \times 3$. Given such a notion of parallel cost, the definition of the parallel matrix chain multiplication problem is straightforward.

Let us define $C_p[i, j]$ to be the minimum parallel cost of the matrix chain multiplication $A_i A_{i+1} \cdots A_j$. Then, the problem of finding a multiplication ordering that obtains the minimum parallel cost $C_p[1, n]$ is the parallel matrix chain multiplication problem. Notice that the above definition of parallel gives a natural dynamic programming solution to the optimization problem that requires $O(n^3)$ steps.

Correspondingly, a parallel polygon triangulation problem may be defined as follows. Let $P = \langle v_0, v_1, \cdots, v_r, \cdots, v_{n-1}\rangle$, be a convex polygon with corresponding
weights \( w_i \) at each vertex and a weight function \( W \) defined on triangles formed by sides and chords of \( P \) as \( W(\Delta v_i v_j v_k) = w_i \cdot w_j \cdot w_k \). If the \( n \)-gon is partitioned in order, then the \emph{out-most cut} of a polygon is defined as the first cut point (vertex \( v_i \)) that divides the original polygon into three parts: the triangle \( \Delta v_0 v_i v_{n-1} \), the \((i+1)\)-gon \( \langle v_0 v_1, \ldots, v_i \rangle \) (called \textit{left-subpolygon}) and \((n-i)\)-gon \( \langle v_i, \ldots , v_{n-1} \rangle \) (called \textit{right-subpolygon}). (Note that both the left-subpolygon and the right-subpolygon are called side-subpolygons.) The \textit{parallel cost} \( C_p[0, n - 1] \) of the polygon triangulation, \( P = \langle v_0, v_1, \ldots, v_i, \ldots, v_{n-1} \rangle \), with \( v_i \) as the outermost cut, is the maximum of the parallel cost the left subpolygon(\( \langle v_0, \ldots, v_i \rangle \)) and the parallel cost right subpolygon(\( \langle v_i, \ldots , v_{n-1} \rangle \)) plus the weight of the triangle \( \Delta v_0 v_i v_{n-1} \), i.e.,

\[
C_p[0, n - 1] = \text{Max}\{C_p[0, i], C_p[i, n - 1]\} + W(\Delta v_0 v_i v_{n-1}).
\]

The problem of finding a polygon triangulation of minimum parallel cost \( C_p[0, n - 1] \) for the polygon \( \langle v_0, v_1, \ldots , v_{n-1} \rangle \) is called the \textbf{parallel polygon triangulation problem}. As in the sequential case, it can be shown that the \textit{parallel polygon triangulation problem} is equivalent to the \textit{parallel matrix chain multiplication problem}.

The parallel version of the matrix chain product problem and its equivalent polygon triangulation version originate from the problem of computing matrix chain products on MIMD parallel machines with a tree architecture. We describe this problem further in Chapter 4.

1.3 Classification of parallel computer models and parallel algorithms

Parallel algorithm design often depends on the available parallel architectures. In a situation of rapid development in parallel computing, it is not surprising that no
entirely satisfactory taxonomy of machines has yet been established. Two kinds of early classifications are very useful. Their basic concepts are still very popular today, even though they do not adequately reflect current architecture design. One kind of classification was established by Flynn (1966) [19]. He classified computers into four types based on the number of active instruction streams and data streams:

- SISD: single instruction stream, single data stream
- SIMD: single instruction stream, multiple data stream
- MISD: multiple instruction stream, single data stream
- MIMD: multiple instruction stream, multiple data stream

Another classification, suggested by Gurd (1988) [14], is a refinement of Flynn's. Rather than concentrating on the number of active instruction streams, the emphasis is put on the relationship between processing elements, memory modules, and interconnection network. Gurd classifies parallel computers into two types: integrated memory, and shared memory. In the former case, there is no memory in the system other than that which is locally addressable within the processing elements. In the latter case, there is a central memory (global or shared) that can be addressed from any of the processing elements, irrespective of whether or not they contain their own locally addressable memory.

A more modern classification [9] divides parallel machine models into two classes. One class is the multicomputer. A multicomputer is comprised of a number of Von Neumann computers linked by an interconnection network. In this model, each computer executes its own program. This program may access local memory and may
send and receive messages over the network. Messages are used to communicate with other computers or equivalently, to read and write remote memories. In the idealized network, the cost of sending a message between two nodes is independent of both node location and other network traffic, but does depend on message length. This architecture is most similar to what is often called the distributed-memory MIMD computer. However, the cost of sending a message between two nodes in a MIMD computer may not be independent of node location and other network traffic.

Another important class of parallel computer is the multiprocessor, or shared-memory MIMD computer. In multiprocessors, all processors share access to a common memory, typically via a bus or a hierarchy of buses. In the idealized Parallel Random Access Machine (PRAM), often used in theoretical studies of parallel algorithms, any processor can access any memory element in the same amount of time.

1.4 Parallel random access machines and parallel algorithms

The PRAM is a theoretical machine which abstracts away many of the architectural and hardware constraints of parallel machines. It offers a powerful, architecture-independent programming model without imposing limitations on parallelism, and therefore allows parallel computation to be studied. The PRAM model essentially consists of a (possibly infinite) number of processors, accessing and communication via a global memory (possibly of infinite size). The processors operate in lock-step. In each step of the computation, every processor performs a read-memory, compute, write memory cycle. Memory accesses are of uniform cost and take unit time. There is no notion of memory locality.
Most theoretical parallel algorithms study are based on the PRAM model. These algorithms are classified as the following four types [7].

- **EREW**: exclusive read and exclusive write,
- **CREW**: concurrent read and exclusive write,
- **ERCW**: exclusive read and concurrent write, and
- **CRCW**: concurrent read and concurrent write.

Of these types of algorithms, CRCW and CREW are the most popular. A PRAM that supports only CREW algorithms is called an CREW PRAM, and one that supports CRCW algorithms is called a CRCW PRAM.
2. PARALLEL ALGORITHMS FOR THE MATRIX CHAIN PRODUCT PROBLEM

The first polynomial time algorithm for the matrix chain product problem was discovered by Godbole [12] and ran in $O(n^3)$ time. Later Hu and Shing [17] gave an $O(n \log n)$ sequential time algorithm. This algorithm was later proven to be optimal [21]. Unfortunately, all of the above algorithms seem to be highly sequential.

The first approach to design a parallel algorithm for these problems is based on dynamic programming. Using this technique, Valiant et al. [27] gave an algorithm which runs in $O(\log^2 n)$ time using $n^9$ processors. This result was improved by Rytter [24] to $n^6/\log n$ processors on a CREW PRAM. Huang et al. [15] and Galil and Park [10] modified Rytter’s algorithm and reduced the number of processors to $n^6/\log^6 n$ and $n^8/\log^8 n$ respectively. The most recent research on the parallel algorithms of these problem is based on Hu and Shing’s sequential algorithm. A. Czumaj [8] and P. Ramanan [22] gave algorithms which run in $O(\log^3 n)$ time using $O(n^2/\log^3 n)$ processors and $O(\log^4 n)$ time using $n$ processors, respectively.

It is easy to see that there is a gap between the best sequential and the best parallel algorithm. One approach to attacking this gap is to design fast parallel algorithms that produce near optimal results. In the sequential case, F. Y. Chin [6] gave an $O(n)$ sequential algorithm for the matrix chain multiplication problem that produced a solution with an error ratio of 25%, i.e., it is guaranteed to produce a
solution that is within 1.25 times the optimal cost. Based on Chin’s algorithm, Hu and Shing [16] reduced the error ratio to 15%. Here we present a new, very fast parallel algorithm to find the near-optimal computation orderings of matrix chain products. The new algorithm has the same error ratio as Hu and Shing’s sequential approximate algorithm.

2.1 Fundamental concepts and related theory

_Fan_ is an important concept in the polygon triangulation problem. It is defined as follows.

**Definition 1** A fan of a (sub)polygon $P = (v_0, v_1, \cdots, v_i, \cdots, v_n)$ is a triangulation of the (sub)polygon in which the vertex $v_0$ of smallest weight is connected to all the other vertices (see figure 2.1); $v_0$ is called the center of the fan. We use $F(v_0|v_1, \cdots, v_i, \cdots, v_n)$ to denote the fan of $P$; $C_F(v_0|v_1, \cdots, v_i, \cdots, v_n)$ denotes its cost. We have that

$$C_F(v_0|v_1, \cdots, v_i, \cdots, v_n) = w_0 \times \sum_{i=1}^{n} w_i \times w_{i+1}$$

where $w_i$ is the weight of vertex $v_i$.

In any partition of an $n$-gon, every arc dissects a _unique_ quadrilateral. Let $V_x, V_y, V_z, V_w$ be the four vertices of an inscribed quadrilateral and let $V_x-V_z$ be the arc which dissects the quadrilateral (see figure 2.2), then we have the following definition.

**Definition 2** $V_x-V_z$ is said to be a vertical arc if it satisfies

$$\min(W_x, W_z) < \min(W_y, W_w)$$
Figure 2.1 A fan

or

\[
\min(W_x, W_z) = \min(W_y, W_w), \quad \max(W_x, W_z) \leq \max(W_y, W_w).
\]

\(V_x-V_z\) is said to be a horizontal arc if it satisfies

\[
\min(W_x, W_z) > \min(W_y, W_w), \quad \max(W_x, W_z) < \max(W_y, W_w).
\]

For brevity, we use the terms h-arc and v-arc to denote horizontal arcs and vertical arcs.

After we defined the h-arc and v-arc, we have the following important lemma [17].

**Lemma 2.1 (Hu and Shing[17])** All arcs in an optimum partition must be either vertical arcs or horizontal arcs.

Another important concept which we want to introduce here is potential horizontal arc (ph-arc). It is defined as follows.

**Definition 3** An arc \( r = v_i-v_j \), \(1 \leq i < j \leq n\), is called a potential horizontal arc (ph-arc) if \( w_k \geq \max(w_i, w_j) \) for all \( k, i < k < j \). A potential h-arc will dissect a polygon
into two parts, and the subpolygon which contains the larger vertices is called the upper subpolygon.

A set of arcs are called compatible if all the arcs can exist simultaneously in a partition. Then we have the following lemma:

**Lemma 2.2** All the ph-arcs are compatible.

Proof: The proof is by contradiction. Assume that there exist two ph-arcs $v_i-v_j$ and $v_p-v_q$ that are not compatible (intersect each other). Then we must have either $p < i < q, j > q$ or $i < p < j, q > j$. In both cases at least one of $v_i-v_j$ and $v_p-v_q$ cannot be a ph-arc. □

**Lemma 2.3 (Hu and Shing [17])** If none of the ph-arcs appears in the optimum partition of $n$-gon, the optimum partition must be the fan of the $n$-gon.

The following problems, which we need in our algorithm, are called a class of minimum computation problems.
• **The all nearest smaller value pairs problem**

Let \( A = (a_0, a_1, \cdots, a_n) \) be an array. For each \( a_i (0 \leq i \leq n) \), find the nearest element to its left (and the nearest element to its right) that is less than \( a_i \), if such an element exists. That is, for each \( i, 1 \leq i \leq n \), find the maximal \( j \) \((0 \leq j < i)\) and the minimal \( k \) \((i < k \leq n)\) such that \( a_j < a_i \) and \( a_k < a_i \).

• **The prefix minima problem**

Let \( A = (a_0, a_1, \cdots, a_n) \) be an array. For each \( a_i (0 \leq i \leq n) \), find the minimum among \( a_0, a_1, \cdots, a_i \).

### 2.2 A heuristic sequential near-optimal algorithm

Chin \[6\] was the first to give a linear-time \( O(n) \) algorithm for the matrix chain multiplication problem. Hu and Shing \[16\] improved this algorithm by reducing the error ratio. Hu and Shing’s algorithm is based on the following intuitions.

- If a vertex has a very large weight, then it should be cut in the optimal partition.
- If none of vertices has a very large weight, then we should join the smallest vertex with all other vertices.

In their algorithm, the following notation is important.

• **Large and Primary h-arc**

Let \( v_m \) be the smallest vertex of a convex polygon, and \( v_t \) be a vertex with \( v_k \) and \( v_c \) as its two neighbors. Define vertex \( v_t \) to be *large* if and only if

\[
\frac{1}{v_k} + \frac{1}{v_c} > \frac{1}{v_t} + \frac{1}{v_m}.
\]
From above formula, we must have $v_t > v_k$ and $v_t > v_c$. The vertex $v_t$ is called the *local maximum*. If we join $v_k$ and $v_c$, the arc $v_k - v_c$ is called a primary $h$-arc.

- Procedure $\text{cutoff}(v_t)$:

  Let $v_k$ and $v_c$ be two neighbors of vertex $v_t$. Join $v_k$ and $v_c$, i.e., cut off $v_t$. (See figure 2.3)

![Figure 2.3 $\text{cutoff}(v_t)$](image)

- Procedure $\text{fan}(v_t)$ and $\text{fan-arc}$

  Join $v_t$ with all the other vertices in the stack. These arcs are called $\text{fan-arcs}$.

- $Pop(v_i)$

  Pop a given vertex $v_i$ off the stack.

- $Push(v_i)$

  Push a given vertex $v_i$ onto the stack.
• Top

Return the top of the stack.

Both Chin, and Hu and Shing describe their heuristic linear-time algorithm as follows.

**Hu and Shing’s Algorithm**

The *input* is a sequence of weights \( w_0, w_1, \ldots, w_{n-1} \) and a polygon \( \langle v'_0, v'_1, \ldots, v'_{n-1} \rangle \).

The first step of the algorithm is to find the smallest vertex \( v'_i \) (with the smallest weight \( w_i \)) and shift the indices of vertices so that \( w_i \) is the first element of a sequence, that is, \( v_0 = v'_i, v_1 = v'_{i+1} \mod n, \ldots, v_j = v'_{j+1} \mod n, \ldots \). The remainder of the algorithm follows.

```plaintext
Begin
    push(v0);
    i = 1;
    for i <= n - 1 do
        if top != v0 and top is large then
            cutoff(top); /* produce a h-arc */
            pop(top);
        else
            push(vi);
        endif
        i++;
    end
    fan(v0) /* finish triangulation */
end
```

What is especially interesting is that, in most cases, Hu and Shing’s algorithm yields the optimal solution or a solution which takes only a few percent worse than the optimal one. The error ratio is inversely proportional to \( n \). In general, if the value
$t = \max_i \{ \frac{w_i}{v_0} \}$ has an upper bound, then Hu and Shing [16] give the maximum error ratio for any given input $n$. This ratio is given by the following formula.

$$\max \text{imum error ratio} \quad R = \frac{t - 1}{t^2 + t + (n - 4)}$$

From Hu and Shing's algorithm, we have the following important observations.

**Observation 2.1** Any arc in the partition constructed by Hu and Shing's algorithm is either a *primary h-arc* or a *fan-arc*, and once all the primary $h$-arcs are fixed (yielded by *cutoff procedure*), the whole triangulation is fixed.

**Observation 2.2** All the primary $h$-arcs constructed by Hu and Shing's algorithm are also $p+h$-arcs.

**Observation 2.3** All the fan-arcs constructed by Hu and Shing's algorithm are also $v$-arcs.

**Observation 2.4** During the execution of the loop in Hu and Shing's algorithm, there is always only one non-partitioned convex polygon, which always contains the smallest weight vertex $v_0$. The procedure $\text{cutoff}(v_t)$ is called if and only if $v_t$ is large.

2.3 A very fast near optimal parallel algorithm

The algorithm which we mention here is based on Hu and Shing's sequential algorithm. From the observations in last section, we can see that the key step in Hu and Shing's algorithm is finding all the *primary $h$-arcs*. After all the primary $h$-arcs are fixed, we can just call $\text{fan}(v_0)$ procedure to finish the partition. We know that all the primary $h$-arcs are $p+h$-arcs, i.e., the the set of primary $h$-arcs is a subset of the
set of ph-arcs. This means that after we find all the ph-arcs of a convex polygon, we can find all primary h-arcs by checking each ph-arc to see if it is a primary h-arc. To find all the ph-arcs of a given convex polygon, we use the following lemma.

**Lemma 2.4 (Ramanan [22])** Finding all the ph-arcs of a given convex polygon is equivalent to all nearest smaller value pairs problem.

The remaining problem is to determine whether a ph-arc is a primary h-arc. For this, we use the following lemma.

**Lemma 2.5** Let \((v_k, v_c)\) be a primary h-arc obtained by Hu and Shing’s algorithm, then all the ph-arcs \(v_i - v_j\), for \(k \leq i < j \leq c\), are primary h-arcs.

Proof: If \((v_k, v_c)\) is a primary h-arc, then there is no fan-arc which connects \(v_0\) with any \(v_i\), where \(k < i < c\). So, all of the upper subpolygon must be triangulated by primary h-arcs.

We can implement Hu and Shing’s heuristic algorithm by solving the following problems.

**The Basic polygon triangulation problem**

Hu and Shing [17] define a basic polygon to be a convex polygon \(P = (v_0, v_1, \ldots, v_{n-1})\), where \(v_0\) has the smallest weight, \(v_1\) has the second smallest weight, and \(v_{n-1}\) has the third smallest weight. (See figure 2.4.) We have the following observation.

**Observation 2.5** The arc \(v_1 - v_{n-1}\) is a ph-arc.

We can prove the following crucial lemma.

**Lemma 2.6** The arc \(v_1 - v_{n-1}\) is also a primary h-arc.

Proof: (By contradiction). The key point is to prove that the procedure \(cutoff(v_i)\) is
performed for every vertex $v_i$ except for $v_1$ and $v_{n-1}$. To see this, notice that every vertex $v_i$, with $1 < i < n - 1$ is bigger than either $v_1$ or $v_{n-1}$. Thus, neither $cutoff(v_1)$ nor $cutoff(v_{n-1})$ can ever be executed. It follows that the vertices $v_0, v_1, \text{ and } v_{n-1}$ will always be kept in the remaining subpolygon until the loop of Hu and Shing's algorithms reaches the point $v_{n-1}$.

To see that $cutoff(v_i)$ is called for other vertices, assume that the current subpolygon during the execution Hu and Shing's algorithm is $P' = (v_0, v_1, v_i, \ldots, v_j, v_{n-1})$. It must be the case that some vertex $v_i$ between $v_1$ and $v_{n-1}$ is large in the sense that its nearest neighbors have smaller weights. (Otherwise, the weights of the vertices would satisfy $w_1 < w_i < \ldots < w_j < w_{n-1}$, which is impossible.) Hu and Shing's algorithm will then call $cutoff(v_i)$. This process continues until 4 vertices $v_0, v_1, v_j, v_{n-1}$ remain. At this point, Hu and Shing's algorithm calls $cutoff(v_j)$ and thus creates the edge $v_1 - v_{n-1}$.

From the above lemmas, we are lead to the following important theorem.
Theorem 2.1 All the ph-arcs in the basic polygon are also primary h-arcs.

Cutting a polygon into basic subpolygons
We can partition a convex polygon into smaller non-intersecting basic subpolygons as follows.

1. Divide a polygon into two parts by joining the smallest vertex with the second smallest one.

2. Consider two obtained subpolygons independently.

3. Find the third smallest vertex in each part, and connect them with the smallest vertex.

4. Execute the above procedure repeatedly.

Then the following lemma will be very helpful to solve this problem.

Lemma 2.7 Cutting a polygon into basic subpolygons is equivalent to the prefix minima problem.

Proof: (See figure 2.5.) The above process is equivalent to the following procedure.

1. Find the prefix minima in the whole range except the smallest one $v_0$. Rename the vertex $v_{n-1}$, and connect $v_{n-1}$ to $v_0$. This cuts the original polygon into two subpolygons.

2. Rearrange the indices of vertices in both subpolygons, so that they satisfy the order $v_0, v_1, \ldots, v_{n-1}$.

3. Join $v_i$ with $v_0$ if and only if this vertex is minimal in the range $v_1, \ldots, v_i$. 
Figure 2.5 Cutting a polygon into basic subpolygons

Obviously, it is equivalent to solving the prefix minima problem.

If the arc $v_0-v_i$ (including $v_0-v_{n-1}$) is also an arc in the original convex polygon, we call it primary $v$-arc. The following lemma is also important.

**Lemma 2.8** All the primary $v$-arcs are also fan-arcs obtained by Hu and Shing's algorithm.

Proof: (By Contraposition) If the arc $v_0-v_i$ is not a fan-arc in the algorithm, then the procedure cutoff($v_i$) must be called in the loop of Hu and Shing's algorithm. This means that there must exist two vertices $v_k$ and $v_c$, where $k < i$, and $c > i$, such that $v_i > v_k$, and $v_i > v_c$. This is impossible if $v_i$ is the prefix minima. So the primary $v$-arc must be also a fan-arc.

We produce the same partition as in Hu and Shing's algorithm by cutting the polygon into basic subpolygons and then finding the all $ph$-arcs in each basic sub-polygon. We also map these subproblems into the *all nearest smaller value pairs problem* and the *prefix minima problem*. We also know that for the convex polygon
triangulation problem, all the weight input values are integers, and we can assume the input integer value in the domain \([1...s]\). After introducing the following important lemmas given by O. Berkman [2], we will reach the final theorem.

**Lemma 2.9 (O. Berkman)** The all nearest smaller value pairs problem with integer input values in the domain \([1...s]\) can be solved in \(O(\log \log s + \log^* n)\) running time using \(n/\log \log \log s\) processors on a **CRCW PRAM**. (The function \(\log^* n\) is defined as follows. Let the function \(\log^{(i)} n\) be defined as \(\log(\log^{(i-1)} n)\) for \(i > 1\), and let \(\log^{(1)} n = \log n\). Then, \(\log^* n = \min i : \log^{(i)} n \leq 2\). This function is very slowly increasing.)

**Lemma 2.10 (O. Berkman)** The prefix minima problem with integer input values in the domain \([1...s]\) can be solved in \(O(\alpha(n))\) running time using \(n/\alpha(n)\) processors on a **CRCW PRAM**, where \(\alpha(n)\) is the very slowly increasing Inverse-Ackerman function.

**Theorem 2.2** The problem of finding a near-optimal order of matrix chain products and the problem of finding a near-optimal partition of a convex polygon with the weight value in the domain \([1...s]\) can be solved in \(O(f(n) + \log \log \log s)\) running time using \(n/T(s,n)\) processors on a **CRCW PRAM**, where \(f(n)\) is either \(\log^* n\) or \(\alpha(n)\) and \(T(s,n) = \min(\log \log \log s, \alpha(n))\).

**Proof:** We can solve this problem using the following algorithm.

1. Cut the original polygon into basic subpolygons. According to the above lemma, this is equivalent to solving the prefix minima problem, and can be solved in \(O(\alpha(n))\) running time using \(n/\alpha(n)\) processors on a **CRCW PRAM**.
2. Find the \textit{ph-arcs} for each basic subpolygon. This is equivalent to solving the all nearest smaller value pairs problem. According to the above lemma, this can be solved in $O(\log \log \log s + \log^* n)$ running time using $n/\log \log \log s$ processors.

According to \textbf{Lemma 2.8} and \textbf{Theorem 2.1}, we know that all the primary $v-$arcs to cut the original polygon into basic subpolygons are also the fan-arcs obtained by Hu and Shing's algorithm and all the ph-arcs in the basic subpolygons are also primary h-arcs. So, by step 1 and 2 we get the same partition as Hu and Shing's algorithm. This means we get the same error ratio, i.e., 0.15, as Hu and Shing's algorithm.

When $n/T(s, n)$ processors are used, we can solve the whole problem in $O(f(n) + \log \log \log s)$ running time with an error ratio of 0.15. $\square$
3. PARALLEL MATRIX CHAIN PRODUCT ALGORITHM

3.1 Introduction

In chapter 1, we have mentioned about that parallel matrix chain product problem is originated from computing matrix chain product on MIMD parallel computers with parallel architecture. In fact, we can think of the problem as finding an optimal multiplication ordering under the following three assumptions.

1. A single matrix multiplication is done on one computer.

2. The communication cost between machines is free.

3. The machines are connected in a binary tree architecture.

To minimize the cost of computing the matrix chain $M_1 \times M_2 \times \cdots M_n$ on the parallel computers, how should we allocate the matrices in each computers?

Consider the following example.

Example 3.1 Computing the matrix chain $M = (M_1 \times M_2) \times (M_3 \times M_4)$, where $M_1, M_2, M_3, M_4$ are $3 \times 5, 5 \times 4, 4 \times 10$ and $10 \times 3$ matrices, on the parallel machine with binary tree architecture, as shown in figure 3.1.

Here, every node(or root) represents a Von Neumann computer. The total cost should be equal to the waiting cost (wait for the data to be ready) $C_{\text{wait}}$ plus the
computing cost $C_{comp}$. The total cost $Cost(root)$ to compute $M$ should follow the formula.

$$Cost(root) = C_{wait}(root) + C_{comp}(root).$$

For this case, the root has two children nodes, node1 and node2. The waiting cost of the root should be equal to the maximum of its two children cost, that is

$$C_{wait}(root) = Max\{Cost(node1), Cost(node2)\}$$

where

$$Cost(node1) = \text{the cost of multiplying } M_1 * M_2$$

and

$$Cost(node2) = \text{the cost of multiplying } M_3 * M_4.$$ 

As for $Cost(node1)$ and $Cost(node2)$, they also include two parts, $C_{wait}$ and $C_{comp}$, but they do not have any children. In this case, the value of $C_{wait}$ should be equal to zero.

Figure 3.1 Model for computing matrix chain product on parallel machine
Here, we have

\[ C_{\text{comp}}(\text{node1}) = 3 \times 5 \times 4 = 60, \]

\[ C_{\text{comp}}(\text{node2}) = 4 \times 10 \times 3 = 120, \]

and

\[ C_{\text{comp}}(\text{root}) = 3 \times 4 \times 3 = 36. \]

So, the total cost to compute \( A_{\text{il}} \) is

\[
\begin{align*}
\text{Cost}(&\text{root}) = \max\{\text{Cost}(&\text{node1}), \text{Cost}(&\text{node2})\} + C_{\text{comp}}(\text{root}) \\
&= \max\{C_{\text{comp}}(&\text{node1}), C_{\text{comp}}(&\text{node2})\} + C_{\text{comp}}(\text{root}) \\
&= \max\{60, 120\} + 36 \\
&= 156.
\end{align*}
\]

The result is just the same as what we have discussed in Chapter 1. In fact, the parallel matrix chain product problem (or parallel polygon triangulation problem) is an appropriate mathematical description of this case. Comparing this case with the parallel polygon triangulation problem, the \( \text{Cost}(\text{root}) \) can be seen as the total parallel cost of the polygon, the \( \text{Cost}(\text{node1}) \) can be seen as the parallel cost of the left-subpolygon, the \( \text{Cost}(\text{node2}) \) can be seen as the parallel cost of the right-subpolygon, and the \( C_{\text{comp}}(\text{root}) \) can be seen as the weight of the triangle.

After we have parenthesized the matrix chain to minimize the parallel cost of its objective function, we can represent the parenthesized matrix chain as a parse tree. Based on the parse tree, we know how we should allocate the matrices to each computer and the parallel schedule.
Example 3.2 Assume we want to compute the following matrix chain.

\[ M = A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7 \times A_8 \times A_9 \]

If the multiplication ordering of the matrix chain is \(((A_1 \times A_2) \times (A_3 \times A_4))(((A_5 \times A_6) \times A_7) \times (A_8 \times A_9))\), then we can represent it as the following parse tree. (See figure 3.2.) After we get the above parse tree, we can allocate the matrices to each computer as shown in figure 3.3, and evaluate \( M \) using recursive doubling algorithm as follows.

**Recursive Doubling Algorithm for Computing M**

- **Step 1:**
  - Node 3: \( A_1 \times A_2 \), store as \( M'_1 \).
  - Node 4: \( A_3 \times A_4 \), store as \( M'_2 \).
- Node 5: \((A_5 \times A_6) \times A_7\), store as \(M'_1\),

- Node 6: \(A_8 \times A_9\), store as \(M''_2\),

- **step 2:**
  - Node 1: \(M'_1 \times M'_2\), store as \(M'\),
  - Node 2: \(M''_1 \times M''_2\), store as \(M''\),

- **step 3:**
  - root: \(M' \times M''\), output result.

Computations within each step are performed simultaneously.

![Figure 3.3 Parallel computing using recursive doubling algorithm](image-url)

Here, we will discuss a sequential algorithm to obtain the optimal multiplication ordering of matrix chain based on minimizing the *parallel cost*. As we know, Hu and
Shing give a $O(n \log n)$ algorithm for sequential matrix chain product problem. The next question is whether we can transplant their algorithm to parallel case so that we can get a similar result. Unfortunately, parallel matrix chain product problem is very different from sequential matrix chain product problem. The cornerstone of Hu and Shing's theory, theorem 1 [17], does not hold in the parallel case. Here, we give a simple example to explain the difference between the two cases.

Assume we have a polygon partial triangulation as shown in figure 3.4.

![Figure 3.4 Example for Polygon triangulation](image)

For the sequential polygon triangulation case, we know that the total cost will contain the two items $W(\Delta v_1 v_2 v_3)$ and $W(\Delta v_1 v_3 v_4)$. But, for the parallel polygon triangulation case, we just know that the parallel cost will be bigger than the sum of the two items. We are not sure whether these two items will be counted or not in the total cost because we do not know if the out-most cutting point is $V_4$ or $V_5$. We also do not know which part (the left-subpolygon or the right-subpolygon) will have the biggest cost.
3.2 Some new notations for parallel polygon triangulation

**Definition 4** A triangulation of n-gon is said to be potentially-optimal if it obtains the optimal (minimized) parallel cost. If all the triangulations of its side-subpolygons are optimal, we call a triangulation completely-optimal. For brevity, we use \( p\text{-optimal} \) and \( c\text{-optimal} \) to denote potentially-optimal and completely-optimal. For example, see figure 3.5).

![Image of triangulation diagrams]

Figure 3.5 *A completely-optimal triangulation and a potentially-optimal triangulation*

**Example 3.3** Let \( P = (V_1, V_2, V_3, V_4, V_5, V_6, V_7) \) be a 7-gon with weights \( W = (11, 17, 18, 19, 20, 23, 25) \). The optimal out-most cut point is \( V_5 \).

- For the \( c\text{-optimal} \) case, we can evaluate the optimal parallel triangulation cost \( C_p[1, 7] \) as follows.

\[
C_p[1, 7] = \max\{C_p[1, 5], C_p[5, 7]\} + W(\Delta v_1v_5v_7)
\]
\[
\begin{align*}
&= \max\{W(\triangle v_1v_3v_5) + W(\triangle v_3v_4v_5), W(\triangle v_5v_6v_7)\} + W(\triangle v_1v_5v_7) \\
&= \max\{10800, 11500\} + 5500 \\
&= 17000
\end{align*}
\]

- For the \textit{p-optimal} case, we have the following.

\[
\begin{align*}
C_p[1, 7] &= \max\{C_p[1, 5], C_p[5, 7]\} + W(\triangle v_1v_3v_7) \\
&= \max\{W(\triangle v_1v_2v_3) + W(\triangle v_1v_3v_4) + W(\triangle v_1v_4v_5), W(\triangle v_5v_6v_7)\} \\
&\quad + W(\triangle v_1v_5v_7) \\
&= \max\{11308, 11500\} + 5500 \\
&= 17000
\end{align*}
\]

From this example, we can see that \textit{p-optimal} triangulation has the same out-most cut point and same parallel cost as its corresponding \textit{c-optimal} triangulation. In fact, for a \textit{c-optimal} with one side-subpolygon having bigger sub-parallel-cost, we can chose any triangulation for the other side-subpolygon, as long as the sub-parallel-cost of this triangulation remains less than or equal to the fixed sub-parallel-cost of the other side-subpolygon.

The existence of \textit{p-optimal} triangulations for the parallel polygon triangulation problem give us a chance to break the rule that the globally optimal solution must guarantee locally optimal solutions.

With the above definition, we have the following lemma.
Lemma 3.1 If a convex polygon $P = (v_0, v_1, \cdots, v_i, \cdots, v_{n-1})$ with weight vector $W = (w_0, w_1, \cdots, w_i, \cdots, w_{n-1})$ satisfies the condition

$$w_0 < w_1 < \cdots < w_i < \cdots < w_{n-1}$$

then its c-optimal triangulation will always contain the arc $v_0-v_2$. (Such a polygon is called a unimodal polygon.)

Proof: (by strong induction)

1. Base case: For the c-optimum partition of a triangle, the lemma holds. For the c-optimum partition of a 4-gon, only two cases are possible: a) $v_0$ connects $v_2$; or b) $v_1$ connects $v_3$. The parallel cost for case (a) is $W(\Delta v_0 v_1 v_2) + W(\Delta v_0 v_2 v_3) = w_0 w_1 w_2 + w_0 w_2 w_3$; the parallel cost of case (b) is $W(\Delta v_0 v_1 v_3) + W(\Delta v_1 v_2 v_3) = w_0 w_1 w_3 + w_1 w_2 w_3$. Obviously, we have $w_0 w_1 w_2 < w_0 w_1 w_3$ and $w_0 w_2 w_3 < w_1 w_2 w_3$. The case (a) must be the c-optimum partition.

2. Hypothesis: Assume the lemma holds for all $k$-gons ($3 \leq k \leq n - 1$).

3. Induction: Consider a $k + 1$-gon. No matter which point from $v_1$ to $v_k$ is the out-most cutting point, the left-subpolygon must be a $m$-gon with $m \leq k$. For a c-optimal triangulation, we must guarantee the $m$-gon triangulation is optimal. According to 2), the arc $v_0-v_2$ must be an arc in the optimal partition.

So, we have the lemma holds for every $n$-gon. 

The more interesting thing is that we cannot use this lemma recursively for a unimodal polygon in the parallel case. For the sequential polygon triangulation problem, we can easily use this lemma recursively to obtain a fan. But, for the parallel triangulation
case, the triangulation of an unimodal polygon is not always a fan. The key is we need to determine the out-most cutting point to use the lemma further.

3.3 Perfectly-balanced polygon parallel triangulation

A perfectly-balanced polygon is defined as a convex polygon with the same weight at each vertex. This corresponds to a matrix chain with every matrix having the same dimension. For the sequential case, an arbitrary triangulation or multiplication ordering should be optimal. But for parallel case, it makes a difference. An arbitrary triangulation can be as bad as $O(n/\log n \cdot C_{\text{opt}})$, where $C_{\text{opt}}$ is the optimal parallel cost. We have the following theorem.

**Theorem 3.1** If every matrix $A_i$ in the matrix chain $A_1, A_2, \ldots, A_i, \ldots, A_n$ has the same dimension $m$, or the perfect-balanced polygon $P = (v_0, v_1, \ldots, v_i, \ldots, v_{n-1})$ has the same weight $m$ at each vertex, then the optimal parallel cost is $\lceil \log n \rceil \cdot m^3$. We can use a $O(n)$ algorithm to find it.

**Proof:** The parallel polygon triangulation problem is equivalent to the parallel matrix chain product problem. So let us just consider the latter case. Any multiplication ordering of a matrix chain can be expressed as a parse tree (see figure 3.6): Because each matrix has same dimension, every pair of matrices has the same cost of $m^3$. The total parallel cost will only depend on the depth of the parse tree, and it is equal to the $\text{depth} \times m^3$. When the parse tree is a complete binary tree, it obtains its shortest height. In this case, the $\text{depth}$ is $\lceil \log n \rceil$. We can create such a multiplication ordering by cutting the matrix chain at the half point recursively. The running time of the
algorithm satisfies the following recurrence relation.

\[ T(n) = 2T\left(\frac{n}{2}\right) + c, \]

where \( c \) is a constant. The solution to this recurrence is \( T(n) = O(n) \). □

**Corollary 3.1** If the dimension of each matrix is bounded below by \( k \), then the optimal parallel cost of the matrix chain is bounded below by \( \lceil \log n \rceil \cdot k^3 \).

**Proof:** Let \( M \) be the original matrix chain, i.e., \( M = A_1 \times A_2 \times \cdots \times A_i \times \cdots \times A_n - 1 \).

Let \( M' \) be corresponding matrix chain, i.e., \( M' = A'_1 \times A'_2 \times \cdots \times A'_i \times \cdots \times A'_{n-1} \),

where for each \( i \), \( A_i \) has the dimension \( k \). Because the dimension of each matrix in \( M \) is bounded below by \( k \), the cost of every matrix multiplication in \( M \) is bounded
below by $k^3$. Let $C$ and $C'$ be the parallel cost of $M$ and $M'$ respectively when they have the same multiplication ordering as that given in Theorem 3.1. Then we must have $C \leq C'$, and $C' = \lfloor \log n \rfloor \cdot k^3$. Obviously, the optimal parallel cost of $M$ is bounded below by $C$. So, the corollary holds. □

**Corollary 3.2** If the dimension of each matrix is bounded above by $k$, then the optimal parallel cost of the matrix chain is bounded above by $[\log n] \cdot k^3$

**Proof:** Let $M$ be the original matrix chain, i.e., $M = A_1 \times A_2 \times \cdots \times A_i \times \cdots \times A_n = 1$. Let $M'$ be corresponding matrix chain, i.e., $M' = A'_1 \times A'_2 \times \cdots \times A'_i \times \cdots \times A'_{n-1}$, where for each $i$, $A_i$ has the dimension of $k$. Because the dimension of each matrix in $M$ is bounded below by $k$, the cost of multiplication for every pair of matrices in $M$ is bounded below by $k^3$. Let $C$ and $C'$ be the parallel cost of $M$ and $M'$ respectively when they have the same multiplication ordering as the optimal solution for $M$, i.e., $C$ is $C_{\text{opt}}$. Then we have: $C_{\text{opt}} \geq C$. We also know the optimal cost of $M'$ is $[\log n] \cdot k^3$, so $C \geq [\log n] \cdot k^3$. Therefore $C_{\text{opt}} \geq [\log n] \cdot k^3$. □

**Corollary 3.3** Let $k$ be the smallest dimension of any matrix, and $l$ be the largest dimension. We can use an $O(n)$ algorithm to produce a solution that is at most $(\frac{1}{k})^3$ times larger than the optimal solution.

**Proof:** According to the above theorem and corollaries, we can use an $O(n)$ algorithm to get a multiplication ordering whose parallel cost $C$ satisfies the following formula.

$$\lfloor \log n \rfloor \cdot k^3 \leq C_{\text{opt}} \leq C \leq [\log n] \cdot l^3.$$

Then, we have $C \leq (\frac{1}{k})^3 [\log n] \cdot k^3 \leq (\frac{1}{k})^3 \cdot C_{\text{opt}}$. □
We can see Corollary 3.3 as a near optimal algorithm, especially if $\frac{t}{k}$ is not too large. Obviously, it is far from tight.

### 3.4 Polynomial algorithm for parallel matrix chain product problem

One way to solve the parallel matrix chain product problem is the brute-force method of exhaustively checking all possible multiplication orderings. This approach takes $\Omega(4^n/n^{3/2})$ steps. So, it is a poor strategy for this problem. Fortunately, the parallel matrix chain product problem has the following two key ingredients: *optimal substructure* and *overlapping subproblems*. These make dynamic programming applicable.

Let $C[i,j]$ be the optimal parallel cost to compute the matrix $A_{i..j}$. Then, the total optimal parallel cost is $C[1,n]$. Assume that for each $i$, $A_i$ is $p_{i-1} \times p_i$ matrix. We define $C[i,j]$ as follows.

$$C[i,j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{\max(C[i,k], C[k+1,j]) + p_{i-1}p_kp_j\} & \text{if } i < j
\end{cases}$$

Let us define $S[i,j]$ to be a value of a $k$ at which we can split the product $A_iA_{i+1} \cdots A_j$ to obtain an optimal parallel multiplication ordering. The following pseudo-code will give an algorithm to determine the parallel optimal order of a matrix chain. It assumes that matrix $A_i$ has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, ..., n$. The input is a sequence $(p_0, p_1, ..., p_n)$, where $\text{length}[p] = n + 1$.

**MATRIX-CHAIN-ORDER(P)**

```pseudo
c = length[P] - 1
for i = 1 to n do
    begin
        C[i,i] = 0
```
for 1 = 1 to n - 1 do
begin
for i = 1 to n - 1 /*compute C[i,i+1] */
begin
j = i + 1
q = max(C[i,i], C[i+1,j]) + p(i-1)*p(i)*p(j)
k' = i
for k = i+1 to j-1 do
begin
q' = max(C[i,k], C[k+1,j]) + p(i-1)*p(k)*p(j)
if q > q' then
begin
q = q'
k' = k
end
end
C[i,j] = q
S[i,j] = k'
end /* for i loop */
end /* for l loop */
return C and S
End

If we only take multiplication as the basic operation, the running time $R(n)$ should satisfy the following formula.

$$R(n) = \sum_{l=1}^{n-1} \sum_{i=1}^{n-l} 2 + \sum_{k=i+1}^{i+l-1} 2 = \sum_{l=1}^{n-1} \sum_{i=1}^{n-l} 2l = 2 \sum_{l=1}^{n-1} (n-l)l$$

$$= 2n \sum_{l=1}^{n-1} l - 2 \sum_{l=1}^{n-1} l^2$$

$$= 2n \frac{n(n-1)}{2} - 2\left( \frac{(n-1)^3}{3} + \frac{(n-1)^2}{2} + \frac{n-1}{6} \right)$$

$$= n^3 - n^2 - \frac{2}{3}(n-1)^3 + (n-1)^2 + \frac{n-1}{3}$$

$$= \Theta(n^3).$$
So far, we have used the $\Theta(n^3)$ algorithm to obtain the total optimal parallel cost and the optimal multiplication ordering of a matrix chain. We also saved the results in the $C$ and $S$ tables. We can use the following recursive algorithm to print the multiplication ordering out.

**Parenth(i,j)**

**Begin**

if $i = j$ then print ``Ai``

else

print ``(``

parenth(i,S[i,j])

print ``)``

parenth(S[i,j]+1,j)

print ``)``

**End**

We have implemented the above algorithm using C. Let us give the following running example.

**Example 3.4** Let the matrix chain is $M$,

\[
M = A_7^{1} \times A_8^{2} \times A_9^{3} \times A_{10}^{4} \times A_{11}^{5} \times A_{12}^{6} \times A_{13}^{7}
\]

Then the **Input** is $P = (7, 8, 9, 10, 11, 12, 13, 14)$.

The **Output** is $(((A^1 A^2)A^3)(A^4 A^5))(A^6 A^7))$.

3.5 Parallel polygon triangulation and GNUPLOT

As we know, the parallel polygon triangulation problem is equivalent to parallel matrix chain product problem. It seems that when we obtain the optimal parallel multiplication ordering of a matrix chain, we also obtain the optimal parallel triangulation of its corresponding convex polygon. But what is the exact relation between
matrix chain multiplication ordering and polygon triangulation? How can we draw the triangulation using GNUPLTO\textsuperscript{2} based on the information we have obtained from the matrix chain multiplication ordering? GNUPLTO\textsuperscript{2} is a software tool which can link the points sequentially if given the coordinates of these points as input. In the last section, we obtained the parallel optimal order of the matrix chain multiplication ordering, and stored it in $S$ table. The $S$ table is actually a two dimension table. How can we arrange it in linear sequence so that GNUPLTO\textsuperscript{2} can use it?

First, let us determine the exact relation between matrix chain multiplication orderings and polygon triangulation. Look at the following example. (See figure 3.7.)

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.7.png}
\caption{The relation between matrix chain multiplication ordering and polygon triangulation}
\end{figure}

For the matrix chain $A_1 \cdots A_7$, we arrange the vertices $V_0 \cdots V_7$ so that each vertex corresponds to a position between the matrix and $V_0$ at the position before $A_1$, $V_7$.
behind $A_7$; link the adjacent vertices and link $V_7$ to $V_0$. Then we get a circle which has the same topology structure as the convex polygon to which the matrix chain correspond. If there exists a '(' at the position to which a vertex correspond, that means this vertex joins another vertex behind it; If there exists a ')', that means this vertex joins another vertex before it. The linking line segment corresponds to the nearest pair of '(% and ')'. Then we can define the vertex structure as follows.

```c
struct vertex {
    float x;    /* x - coordination */
    float y;    /* y - coordination */
    int left_flag /* the number of '(' */
    int right_flag /* the number of ')' */
}
```

We can use the following algorithm to draw a $n-gon$ triangulation.

```c
Draw_Trian(n)
Begin
    draw the convex polygon
    for i = 0 to n-1
        begin
            if vertex[i].right_flag = 0 /* this vertex does not join any vertex before it */
                do nothing
            otherwise
                for j = vertex[i].right_flag to 0
                    begin
                        find the biggest k from 0 to i /* find the nearest pair so that vertex[k].right_flag != 0 of '(' and ')' */
                        join vertex[i] and vertex[k]
                    end
        end
End
```
We have implemented the above algorithm using C. Consider the following as an example.

**Input:** $P = \{7, 8, 9, 10, 11, 12, 13, 14\}$.

**Output:** See figure 3.8.

![Diagram of a polygon with labeled vertices V0 to V7 and triangulation lines.](image)

Figure 3.8 *GNUPLPOT for polygon triangulation*

This polygon is a *unimodal polygon*. The triangulation for sequential case should be a *fan*, but for parallel case, we can see it is different.
4. CONCLUSION AND FUTURE RESEARCH

In this thesis, we have given a very fast near-optimal parallel algorithm for matrix chain product problem and suggested the parallel matrix chain product and polygon triangulation problem. The following questions still need to be resolved.

- **Sequential algorithm for matrix chain product problem with non-formal weight function**
  
  In the introduction, we mentioned that research on sequential algorithms for these problems is essentially complete. There, we assume the weight function is $p \times q \times r$. How about the weight function $p + q + r$? Do all the theorems and algorithms hold? The answer is 'Yes'. In fact, it was shown that if the weight function is a formal function (i.e., non-negative monotonically non-decreasing function) then all the theorems and algorithms hold. However, questions surrounding the best sequential algorithms for these problems with non-formal weight functions remain open.

- **Optimal parallel algorithm for matrix chain product problem**

  As we know, the best sequential algorithm takes $O(n \log n)$ time, and the best parallel algorithm takes $O(\log^4 n)$ time using $n$ processors. The total work for the parallel algorithm is $O(n \log^4 n)$ steps. There still exists a gap between best
sequential algorithm and parallel algorithm. The future research will seek to reduce the total work of a parallel algorithm to $O(n \log n)$.

- **Parallel matrix chain product problem**

  Even though we have given an $O(n^3)$ algorithm for this problem, it is far from optimal. It can be shown the lower bound of this problem should also be $\Omega(n \log n)$. It is a more interesting and more difficult problem. The research on the unimodal polygon parallel triangulation may be helpful to solving this problem.
BIBLIOGRAPHY


A. THE SOURCE CODE IN C

```c
#define MAX_POINTS 100
#define PI 3.1415926
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>

typedef struct{
    float x;
    float y;
    float weight;
    int left_flag;
    int right_flag;
} POINT;

typedef struct{
    float rows;
    float cols;
} MATRIX;

POINT points[MAX_POINTS];
MATRIX matrix[MAX_POINTS-1];

char s_temp[2*MAX_POINTS];
int final_order[4*MAX_POINTS];
int s_paren[MAX_POINTS][MAX_POINTS];
float cost[MAX_POINTS][MAX_POINTS];
int point_counter = 0;
int retrieve_counter;
int last_draw_point = 0;
int final_order_counter = 0;

static int input_points(void );
static float max_re(float,float);
static void parenth(int,int);
```
void main()
{
    int i;
    int l;
    int j;
    int k;
    int k_ta;
    int n;
    float q;
    float q_ta;

    n = input_points()-1;
    for(i = 0; i < n; i++){
        cost[i][i] = 0;
    }
    for(l = 1; 1 < n; l++){
        for(i = 1; i < n-l+1; i++){
            j = i + l;
            q = max_re(cost[i-1][i-1],cost[i][j-1]) + matrix[i-1].rows*matrix[i-1].cols*
                matrix[j-1].cols;
            k_ta = i;
            for(k = i+1; k < j; k++){
                q_ta = max_re(cost[i-1][k-1],cost[k][j-1]) +
                    matrix[i-1].rows*matrix[k-1].cols*matrix[j-1].cols ;
                if(q > q_ta){
                    q = q_ta;
                    k_ta = k;
                }
            }
            cost[i-1][j-1] = q;
            s_paren[i-1][j-1] = k_ta-1;
        }
    }
    parenth(0,n-1);
    for(i = 0; s_temp[i] != '\0'; i++){
        switch(s_temp[i]){
case `(\':
    points[point_counter].left_flag++;
    break;
  case `)`:
    points[point_counter].right_flag++;
    break;
  case `A`:
    point_counter++;
    break;
  default:
    break;
  }
  }

for(i = 0; i < point_counter+1; i++){
    if(points[i].right_flag == 0);
    else{
        for(k = last_draw_point;k <= i; k++){
            final_order[final_order_counter] = k;
            final_order_counter++;
        }
        retrieve_counter = i;
        for(j = 0; j < points[i].right_flag; ){
            retrieve_counter--;
            if(points[retrieve_counter].left_flag == 0) ;
            else{ /* find a match */
                j++;
                points[retrieve_counter].left_flag--;

                final_order[final_order_counter++] = retrieve_counter;
                final_order[final_order_counter++] = i;
            }
        }
        last_draw_point = i;
    }
}
for(i = 0; i < n+1; i++){
    fprintf(stdout,"%f %f \n", points[i].x, points[i].y);
}

fprintf(stdout,"#s \n", s_temp);
fprintf(stdout,"#d \n", point_counter+1);
fprintf(stdout,"#the outmost paren point is: d \n",
    s_paren[0][point_counter-1]+2);

fprintf(stdout,"#cost is : f \n", cost[0][point_counter-1]);

fprintf(stdout,"#The left side cost is : f \n",
    cost[0][s_paren[0][point_counter-1]]);
fprintf(stdout,"#The right side cost is : f \n",
    cost[s_paren[0][point_counter-1]+1][point_counter-1]);

for(i = 0; i < final_order_counter; i++){
    fprintf(stdout,"%f %f \n", points[final_order[i]].x, points[final_order[i]].y);
}

int input_points()
{
    int i;
    int k = 0;
    float a;
    double theta;
    int scanf_return;
    while((scanf_return = fscanf(stdin, "%f", &a)) != EOF)
    {
        if (scanf_return < 1)
        {
            fprintf(stderr, "Input Error\n");
            exit(1);
        }
        else
        {
            points[k].weight = a;
            k++;
        }
    }
    for(i = 0; i < k; i++){
        theta = 1.5*PI - (float)i*2.0*PI/(float)k;