A STUDY ON COMPUTER ALGORITHMS INVOLVING MULTIPLICATION

BY

MD. SANAUL HOQUE

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF

MASTER OF SCIENCE IN ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

September 1996

The thesis A STUDY ON COMPUTER ALGORITHMS INVOLVING MULTIPLICATION, submitted by MD. SANAUL HOQUE, Roll No. 901814P, Session 1988-89, Registration No. 84182 to the Computer Science and Engineering Department of Bangladesh University of Engineering and Technology has been accepted as satisfactory for partial fulfillment of the requirements for the degree of M.Sc. Engg. in Computer Science and Engineering and approved as to its style and contents. Examination held on September 24, 1996.

DECLARATION

I, hereby, declare that the work presented in this thesis is done by me under the supervision of Dr. Mohammad Kaykobad, Associate Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka. I also declare that neither this thesis nor any part thereof has been submitted elsewhere for the award of any degree or diploma.

Countersigned

[illegible]

Dr. Mohammad Kaykobad
(Supervisor)

[illegible]

(Md. Sanaul Hoque)

ACKNOWLEDGEMENTS

All praises are for the Almighty.

The author would like to express his sincere gratitude and profound indebtedness to his supervisor Dr. M. Kaykobad, Associate Professor and Head, Department of Computer Science and Engineering(CSE), BUET, Dhaka, for his constant guidance, helpful advice, invaluable assistance and endless patience throughout the progress of this work, without which the work could not have been completed.

The author gratefully acknowledges the kind support and encouragement extended to him by Dr. Md. Shamsul Alam, Professor, CSE Department, BUET. The author also acknowledges support extended to him by Dr. A.B.M. Siddique Hossain, Ex-Head, CSE Department.

The author conveys his gratitude to Professor M. Shamsher Ali, Vice-Chancellor, Bangladesh Open University for agreeing to act as an External examiner and showing keen interest in the work even amidst his heavy scientific and social preoccupations in national and international arena.

The author is indebted to Mr. Ahmad Fuad Rezaur Rahman, Lecturer (on leave), CSE Dept., BUET, Mr. Md. Yasin Ali, Lecturer (on leave), BIT, Chittagong, Mr. Mohammad Manzur Murshed and Mr. Sandeepan Sanyal, Ex-Lecturers, CSE Dept, BUET for their effort in collecting and sending relevant papers from abroad, and generous assistance in the thesis work.

The author also acknowledges the constant support and patience of his family members. The arduous task of having completed the thesis could not have been accomplished without the understanding he received from them.

Finally, the author acknowledges with hearty thanks the all out cooperation of his friends, particularly Abdul Mannan and the faculty members of the CSE Department, who gave their time, their knowledge and their best advice.

ABSTRACT

This research work has been aimed at investigating algorithms involving multiplication. In particular three classes of problems namely, polynomial evaluation, matrix multiplication and chain matrix multiplication have been considered.

The problem of polynomial evaluation has been considered with preprocessing which has been proven efficient in case it is evaluated at many points. Belaga, Peterson-Stockmeyer and Knuth's method of preprocessing have been subject to numerous experiments, and Belaga's method has outperformed the remaining algorithms in terms of computational time requirement for evaluating preprocessed polynomials, whereas Belaga's preprocessing algorithm was found to be the most time consuming.

Matrix multiplication algorithms were tested with both integer and real data. We have considered order of the matrices in the range 4-128 for our experiment. It was observed that in this range none of the prospective algorithms of better orders could perform better than $O(n^3)$ classical algorithm and algorithm using Winograd's identity. Multiplication algorithm using Winograd's identity came out superior in both the cases of integer and real data elements. This finding does not totally disagree with the previous assertion that Strassen's algorithm become competitive only after order of the involved matrices exceed 120. However, our numerical experiments do not indicate that even after the order exceeds 120 it can have any competitive edge over Winograd's identity or even classical algorithm for orders near 120. We have also introduced a new preprocessing method for converting an arbitrary system of linear equations into a positive definite linear systems for which convergent iterative schemes exist. This preprocessing was done by recursively using Strassen's scheme to compute A^tA by using only two-thirds of the cost required to multiply two arbitrary matrices using Strassen's scheme. This has resulted in an algorithm with 33% savings over direct application of Strassen's multiplication scheme.

For chain matrix multiplication, dynamic programming scheme as well as heuristic algorithms of Chin and Hu-Shing were considered. Both the heuristic algorithms performed very well producing optimal sequences in reasonable amount of computation with Hu-Shing's algorithm performing better. Moreover, both these methods produced solutions whose deviation from the optimal solution was found to be decreasing with the number of matrices in the chain.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

An algorithm is any well defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a finite amount of time. An algorithm is thus a sequence of computational steps that transform the input into the output. An algorithm is usually specified as programs written in pseudocode.

An algorithm is said to be correct if, for every input instance it halts with the correct output. An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired result.

Algorithms devised to solve the same problem often differ dramatically in their efficiency. A good algorithm is like a sharp knife - it does exactly what it is supposed to do with a minimum amount of applied effort. Using a wrong algorithm to solve a problem is like trying to cut a steak with a screwdriver. It may eventually produce a digestive result but requires considerably more effort than necessary, and the result is unlikely to be aesthetically pleasing.

Analyzing an algorithm mean predicting the resources that the algorithm requires. By analyzing several candidate algorithms for a problem, the most efficient one can be easily identified. Such analysis may indicate more than one viable candidate, but several inferior algorithms are usually discarded in the process.

Algorithms can be evaluated by a variety of criteria. Mostly the interest is in the rate of growth of time or space required to solve larger and larger instances of a problem..

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The limiting behavior of the complexity as size grows is called the *asymptotic time complexity*. If an algorithm processes inputs of size n in time an^2+bn+c for some constants a, b, c, then the time complexity of that algorithm is $O(n^2)$. Similar definition can be given for *space complexity* and *asymptotic space complexity*. The asymptotic complexity of an algorithm ultimately determines the size of problems that can be solved by the algorithm.

If for a given size, the complexity is taken as the maximum complexity over all inputs of that size, then the complexity is called the *worst case complexity*. The worst case running time of an algorithm is an upper-bound on the running time. Knowing this gives a guarantee that the algorithm will never take any longer.

If the complexity is taken as the average complexity over all inputs of a given size then the complexity is called the *expected complexity* (also called *average complexity*). The expected complexity of an algorithm is more difficult to ascertain because it may not be apparent what constitutes an average input for a particular problem. Most often it is assumed that all inputs of a given size are equally likely. A randomized algorithm can sometimes force it to hold.

In analyzing algorithms, the *accuracy* of the computed result is another important criterion. The computer being a finite state machine, is capable of representing numbers only to a finite number of digit positions. As a result, most numbers are rounded if they are too long for the computer to represent exactly. Thus, some algorithms may produce approximations that are wildly inaccurate.

IMPORTANCE OF THE STUDY

Classical science is based on observation, theory and experimentation. Unfortunately, scientists cannot use physical experiments to test many of their theories. High speed computing allows them to test their hypothesis in another way by developing a numerical simulation. Many important scientific problems are so complex that solving them via numerical simulation requires extra ordinarily speedy computer.

Ever since conventional computers were invented, their speed has steadily increased to match the needs of emerging applications. One might suspect that this would decrease the importance of efficient algorithms. However, just the opposite is true. Fundamental physical limitations make it impossible to achieve further improvements in the speed of these computers indefinitely. Recent trends show that the performance of these computers is beginning to saturate. A cost-performance comparison over the last few decades revealed that at the lower end, performance increases almost linearly (or even faster than linearly) with cost. However, beyond a certain point, each curve starts to saturate, and even smaller gains in performance come at an exorbitant increase in cost.

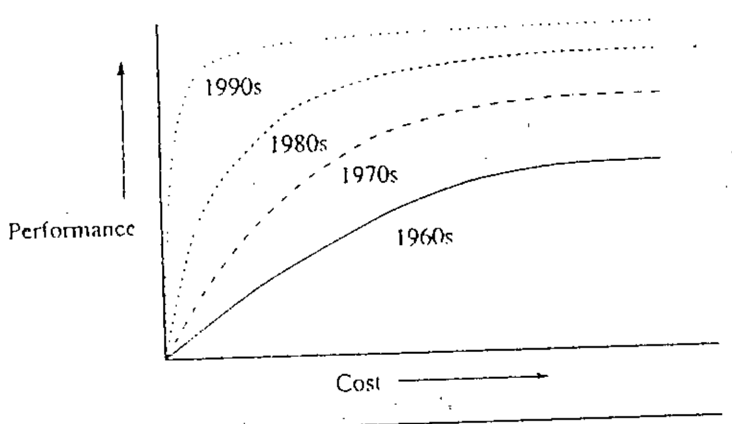

Figure 1.1 Cost versus performance curve and its evolution over the decades.

Instead it can be shown that speed gain that can be achieved through algorithmic improvements is much more prominent than that can be achieved from a high speed next generation computer.

The third argument that may be placed in favour of hardware is the use of parallel computers. This is a comparatively new field. Besides many algorithms suitable for conventional, single processor computers are not appropriate for parallel architectures. Many algorithms with inherent parallelism have a higher computational complexity than the best sequential counterpart. Thus total system performance depends on choosing efficient algorithms as much as on choosing fast hardware.

Many of the scientific calculations are repetitive in nature. Saving a small amount of computational time for frequently used program segments can result in large overall savings. Since computer arithmetic is in the heart of every computational work, a marginally better algorithm for one of these basic operations will result in tremendous savings.

HISTORICAL PERSPECTIVE

Algorithms for doing elementary arithmetic operations such as addition, multiplication, and division have a very long history, dating back to the origins of algorithm studies in the work of the Arabian mathematician al-Khowarizmi, with roots going even further back to the Greeks and the Babylonians.

W.G. Horner[1] taught an elegant way to rearrange a polynomial for computation. He gave this rule early in the nineteenth century. Though the fame goes with Horner, this idea was used by Sir Isaac Newton[2] in 1711, 150 years earlier than Horner.

A. M Ostrowski[3], in 1954, was the first to ask whether a good algorithm, Horner's rule for polynomial evaluation, was the best method possible. This began the modern history of arithmetic complexity. Ostrowski was only partially successful in answering the fundamental question he posed.

In 1966, V. Y. Pan[4] established that Horner's rule is optimal with respect to the number of multiplications/divisions for the evaluation of a polynomial. Z. Kedem and D. Kirkpatrick, in 1974, provided lower bounds on addition/subtraction operations for a number of problems including polynomial evaluation and matrix multiplication. Initially all scientific endeavour to polynomial evaluation were attempts to bring about efficient evaluation schemes by means of conceptually new representation forms.

In 1955, Motzkin[6] introduced the idea of preprocessing the polynomial coefficients. Belaga[5], in 1958, established the lower bound on polynomial evaluation with preprocessed coefficients. Pan[7], in 1959, proposed a form of economical evaluation of a polynomial. Since then, several other schemes of polynomial evaluation were brought into light. Rabin and Winograd[8], in 1971, presented a number of rational preconditioning methods.

Multiplication and division, of the four basic arithmetic operations, are the most time consuming. Thus minimum computation time can be achieved if multiplications can be done fast. Karatsuba[9] suggested a method for doing multiplication with running time of order $n^{\log 3}$. A.L.Toom[10], in 1963, gave a new idea to further improve Karatsuba's method. S.A. Cook[11] showed how this method can be adapted in a computer program. V. Strassen and A. Schönage[12] jointly discovered a 2-base-FFT based scheme to multiply large integers.

The classical matrix multiplication algorithm seemed irreducible for a long time. In 1962, A. Karatsuba[9], with Yu Ofman, developed a method for multiplying two matrices. His algorithm uses fewer number of additions and the same number of multiplications as compared to that of the classical method. S. Winograd[14], in 1967, discovered a method based on the identity involving the sum of two pairwise products. V. Strassen[15], in 1969, observed that a pair of 2 x 2 matrices can be multiplied in 7 multiplications, instead of the usual 8. This single result has provided the greatest impetus to the field of computational complexity.

After Strassen's $O(n^{\log 7})$ algorithm, numerous attempts were made to improve it. S. Winograd[16] further improved Strassen's method. He presented an algorithm which uses 7 multiplications and 15 additions to multiply two 2 x 2 matrices. Victor Pan[17], in 1978, discovered that it could be lowered to $O(n^{2.795})$. This breakthrough led to further intensive analysis of the problem, and the combined efforts of D. Bini, M. Caporani, G. Lotti, F. Romani, A. Schönhage, V. Pan, S. Winograd, and D. Coppersmith (see [30], p.482) culminated in constructions that have an asymptotic running time of $O(n^{2.5161})$. The asymptotically most efficient algorithm to date, due to V. Pan[18], has a running time of $O(n^{2.496})$.

Francis Y. Chin[19], in 1978, presented an $O(n)$ algorithm for determining a near-optimal computation order for chain-matrix multiplication problem. A.K. Chandra[13] presented an $O(n)$ algorithm which produces an order that requires no more than twice the optimum computation time. T.C. Hu[20], along with Shing, presented a heuristic algorithm to find the optimum order.

THESIS ORGANIZATION

This thesis comprises six chapters. Chapter one provides the motivation for the study. It explains what algorithm is, what complexity is, why this analysis is needed. It also gives some insights into the various developments in the field of arithmetic computation.

Chapter two deals with the evaluation of polynomials. Initially it describes some polynomial representation techniques and their associated evaluation algorithms. Then it introduces the concept of preconditioning and describes algorithms that uses preconditioning. Algorithms to evaluate polynomials with complex arguments are also described here. In addition, techniques for evaluating several polynomials at several different points simultaneously are presented in this chapter.

Chapter three is dedicated to matrix multiplication. Operation on matrices are at the heart of scientific computing. Though classical method of matrix multiplication is widely used, several other efficient algorithms have been developed. This chapter presents Strassen's surprising algorithm and several other algorithms.

Chapter four describes techniques that should be employed while performing chain matrix multiplication. The way matrices are parenthesized has a dramatic impact on the cost of evaluating the product. This chapter presents a dynamic programming approach to determine the optimum order. In addition, it also presents two other algorithms, one by F.Y. Chin and the other by Hu and Shing, that can efficiently determine near-optimal order.

Chapter five is intended for the experimental results based on the algorithms discussed so far in the previous chapters. This chapter compares the complexities of similar algorithms.

In chapter six, conclusions were made on the findings and some issues for further research in those directions are recommended.

CHAPTER TWO

POLYNOMIAL EVALUATION

Evaluation of polynomials is one of the most widely used operations in practical computation. In fact, many algorithms entail the evaluation of one or more polynomials at a large number of points. A polynomial is, generally, an expression of the form

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \qquad \text{.................................}\textbf{(2.1)}$$

where the coefficients a_n , a_{n-1} , ... , a_1 , a_0 are elements of some algebraic system (e.g., integers, floating point numbers, complex quantities etc.) and x is an indeterminate. If $a_n \neq 0$, n is referred to as the degree of the polynomial.

Evaluation of polynomials occur while computing transcendental or more complex algebraic expressions. Trigonometric functions like sine, cosine etc. and exponential and logarithmic functions are sometimes expressed as a polynomial and their evaluation depends on the evaluation of those polynomials.

REPRESENTATION OF POLYNOMIALS

Any particular polynomial may be expressed in a variety of ways. We can represent a polynomial of degree n by its value at $n+1$ points, by its roots, or by its value and all derivative values at a single point. Each of these representations exactly describe the polynomial but where numerical evaluation is concerned, the various forms show different properties.

The following section describes various ways of evaluating a polynomial, depending on the form in which it is presented.

The Power Form : The most common and widely used form of a polynomial is the power form shown in eqn. (2.1). This is sometimes referred to as *coefficient representation* of a polynomial. A polynomial given in power form can be evaluated at a point x_0 using the following algorithm. **Algorithm 2.1** requires $2n$ multiplication, n addition and $(2n+2)$ assignments.

```
procedure powerform ($A$, $n$, $x_0$)
        $s \leftarrow a_0$ ;   $r \leftarrow 1$
        for $i \leftarrow 1$ to $n$ step $1$ do
                $r \leftarrow r * x_0$
                $s \leftarrow a_i * r + s$
        repeat
return ($s$)
end powerform
```

Algorithm 2.1

The Nested Form : W. G. Horner[1] suggested a different representation for a polynomial. This form is known as nested form or Horner form. A polynomial in Horner form is expressed as

$$A(x) = (\cdots (a_n x + a_{n-1})x + \cdots + a_1)x + a_0 \qquad \text{.......}(2.2)$$

This suggests the evaluation algorithm for Horner form as

```
procedure horner ($A$, $n$, $x_0$)
        $s \leftarrow a_n$
        for $i \leftarrow (n-1)$ to $0$ step $-1$ do
                $s \leftarrow s * x_0 + a_i$
        repeat
return ($s$)
end   horner
```

Algorithm 2.2

Algorithm 2.2 requires n multiplication, n additions and $(n+1)$ assignments.

A slight variation of Horner's form states a polynomial as

$$A(x) = (\ldots(a_{2p} x^2 + a_{2p-2})x^2 + \ldots)x^2 + a_0$$
$$+((\ldots(a_{2q-1} x^2 + a_{2q-3})x^2 + \ldots)x^2 + a_1)x$$
$$\text{where } p = \lfloor n/2 \rfloor \quad \text{and } q = \lceil n/2 \rceil \qquad \text{.......}(2.3)$$

We can evaluate this polynomial using the algorithm shown in next page. This algorithm requires $(n+1)$ multiplications and n additions. Though slightly expensive as compared to Horner's algorithm, this algorithm is particularly useful if one wishes to evaluate both $A(x_0)$ and

A$(-x_0)$. This is accomplished with just one extra addition operation, as such two values are obtained almost at the cost of one.

```
procedure horner_variation($A$, $n$, $x_0$)
        $u \leftarrow 2 * \lfloor n/2 \rfloor$;    $v \leftarrow 2 * \lceil n/2 \rceil - 1$
        $s_1 \leftarrow a_u$ ;   $s_2 \leftarrow a_v$
        $r \leftarrow x_0 * x_0$
        for ($i = u-2, j = v-2$ ; $i \geq 0; j \geq 0$ ; $i = i-2$, $j=j-2$ ) do
                $s_1 \leftarrow s_1 * r + a_i$
                $s_2 \leftarrow s_2 * r + a_j$
        repeat
return ( $s_1 + s_2 * x_0$ )
end horner_variation
```

Algorithm 2.3

The Root Product Form : The Root Product form of a polynomial is given by

$$A(x) = a_0 \prod_{i=1}^{n} (x - \gamma_i) \qquad \text{.................(2.4)}$$

where, $A(\gamma_i) = 0$. The roots γ_i's can be real or complex.

Important cases of polynomials in this form arise, for example, in statistics. The computational algorithm for this representation can be expressed as

```
procedure root_product ($\gamma$, $a_0$ , $n$, $x_0$)
        $s \leftarrow a_0$
        for $i \leftarrow 1$ to $n$ do
                $s \leftarrow s * (x_0 - \gamma_i)$
        repeat
return ($s$)
end root_product
```

Algorithm 2.4

It can be easily seen that evaluation algorithm for the Root Product form of representation requires n multiplications and n additions.

The Lagrange Form : A polynomial can be represented by their values at different points. Given $(n+1)$ points $[x_i, f(x_i)]$, we can uniquely describe a polynomial $A(x)$ of degree $\le n$ that goes through these $(n+1)$ points. A polynomial has many different point-value representations, since any set of $(n+1)$ distinct points can be used as a basis for the representation. A polynomial thus can be expressed as

$$A(x) = \sum_{1 \le i \le n} \left[\prod_{i \ne j} \frac{x - x_j}{x_i - x_j} \right] f_i \qquad \text{.................(2.5)}$$

This representation is quite convenient for many operations on polynomials, such as addition, subtraction, multiplication, etc. We can compute the polynomial at any point $x = x_0$ using **Algorithm 2.5** given below. This algorithm requires $(2n^2 + 2n)$ multiplications and $(2n^2 + 3n + 1)$ additions which is discouragingly very high.

```
procedure lagrange ($X, F, n, x_0$)
        $s \leftarrow 0$
        for $i \leftarrow 0$ to $n$ step $1$ do
                $p \leftarrow f_i$
                for $j \leftarrow 1$ to $n$ step $1$ do
                        if $i \ne j$ then
                                $p \leftarrow p * (x_0 - x_j) / (x_i - x_j)$
                        endif
                repeat ( $j$ )
                $s \leftarrow s + p$
        repeat ( $i$ )
return ( $s$ )
end    lagrange
```

Algorithm 2.5

The Newton Form : The Newton form of a polynomial is a modified form of Lagrange's form. This polynomial form arises in the form of interpolating a function given in tabular form. The form does not occur directly but is arrived at after some manipulation on the direct interpolating polynomial.

Lagrange's polynomial of degree two can be expressed as

$$P_2(x) = f_1\frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + f_2\frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + f_3\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$$

and that of degree one can be expressed as

$$P_1(x) = f_1\frac{x-x_2}{x_1-x_2} + f_2\frac{x-x_1}{x_2-x_1}$$

Subtracting,

$$P_2(x) - P_1(x) = f_1\frac{(x-x_2)}{(x_1-x_2)}\left[\frac{(x-x_3)}{(x_1-x_3)} - 1\right] + f_2\frac{(x-x_1)}{(x_2-x_1)}\left[\frac{(x-x_3)}{(x_2-x_3)} - 1\right] + f_2\frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$$

Now $P_2(x) - P_1(x)$ is a second degree polynomial that vanishes at $x = x_1$ and x_2, and must therefore be a multiple of $(x - x_1)(x - x_2)$.

$$\therefore \quad P_2(x) - P_1(x) = \alpha_2(x-x_1)(x-x_2).$$

Similarly, it can be shown that

$$P_1(x) - P_0(x) = \alpha_1(x-x_1)$$

therefore,

$$\begin{aligned} P_2(x) &= \alpha_2 (x-x_1)(x-x_2) + P_1(x) \\ &= \alpha_2 (x-x_1)(x-x_2) + \alpha_1 (x-x_1) + P_0(x) \\ &= \alpha_2 (x-x_1)(x-x_2) + \alpha_1 (x-x_1) + \alpha_0 \end{aligned}$$

Generalizing, we can write

$$P_n(x) = \alpha_n(x-\beta_n)(x-\beta_{n-1}) \ldots (x-\beta_1) + \alpha_{n-1}(x-\beta_{n-1})(x-\beta_{n-2}) \ldots (x-\beta_1) + \ldots + \alpha_1(x'-\beta_1) + \alpha_0 \qquad \ldots\ldots\ldots\ldots(2.6)$$

This is the Newton form of a polynomial. The computational algorithm based on Newton form may be stated as follows :

```
procedure newton ($\alpha$, $\beta$, $n$, $v$)
        $s \leftarrow \alpha_n$
        for $i \leftarrow n$ to $1$ step $-1$ do
                $s \leftarrow s * (v - \beta_i) + \alpha_{i-1}$
        repeat
return ($s$)
end newton
```

Algorithm 2.6

This algorithm requires n multiplications and $2n$ additions.

Orthogonal Form : The orthogonal polynomial form is given by

$$P_n(x) = \sum_{k=0}^{n} b_k Q_k(x) \qquad \ldots\ldots\ldots\ldots(2.7)$$

where the orthogonal polynomials, $\{Q_i(x),\ i = 0, 1, \ldots, n\}$ satisfy a recurrence relation $Q_{i+1}(x) = (A_i x + B_i)\,Q_i(x) - C_i Q_{i-1}(x)$ with $A_i \neq 0$, $Q_0(x) = 1$, $Q_{-1}(x) = 0$ and where A_i, B_i and C_i are independent of x. The computational algorithm for orthogonal representation is,

```
procedure orthogonal ( $A, B, C, n, b, x_0$ )
        $V_n \leftarrow b_n$
        $V_{n-1} \leftarrow (A_{n-1} x_0 + B_{n-1})V_n + b_{n-1}$
        for $k \leftarrow n-2$ to $0$ by $-1$ do
                $V_k \leftarrow (A_k x_0 + B_k)V_{k+1} - C_k V_{k+2} + b_k$
        repeat
return ($V_0$)
end orthogonal
```

Algorithm 2.7

This scheme of evaluation requires $(3n-1)$ multiplications and $(3n-1)$ additions.

Chebyshev polynomial $T_n(x)$ is a classical orthogonal polynomial. Here,

$$P_n(x) = \sum_{k=0}^{n} b_k T_k(x) \qquad \text{......................(2.8)}$$

and $B_i = 0$, $C_i = 1$ *for* $i \geq 0$, $A_i = 2$ *for* $i \geq 1$ *and* $A_0 = 1$.

Clenshaw[21] developed an algorithm (referred to as Chebyshev-Clenshaw algorithm) to evaluate a Chebyshev polynomial.

```
procedure chebyshev_clenshaw ( $n, b, x_0$ )
        $Y \leftarrow 2 * x_0$
        $V_n \leftarrow b_n$
        $V_{n-1} \leftarrow Y * V_n + b_{n-1}$
        for $k \leftarrow (n-2)$ to $1$ step $-1$ do
                $V_k \leftarrow Y * V_{k+1} - V_{k+2} + b_k$
        repeat
return ($x_0 * V_1 - V_2 + b_0$)
end chebyshev_clenshaw
```

Algorithm 2.8

The evaluation of a general polynomial as a weighted sum of Chebyshev polynomials using the above algorithm requires $(n+1)$ multiplications and $2n$ additions.

Bakhvalov[22] developed another algorithm to evaluate Chebyshev polynomials. This algorithm is referred to as Chebyshev-Bakhvalov algorithm and is given below.

```
procedure chebyshev_bakhvalov ( $n, b, x_0$ )
        for $k \leftarrow 0$ to $n-2$ do
                $D_k \leftarrow (b_k - b_{k+2}) / 2$
        repeat
        $D_{n-1} \leftarrow b_{n-1} / 2$
        $V_n \leftarrow b_n / 2$
        $Y \leftarrow 2 * x_0$
        $V_{n-1} \leftarrow Y * V_n + D_{n-1}$
        for $k \leftarrow n-2$ to $0$ by $-1$ do
                $V_k \leftarrow (Y * V_{k+1} - V_{k+2}) + D_k$
        repeat
return ($V_0$)
end chebyshev_bakhvalov
```

Algorithm 2.9

This algorithm requires $(n+1)$ multiplications, $(n+1)$ divisions by a factor of 2, and $(3n-1)$ additions.

Algorithm Chebyshev_Clenshaw is more efficient than algorithm Chebyshev_Bakhvalov in terms of the number of arithmetic operations, but for certain polynomials the use of the latter may be preferable on the grounds of numerical accuracy.

PREPROCESSING OF COEFFICIENTS

There are many possible representations of a polynomial, all of which may be used as inputs to a computation. A.M. Ostrowski[3] showed that at least *n* multiplications and *n* additions are required to evaluate degree n polynomials for $n \leq 4$ with the underlying assumption that the polynomial coefficients were not in any way artificially transformed. Since then, this result has been proved true for all non-negative values of n. This establishes that the Horner algorithm is optimal regarding the number of arithmetic operations involved.

However, it can be shown that if the polynomial coefficients are preprocessed for the evaluation of the polynomial, it requires fewer multiplication and/or additions than the Horner algorithm. This preconditioning involves a lot of additional arithmetic operations, but it has to be done only once. The overall savings on the number of arithmetic operations may be significant if the polynomial is evaluated at many points.

In distinction to the polynomial forms considered earlier, polynomials with preprocessed coefficients do not arise 'naturally' but are obtained in an artificial way, with the sole purpose of facilitating their fast evaluation.

Todd-Motzkin algorithm : Motzkin[6] was first to introduce the idea of preprocessing the polynomial coefficients for the purpose of polynomial evaluation. The Todd-Motzkin approach expresses a degree four polynomial as

$$A(x) = \alpha_4 \, [z(z + x + \alpha_2) + \alpha_3]$$

where, $$z = x(x + \alpha_0) + \alpha_1 \qquad \text{.....................(2.9)}$$

substituting,

$$A(x) = \alpha_4 x^4 + \alpha_4(2\alpha_0+1)x^3 + \alpha_4[\alpha_1+\alpha_2+\alpha_0(\alpha_0+1)]x^2$$
$$+ \alpha_4[(\alpha_1+\alpha_2)\alpha_0+\alpha_1]x + \alpha_4(\alpha_1\alpha_2+\alpha_3)$$

equating like powers of x,

$$\alpha_0 = 0.5\ (a_3 / a_4 - 1)$$

$$\alpha_1 = a_1 / a_4 - \alpha_0\beta. \qquad \text{where, } \beta = a_2 / a_4 - \alpha_0(\alpha_0 + 1)$$

$$\alpha_2 = \beta - 2\alpha_1$$

$$\alpha_3 = a_0 / a_4 - \alpha_1(\alpha_1 + \alpha_2)$$

$$\alpha_4 = a_4$$

Three multiplications and five additions are required to evaluate a degree four polynomial using this approach. Though the preconditioning operation (i.e., evaluation of α's) requires extra 9 multiplications/divisions and 7 additions/subtractions.

Belaga Algorithm :

In 1958, E.C. Belaga[5] formulated two theorems stating that "**For any polynomial A(x) of degree n, there exists a computational evaluation scheme that requires $\lfloor(n+1)/2\rfloor+1$ multiplications and (n+1) additions**" and "**No evaluation scheme exists with less than $\lfloor(n+1)/2\rfloor$ multiplications or with less than n additions.**" These theorems establish the lower bounds on the number of multiplications and additions, under the assumption that some preprocessing of the coefficients is allowed without cost. The general scheme suggested by Belaga can evaluate any polynomial of degree $n \geq 4$ with prior preprocessing of the coefficients.

The scheme for evaluation of a polynomial as suggested by Belaga can be described as

$$P_n(x) = \sum_{k=0}^{n} a_k x^{n-k} = \begin{cases} \alpha_{n+1} V_{\lfloor n/2 \rfloor} + \alpha_0, & \textit{for } n \textit{ even} \\ \alpha_{n+1} x V_{\lfloor n/2 \rfloor} + \alpha_0, & \textit{for } n \textit{ odd} \end{cases} \qquad \text{......(2.10)}$$

where, $V_k = V_{k-1}(V_1 + \alpha_{2k-1}) + \alpha_{2k}, \qquad k = 3, \ldots, \lfloor n/2 \rfloor$

$V_2 = (V_1 + x + \alpha_2)(V_1 + \alpha_3) + \alpha_4$

$V_1 = (x + \alpha_1)\, x$

We can easily see that the Belaga computational scheme requires $\lfloor (n+1)/2 \rfloor + 1$ multiplications and $2\lfloor n/2 \rfloor + 1$ additions.

E. W. Cheney[26] developed a method for computing the parameters α_i 's in the Belaga scheme in terms of $(a_0, a_1, \ldots, a_n)$. Let

$$P_n(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_n = \begin{cases} \alpha_{n+1} V_m + \alpha_0 \\ \\ \alpha_{n+1} x V_m + \alpha_0 \end{cases}, \quad m = \lfloor n/2 \rfloor \qquad \text{......(2.11)}$$

Without restricting the generality of the method, we may assume that $a_0 = \alpha_{n+1} = 1$, and considering

$$V_m = x^{2m} + c_1x^{2m-1} + c_2x^{2m-2} + \ldots + c_{2m}$$

$$= (x^{2m-2} + b_1x^{2m-3} + b_2x^{2m-4} + \ldots + b_{2m-2})(x^2 + \alpha_1 x + \alpha_{2m-1}) + \alpha_{2m} \quad \ldots\textbf{(2.12)}$$

$$\text{where } c_j = a_j\,, \quad j = 1, \ldots, 2m, \quad m = \left\lfloor \frac{n}{2} \right\rfloor$$

Equating like powers of x in eqn. (2.11) gives:

$$c_1 = \alpha_1 + b_1$$

Setting $\alpha_1 \;\; \frac{1}{m}(c_1 \;\; 1)$, we obtain $b_1 \;\; \frac{(m-1)c_1+1}{m}$.

$$c_2 = \alpha_{2m-1} + \alpha_1 b_1 + b_2.$$

or, $\quad b_2 = c_2 - \alpha_1 b_1 - \alpha_{2m-1}$.

for other b's, $\;b_k = c_k - \alpha_1 b_{k-1} - \alpha_{2m-1} b_{k-2}$, $\qquad k = 3, \ldots, 2m-2$

α_{2m-1} is still unknown. To obtain this value, solve

$$c_{2m-1} = \alpha_{2m-4} b_{2m-3} + \alpha_1 b_{2m-1}$$

after substituting the b's in it. At this point, substitute the value of α_{2m-1} into the expressions for $b_{2m-2}, \ldots, b_2$ and obtain their numeric values. Finally, solve the equation

$$c_{2m} = \alpha_{2m-1} b_{2m-2} + \alpha_{2m} \qquad \text{for } \alpha_{2m}.$$

We have now obtained values for $\alpha_n = \alpha_{2m}$, $\alpha_{n-1} = \alpha_{2m-1}$, and α_1. By replacing m by m−1 in eqn.(2.12) and solving the system obtained in the manner similar to the above, we obtain α_{n-2} and α_{n-3}. The process is repeated until all α_j are computed. The parameters of Belaga form for a given polynomial are not unique. In addition, some of the α_j 's may be complex for a polynomial with real natural coefficients.

An example :

Let, $P_7(x) = x^7 - 2x^5 + x^3 - 0.1x$ be the polynomial to be evaluated.

We may write,

$$V_3 = x^6 - 2x^4 + x^2 - 0.1$$

$$= (x^4 + b_1 x^3 + b_2 x^2 + b_3 x + b_4)(x^2 + \alpha_1 x + \alpha_5) + \alpha_6$$

here, $m = 3$, $c_1 = 0$, $c_2 = -2$, $c_3 = 0$, $c_4 = 1$, $c_5 = 0$, $c_6 = -0.1$.

we also get, $\alpha_7 = 1.0$, and $\alpha_0 = 0$.

$\therefore \quad \alpha_1 = -0.333, \;\; b_1 = 0.333$

$b_2 = - 1.89 - \alpha_5 , \quad b_3 = - 0.63 - 0.67\alpha_5$

$b_4 = 0.79 + 1.22\alpha_5 + \alpha_5^2$.

Now, solving $0.26 + 1.04\alpha_5 + \alpha_5^2 = 0$, we get $\alpha_5 = -1.607$ or 2.39.

substituting, α_5 in the equations for b_2 , b_3 , b_4 , we get

$b_2 = - 0.283, \;\; b_3 = 0.447, \;\; b_4 = 1.412.$

finally, $\alpha_6 = c_6 - \alpha_5 b_4 = 2.169$.

Thus we get,

$$V_2 = x^4 + 0.333x^3 - 0.283x^2 + 0.447x + 1.412$$

$$= (x^2 + b_1 x + b_2)(x^2 + \alpha_1 x + \alpha_3) + \alpha_4$$

Equating like powers,

$b_2 = - 0.172 - \alpha_3$

for α_3 , we get, $0 = 0.504 - 0.666\alpha_3$, or $\alpha_3 = 0.757$.

therefore $b_2 = - 0.929$

and $\alpha_4 = 1.412 - \alpha_3 b_2 = 2.115$. This is the end of preprocessing.

For evaluation, we have,

$$V_1 = (x - 0.333)\,x$$

$$V_2 = (V_1 + x - 0.929)(V_1 + 0.757) + 2.115$$

$$V_3 = V_2\,(V_1 - 1.607) + 2.169$$

$$\therefore \quad P(x) = x\,V_3 + 0$$

For x = 2, we can easily calculate

$$V_1 = 3.334$$

$$V_2 = 20.136$$

$$V_3 = 36.944$$

$$P(2.0) = 1.0 \times 2.0 \times 36.944 + 0.0$$

$$= 73.887$$

Paterson-Stockmeyer method :

M. Paterson and L. Stockmeyer[25] have developed a rational preconditioning algorithm in which only rational functions are used in the preconditioning phase. Their method was motivated by computation of polynomials whose coefficients and variable are matrices. This algorithm is developed assuming the polynomial P(x) to be monic and of degree n. We can decompose P(x) as follows :

$$P(x) = (x^{\lceil n/2 \rceil} + \gamma)\,P_1(x) + P_2(x)$$

where,

$$P_1(x) = x^{\lfloor n/2 \rfloor} + \sum_{j=0}^{\lfloor n/2 \rfloor - 1} \alpha_j x^j$$
$$P_2(x) = x^{\lfloor n/2 \rfloor} + \sum_{j=0}^{\lfloor n/2 \rfloor - 1} \beta_j x^j \qquad \text{......................(2.13)}$$

equating like powers of x, we can obtain

$$\gamma = \begin{cases} a_{\lfloor n/2 \rfloor} - 1 & n \textit{ is odd} \\ -1 & n \textit{ is even} \end{cases}$$
$$\alpha_j = a_{\lfloor n/2 \rfloor + j}$$
$$\beta_j = a_j - \gamma\, \alpha_j \qquad \text{...............(2.14)}$$

Only rational operations are needed to compute $\{\alpha_j\}$, γ, $\{\beta_j\}$ from the coefficients of P(x). More over, the idea can be recursively used to compute the monic polynomials $P_1(x)$ and $P_2(x)$.

Not counting the cost of preconditioning nor computing the appropriate powers of x, we can analyze the multiplication and addition complexity of the algorithm. For simplicity, let $n = 2^{k+1}-1$.

Then the cost of multiplication is $f(k+1) = 2f(k) + 1$ where $f(1)=0$.

After simplification, $f(k+1) = 2^k - 1$

or, in terms of n, $f(n) \approx n/2$

Additional log n multiplications will be required for computing the powers of x.

The cost for addition, $g(k+1) = 2g(k) + 2$ where, $g(1) = 1$.

After simplification, $g(k+1) = 3 \cdot 2^k - 2$

or, in terms of n, $g(n) \approx 3n/2$

Hence the complexity after preconditioning is roughly ($n/2$ + log n) multiplications and 3n/2 additions.

Example:

Let the polynomial to be evaluated is

$$P(x) = x^7 - x^6 + 8x^5 - 4x^4 + 6x^3 + 2x^2 - 5x + 1.$$

We may write,

$$P(x) \quad (x^4 + \gamma)(x^3 + \alpha_2 x^2 + \alpha_1 x + \alpha_0) + (x^3 + \beta_2 x^2 + \beta_1 x + \beta_0)$$

equating we get, $\gamma=5$, $\alpha_2 = -1$, $\alpha_1 = 8$, $\alpha_0 = -4$, $\beta_2 = 7$, $\beta_1 = -45$, $\beta_0 = 21$.

Thus $P_1(x) = x^3 - x^2 + 8x - 4$

$= (x^2 + \gamma')(x + \alpha_0') + (x + \beta_0')$

where, $\gamma' = 7$, $\alpha_0' = -1$, $\beta_0' = 3$

and $P_2(x) = x^3 + 7x^2 - 45x + 21$

$= (x^2 + \gamma'')(x + \alpha_0'') + (x + \beta_0'')$

where, $\gamma'' = -46$, $\alpha_0'' = 7$, $\beta_0'' = 343$.

Now, to evaluate P(2.0), we get,

$P_1' = (2+\alpha_0') = 1$, $P_2' = (2+\beta_0') = 5$, $P_1'' = (2+\alpha_0'') = 9$, $P_2'' = (2+\beta_0'') = 345$.

$\therefore$ $P_1 = (2^2+\gamma')P_1' + P_2' = 16$, $P_2 = (2^2+\gamma'')P_1''+P_2'' = -33$.

$\therefore$ $P(2) = (2^4+\gamma)P_1+P_2 = 303$.

Knuth Algorithm :

Donald E. Knuth suggested another preprocessing scheme. His method is described below. Let the polynomial be expressed as

$$P_n(x) = \sum_{i=0}^{n} c_i x^i$$

This arbitrary polynomial of degree n can be expressed as

$$P_n(x) = (x^2-\alpha_n)P_{n-2}(x) + R_n \quad \text{..........................}\mathbf{(2.15)}$$

where, α_n is a constant, P_{n-2} is a polynomial of degree $(n-2)$, and R_n is of degree at most 1. Since α_n is arbitrary, it can be chosen to make R_n a constant. The factoring can be continued in the same way and thus would yield an $\lfloor n/2 \rfloor + 2$ multiplication algorithm.

To establish that appropriate set α_n and R_n can always be obtained, the following scheme can be used. We may write,

$$P_n(x) - R_n = (x^2 - \alpha_n)P_{n-2}(x)$$

which means that $\pm\sqrt{\alpha_n}$ are roots of $P_n(x) - R_n$.

If we let $\bar{c}_0 = c_0 - R_n$, then

$$\sum_{i=1}^{n} c_i \alpha_n^{i/2} + \bar{c}_0 = 0 \qquad \text{..........(2.16)}$$

and,

$$\sum_{i=1}^{n} c_i \alpha_n^{i/2} (-1)^i + \bar{c}_0 = 0 \qquad \text{..........(2.17)}$$

Assuming, $\alpha_n \neq 0$, adding and subtracting the above two equations, we get,

$$\sum_{i=1}^{\lfloor n/2 \rfloor} c_{2i} \alpha_n^i + \bar{c}_0 = 0 \qquad \text{..........(2.18)}$$

and,

$$\sum_{i=1}^{\lceil n/2 \rceil} c_{2i-1} \alpha_n^{i-\frac{1}{2}} = 0 \qquad \text{..........(2.19)}$$

Multiplying eqn. (2.19) by $\sqrt{\alpha_n}$, we get that α_n is a root of the eqn (2.19)

Some of the α's may be complex.

J. Eve[32] gave a theorem which can be used to make all the α's real. According to Eve, *If $P_n(x) = \sum c_i x^i$ has $n-1$ roots with non negative real parts, then all the roots of*

$$Q_n(x) \doteq \sum_{i=1}^{\lceil n/2 \rceil} c_{2i-1}\, x^{i-1}$$

are real.

Thus, $P_n(x)$ can be altered so that all its roots will have non negative real parts. If $\bar{P}_n(x) = P_n(x - r)$, where r is a positive number large enough to guarantee that all the roots of $\bar{P}_n(x)$ have positive real parts.

There is a very simple, though inefficient, way to determine the value of r. It has been established that any root of a polynomial $A(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$ will satisfy,

$$|x_k| < \left|1 + \frac{A}{a_0}\right|, \quad \textit{where} \quad A = \max\{|a_1|, |a_2|, \cdots, |a_n|\}.$$

So, by selecting r equal to an amount $\left|1 + \frac{A}{a_0}\right|$, $\bar{P}_n(x)$ will have roots, all with non negative real parts.

To evaluate $P_n(x)$, evaluation of $\bar{P}_n(x + r)$ will give the desired result.

CHAPTER THREE

MATRIX MULTIPLICATION

A matrix is a set of numbers arranged in a rectangular array written between parentheses or double lines on either side of the array. For example,

$$A \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & & & a_{2n} \\ a_{31} & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{ml} & a_{m2} & \cdot & \cdot & a_{mn} \end{bmatrix}$$

In the matrix, A = [a_{ij}], where for i =1,2,...,m and j =1,2,...,n, the element of the matrix in row i and column j is a_{ij}. The elements of a matrix are numbers from a number system, such as, real numbers, complex numbers etc. A matrix of m rows and n columns is called an (m x n) matrix. Matrices are usually denoted by capital bold faced letters. A matrix does not have any quantitative value. Matrices can be added, subtracted, multiplied, inverted, transposed etc.

Such sets and arrangements occur in various branches of applied mathematics. In many cases they are sets of coefficients of linear transformations or systems of linear equations arising, for instance, from electrical networks, frameworks in mechanics, curve fitting in statistics, and transportation problems. Matrices are useful because they enable us to consider an array of many numbers as a single object, denote it by a single symbol, and perform calculations with these symbols in a very compact form. The *mathematical shorthand* thus obtained is very elegant and powerful and is suitable for various problems. It entered engineering mathematics about sixty years ago and is of increasing importance in various branches.

Matrix multiplication is principally used in many successful algorithms of linear algebra, for example, solving a set of linear algebraic equations, matrix inversion, evaluation of the determinant, boolean matrix multiplication, etc. The complexity of a variety of algorithms such as those for performing transitive closure of graphs, parsing of context-free languages, etc., can be shown to depend on the complexity of matrix multiplication. In other words, if faster algorithms for matrix multiplication are developed, they may be applicable in speeding up the algorithms for solving a variety of interesting problems.

Some recently developed matrix multiplication algorithms are discussed here. These methods compute the product of two matrices using significantly fewer arithmetic operations compared with the classical technique. These algorithms are yet of theoretical interest only as they can actually supersede the standard method only when applied to solve problems of truly large size. Still, these new algorithms form a basis for the development of genuinely efficient algorithms for this important class of problems.

THE CLASSICAL METHOD

Two matrices can only be multiplied if the number of columns in the first matrix equals the number of rows in the second one and multiplication of matrices is not commutative. If **A** = ($\mathbf{a}_{ij}$) is an m x n matrix, **B** = (b_{jk}) is an n x p matrix, then their matrix product **C** = **AB** is an m x p matrix **C**=($\mathbf{c}_{ik}$), where,

$$c_{ik} \quad \sum_{i \le j \le n} a_{ij} b_{jk} \qquad \text{where, } 1 \le i \le m,\ 1 \le k \le p \qquad \ldots\ldots\ldots\ldots\ldots\ldots\mathbf{(3.1)}$$

The total number of operations required by this process is *mnp* multiplication and *mp(n - 1)* additions. If the matrices are square, i.e., $m = n = p$, the above process performs n^3 multiplications and $n^2(n-1)$ additions.

KARATSUBA'S METHOD

In 1962, A. Karatsuba, with Yu Ofman, developed a method for multiplying two matrices. His algorithm uses fewer number of additions and the same number of multiplications as compared with that of the classical method.

Let **C** be the product of the two matrices **A** and **B** each of order $m2^k$. We may represent A, B, C, as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},\ C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

The elements $\mathbf{C}_{ij}$ can now be computed using the formula

$$\mathbf{C}_{11} = \mathbf{A}_{11}(\mathbf{B}_{11}+\mathbf{B}_{21}) + (\mathbf{A}_{12}-\mathbf{A}_{11})\mathbf{B}_{21},$$

$$\mathbf{C}_{12} = \mathbf{A}_{12}(\mathbf{B}_{22}+\mathbf{B}_{12}) - (\mathbf{A}_{12}-\mathbf{A}_{11})\mathbf{B}_{12},$$

$$\mathbf{C}_{21} = \mathbf{A}_{21}(\mathbf{B}_{11}+\mathbf{B}_{21}) + (\mathbf{A}_{22}-\mathbf{A}_{21})\mathbf{B}_{21},$$

$$\mathbf{C}_{22} = \mathbf{A}_{22}(\mathbf{B}_{22}+\mathbf{B}_{12}) - (\mathbf{A}_{22}-\mathbf{A}_{21})\mathbf{B}_{12} \qquad \text{.........................(3.2)}$$

Another possibility is to use

$$\mathbf{C}_{11} = \mathbf{A}_{12}(\mathbf{B}_{11}+\mathbf{B}_{21}) - (\mathbf{A}_{12}-\mathbf{A}_{11})\mathbf{B}_{11},$$

$$\mathbf{C}_{12} = \mathbf{A}_{11}(\mathbf{B}_{11}+\mathbf{B}_{21}) - (\mathbf{A}_{12}-\mathbf{A}_{11})\mathbf{B}_{11},$$

$$\mathbf{C}_{21} = \mathbf{A}_{22}(\mathbf{B}_{12}+\mathbf{B}_{22}) + (\mathbf{A}_{22}-\mathbf{A}_{21})\mathbf{B}_{12},$$

$$\mathbf{C}_{22} = \mathbf{A}_{21}(\mathbf{B}_{12}+\mathbf{B}_{22}) - (\mathbf{A}_{22}-\mathbf{A}_{21})\mathbf{B}_{12}. \qquad \text{.......................(3.3)}$$

Let $\alpha_{m,k}$ denote the algorithm to multiply matrices of order $n = m2^k$, where $\alpha_{m,\,0}$ is the classical algorithm for matrix multiplication with m^3 multiplication and $m^2(m-1)$ additions. If $M\alpha_{m,\,k}$ denotes the number of multiplication required by $\alpha_{m,k}$, then

$$\begin{aligned} M\alpha_{m,k} &= 8\ M\alpha_{m,k-1} \\ &= 8^k\ M\alpha_{m,0} \\ &= 8^k\, m^3 \\ &= n^3 \end{aligned} \qquad \text{.........................(3.4)}$$

Let $S\alpha_{m,\,k}$ define the number of additions and subtractions. Since the algorithm requires 8 additions/subtractions of matrices of order $m2^{k-1}$ and also the 8 matrix multiplication of order $m2^{k-1}$ requires $S\alpha_{m,\,k-1}$ additions each. Thus

$$
\begin{aligned}
S\alpha_{m,k} &= 8\,(m2^{k-1})^2 + 8S\alpha_{m,k-1} \\
&= 8^k m^3 - 2(4^k)m^2 \\
&= (m2^k)^3 - 2(m2^k)^2 \\
&= n^3 - 2n^2 \qquad \text{.........................(3.5)}
\end{aligned}
$$

Thus when used recursively, Karatsuba method saves n^2 additions.

WINOGRAD'S METHOD

Samuel Winograd discovered, in 1967, that there is a way to trade about half of the multiplications for additions as compared to that of the classical algorithm. This method is based on the identity involving the sum of two pairwise products, as shown

$$z_{ik} = \sum_{i \le j \le \frac{n}{2}} (x_{i,2j} + y_{2j-1,k})(x_{i,2j-1} + y_{2j,k}) - a_i - b_k + c_{ik} \qquad \text{....................(3.6)}$$

where,
$$a_i = \sum_{1 \le j \le \frac{n}{2}} x_{i,2j}\, x_{i,2j-1}$$

$$b_k = \sum_{1 \le j \le \frac{n}{2}} y_{2j-1,k}\, y_{2j,k}$$

$$c_{ik} = \begin{cases} 0 & n \text{ even} \\ x_{in} y_{nk} & n \text{ odd} \end{cases}$$

This identity can be used in multiplying matrices. Let A and B be two matrices of dimension m x n and n x p, respectively. The algorithm for computing C=AB using the Winograd Method is given as

(assuming, n = 2k)

$$c_{ij} = \sum_{u=1}^{k} (a_{i,2u-1} + b_{2u,j})(a_{i,2u} + b_{2u-1,j}) - f_i - g_j \qquad i=1, \dots, m \qquad j=1, \dots, p$$

...........**(3.7)**

where, $$f_i = \sum_{u=1}^{k} a_{i,2u-1}\, a_{i,2u} \qquad i = 1,2, \dots, m$$

$$g_i = \sum_{u=1}^{k} b_{2u-1,j}\, b_{2u,j} \qquad j = 1,2, \dots, p$$

If n is not even, we can easily pad a 0 column to matrix A and 0 row to matrix B and apply the algorithm accordingly. The total number of operations required by this process is $\frac{mnp}{2} + \frac{n}{2}(m+p)$ multiplication and $\frac{3}{2}n\,mp + mp + \left(\frac{n}{2} - 1\right)(m+p)$ additions. If the matrices are square of dimension n by n, we shall need $\left(\frac{n^3}{2} + n^2\right)$ multiplications and $\left(\frac{3}{2}n^3 + 2n^2 - 2n\right)$ additions. By using Winograd's method, the number of multiplication is about halved while some price for this reduction is paid in terms of an increased number of additions required.

STRASSEN'S METHOD

In 1968, Volker Strassen discovered a clever scheme for multiplying matrices. Strassen's method computes the product of 2x2 matrices with only seven multiplication and eighteen additions. By using the method recursively, he was able to multiply two n by n matrices in times $O(n^{\log 7})$, which is of order approximately $n^{2.81}$.

Let **A**, **B** are two matrices of order $n\ (= m2^k)$ to be multiplied, we may write them as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

and their product is

$$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$ where the A_{ik}, B_{ik}, C_{ik} are matrices of order $m2^{k-1}$.

Then we compute

$\mathrm{I} = (\mathbf{A}_{11}+\mathbf{A}_{22})(\mathbf{B}_{11}+\mathbf{B}_{22})$,

$\mathrm{II} = (\mathbf{A}_{21}+\mathbf{A}_{22})\mathbf{B}_{11}$,

$\mathrm{III} = \mathbf{A}_{11}(\mathbf{B}_{12}-\mathbf{B}_{22})$,

$\mathrm{IV} = \mathbf{A}_{22}(-\mathbf{B}_{11}+\mathbf{B}_{21})$,

$\mathrm{V} = (\mathbf{A}_{11}+\mathbf{A}_{12})\mathbf{B}_{22}$,

$\mathrm{VI} = (-\mathbf{A}_{11}+\mathbf{A}_{21})(\mathbf{B}_{11}+\mathbf{B}_{12})$,

$\mathrm{VII} = (\mathbf{A}_{12}-\mathbf{A}_{22})(\mathbf{B}_{21}+\mathbf{B}_{22})$, ...**(3.8)**

C_{11} = I+IV-V+VII,

C_{21} = II+IV,

C_{12} = III+V

C_{22} = I+III-II+VI ....(3.9)

Let $\alpha_{m,k}$ denote the algorithm to multiply matrices of order $m2^k$, where $\alpha_{m,0}$ is the classical algorithm for matrix multiplication and requires m^3 multiplication and $m^2(m-1)$ additions. It follows that $\alpha_{m,k}$ requires 7 multiplication of matrices of order $m2^{k-1}$. Denoting by $M\alpha_{m,k}$ the number of multiplication required by algorithm $\alpha_{m,k}$, we get

$$\begin{aligned} M\alpha_{m,k} &= 7\, M\alpha_{m,k-1} \\ &= 7^k\, M\alpha_{m,0} \\ &= 7^k m^3 \qquad \text{.......................(3.10)} \end{aligned}$$

Similarly, $S\alpha_{m,k}$ defines the number of additions and subtractions required by the algorithm $\alpha_{m,k}$. Now from eqns. (3.8) and (3.9), it follows that $\alpha_{m,k}$ requires 18 additions of matrices of order $m2^{k-1}$. Also the multiplication of matrices of order $m2^{k-1}$, each requires $S\alpha_{m,k-1}$ additions. Thus

$$\begin{aligned} S\alpha_{m,k} &= 18\,(m2^{k-1})^2 + 7S\alpha_{m,k-1} \\ &= 18\,(m2^{k-1})^2\,[1 + 7/2^2 + 7^2/2^4 + ... + 7^{k-1}/(2^{k-1})^2] + 7^kS\alpha_{m,0} \\ &= (5+m)m^2\, 7^k - 6(m2^k)^2 \qquad \text{.........................(3.11)} \end{aligned}$$

From eqns. (3.10) and (3.11), the total number of arithmetic operations is

$$T\alpha_{m,k} = M\alpha_{m,k} + S\alpha_{m,k}$$

$$= 7^k m^3 + (5+m)m^2 7^k - 6(m2^k)^2$$

$$= (5+2m)m^2 7^k - 6(m2^k)^2$$

setting $k = \lfloor \log n - 4 \rfloor$ and $m = \lfloor n2^{-k} \rfloor +1$, with n assumed to be ≥ 16, and introduced to include all n in the range $m2^{k-1} < n \leq m2^k$, we get,

$$T\alpha_{m,k} < [5+2(n2^{-k})+1](n2^{-k}+1)^2 7^k$$

$$< 2n^3(7/8)^k + 12.03n^2(7/4)^k$$

now $16.2^k \preceq n,\ k \geq 0$

$$= [2(8/7)^{\log n - k} + 12.03(4/7)^{\log n - k}]\, n^{\log 7}$$

$$\preceq 4.7\, n^{\log 7} \qquad \text{........................(3.12)}$$

Thus using Strassen's matrix multiplication algorithm, the product of two matrices of order n can be computed using no more than 4.7 $n^{\log 7}$ arithmetic operations.

WINOGRAD VARIANT OF STRASSEN'S METHOD

S. Winograd further improved Strassen's method. He presented the following algorithm which uses 7 multiplication and 15 additions to multiply two 2 x 2 matrices. Let **A** and **B** be two matrices of order $n = m2^k$ and their product be denoted by **C**. We write

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

and their product is, $$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

where the $\mathbf{A}_{ik}$, $\mathbf{B}_{ik}$, $\mathbf{C}_{ik}$ are matrices of order $m2^{k-1}$. To evaluate $\mathbf{C}$, we compute

(Q1) $= \mathbf{A}_{21} - \mathbf{A}_{11}$

(Q2) $= \mathbf{A}_{11} + \mathbf{A}_{12}$

(Q3) $= \mathbf{A}_{12} - (Q1)$

(Q4) $= \mathbf{A}_{22} - (Q3)$

(Q5) $= \mathbf{B}_{22} - \mathbf{B}_{21}$

(Q6) $= \mathbf{B}_{12} - \mathbf{B}_{11}$

(Q7) $= \mathbf{B}_{21} - (Q5)$

(Q8) $= \mathbf{B}_{21} - (Q7)$ ....**(3.13)**

(P1) $= \mathbf{A}_{21}\mathbf{B}_{11}$

(P2) $= \mathbf{A}_{22}\mathbf{B}_{11}$

(P3) $= (Q1)(Q5)$

(P4) $= (Q2)(Q6)$

(P5) $= (Q4)\mathbf{B}_{22}$

$(P6) = \mathbf{A}_{12}(Q8)$

$(P7) = (Q3)(Q7)$

$(Q9) = (P1) + (P7)$

$(Q10) = (Q9) + (P7)$

$(Q11) = (P4) + (P5)$ ....**(3.14)**

$\mathbf{C}_{11} = (Q10) + (P6)$

$\mathbf{C}_{12} = (Q10) + (P4)$

$\mathbf{C}_{21} = (P1) + (P2)$

$\mathbf{C}_{22} = (Q9) + (Q11)$ ....**(3.15)**

Using Strassen's analysis of the number of arithmetic required to multiply n by n matrices, we get,

$$M\alpha_{m,k} = 7^k m^3 \qquad \text{..........................}\mathbf{(3.16)}$$

and

$$\begin{aligned} S\alpha_{m,k} &= 15\,(m2^{k-1})^2 + 7S\alpha_{m,k-1} \\ &= 15\,(m2^{k-1})^2\,[1 + 7/2^2 + 7^2/2^4 + \dots + 7^{k-1}/(2^{k-1})^2] + 7^k S\alpha_{m,0} \\ &= (4+m)m^2 7^k - 5(m2^k)^2 \qquad \text{..........................}\mathbf{(3.17)} \end{aligned}$$

From eqns. (3.16) and (3.17) the total number of arithmetic operations is

$$\begin{aligned} T\alpha_{m,k} &= M\alpha_{m,k} + S\alpha_{m,k} \\ &= 7^k m^3 + (4+m)m^2 7^k - 5(m2^k)^2 \\ &= (4+2m)m^2 7^k - 5(m2^k)^2 \end{aligned}$$

setting $k = \lfloor \log n - 4 \rfloor$ and $m = \lfloor n2^{-k} \rfloor + 1$, we get,

$$T\alpha_{m,k} < [4+2(n2^{-k}+1)](n2^{-k}+1)^2 7^k$$
$$< 2n^3(7/8)^k + 10.58n^2(7/4)^k$$

now $16.2^k \leq n, \; k \geq 0$

$$= [2(8/7)^{\log n - k} + 10.58(4/7)^{\log n - k}]\, n^{\log 7}$$
$$\leq 4.54\, n^{\log 7} \qquad \text{..........................(3.18)}$$

Thus by using Winograd's improved method, total arithmetic operations can be reduced to $4.54\, n^{\log 7}$.

TRILINEAR FORM

The evaluation of the product of *n* x *p* by *p* x *n* matrices and decomposing the trace product of three matrices of dimension *n* x *p* by *p* x *m* by *m* x *n* are two equivalent problems. Thus the bilinear form of the traditional algorithm

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \, , \quad i=1,\dots,m \, , \; j=1,\dots,p$$

can be represented in trilinear form as

$$\sum_{i,j,k} a_{ij} b_{jk} c_{ki}$$

here the coefficients of C_{ij} is the (j,i)th element of the product C.

To evaluate product of matrices, we may consider the trilinear form as

$$\begin{aligned}
&\sum_{i,j,k} a_{ij}\, b_{jk}\, c_{ki} \\
= \quad &\sum_{i+j+k \text{ even}} (a_{ij} + a_{k+1,i+1})(b_{jk} + b_{i+1,j+1})(c_{ki} + c_{j+1,k+1}) \\
- \quad &\sum_{i,k} (a_{k+1,i+1}) \sum_{j,\, i+j+k \text{ even}} (b_{jk} + b_{i+1,j+1})\, c_{ki} \\
- \quad &\sum_{i,j} a_{ij} b_{i+1,j+1} \sum_{k,\, i+j+k \text{ even}} (c_{ki} + c_{j+1,k+1}) \\
- \quad &\sum_{j,k} \sum_{i,\, i+j+k \text{ even}} (a_{ij} + a_{k+1,j+1})\, b_{jk}\, c_{j+1,k+1}
\end{aligned}$$

....................**(3.19)**

This method is inferior to Strassen's method but when n>6, this algorithm becomes faster than the classical method.

VICTOR PAN'S METHOD

There is more than one way in which the trilinear form can be represented. After considering various trilinear representation forms, Victor Pan derived a new algorithm which yields results superior to Strassen's method. The Pan algorithm represents the trilinear form in the following manner :

$$\sum_{i,j,k} a_{ij}\, b_{jk} c_{ki} = T_0 - T_1 - T_2 - T_3 ,$$ ....**(3.20)**

where,

$$
\begin{aligned}
T_0 = \sum_{i,j,k \in S'(s)} & (a_{ij}+a_{jk}+a_{ki})(b_{jk}+b_{ki}+b_{ij})(c_{ki}+c_{ij}+c_{jk}) \\
& - (a_{ij}-a_{jk}+a_{ki})(b_{jk}+b_{ki}-b_{ij})(-c_{ki}+c_{ij}+c_{jk}) \\
& - (-a_{ij}+a_{jk}+a_{ki})(b_{jk}-b_{ki}+b_{ij})(c_{ki}+c_{ij}+c_{jk}) \\
& - (a_{ij}+a_{jk}-a_{ki})(-b_{jk}+b_{ki}+b_{ij})(c_{ki}-c_{ij}+c_{jk}) \\
& - (a_{ij}+a_{jk}-a_{ki})(-b_{jk}+b_{ki}+b_{ij})(c_{ki}-c_{ij}+c_{jk}) \\
& - (-a_{ij}+a_{jk}+a_{ki})(b_{jk}-b_{ki}+b_{ij})(-c_{ki}+c_{ij}-c_{jk}) \\
& - (a_{ij}-a_{jk}+a_{ki})(b_{jk}+b_{ki}-b_{ij})(-c_{ki}+c_{ij}+c_{jk}) \\
& + (a_{ij}+a_{jk}+a_{ki})(b_{jk}+b_{ki}+b_{ij})(c_{ki}+c_{ij}+c_{jk})
\end{aligned}
$$

$$
\begin{aligned}
T_1 = \sum_{1 \le i,j \le s} & a_{ij}b_{ij}\Big[(s-2w_{ij})\,c_{ij}+\sum{}'(c_{ki}+c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-w_{ij})\,c_{ij}+\sum{}'(-c_{ki}+c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-w_{ij})\,c_{ij}+w_{ji}c_{ji}+\sum{}'(c_{ki}-c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-w_{ij})\,c_{ij}-\sum{}'(c_{ki}+c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-w_{ij})\,c_{ij}-\sum{}'(c_{ki}+c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-w_{ij})\,c_{ij}-w_{ji}c_{ji}+\sum{}'(c_{ki}-c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-w_{ij})\,c_{ij}+\sum{}'(-c_{ki}+c_{jk})\Big] \\
& + a_{ij}b_{ij}\Big[(s-2w_{ij})\,c_{ij}+\sum{}'(c_{ki}+c_{jk})\Big]
\end{aligned}
$$

$$
\begin{aligned}
T_2 = \sum_{1 \le i,j \le s} & \{a_{ij}\sum{}'(b_{ki}+b_{jk})\,c_{ij} - a_{ij}\sum{}'(b_{ki}+b_{jk})\,c_{ij} \\
& + a_{ij}\sum{}'(b_{jk}-b_{ki})\,c_{ij} + a_{ij}\sum{}'[(b_{ki}-b_{jk})-w_{ji}b_{ji}]\,c_{ij} \\
& + a_{ij}[\sum{}'(b_{ki}-b_{jk})-w_{ji}b_{ji}]\,c_{ij} + a_{ij}\sum{}'(b_{jk}-b_{ki})\,cij \\
& - a_{ij}\sum{}'(b_{ki}+b_{jk})\,c_{ij} + a_{ij}\sum{}'(b_{ki}+b_{jk})\,c_{ij}\}
\end{aligned}
$$

$$\begin{aligned} T_3 = \sum_{1 \le i,j \le s} \{ & \sum{}' (a_{ki} + a_{jk}) b_{ij} c_{ij} + \sum{}' [(a_{ki} - a_{jk}) - w_{ji} a_{ji}] b_{ij} c_{ij} \\ & - \sum{}' (a_{ki} + a_{jk}) b_{ij} c_{ij} + \sum{}' (a_{jk} - a_{ki}) b_{ij} c_{ij} \\ & + \sum{}' (a_{jk} - a_{ki}) b_{ij} c_{ij} - \sum{}' (a_{ki} + a_{jk}) b_{ij} c_{ij} \\ & + \sum{}' [(a_{ki} - a_{jk}) - w_{ji} a_{ji}] b_{ij} c_{ij} + \sum{}' (a_{ki} + a_{jk}) b_{ij} c_{ij} \} \end{aligned}$$

Here,

$$S'(s) = S_1'(s) \cup S_2'(s)$$
$$S_1'(s) \quad \{(i,j,k),\ 1 \le i \le j < k \le s\}$$
$$S_2'(s) \quad \{(i,j,k),\ 1 \le k < j \le i \le s\}$$
$$n = 2s, \quad \hat{i} = i + s, \quad \hat{j} = j + s, \quad \hat{k} = k + s,$$
$$w_{pq} = \begin{cases} 1 & \textit{if } p = q, \\ 0 & \textit{if } p \ne q, \end{cases}$$
$$\sum{}' \quad \sum_{k=1}^{s} \quad \textit{if } i \ \ j \textit{ then } k \ne i.$$

It can be seen that the number of terms in T_0 is $8(s^3-s)/3$, and each of T_1, T_2, T_3 has $8s^2$ terms. Therefore the complexity of the algorithm is

$$8(s^3 - s)/3 + 24s^2$$
$$= (n^3 - 4n)/3 + 6n^2$$

This is still $O(n^3)$ but still a reduction in the number of multiplications comes from the low coefficients of the dominant term in the complexity function.

MATRIX MULTIPLICATION IN PREPROCESSING OF SYSTEMS OF LINEAR EQUATIONS

The system of linear equations is one of the most important problems that occur in the solution of many practical problems. Direct methods like Gaussian elimination and triangular decomposition solve the problem in $O(n^3)$ arithmetic operations. Indirect iterative methods like Gauss-Seidel, Jacobi take $O(n^2)$ arithmetic operations. Thus iterative schemes are attractive from computational point of view. But the main drawback is that these iterative methods cannot necessarily ensure convergence of an arbitrary system of linear equations. For convergence, the systems must satisfy some stringent constraints.

Let us consider the non singular system of linear equations : Ax=b, where A is an arbitrary n x n matrix. Solving the above equation is equivalent to solving $A^t Ax = A^t b$. This preprocessing makes the system symmetric and positive definite for which there are many convergent iterative schemes of $O(n^2)$.

The cost involvement in the preprocessing can be calculated as shown below. For simplicity of derivation, let us assume that $n=2^k$ for some integer k.

Let, $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ where A_{ij} is a matrix itself.

then, $A^t = \begin{bmatrix} A_{11}^t & A_{21}^t \\ A_{12}^t & A_{22}^t \end{bmatrix}$

therefore, $$A^t.A = \begin{bmatrix} A_{11}^t A_{11} + A_{21}^t A_{21} & A_{11}^t A_{12} + A_{21}^t A_{22} \\ A_{12}^t A_{11} + A_{22}^t A_{21} & A_{12}^t A_{12} + A_{22}^t A_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \qquad \text{.....................}\textbf{(3.21)}$$

The product of two matrices thus can be defined as follows :

$$\begin{aligned}
P_0 &\quad A_{11}^{t}A_{11} \\
P_1 &\quad A_{22}^{t}A_{22} \\
P_2 &= A_{12}^{t}A_{12} \\
P_3 &= A_{21}^{t}A_{21} \\
P_4 &= A_{11}^{t}A_{12} \\
P_5 &= A_{21}^{t}A_{22} \\
C_{11} &= P_0 + P_3 \\
C_{22} &\quad P_2 + P_1 \\
C_{12} &\quad P_4 + P_5 \\
C_{21} &= C_{12}^{t}
\end{aligned} \qquad \text{..............................} \textbf{(3.22)}$$

Therefore, computation of A'A of order 2^k requires 4 multiplications of matrices of order 2^{k-1} with their transposes and 2 general matrix multiplications of order 2^{k-1}.

If $M^s(k)$ denote the complexity of multiplication of matrix A of order 2^k with its transpose and M(k) denote the complexity of multiplication of two arbitrary matrices of order 2^k, then

$$\begin{aligned}
M^s(k) &= 4M^s(k-1) + 2M(k-1) \\
&= 4[4M^s(k-2) + 2M(k-2)] + 2M(k-1) \qquad \text{...........} \textbf{(3.23)} \\
&= 4^{k-1}M^s(1) + 2\left[4^{k-2}M(1) + 4^{k-3}M(2) + \ldots + 4^{k-k}M(k-1)\right]
\end{aligned}$$

Now multiplication of a 2x2 matrix with its transpose requires only 5 multiplications. Thus, $M^s(1) = 5$. If we use Strassen's multiplication scheme which requires 7^k multiplications for multiplication of two matrices of order 2^k. Then, $M(k) = 7^k$.

$$\begin{aligned} M^s(k) &= 4^{k-1}.5 + 2\left[4^{k-2}.7^1 + 4^{k-3}.7^2 + \dots + 4^0.7^{k-1}\right] \\ &= \frac{5}{4}\cdot 4^k + 2\times 4^{k-1}\left[\frac{7}{4} + \frac{7^2}{4^2} + \dots + \left(\frac{7}{k}\right)^{k-1}\right] \\ &= \frac{5}{4}(2^k)^2 + 2\times 4^{k-1}\,\frac{\frac{7}{4}\left(\frac{7^{k-1}}{4^{k-1}} - 1\right)}{\frac{7}{4} - 1} \\ &= \frac{5}{4}n^2 + 2\times 4^{k-1}\,\frac{7\left(\frac{7^{k-1}}{4^{k-1}} - 1\right)}{3} \\ &= \frac{5}{4}n^2 + \frac{2}{3}\times\left[7^k - 7\times 4^{k-1}\right] \\ &= \frac{5}{4}n^2 + \frac{2}{3}n^{\log_2 7} - \frac{7}{6}n^2 \\ &= \frac{1}{12}n^2 + \frac{2}{3}n^{\log_2 7} \end{aligned}$$

..........................(3.24)

In a similar fashion, the complexity of addition can be computed. Let, $A^s(k)$ denote the number of addition operations required for multiplication of one matrix of order 2^k with its transpose and $A(k)$ denote the number of addition operations required for multiplication of one matrix with any arbitrary matrix of order 2^k.

Since, every matrix multiplication with its transpose requires 3 matrix additions. Two additions of symmetric matrices of order 2^{k-1} and one addition of two non symmetric matrix. Therefore,

$$\begin{aligned} A^s(k) &= 4A^s(k-1) + 2A(k-1) + 2\left(2^{k-1}\right)^2 + 2^{k-1} \\ &= 4\left[4A^s(k-2) + 2A(k-2) + 2\left(2^{k-1}\right)^2 + 2^{k-1}\right] + 2A(k-1) + 2\left(2^{k-1}\right)^2 + 2^{k-1} \\ &= 4^{k-1}A^s(1) + 2\left[4^{k-2}A(1) + 4^{k-3}A(2) + \dots + 4^{k-k}A(k-1)\right] \\ &\quad + 2\left[4^{k-2}(2^1)^2 + 4^{k-3}(2^2)^2 + \dots + 4^{k-k}(2^{k-1})^2\right] \\ &\quad + \left[4^{k-2}2^1 + 4^{k-3}2^2 + \dots + 4^{k-k}2^{k-1}\right] \end{aligned}$$

..........................(3.25)

Now, only 3 additions are required while multiplying a 2x2 matrix with its transpose, i.e., $A^s(1)=3$. If we use Winograd's variation of Strassen's matrix multiplication scheme, then $A(k) \quad 5\cdot[7^k \quad 4^k]$. Therefore,

$$\begin{aligned}
A^s(k) \quad & \frac{3}{4}n^2 + 10\left[4^{k-2}(7^1 \quad 4^1) + 4^{k-3}(7^2-4^2) + \ldots + 4^0(7^{k-1}-4^{k-1})\right] \\
& + 2\left[4^{k-1} + 4^{k-1} + \ldots + 4^{k-1}\right] \\
& + \left[2^{2k-3} + 2^{2k-4} + \ldots + 2^{k-1}\right] \\
& \frac{3}{4}n^2 + 10 \times 4^{k \quad 1}\frac{\frac{7}{4}\left(\frac{7^{k-1}}{4^{k \quad 1}} \quad 1\right)}{\frac{7}{4} \quad 1} \quad 8(k \quad 1)\,4^{k \quad 1} + 2^{2k \quad 3} \times \frac{1 \quad \frac{1}{2^{k \quad 1}}}{1 \quad \frac{1}{2}} \\
= \quad & \frac{3}{4}n^2 + \frac{10}{3}4^{k-1}\left(\frac{7^k}{4^{k-1}} - 7\right) - 2n^2\log_2 n + 2n^2 + 2^{2k-2}\left(1 - \frac{1}{2^{k-1}}\right) \\
& \frac{10}{3}n^{\log_2 7} \quad \frac{17}{6}n^2 \quad 2n^2\log_2 n \quad \frac{1}{2}n
\end{aligned}$$

..........................**(3.26)**

To compute $A^t b$ will need n^2 multiplications and $(n-1)^2$ additions. Thus total arithmetic operation for the preprocessing is

$$\begin{aligned}
T^s(k) \quad = \quad & M^s(k) + A^s(k) + n^2 + (n-1)^2 \\
& \frac{1}{12}n^2 + \frac{2}{3}n^{\log_2 7} + \frac{10}{3}n^{\log_2 7} \quad \frac{17}{6}n^2 - 2n^2\log_2 n - \frac{1}{2}n + n^2 + (n-1)^2 \\
& 4n^{\log_2 7} \quad \frac{3}{4}n^2 \quad 2n^2\log_2 n + \frac{3}{2}n + 1
\end{aligned}$$

..........................**(3.27)**

As this is a recursive method, space requirement is high. At the ith step of recursion, maximum number of storage required when matrix C is determined from matrix P_0 to P_5. In all previous stages of recursion, 6 matrices of specified size are stored. Thus total requirement is given by

$$6\left(\frac{n}{2}\right)^2 + 6\left(\frac{n}{4}\right)^2 + 6\left(\frac{n}{8}\right)^2 + \ldots + 6\left(\frac{n}{2^{i-1}}\right)^2 + 6\left(\frac{n}{2^i}\right)^2$$

$$6n^2\left[\frac{1}{4} + \frac{1}{16} + \ldots + \frac{1}{4^{i-1}} + \frac{1}{4^i}\right]$$

$$\frac{6n^2}{3}\left(1 \quad \frac{1}{4^i}\right)$$

$$\leq 2n^2 \qquad \text{.......................}\mathbf{(3.28)}$$

For i=1, this space requirement is maximum, and equals $2n^2$ data words.

CHAPTER FOUR

CHAIN MATRIX MULTIPLICATION

Let $M_1 \times M_2 \times \ldots \times M_n$ be a chain of matrices to be multiplied where M_i is a $k_i \times k_{i+1}$ matrix. Matrix multiplication is associative and thus this chain may be evaluated in several different ways. Of these, two possibilities are $(\ldots((M_1 \times M_2) \times M_3) \times M_4) \times \ldots) \times M_n$ and $(M_1 \times (M_2 \times (\ldots \times (M_{r-1} \times M_n) \ldots)$. It can be shown that the number of different ways to evaluate such a matrix product chain is $\frac{[2(n-1)]!}{n!\,(n-1)!}$ which is very large even when n is relatively small. All of these arrangements yield the same result. But the way one order the multiplication operation can have a dramatic impact on the cost of evaluating the product. Here the term cost denotes the number of scaler multiplications needed to compute the product. If the classical method is used, multiplication of a $p \times q$ matrix by a $q \times r$ matrix requires pqr scaler multiplications and the product is a $p \times r$ matrix. Though there are other efficient (see Chapter 4) matrix multiplication methods, the multiplication cost is considered pqr in this chapter.

Let $P(n)$ denote the number of alternative parenthesization of a sequence of n matrices. Since we can split a sequence of n matrices between the kth and (k+1)st matrices for any k = 1,2,...,n−1 and then parenthesize the two resulting subsequences independently, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)\,P(n-k), & \text{if } n \geq 2 \end{cases} \qquad \text{.................}\textbf{(4.1)}$$

The solution to this recurrence is the sequence of Catalan numbers.

$\therefore \quad P(n) = C(n - 1)$

$$\begin{aligned} where,\ C(n) &= \frac{1}{n+1}\binom{2n}{n} \\ &= \frac{4^n}{\sqrt{\pi}\, n^{3/2}}(1 + O(n)) \\ &= O(4^n / n^{3/2}) \end{aligned} \qquad \text{.................}\textbf{(4.2)}$$

The number of possible arrangements is thus exponential in *n*, and the brute force method of exhaustive search is therefore a poor strategy for determining the optimal parenthesization of a matrix chain. To find the optimum order of multiplication several schemes may be employed.

DYNAMIC PROGRAMMING APPROACH

Dynamic programming approach is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions. An optimal sequence of decisions may be found by making the decisions one at a time and never making an erroneous decision. This technique can be used to find the order of the matrix multiplication that minimizes the total number of scaler multiplications used.

Let the notations $M_{i..j}$ denote the resulting matrix from evaluating the product $M_i M_{i+1} \dots M_j$. An optimal ordering splits the product $M_1 M_2 \dots M_n$ between M_p and M_{p+1} for some p in the range $1 \leq p \leq n$. That is, for some value p, the matrices $M_{1..p}$ and $M_{p+1..n}$ are first computed and then these are multiplied to produce the final product $M_{1..n}$. The cost is thus the cost of computing the matrix $M_{1..p}$, plus the cost computing $M_{p+1..n}$, plus the cost of multiplying them together. Thus it can be written,

$$C_{i,j} = \begin{cases} 0, & \text{when } i \; j \\ \min\limits_{i\leq p\leq j} \left[C_{i..p} + C_{p+1..j} + k_{i-1} k_p k_j\right], & \text{when } i<j \end{cases} \qquad \dots\dots\dots\dots\dots\dots (4.3)$$

where $C_{i..j}$ denote the optimum cost for generating $M_{i..j}$. The computation is done `bottom up', saving computed answers to small partial problems to avoid recomputation. In only one way one can multiply M_1 by M_2, M_2 by M_3, ... , M_{n-1} by M_n and these costs are recorded. Then the best way to multiply successive triples are computed using all the information computed so far. For example M_1 M_2 M_3 can be computed either by computing M_1 x M_2 first or M_2 x M_3 first. These costs have already been computed in the previous step and no need to compute it again. The minimum cost of the two approach is computed and saved again for reference in the following steps. This procedure continues until successive groups of n matrices is formed and thus generate the best way to order a matrix-chain.

The pseudocode is given below.

```
procedure dynamic ( k, n, cost, order )
        for i ← j to n do
                for j ← i+1 to n do
                        cost [i][j] ← ∝
                repeat (j)
        repeat (i)
        for i ← 1 to n do
                cost [i][i] ← 0
        repeat (i)
        for j ← 1 to n-1 do
                for i ← 1 to n-j do
                        for m ← i+1 to i+j do
                                t ← cost [i][m-1] = cost [m][i+j] + k[i] * k[m] * k[i+j+1]
                                if (t < cost [i][i+j]) then
                                        cost [i][i+j] ← t
                                        order [i][i+j] ← m
                                endif
                        repeat (m)
                repeat (i)
        repeat (j)
end dynamic
```

Algorithm 4.1

In the above algorithm, the loops are nested three deep and each loop index takes on at most n values. Thus the running time of this algorithm is $O(n^3)$. It requires $O(n^2)$ space to store **cost** and **order** tables. Thus this algorithm is much more efficient than the exponential time method of enumerating all possible combinations and checking each one.

CHIN'S METHOD

Francis Y. Chin, in 1978, presented an $O(n)$ algorithm for determining a near-optimal computation order of matrix chain products [19]. This algorithm takes less than 25 percent longer than the optimal time. Although, in most cases, the algorithm yields the optimal order or an order which takes only a few percent (usually less than one percent on the average) longer than the optimal time.

Consider the evaluation of the product of n matrices,

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

where, M_i is a $k_{i-1} \times k_i$ matrix with each $k_i \geq 1$.

Chin establishes the following two theorems to derive a near-optimal order.

Theorem 1 : *For all i, if*

$$\left(\frac{1}{k_{i-1}} + \frac{1}{k_{i+1}}\right) > \left(\frac{1}{k_i} + \frac{1}{k_*}\right), \quad \text{where } k_* = \min \{ k_i \}$$

holds, then $(M_i \times M_{i+1})$ *must be in the optimal order.*

Theorem 1 provides a sufficient condition to determine whether two matrices are associated in optimal order. Each pair of adjacent matrices in a matrix chain are scanned and associated if they satisfy Theorem 1. The matrix chain is shortened by replacing all the associated terms in the order with single matrices. This procedure is iterated until no remaining part of adjacent matrices satisfies the above theorem and no more shortened matrix chain can be done. Theorem 2 comes into action at this stage.

Theorem 2 : *Given a reduced matrix chain, the order*

$$M = (M_1 \times \ldots (M_{m-1} \times M_m)\ldots) \times (\ldots(M_{m+1} \times M_{m+2}) \times \ldots \times M_n), \quad \textit{where } k_m = k_*$$

guarantees that $T < 1.2485\ T_{opt}$

So, after obtaining the reduced matrix chain, index *m* is searched for so that $k_m = k_*$ and the matrices are associated both ways from *m*. The algorithm in pseudocode is given below.

```
procedure mat_chain (K, V, n)
// K denotes the dimensions of the matrices,
            V the order vector to be returned and
            n the number of matrices in the chain//
        V[ ] ← 0; c ← 1;  b ← n - 1
        for i = 0 to n do
                $r_i$ = 1 / $k_i$
                if ( rm < $r_i$ ) rm = $r_i$
        repeat
        j ← 0 ; s[j] ← 0
        for i = 1 to n-1 do
                j ← j+1 ; s[j] ← i
                while ( j > 0 AND $r_{s[j]}$ + rm < $r_{s[j-1]}$ + $r_{i+1}$ ) do
                        $v_{s[j]}$ ← c
                        c ← c+1 ; j ← j - 1
                endwhile
        repeat
        $j \leftarrow j + 1$; $s[j] \leftarrow n$; $k \leftarrow 0$; $flag \leftarrow 0$
        while ($flag = 0$) do
            $flag \leftarrow 1$
            if ($r_{s[k]} < r_{s[j]}$) then
                if ($r_{s[j]} + rm < r_{s[j-1]} + r_{s[k]}$) then
                    $j \leftarrow j - 1$
                    $v_{s[j]} \leftarrow b$
                    $b \leftarrow b - 1$
                    $flag \leftarrow 0$
                elseif ($r_{s[k]} + rm < r_{s[k+1]} + r_{s[j]}$) then
                    $k \leftarrow k + 1$
                    $v_{s[k]} \leftarrow b$
                    $b \leftarrow b - 1$
                    $flag \leftarrow 0$
                endif
            endif
        for $i = m-1$ to $s[k]+1$ step $-1$ do
            if ($v_i = 0$) then
                $v_i \leftarrow c$
                $c \leftarrow c+1$
            endif
        repeat
        for $i = m+1$ to $s[j]-1$ do
            if ($v_i = 0$) then
                $v_i \leftarrow c$
                $c \leftarrow c+1$
            endif
        repeat
        if ( $v_m = 0$ AND $m \neq 0, n$) then
            $v_m \leftarrow c$
        endif
    end procedure
```

Algorithm 4.2

In the above algorithm, calculation of r_i and rm can be done in $O(n)$ time. Initially $c = 0$ and $b = n - 1$; besides throughout the program $b \geq c - 1$. The procedure has four different loops.

In each loop, either c is increased by 1 or b is decreased by 1 after assigning the value of c or b to v_i. Since at most (n - 1) associations can be done, the sum of repetitions in all the loops is no more than n. Thus the algorithm is $O(n)$.

HU-SHING HEURISTIC METHOD

T.C. Hu and M.T. Shing[20] proposed a heuristic algorithm to find a near-optimum order for multiplying a number of matrices. They drew an analogy between matrix chain product problem and the problem of partitioning a convex polygon into non-intersecting triangles, and develop an $O(n)$ heuristic algorithm which has a 15% error in the worst case. .

The one-to-one correspondence between the ordering of a chain of *n* matrices and the partitioning of a convex *(n + 1)*-gon can be established as follows. For the *(n + 1)*-gon, the side V_1-V_{n+1} is drawn horizontally. All other sides are considered in the clockwise direction. Every vertex V_i of the polygon is assigned a weight k_i. Let the cost of a triangle be the product of the weights of its three vertices, and the cost of partitioning a polygon is the sum of the costs of all its non-intersecting triangles. Then it can be assumed that each side represent a matrix in the chain where the dimensions of a matrix are the two weights associated with the two end vertices of that side. For example, the V_1-V_2 side represent the first matrix M_1 in the chain, the V_2-V_3 side denote M_2, and so on. Then the base V_1-V_n represent the resulting product matrix M. The cost of partitioning is then the cost of the matrix chain product.

Given an *$(n+1)$*-sided convex polygon, the number of ways to partition the polygon into $(n-1)$ triangles by non-intersecting diagonals is the Catalan numbers.

To optimally partition a polygon, several theorems were presented and established by Hu and Shing.

Theorem : *In every optimum partition of a polygon, the smallest vertex V_1 is always connected with the second smallest vertex V_2 and also with the third smallest vertex V_3.*

This fact can be used recursively. To find the optimal partition of a given polygon, it is decomposed into subpolygons by joining the smallest vertex with the second smallest and third smallest vertices repeatedly, until each of these subpolygons has the property that its smallest vertex is adjacent to both its second smallest and third smallest vertices. Such a polygon can be referred to as a *basic polygon.*

Theorem : *(i) A necessary but not sufficient condition for $V_2 - V_3$ to exist in an optimum partition of a basic polygon is*

$$\frac{1}{k_1} + \frac{1}{k_4} \leq \frac{1}{k_2} + \frac{1}{k_3}$$

(ii) If V_2 and V_3 are not connected in an optimum partition, then V_1 and V_4 are always connected in that optimum partition.

Theorem : *Let k_1 be the minimum vertex of a general convex polygon, and k_m be a local maximum vertex, with k_p and k_q as its two neighbors, i.e., $k_m > k_p$ and $k_m > k_q$. If*

$$\frac{1}{k_p} + \frac{1}{k_q} > \frac{1}{k_1} + \frac{1}{k_m}$$

then $w_p - w_q$ will exist in the optimum partition of the polygon.

Based on the above observations, T. C. Hu and M. T. Shing gave the following heuristic algorithm. The algorithm is based on two intuitions :

(i) if a vertex has a very large weight, it should be cut out in the optimum partition;

(ii) if none of the vertices has a very large weight, the fan[1] with the smallest vertex k_1 as its center should be near optimum.

Thus the algorithm can be implemented in the following manner.

STEP 1: Start from the smallest vertex k_1, travel in the clockwise direction around the polygon, and push the weights of the vertices successively onto a stack. Thus k_1 will be at the bottom of the stack. Let k_t be the top element on the stack, k_{t-1} be the element immediately below k_t, and k_c be the element to be pushed onto the stack. If there are two or more vertices on the stack and $\frac{1}{k_{t-1}} + \frac{1}{k_c} > \frac{1}{k_1} + \frac{1}{k_t}$, then join $k_{t-1} - k_c$ and pop k_t off

[1] A partition which is made up entirely of arcs joining the smallest vertex to all other vertices is called a *fan*.

the stack, else push k_c onto the stack. Repeat this step until the n^{th} vertex has been pushed onto the stack.

STEP 2: If there are more than 3 vertices on the stack, join $k_1 - k_{t-1}$, pop k_t off the stack and repeat this step, else stop.

The STEP 1 in the above algorithm cuts out a vertex if its weight is sufficiently large and STEP 2 joins k_1 to all vertices which have not been cut off. If there are two or more vertices with weights equal to the smallest weight w_1, arbitrarily any one can be chosen as vertex k_1.

CHAPTER FIVE

EXPERIMENTAL RESULTS

This chapter is intended to present the experimental results based on the algorithms discussed so far in the previous chapters. Three different arithmetic operations were considered for this study. This chapter comprises three sections, each dealing with one type of arithmetic operation. Each section begins with a brief description of the experimental setup for that section. It should be emphasized that one can go a long way without limit if one likes to carry out experiments with the algorithms in all possible respects. But, in this study, some representative properties of the algorithm were chosen as the basis of the experiments. Mainly the time complexity has been chosen to be the major concern, though in some cases (where appropriate) space complexity, number of multiplications etc. were also considered.

The PC used for the experiments bears the following properties:

Processor	:	486SX
RAM	:	4 MB
Clock Speed	:	33 MHz

All codes were written in C and were compiled using Turbo C++ 3.0 compiler of Borland International.

Polynomial Evaluation Algorithms :

This section deals with experiments on the algorithms for polynomial evaluation. Polynomial Evaluation is one of the most frequently occurring task in numerical computation. Chapter two presented various polynomial representation techniques. These representations occur naturally and do not offer reduced number of multiplications for computation purpose. For this reason, the standard representation of polynomial is assumed throughout this study. Since, in many computation, polynomials are usually evaluated at many different points, some artificial representation that offer better computation time are tried. The latter half of chapter two discusses three evaluation algorithms that involved preprocessing.

As the test data, several data sets of varying order were prepared. The Belaga algorithm sometimes generated complex quantities. Since complex arithmetic require extra operations, only those data sets, that do not generate complex Belaga coefficients, were used so that a comparative study can be carried out. The experimental findings are presented in Fig. 5.1 and Fig. 5.2.

Matrix Multiplication Algorithms :

In Chapter three, seven different algorithms for matrix multiplication were presented. To test these algorithms, ten sets of square matrices of order 4, 8, 16, 32, 64, 128 were created. The matrix elements were chosen to be both integers (as they require less memory space) and floating point numbers and were generated randomly.

90292

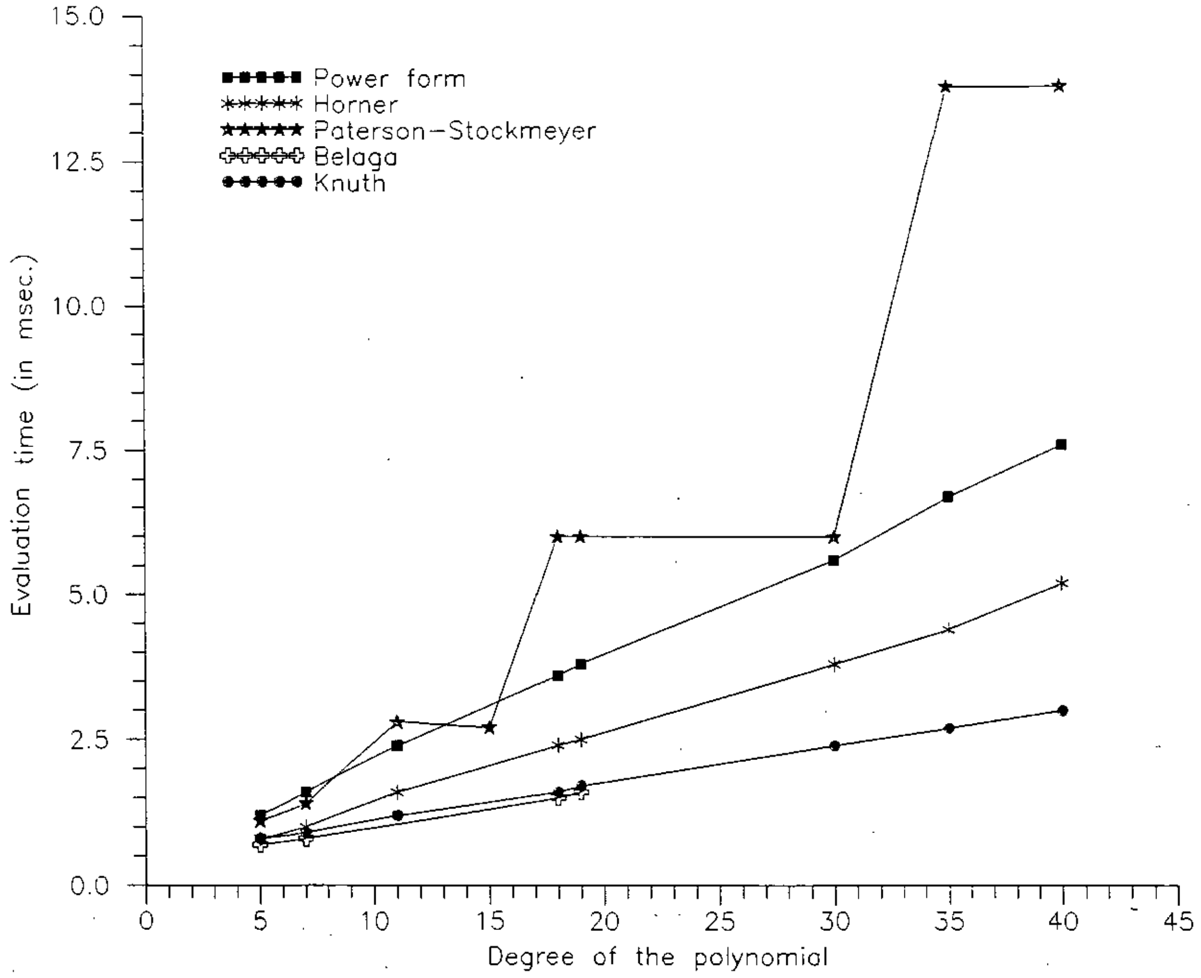

Fig. 5.1 Evaluation time for different algorithms

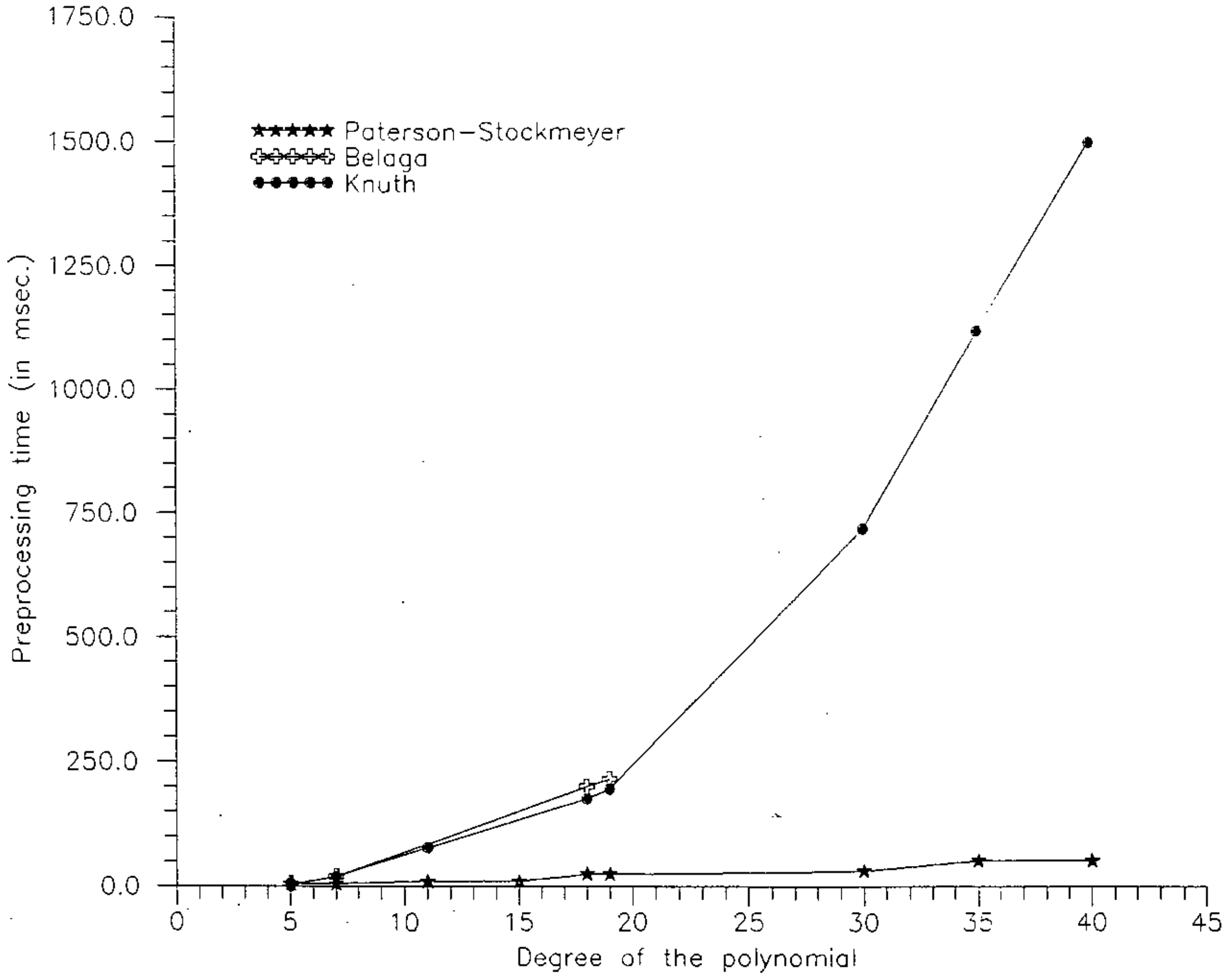

Fig. 5.2 Preprocessing time for different algorithms

The main interest was on the computation time. In addition, the memory requirement for some of these algorithms is substantial. In addition to data and product matrices, intermediate results of the manipulation need to be stored temporarily. The total volume of memory required may become an important factor which limit the size of the problem that can be solved on a particular computer. Speed of the process is also influenced by the amount of auxiliary memory used.

In most of the matrix multiplication algorithms, number of costly multiplication is usually reduced at the cost of extra addition/subtractions. In addition, memory accesses for intermediate and initial data play a considerable role in the computation time. Additional complexities are added by logical comparisons, procedure calls, looping overheads etc.

Based on the above considerations, criteria selected to measure the performance of these algorithms are

a) time complexity
b) space complexity
c) number of fundamental arithmetic operations
d) number of assignments and memory accesses

The experimental findings are presented in TABLE 5.1 through TABLE 5.7 and Fig. 5.3 through Fig. 5.12.

We have performed an experiment of solving systems of linear equations by preprocessing the system to obtain a positive definite system which can be solved by using many convergent iterative schemes. In this case matrix A of the system of linear equations is premultiplied by its transpose to obtain a positive definite symmetric system. For computing

AA' we used Strassen's scheme recursively to obtain some savings for finding the transposed product. It may be recollected that solving a system of linear equations is equivalent to matrix multiplication in terms of complexity. Approximate savings that can result from computing AA' by our method compared to multiplying two general matrices using Strassen's scheme has been presented in TABLE 5.8. A 90 MHz Pentium machine with 16MB RAM and 512 KB cache was used for this experiment.

TABLE 5.1 Experimental results for matrix multiplication using Classical method

order		4	8	16	32	64	128
time in msec.	integer data	0.21978	1.813187	14.120879	109.8901	901.0989	7241.7582
	float data	7.197802	56.593407	460.54945	3626.374	29065.924	223516.48
no. of additions		64	512	4096	32768	262144	2097152
no. of multiplications		64	512	4096	32768	262144	2097152
memory access	array read	192	1536	12288	98304	786432	6291456
	array write	64	512	4096	32768	262144	2097152
	other read	0	0	0	0	0	0
	other write	0	0	0	0	0	0
space		48	192	768	3072	12288	49152

TABLE 5.2 Experimental results for matrix multiplication using Karatsuba's method

order		4	8	16	32	64	128
time in msec.	integer data	0.824176	8.186813	67.69231	494.505	3846.154	31208.79
	float data	10.93407	93.35165	769.2308	6186.813	50659.34	401263.7
no. of additions		112	960	7936	64512	520192	4177920
no. of multiplications		64	512	4096	32768	262144	2097152
memory access	array read	288	2432	19968	161792	1302528	10452992
	array write	80	704	5888	48128	389120	3129344
	other read	57	417	3297	26337	210657	1685217
	other write	42	346	2778	22234	177882	1423066
no. of comparisons		9	73	585	4681	37449	299593
procedure calls		30	254	2046	16382	131070	1048574
space		88	376	1528	6136	24568	98296

TABLE 5.3 Experimental results for matrix multiplication using Winograd's identity

order		4	8	16	32	64	128
time in msec.	integer data	0.219780	1.538462	11.043956	82.912088	651.09890	5115.3846
	float data	10.38461	64.175824	439.5604	3230.7692	24780.220	192692.31
no. of additions		160	1024	7168	53248	409600	3211264
no. of multiplications		48	320	2304	17408	135168	1064960
memory access	array read	224	1408	9728	71680	548864	4292608
	array write	32	128	512	2048	8192	32768
	other read	129	897	6657	51201	401409	3178497
	other write	129	897	6657	51201	401409	3178497
no. of comparisons		0	0	0	0	0	0
procedure calls		0	0	0	0	0	0
space		56	206	800	3136	12316	49408

TABLE 5.4 Experimental results for matrix multiplication using Strassen's method

order		4	8	16	32	64	128
time in msec.	integer data	1.098901	10.43956	78.57143	516.4835	3956.044	25824.176
	float data	16.098901	130.21978	983.5165	7219.7802	51373.63	364450.55
no. of additions		145	1027	7201	50419	352945	2470627
no. of multiplications		49	343	2401	16807	117649	823543
memory access	array read	206	1466	10286	72026	504206	3529466
	array write	47	341	2399	16805	117647	823541
	other read	117	824	5773	40416	282917	1980424
	other write	91	651	4571	32011	224091	1568651
no. of comparisons		8	57	400	2801	19608	137257
procedure calls		45	346	2453	17202	120445	843146
space		100	436	1780	7156	28660	114676

TABLE 5.5 Experimental results for matrix multiplication using Strassen's method (Winograd variation)

order		4	8	16	32	64	128
time in msec.	integer data	1.703297	14.450549	100.000	763.7363	5329.670	35604.396
	float data	17.967033	141.26374	1098.901	7527.473	53736.264	x
no. of additions		121	856	6001	42016	294121	2058856
no. of multiplications		49	343	2401	16807	117649	823543
memory access	array read	158	1124	7886	55220	386558	2705924
	array write	44	317	2228	15605	109244	7764717
	other read	215	1510	10575	74030	518215	3627510
	other write	177	1262	8857	62022	434177	3039262
no. of comparisons		8	57	400	2801	19608	137257
procedure calls		60	466	3308	23202	162460	1137266
space		136	616	2536	10216	40936	163816

TABLE 5.6 Experimental results for matrix multiplication using Trilinear form

order		4	8	16	32	64	128
time in msec.	integer data	0.714286	5.49451	38.4615	329.670	2637.363	212663.74
	float data	27.4725	214.2857	1703.297	13571.429	108406.59	868461.54
no. of additions		320	2560	20480	163840	1310720	10485760
no. of multiplications		80	448	2816	19456	143360	1097728
memory access	array read	544	4229	33280	264192	2105344	16809984
	array write	192	1536	12288	98304	786432	6291456
	other read	218	1618	12578	99394	790658	6308098
	other write	177	1201	8865	68161	534657	4235521
no. of comparisons		116	840	6416	50208	397376	3162240
procedure calls		0	0	0	0	0	0
space		48	192	768	3072	12288	49152

TABLE 5.7 Experimental results for matrix multiplication using Pan's method

order		4	8	16	32	64	128
time in msec.	integer data	0.54945	4.3956	24.72527	154.94505	1061.5385	7807.6923
	float data	26.5384	151.64835	884.6154	5587.9121	38351.648	282637.36
no. of additions		350	2324	14920	101136	729120	5498944
no. of multiplications		141	673	3425	19265	120961	833793
memory access	array read	316	1816	10544	66144	451776	3295616
	array write	90	516	3016	19088	131360	963136
	other read	236	1304	7408	45792	309696	2245504
	other write	164	816	4096	22144	132864	882176
no. of comparisons		30	188	1144	7408	51680	381888
procedure calls		0	0	0	0	0	0
space		48	192	768	3072	12288	49152

TABLE 5.8 Performance of Calculating A^tA and AB

k	Computation time*		% of saving
	A^tA	AB	
3	0.0125	0.03	58.33
4	0.125	0.25	50
5	1	1.9	47.37
6	8.5	13.5	37.04
7	61	99	38.38
8	446	705	36.74
9	3205	4975	35.58
10	22825	34961	34.71

* Time shown in clock ticks. 18.21 clock tick = 1 second.

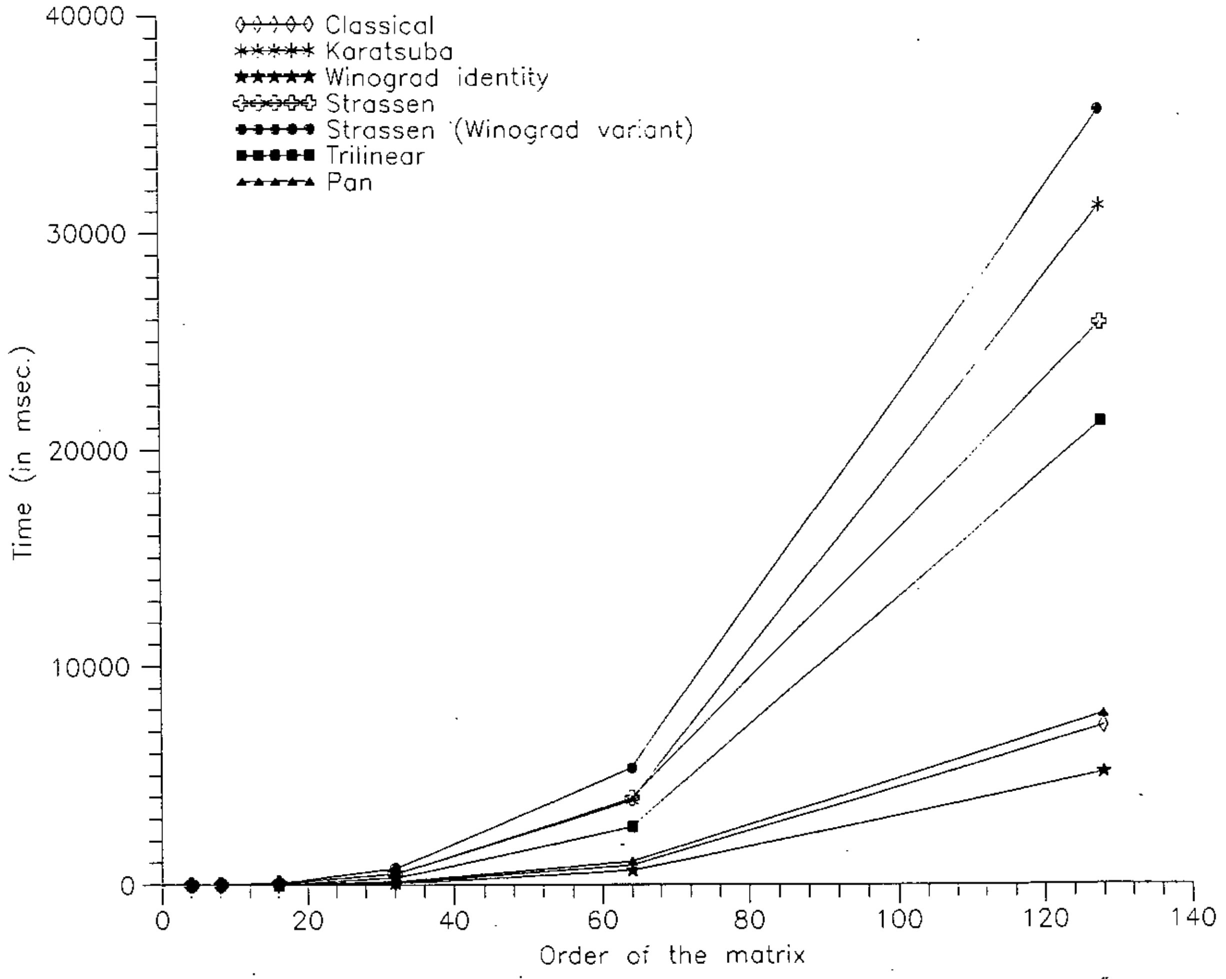

Fig. 5.3 · Matrix Multiplication time (for integer element)

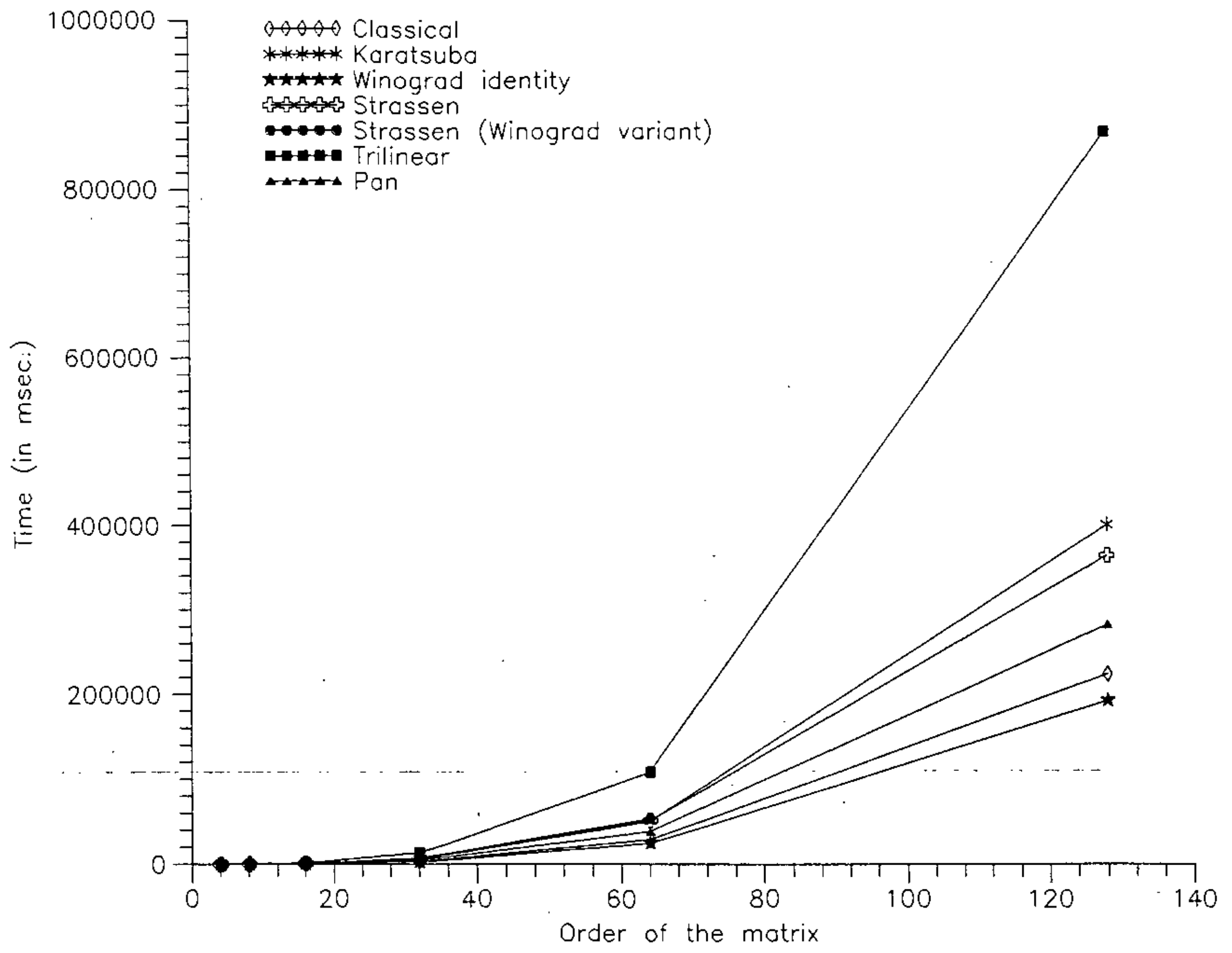

Fig. 5.4 Matrix Multiplication time (for float type element)

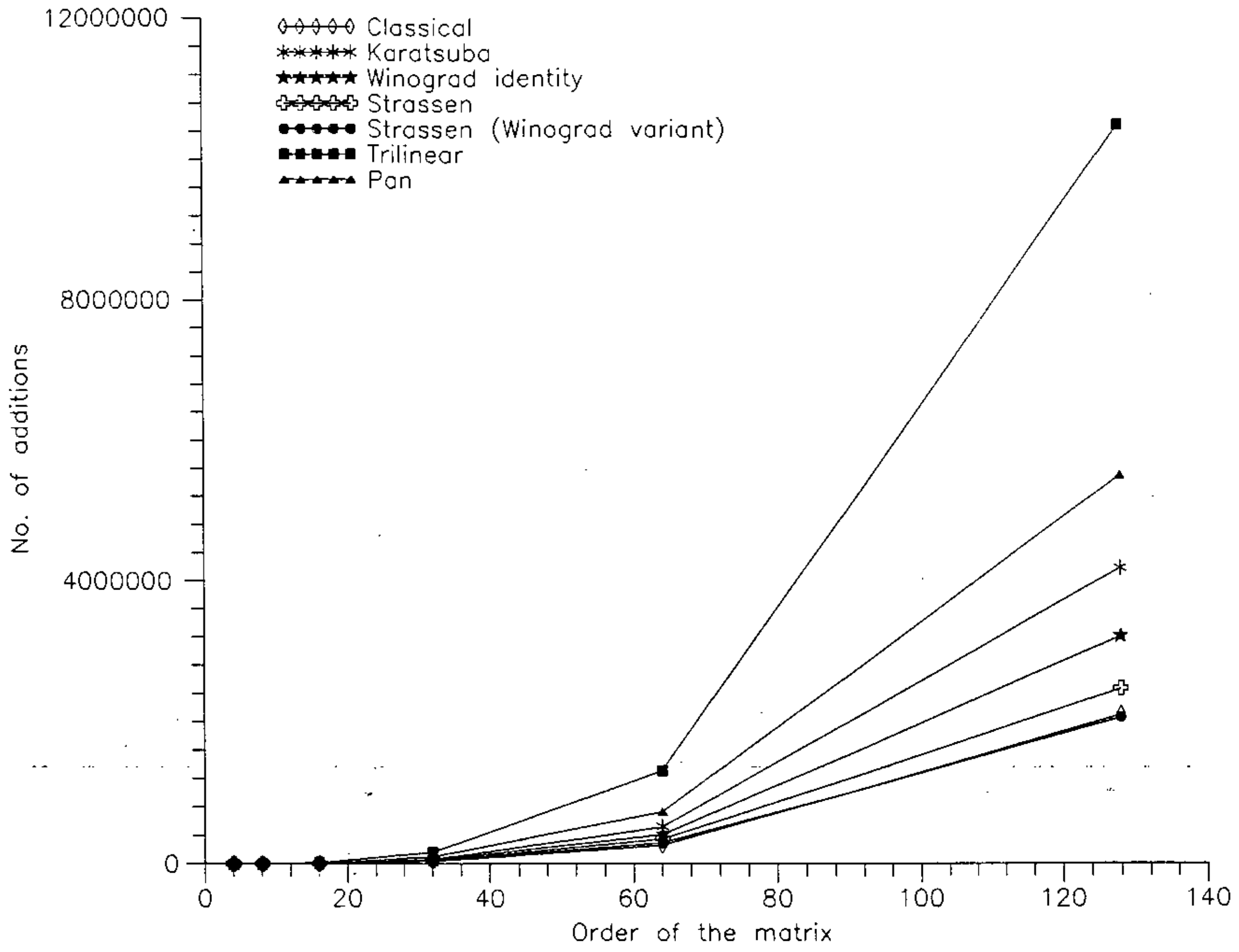

Fig. 5.5 Additions required for matrix multiplication

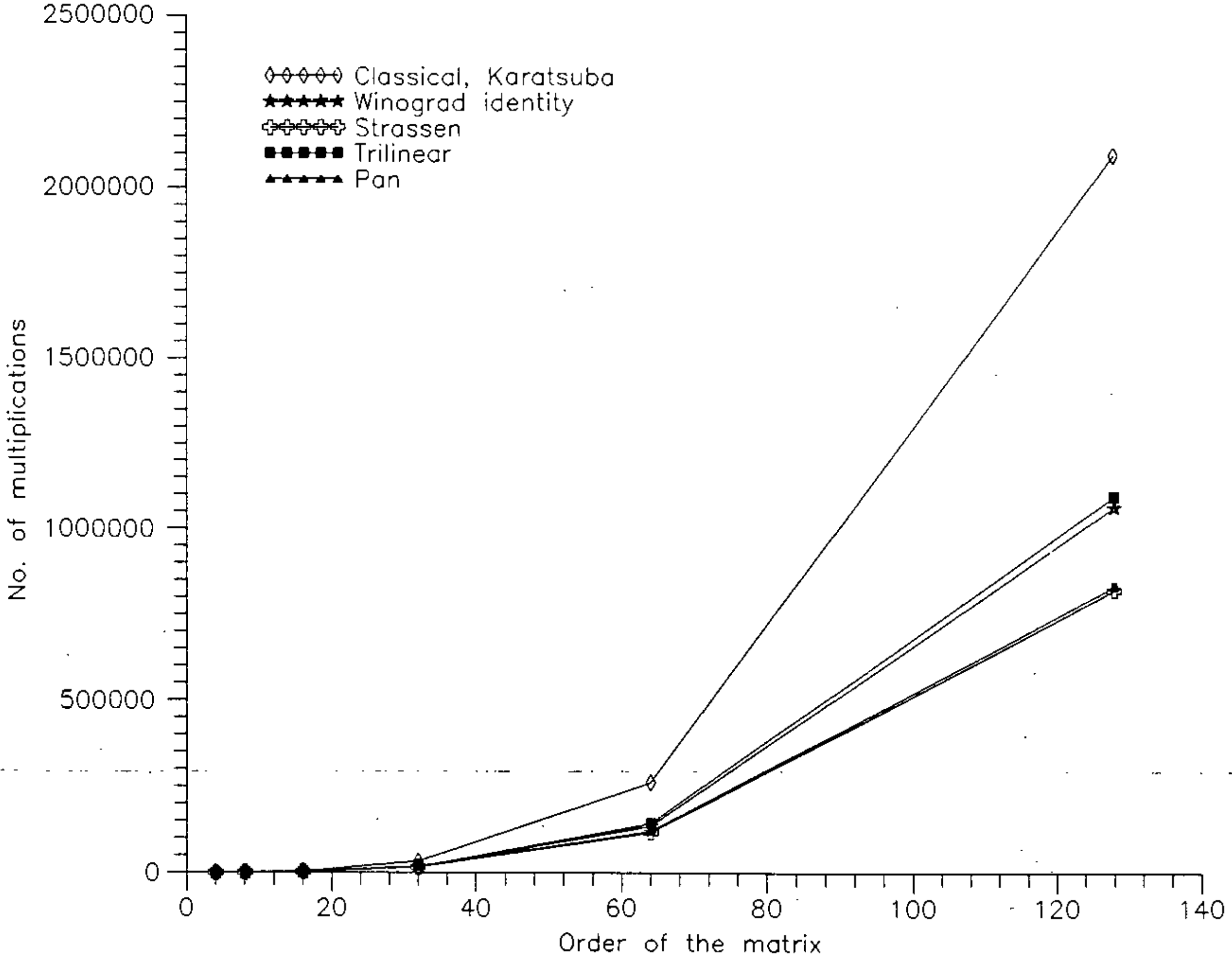

Fig. 5.6 Elementary multiplication requirement for matrix multiplication

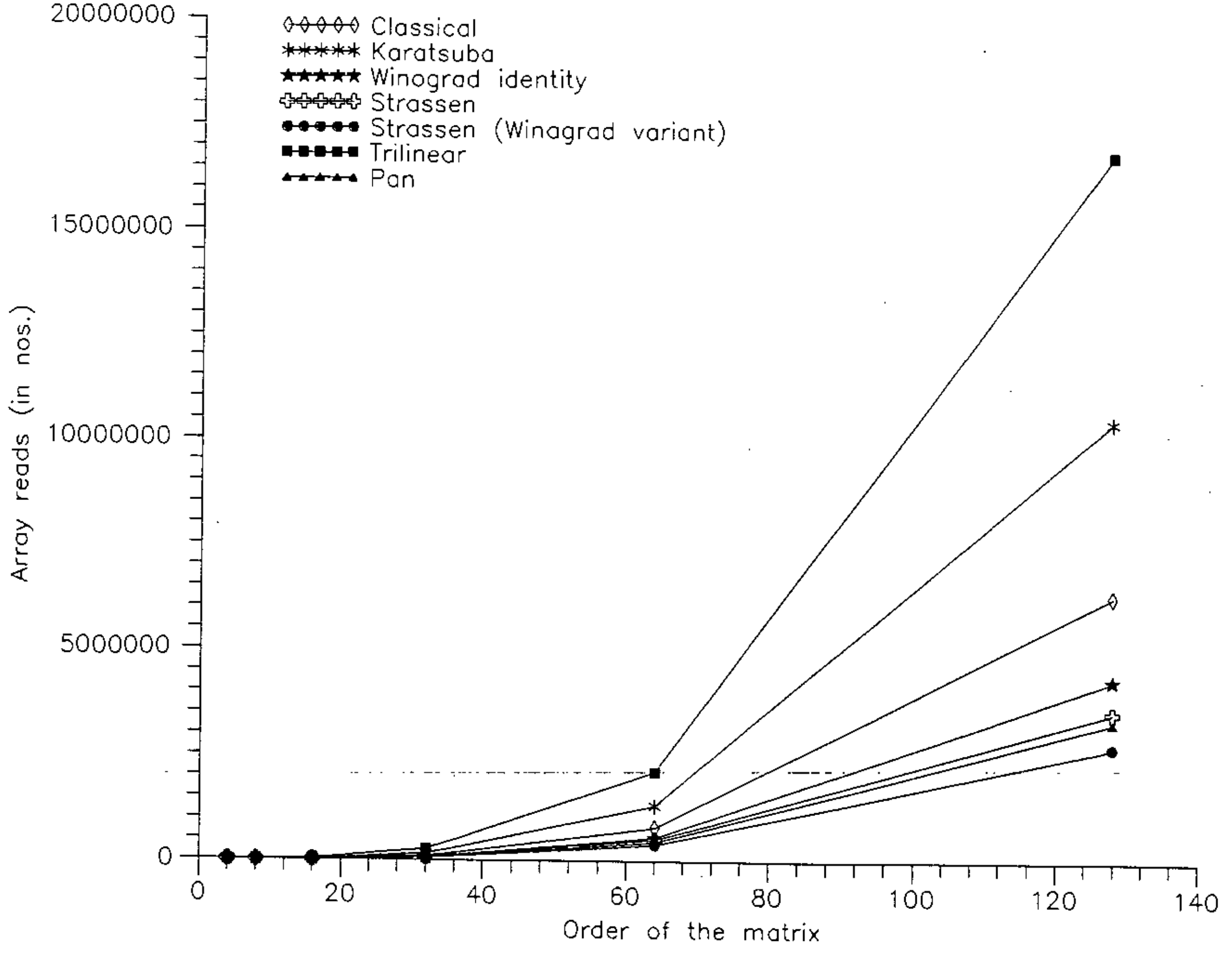

Fig. 5.7 Memory access for reading matrix data

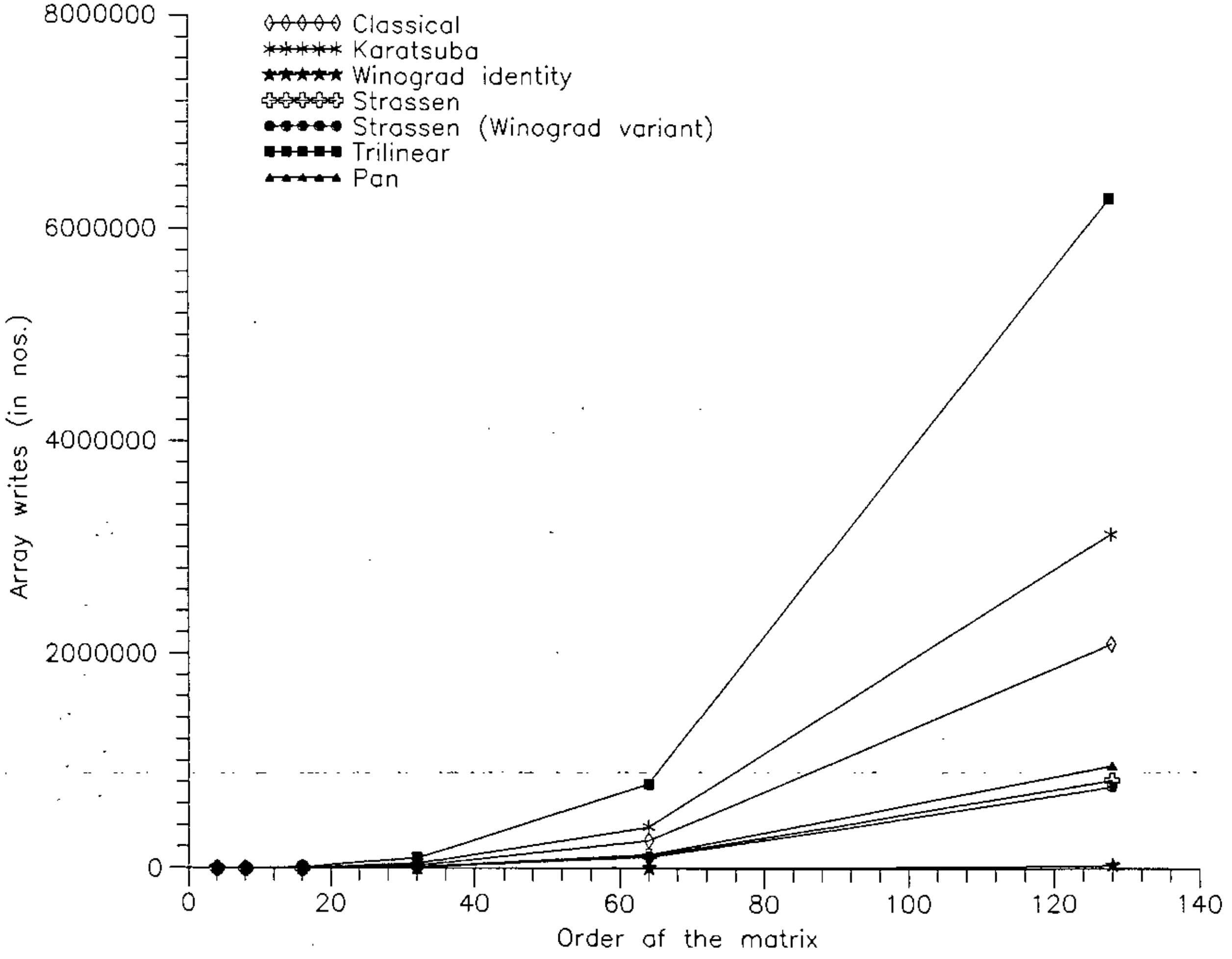

Fig. 5.8 Memory access for writing matrix element

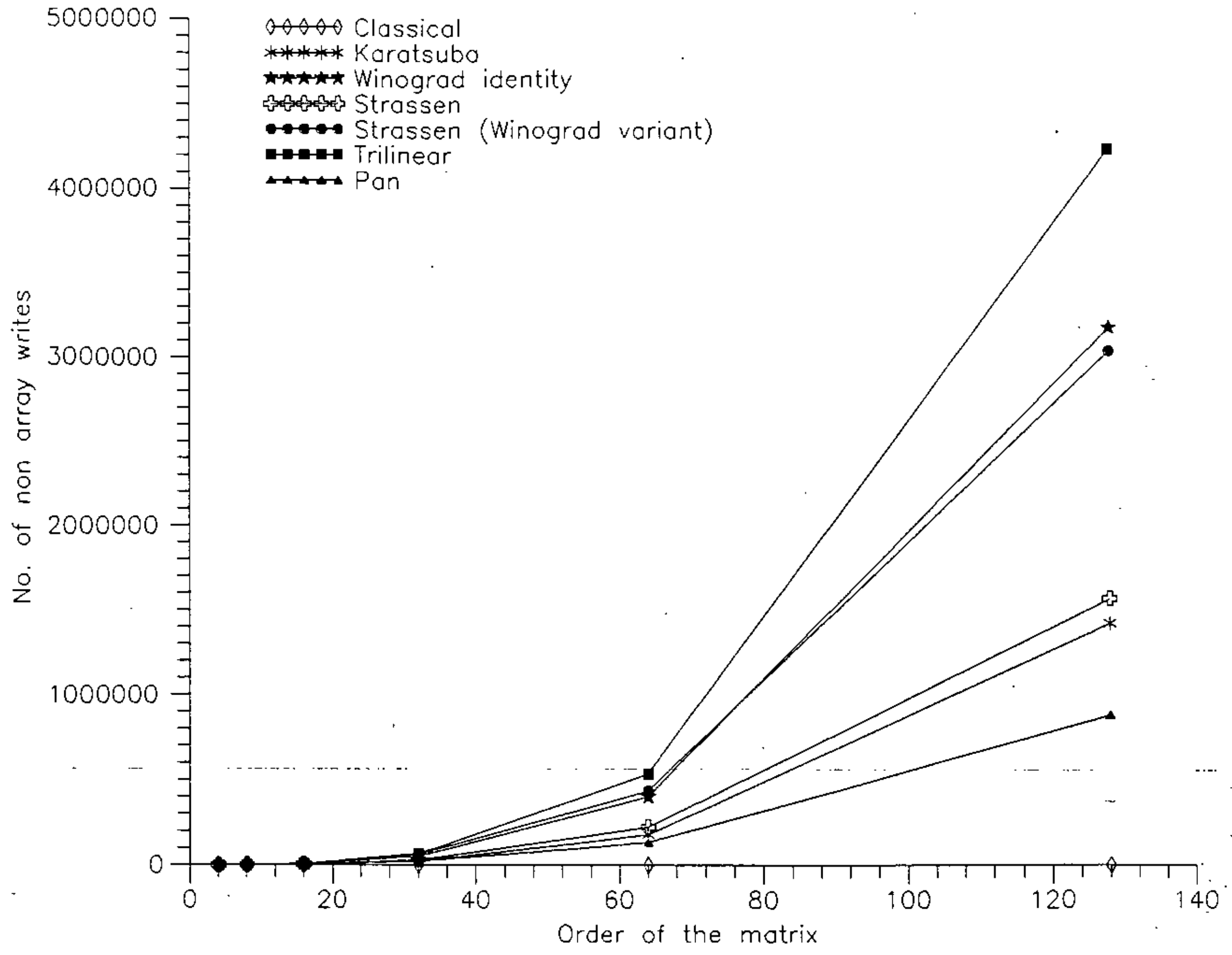

Fig. 5.9 Memory access for writing non–matrix element

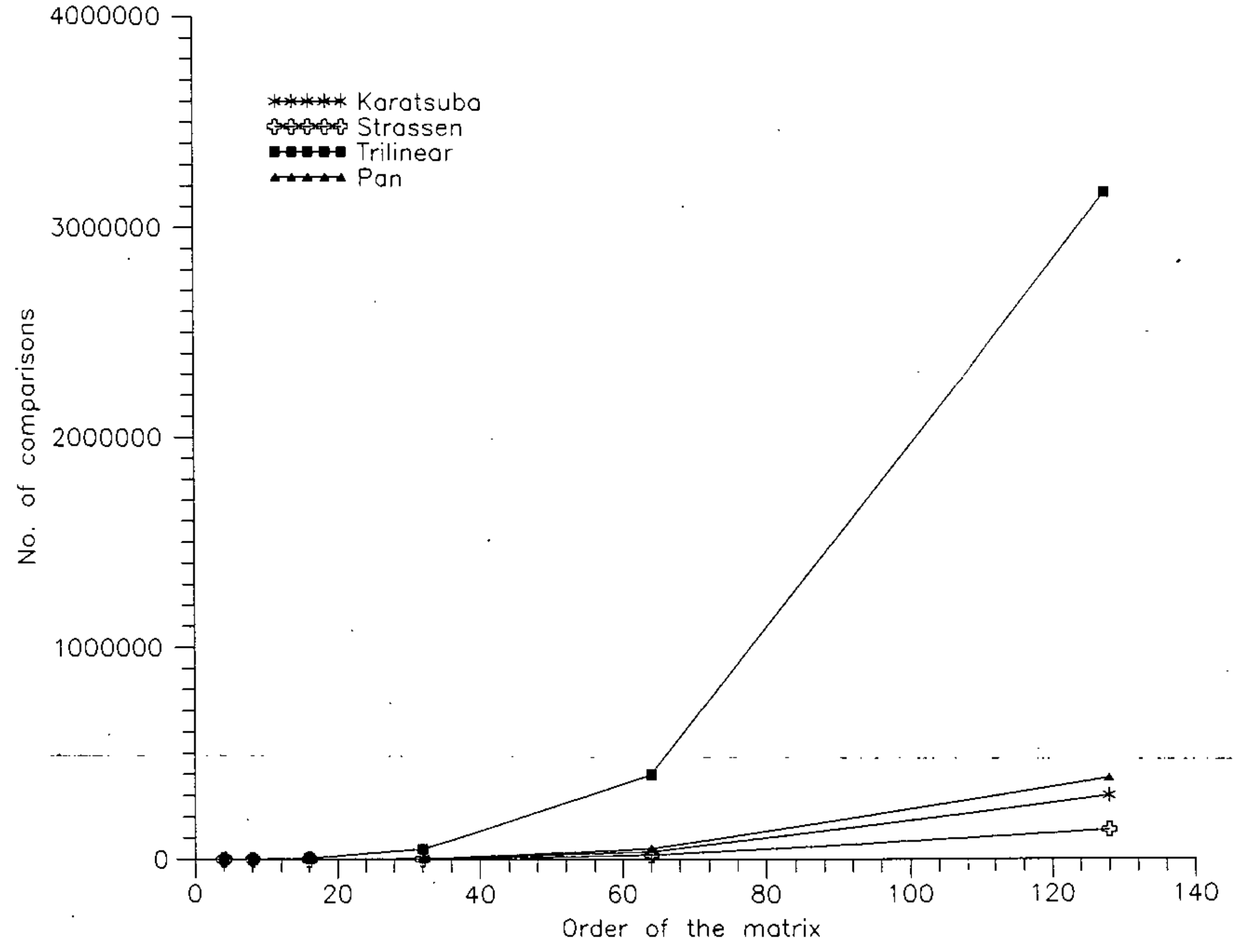

Fig. 5.10 Comparisons made while multiplying matrices

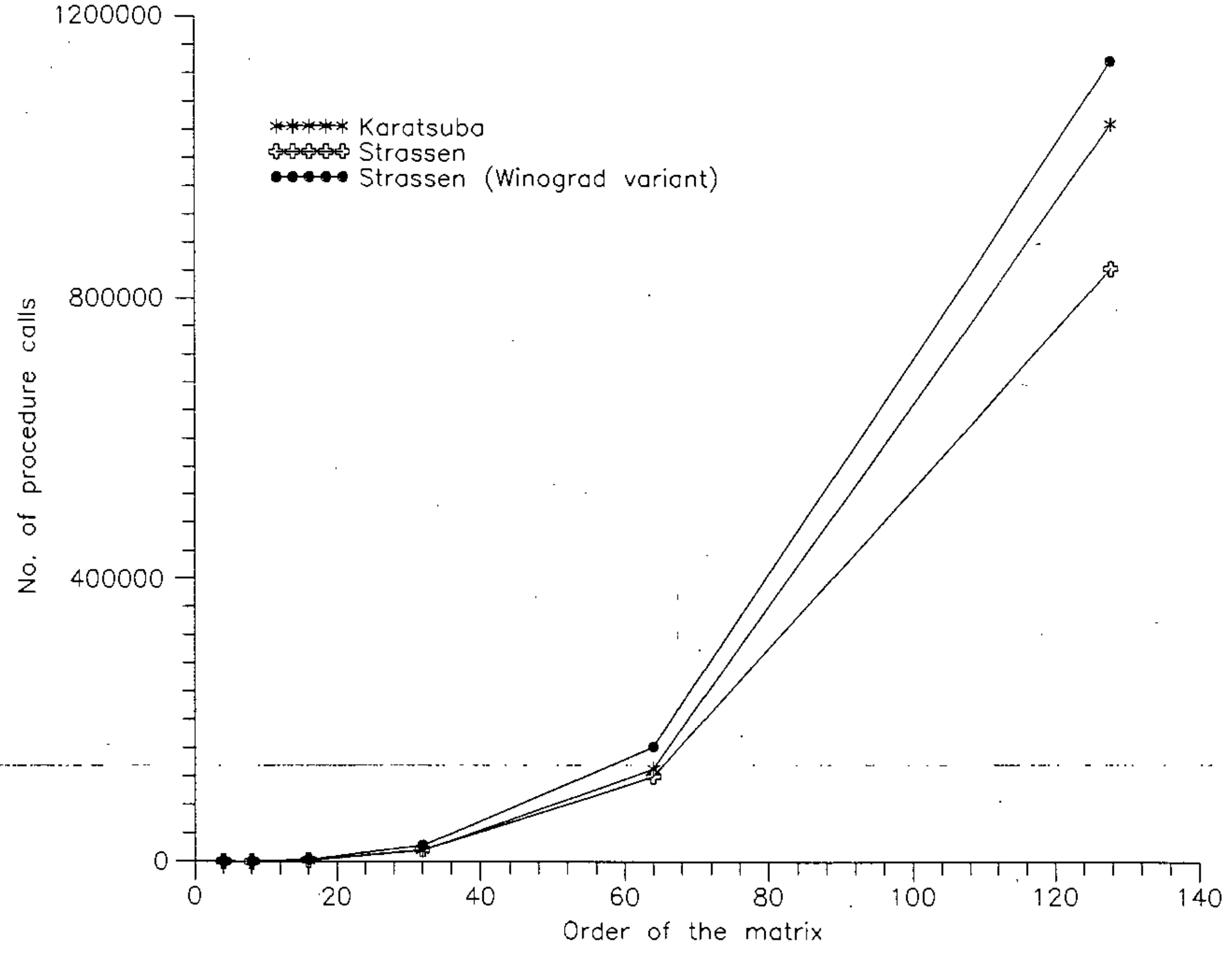

Fig. 5.11 Procedures called during matrix multiplication

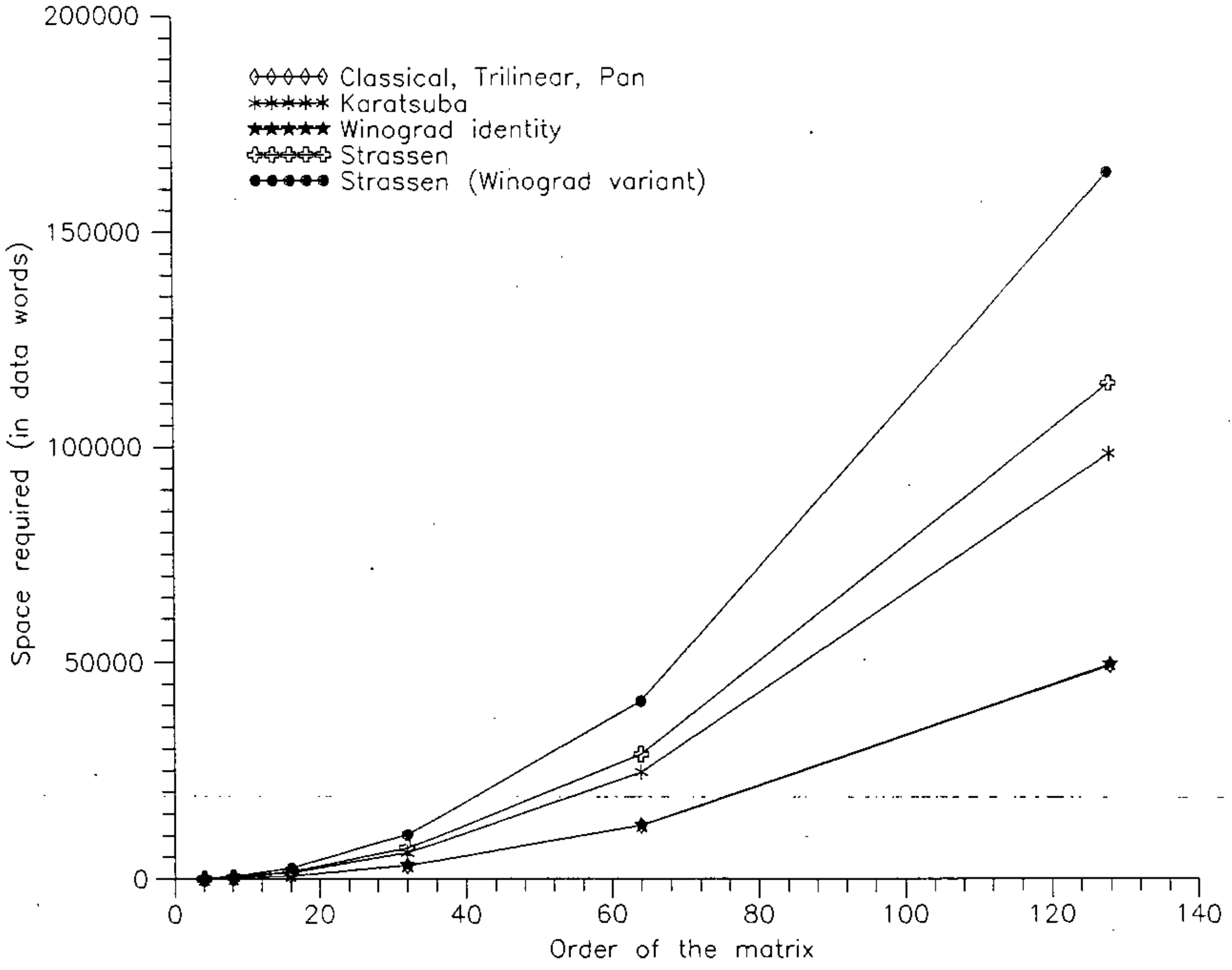

Fig. 5.12 Space requirement for matrix multiplication

Chain Matrix Multiplication :

In chapter four, we presented three different algorithms to obtain the optimum (or near-optimum) order in multiplying a series of matrices. Several data sets were randomly generated to study the behaviour of these algorithms. As before, the principal objective was to study the time complexity. The observations are shown in Table 5.9 and Fig. 5.13.

TABLE 5.9 Time required to obtain the optimum order of multiplication

No. of matrices	Time (in msec)		
	Dynamic Programming	Chin's method	Hu-Shing algorithm
10	2.198	0.109890	0.219780
20	14.835	0.329670	0.439560
30	48.352	0.509451	0.604396
40	114.286	0.69231	0.769231
50	223.077	0.879121	0.989011
60	384.615	1.043956	1.203736
70	607.143	1.263736	1.373626
80	942.857	1.483516	1.593407
90	1297.802	1.653407	1.868132

In addition to time complexity, several other experiments were carried out. As only the dynamic programming approach generates the optimum order for all cases of computation, we observed the deviation (both average and maximum) from optimum cost for the orders generated by the other two algorithms. We also observed how many times these algorithm generate the optimum order.

Another attempt were made to determine the effect of matrix dimensions on the cost. For this, three sets of data (each set with 31 problems) were generated for matrice chains with n=30, 50, 70 with maximum k_i equal to 30, 50, and 70. All the algorithms were tested with these data sets and the results are presented in Table 5.10.

TABLE 5.10 Performance of Heuristic Chain Matrix Multiplication Algorithms

n	Algorithm	Optimum order generated, Max. deviation, Average deviation		
		max. k_i = 30	max. k_i = 50	max. k_i = 70
30	Chin	12, 2.77%, 0.68%	12, 2.52%, 0.64%	12, 2.31%, 0.60%
	Hu-Shing	14, 2.12%, 0.53%	14, 1.91%, 0.49%	14, 1.74%, 0.46%
50	Chin	6, 2.13%, 0.60%	4, 2.02%, 0.59%	4, 1.92%, 0.58%
	Hu-Shing	7, 1.78%, 0.53%	6, 1.69%, 0.50%	6, 1.61%, 0.49%
70	Chin	3, 1.54%, 0.47%	3, 1.48%, 0.46%	3, 1.43%, 0.445%
	Hu-Shing	4, 1.54%, 0.47%	3, 1.48%, 0.45%	4, 1.43%, 0.441%

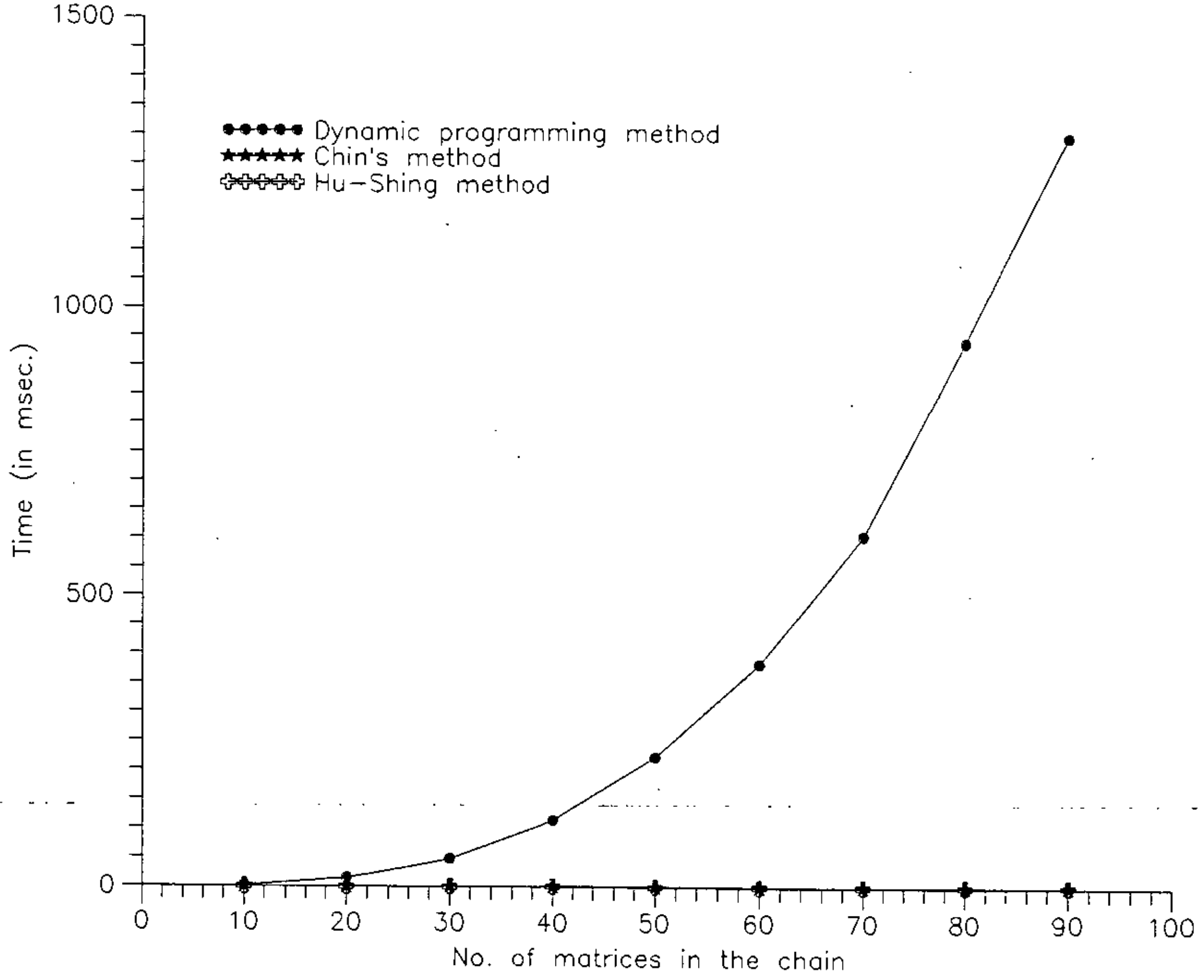

Fig. 5.13a Time to find the multiplication order of a matrix chain

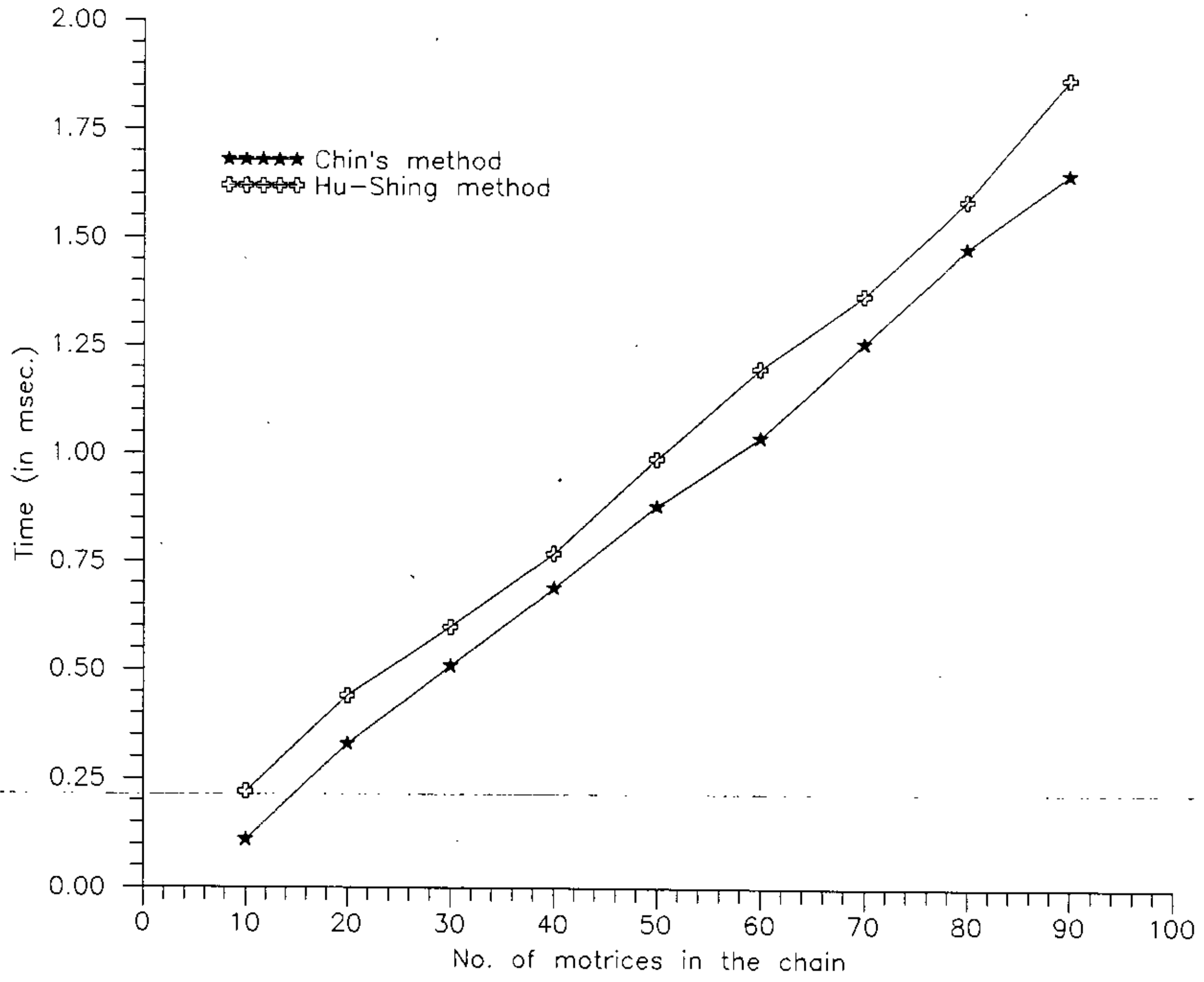

Fig. 5.13b Time to find the multiplication order of a matrix chain

CHAPTER SIX

CONCLUSIONS AND RECOMMENDATIONS

CONCLUSIONS

In this thesis we have considered the problems of polynomial evaluation, matrix multiplication and chain matrix multiplications. Conclusions of experimental results on the implementation of different algorithms for solving these problems have been presented below.

We have considered the problem of polynomial evaluation along with preprocessing when the same polynomial is evaluated in many different points. Among the preprocessing algorithms Belaga method evaluates the preprocessed polynomial in the fastest possible time, whereas preprocessing time taken by the algorithm is the highest. Belaga has the problem of generating complex Belaga coefficients, in which case time requirement in evaluation will be multiplied by at least four times. Results presented in the thesis correspond to polynomials not generating complex coefficients.

Peterson-Stockmeyer method of preprocessing requires minimum preprocessing time but requires alarmingly high evaluation time with a staircase nature as is evident from the application of binary tree concepts.

Knuth's evaluation time is very close to that of Belaga whereas preprocessing time is less than that of Belaga.

Among the non-preprocessing algorithms Horner's scheme has the best evaluation time.

We have considered matrix multiplication with integer elements and with float elements. Experiments with both integer and real data show that matrix multiplication using Winograd's identity has the best time performance. Classical method's performance is the second best. Theoretically superior methods of Pan, Strassen, Karatsuba and trilinear have performance in this order, and have proved to be inferior for at least lower values of *n*. Although Strassen's method requires lesser number of multiplications, excessive memory accesses, other computational and recursion overheads offset the savings from reduced number of multiplications. Classical, trilinear Pan's and Winograd's identity require minimum space, whereas Strassen's and Winograd's variation of Strassen's method require significantly more space. This excessive space requirement will act as limiting factor for using these methods.

We have also applied a variation of Strassen's method for preprocessing arbitrary system of linear equations to convert it into positive definite systems for which a lot of $O(n^2)$ convergent iterative schemes exist. This preprocessing consists of premultiplying A with A^t. We have recursively used Strassen's method and symmetricity of involved submatrices to cut down cost of multiplication by approximately 33% thus giving us a better method of solving arbitrary systems of linear equations.

In chain matrix multiplication dynamic programming approach has always produced optimal sequence of multiplication at the cost of $O(n^3)$ operations. We have also tried two heuristic methods for finding the best sequence of multiplication. These methods require negligible amount of time compared to dynamic programming approach with Chin's method outperforming Hu-Shing's method. However, solution obtained by Hu-Shing's method was consistently superior to that of Chin's method. Although probability of obtaining the optimal sequence reduced with the increase of number of matrices n in the chain maximum and average deviations of the obtained solution from the optimal reduced with the increase of n. Our experiments show that deviations from the optimal sequence did never cross 3%, and it decreased with the increase of n. Although these algorithms are heuristic in many cases optimal sequences were generated..

RECOMMENDATIONS FOR FURTHER STUDY

In polynomial evaluation we have not considered Belaga method with complex Belaga parameters in order to keep it competitive with other algorithms. One can study this aspect of Belaga method in detail to ascertain its performance in cases where those parameters become complex. We have not also considered polynomials with complex coefficients. So this aspect of the problem has remained untouched, and therefore, one can try to pursue research in this direction as well.

Although our experiments do indicate inferior performance of theoretically superior matrix multiplication algorithms since due to technical reasons we were not able to perform experiments with much larger memory space, the conclusions will perhaps be in their favour if dimensions of matrices involved could have been increased significantly. In absence of enough primary memory one can attempt to perform these experiments using secondary memory.

One can attempt to derive a better time complexity for these algorithms incorporating memory access time, looping overheads, logical operations, procedure calls and recursion overheads among other items.

One can also try to ascertain performance of all the popular parallel algorithms for matrix multiplications.

REFERENCES

[1] Horner, W. G., Philosophical Transactions, Royal Society of London, vol. 109, pp.308-335.

[2] Newton, I., *De Analysi per Æquationes Infinitas,* 1969.

[3] Ostrowski, A. M., *On two problems in abstract algebra connected with Horner's rule,* Studies in Mathematics and Mechanics, Academic Press, N.Y. pp.40-48, 1954.

[4] Pan, V. Ya., *Methods of computing values of polynomials,* Uspekhi Math. Nauk, vol. 21, pp. 103-134, 1966 (in Russian). English translation in Russian Math. Surv., vol.21, pp. 105-136.

[5] Belaga, E. C., *Some problems in the computation of polynomials,* Dokl. Akad. Nauk., SSSR, vol. 123, pp. 775-777, 1958 (in Russian).

[6] Motzkin, T. C., *Evaluation of polynomials,* Bull. Amer. Math. Soc., vol.61, p.163.

[7] Pan, V. Ya., *Schemes for computing polynomials with real coefficients,* Dokl. Akad. Nauk. SSSR, vol 127, pp.266-269 (in Russian). English translation in Math. Rev., vol. 23, 1962.

[8] Rabin, M. and S. Winograd, *Fast Evaluation of polynomials by rational preparation,* IBM Tech. Report RC3645, Dec. 1971.

[9] Karatsuba, A. and Yu Ofman, *Multiplication of multiple numbers by means of Automata,* Dokl. Akad. Nauk. USSR, vol.145, No. 2, pp.293-294, 1962 (in Russian).

[10] Toom, A. L., *The complexity of a scheme of functional elements realizing the multiplication of integers,* Dokl. Akad. Nauk, SSSR, vol. 150, pp.496-498, 1963.

[11] Cook, S. A., *On the minimum computation time of functions,* Doctoral thesis, Harvard University, Cambridge, Massachusetts, 1966.

[12] Schönage, A. and V. Strassen, *Schenelle Multiplikation Grosser Zahlen*, Computing, vol.7, pp.281-292, 1971.

[13] Chandra, A. K., *Computing matrix chain products in near optimal time*, IBM Research Report RC5625 (#24393), IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1975.

[14] Winograd, S., *On the number of multiplications required to compute certain functions*, Proc. Natl. Acad. Sci., USA, vol.58, pp.1840-1842, 1967.

[15] Strassen, V., *Gaussian elimination is not optimal*, Num. Math., vol.13, pp.354-356, 1969.

[16] Winograd, S., *Some remarks on fast multiplication of polynomials*, in Complexity of Sequential and Parallel Numerical Algorithms, J.F. Traub ed., Academic Press, N.Y., 1973.

[17] Pan, V., *Strassen algorithm is not optimal. Trilinear technique of aggregating, uniting and cancelling for constructing fast algorithms for matrix multiplication*, Proc. 19th Annual Symposium on the Foundation of Computer Science, Ann Arbor, MI, pp.166-176, 1978.

[18] Pan, V., *How can we speed up matrix multiplication ?*, SIAM rev. vol.26, No.3, pp.393-415, 1984.

[19] Chin, F. Y., *An O(n) Algorithm for Determining a Near-Optimal Computation Order of Matrix Chain Products*, Comm. of the ACM, vol. 21, No. 7, pp.544-549, July 1978.

[20] Hu, T. C., *Combinatorial Algorithms*, Addison Wesley Publishing Company, pp. 242-267, 1982.

[21] Clenshaw, C. W., *A note on the summation of Chebyshev series*, MTAC, vol.9, pp.118-120, 1955.

[22] Bakhvalov, N. S., *On the stable evaluation of polynomials*, J. Comp. Math. and Math. Phys, vol.11, No.6, pp.1568-1574, 1971.

[23] Hopcroft, J. E. and Kerr, L. R., *On Minimizing the Number of Multiplications Necessary for Matrix Multiplication*, SIAM J. Appl. Math., vol. 20, No. 1, pp. 30-36, 1971

[24] Dekel, E., Nassimi, D. and Sahni, S., *Parallel Matrix and Graph Algorithms*, SIAM J. Comput., vol. 10, No. 4, pp.657-675, 1981.

[25] Paterson, M. and L. Stockmeyer, *On the Number of Nonscaler Multiplications Necessary to Evaluate Polynomials*, SIAM J. of Computing, vol.2, No. 1, pp.60-66, 1973.

[26] Cheny, E. W., *Algorithms for the Evaluation of Polynomials Using a Minimum Number ofMultiplications*, Technical Note 2, Computation and Data Processing Center, Aerospace Corporation, El Segundo, California, 1962.

[27] Paprzycki, M., Cyphers, C., *Using Strassen's Matrix Multiplication in High Performance Solution of Linear Systems*, Computers Math. Applic., vol.31, No. 4/5, pp.55-61, 1996.

[28] Bjorstad, P., Manne, F., Sorevik, T., and Vajtersic, M., *Efficient Matrix Multiplication on SIMD Computers*, SIAM J. Matrix Anal. Appl., vol. 13, No. 1, pp. 386-401, 1992.

[29] Knuth, D. E., *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley Publishing Company, (second edition), 1973.

[30] Knuth, D. E., *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley Publishing Company, (second edition), 1981.

[31] Knuth, D. E., *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley Publishing Company, 1973.

[32] Eve, J. ,*The Evaluation of Polynomials*, Num. Mathematik, vol. 6, pp. 17-21.

[33] Horowitz E. and S. Sahni, *Fundamentals of Computer Algorithms*, Galgotia Publications, New Delhi, 1990.

[34] Kronsjö L. I., *Algorithms : Their Complexity and Efficiency*, John Wiley and Sons, 1990.

[35] Brigham, O. E., *The Fast Fourier Transform*, Prentice Hall Inc., 1974.

[36] Kaykobad, M., Hoque, S., Akbar, M.M., and Nath, S.K., *An Efficient Preprocessing for Solving Systems of Linear Equations*, to appear in Int. J. of Comput. Math.

[37] Cormen, T. H., C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[38] Sedgewick, R., *Algorithms*, Addison Wesley Publishing Company, 1988.

[39] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley Publishing Company, 1974.

[40] Borodin, A., and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, N.Y., 1975.

[41] Scarborough, J. B., *Numerical Mathematical Analysis*, Oxford and IBH Publishing Company, (third Indian reprint), 1971.

[42] Cohen, J. and Roth, M., *On Implementation of Strassen's Fast Multiplication Algorithm*, Acta Informatica, 6, pp.341-355, 1976.

[43] Schildt, H., *C: The Complete Reference*, Osborne McGrawHill, 1987.