数式処理 *J.JSSAC (2005)*
Vol. 11, No. 3,4, pp. 3 - 19
Noda2005 論文

Matrix Multiplication Made Fast — Practical View of Fast Matrix Operation for Computer Algebra System

Noriko HYODO* Hirokazu MURAO† Tomokatsu SAITO‡

AlphaOmega Inc.*‡
The Univ. of Electro-Communications†

Abstract

This paper gives a brief review of our experimental results of matrix multiplication in a computer algebra system, and explain a characteristic behavior of computing times, with major emphasis laid upon the relation with complexity obtained by theoretical analysis. Furthermore, based on the knowledge obtained throughout the experiments, we propose a new method to represent matrices appropriate for computer algebra systems. So far very little has been studied about how to implement and treat matrices in computer algebra systems. While the representation of polynomials has been extensively studied, there is little arguments for matrix representation with empirical study, and two-dimensional array is used to represent matrices as in numerical matrices. Also, it is very often that even for matrices with symbolic elements, the same argument and analysis of complexity as for numerical processing is used, and reveal very weak connection to real computations. This fact motivated us to investigate the nature of matrix computation, especially in multiplication, and the use of asymptotically fast algorithms. We investigate computational complexity empirically to find a measure to reflect actual computing time.

1 Introduction

Computing time of an algorithm is usually discussed in connection with a quantity, so-called time complexity, expressed in terms of the behavior of a function of the number of operations actually executed. Discussion and analysis performed on numerical processing is often applied to

*noriko@a2z.co.jp
†murao@cs.uec.ac.jp
‡saito@a2z.co.jp

algorithms in computer algebra, regradless of its appropriateness. A simple example of this kind misuse of complexity analysis is on matrices of symbolic elements. Time complexity of matrix operation is often discussed in terms of the order of treated matrices, but such an analysis is almost useless unless all the elements are of equivalent sizes and the costs for operations of elements are equivalent, which cannot be expected for general matrices treated in computer algebra. Then, what can be a good measure for computing time in computer algebra? This is our first question. Another question connected to time complexity is about asymptotically-fast algorithms. As noted in Knuth's famous textbook, it has long been believed that asymptotically-fast algorithms never be fast in practice, however, nowadays, fast algorithms for polynomial arithmetics are indispensable for some kinds of applications. What about fast algorithms for matrix operations? There have been developed a series of fast algorithms for matrix multiplications [6, 7, 2]. Are they useful for symbolic computation?

To obtain an answer for or any knowledge about the above questions empirically, we have been testing Strassen-Winograd algorithm for matrix multiplication with various types of polynomial elements in multiple ways of representations [9, 8, 11, 10, 12]. In this paper, as a simple summary of our experiments, we explain some distinctive results. Our main statement is simple; computing time is almost proportional to the amount of memory used during computation, namely, the size of memory space to which processing is done, and thus, has close relation with so-called space complexity. Also, it turned out that the fast matrix multiplication algorithm may reveals impressive speed for some cases in symbolic computation, as we have expected. There have been various research results with matrix determinant or linear systems in the past, e.g. [4], [3], [1]. In the previous work[4], there is an interesting complexity analysis which uses a polynomial model with expression growth counted into, but its practical usefulness is not clear. For matrix multiplications, there has been development of new algorithms and macroscopic analysis like $O(n^{2.\cdots})$, and there is few practical results, especially in symbolic and algebraic computation.

Another point focused in this paper is the representation of matrices, especially for sparse matrices. We wonder if the usual matrix representation using two-dimensional array of successive memory space is of any significance in the case of computer algebra. Requirement will be efficient access to an element or to a series of elements in a row or a column and so on. In this paper, we propose a new matrix representation, which is basically a list of indexed elements, and show its practical efficiency.

In the sections to follow, we first describe algorithms used in our experiment in Section 2, and then gives a brief summary of typical results of our experiments in Section 4. Section 5 is devoted to describing the new matrix representation. The final section gives our tentative conclusion.

2 Algorithms To Investigate — Matrix Multiplication

Let A and B be $l \times m$ and $m \times n$ matrices, respectively,

$$A = \left(a_{ij} \right) = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{l1} & \dots & a_{lm} \end{pmatrix}, \quad B = \left(b_{ij} \right) = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix},$$

and we consider the multiplication $C = AB$,

$$C = \left(c_{ij} \right) = \left(a_{ik} \right) \left(b_{kj} \right) = AB.$$

There are known two types of algorithms for the multiplications; the one is the well-known standard algorithm using inner-product and the other is the ones with asymptotically fast complexity, which employ the reduction of the number of multiplications of matrix elements by transforming their bilinear forms of the product elements [2].

2.1 Standard Algorithm

The matrix product $C = AB$ is defined and is usually computed by the following formula

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}, \quad \text{for } 1 \le i \le l \text{ and } 1 \le j \le n. \tag{1}$$

Namely, each element c_{ij} of the product is the inner product of two vectors of the i-th row of A and the j-th column of B. So, this standard algorithm is often called with a term “inner product” or “dot product”.

The above algorithm performs lmn multiplications and $ln(m-1)$ additions of matrix elements, at most. Let t_* and t_+ denote the costs for multiplication and additive operation of matrix elements, respectively. Then, the operation count of the algorithm can be described by

$$T_{inn}(l, m, n) = (lmn)t_* + ln(m-1)t_+. \tag{2}$$

For $n \times n$ matrices, the counts are of $O(n^3)$, and the complexity of the algorithm is often said to be

$$O(n^3) \text{ for matrices of order } n.$$

This analysis will be appropriate to the case when the cost of the arithmetic operations can be assumed independent of matrix elements and equivalent for all matrix elements. This is the case with usual numerical processing, however, this simple analysis is almost meaningless in the case of computer algebra because expressions of matrix elements are usually structured, change their sizes during computation and therefore the cost varies.

2.2 A Fast Algorithm: Strassen-Winograd Algorithm

In 1969, Strassen invented a new fast algorithm, which requires fewer multiplications of matrix elements than the above algorithm [6]. The algorithm partitions each of A and B into four submatrices of an equal size, and employs divide-and-conquer strategy.

Let $l_1 = \lfloor l/2 \rfloor$, $m_1 = \lfloor m/2 \rfloor$ and $n_1 = \lfloor n/2 \rfloor$. We let A and B be partitioned into $l_1 \times m_1$ submatrices A_{ij} and $m_1 \times n_1$ submatrices B_{ij}, as follows:

$$\bar{A} \stackrel{\text{def}}{=} \begin{pmatrix} a_{11} & \dots & a_{1\bar{m}} \\ \vdots & \ddots & \vdots \\ a_{\bar{l}1} & \dots & a_{\bar{l}\bar{m}} \end{pmatrix} \to \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ and } \bar{B} \stackrel{\text{def}}{=} \begin{pmatrix} b_{11} & \dots & b_{1\bar{n}} \\ \vdots & \ddots & \vdots \\ b_{\bar{m}1} & \dots & b_{\bar{m}\bar{n}} \end{pmatrix} \to \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

where $\bar{l}$, $\bar{m}$ and $\bar{n}$ denote $2l_1$, $2m_1$ and $2n_1$ respectively. The case when l, m or n is odd requires additional calculations besides the multiplication of $\bar{A}$ and $\bar{B}$ to obtain the true product C, as described later. Consider the multiplication of $\bar{A}$ and $\bar{B}$.

$$\bar{A}\bar{B} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

While the above expression contains 8 multiplications and 4 additions of submatrices, Strassen has shown that the expression can be obtained by 7 multiplications and 18 additive operations.

Strassen's algorithm was further improved by Winograd[7], by reducing the number of additive operations to 15, as shown below.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & w + v + (A_{12} - s_2)B_{22} \\ w + u + A_{22}(B_{21} - t_2) & w + u + v \end{pmatrix},$$

where

$$\begin{aligned} s_1 &= A_{21} + A_{22}, \\ s_2 &= s_1 - A_{11} && = -A_{11} + A_{21} + A_{22}, \\ t_1 &= B_{12} - B_{11}, \\ t_2 &= B_{22} - t_1 && = B_{11} - B_{12} + B_{22}, \\ u &= (A_{11} - A_{21})(B_{22} - B_{12}), \\ v &= s_1 t_1 && = (A_{21} + A_{22})(B_{12} - B_{11}), \\ w &= A_{11}B_{11} + s_2 t_2 && = A_{11}B_{11} + (-A_{11} + A_{21} + A_{22})(B_{11} - B_{12} + B_{22}). \end{aligned}$$

Basically, for multiplications of submatrices, we apply the algorithm recursively.

If l or n is odd, the l-th row c_{lj}, $1 \le j \le n$ or the n-th column c_{in}, $1 \le i \le l$ of the product C is not included in $\bar{A}\bar{B}$, and must be computed separately via Eq. (1). Furthermore, in the case that m is odd, $a_{im}b_{mj}$ must be added to the (i, j) element of $\bar{A}\bar{B}$ to obtain c_{ij}:

$$\begin{pmatrix} c_{11} & \dots & c_{1\bar{n}} \\ \vdots & \ddots & \vdots \\ c_{\bar{l}1} & \dots & c_{\bar{l}\bar{n}} \end{pmatrix} = \bar{A}\bar{B} + \begin{pmatrix} a_{1m} \\ \vdots \\ a_{\bar{l}m} \end{pmatrix}\begin{pmatrix} b_{m1} & \dots & b_{m\bar{n}} \end{pmatrix}$$

Let's count the number of elementwise operations precisely. We assume matrices are square and of order $n = m2^k$, for simplicity. We apply the above fast algorithm recursively until the order of submatrices become m, and for the multiplication of matrices of order m, we use the standard algorithm. Now, the operation count can be described by

$$\begin{aligned} T_{SW}(m,k) &= 7 \times T_{SW}(m, k-1) \\ &\quad +15 \times (\text{cost of matrix addition/subtraction of order } m2^{k-1} : m^2 2^{2(k-1)} t_+) \\ &= 7^k \times T_{inn}(m,m,m) + 15m^2(2^{2k} - 7^k)/(2^2 - 7)t_+ \\ &= 7^k(m^3 t_* + m^2(m-1)t_+) + 5m^2(7^k - 4^k)t_+. \end{aligned} \tag{3}$$

If n is a power of 2, *i.e.*, $n = 2^k$, and the costs t_* and t_+ are regarded as equivalent and constant, the term $7^k = n^{\log_2 7}$ is dominant, and the time complexity $T_{SW}(n)$, as a function of n, is said to be

$$T_{SW}(n) = O(n^{\log_2 7}) \approx O(n^{2.807}).$$

In the following sections, we may call Strassen-Winograd algorithm as SW algorithm, fast algorithm or $O(n^{\log_2 7})$-algorithm, and the standard algorithm as inner-product, classical, or $O(n^3)$ algorithm.

Remark. While in the standard algorithm, all the additions are with the products of matrix elements, in the fast algorithm, only 7 additions/subtractions of submatrices from 15 are with those elements, and the rest (8 additions/subtractions) treat submatrices only of A's or of B's. The cost t_+ of additive operations of matrix elements varies depending on the expressions of operands, and the costs of two types of additive operations mentioned above may differ significantly. In the following, we give an example of detailed analysis appropriate for symbolic computations

3 A Microsopic Analysis of Computing Cost

The question that initially motivated us to peform empirical study is how efficient the fast algorithms for matrix multiplication can be with symbolic computation in practice. In the series of our experiments so far, there are some unexpected results with extraordinary speedup. Our current interest is how and in what cases the algorithm can be fast, and why. In the rest of this section, we shall give a sample analysis of computing time, using simle matrix model with polynomial entries.

We treat square matrices of order $n = m2^k$, and we apply SW algorithm recursively to submatrices until they get as small as of $m \times m$. For $m \times m$ matrices, we use classical algorithm.

3.1 General Case

In matrix product calculations, there are three kinds of element arithmetics; multiplication of a_{ij} and b_{kl}, addition/subtraction of the products, and addition/subtraction of a_{ij}'s or of b_{ij}'s. The time complexity of each kind is denoted as follows:

- t_*: complexity of multiplication of a_{ij} and b_{kl},
- t_{2+}: complexity of addition/subtraction of $(a_{ij}b_{kl})$'s,
- t_+: complexity of addition/subtraction of a_{ij}'s or of b_{ij}'s.

Then, the total complexity (operation count) of the standard algorithm will be given as

$$T_{inn}(n) = n^3 t_* + n^2(n-1)t_{2+}.$$

Now, the complexity for the fast algorithm will be described more precisely than before:

$$\begin{aligned} T_{SW}(m,k) &= 7T_{SW}(m,k-1) + (m2^{k-1})^2 (4t_{+a} + 4t_{+b} + 7t_{2+}) \\ &= 7^k T_{inn}(m) + \frac{m^2(7^k - 2^{2k})}{3}(4t_{+a} + 4t_{+b} + 7t_{2+}) \\ &= 7^k(m^3 t_* + m^2(m-1)t_{2+}) + \frac{m^2(7^k - 4^k)}{3}(4t_{+a} + 4t_{+b} + 7t_{2+}) \end{aligned}$$

where the complexity t_+ for a_{ij}'s and for b_{ij}'s are designated as t_{+a} and t_{+b}.

3.2 Polynomial Entry Models

We assume that matices are univariate and of dense polynomial elements, and count the number of arithmetic operations on their coefficients. Let τ_+ and τ_* denote the costs for addition/subtraction and multiplication of coefficients, respectively. Let the degree of polynomials be d. We consider the two cases; A and B are univariate in different variables in Section 3.2.1, and A and B are univariate in a single variable in Section 3.2.2,

3.2.1 *A* and *B* are univariate in different variables case

Elements a_{ij} and b_{ij} are univariate polynomials with $(d + 1)$ terms, and the variables in a_{ij} and b_{ij} are different.

The complexity of arithmetics is as follows:

Table 1: Growth and the ratio of the numbers of arithmetics ((i) bivariate case)

k	$n = 2^k$	$C_k^{(I)}/C_{k-1}^{(I)}$	$D_k^{(I)}/D_{k-1}^{(I)}$	$C_k^{(I)}/D_k^{(I)}$
3	8			0.840
4	16	8.27	7.48	0.928
5	32	8.13	7.26	1.04
6	64	8.06	7.14	1.17
7	128	8.03	7.08	1.33
8	256	8.02	7.04	1.52

- $t_+ = t_{+a} = t_{+b} = (d+1)\tau_+$: addition/subtraction of $(d+1)$ terms,
- $t_* = (d+1)^2\tau_*$: multiplication of two univariate polynomials in different variables with $(d+1)$ terms,
- $t_{2+} = (d+1)^2\tau_+$: addition/subtraction of the above products.

$$\begin{aligned} T_{inn}^{(I)}(n) &= n^3(d+1)^2\tau_* + n^2(n-1)(d+1)^2\tau_+, \quad n = m2^k, \\ T_{SW}^{(I)}(m2^k) &= 7^k(m^3(d+1)^2\tau_* + m^2(m-1)(d+1)^2\tau_+) \\ &\quad + \frac{m^2(7^k-4^k)}{3}(8(d+1)\tau_+ + 7(d+1)^2\tau_+) \end{aligned}$$

As a reference for actual computing times to be given in the next section, we give some numeric data. Assuming that τ_+ and τ_* are almost equivalent, we count the total number of arithmetic operations. Let $C_k^{(I)}$ and $D_k^{(I)}$ denote the respective values $T_{inn}^{(I)}(2^k)$ and $T_{SW}^{(I)}(2^k)$ with $d = 4$ (and $m = 1$). Table 1 summarizes the growth ratio of these values with respect to k and their ratio.

3.2.2 A and B are univariate in a single variable case

All the elements are univariate polynomials in a single variable with $(d+1)$ terms, and the products are with $(2d+1)$ terms.

The complexity of arithmetics is as follows:

- $t_+ = t_{+a} = t_{+b} = (d+1)\tau_+$: addition/subtraction of $(d+1)$ terms,
- $t_* = (d+1)^2\tau_* + d^2\tau_+$: $(d+1)^2$ products are gathered into $(2d+1)$ coefficients in a univriate polynomial of degree $2d$,
- $t_{2+} = (2d+1)\tau_+$: addition/subtraction of the above products.

$$\begin{aligned} T_{inn}^{(II)}(n) &= n^3((d+1)^2\tau_* + d^2\tau_+) + n^2(n-1)(2d+1)\tau_+ \\ T_{SW}^{(II)}(m2^k) &= 7^k(m^3((d+1)^2\tau_* + d^2\tau_+) + m^2(m-1)(2d+1)\tau_+) \\ &\quad + \frac{m^2(7^k-4^k)}{3}(8(d+1)\tau_+ + 7(2d+1)\tau_+) \end{aligned}$$

In the detailed analysis above, we counted all the mathematical operations, and from the algebraic point of view, no further precise analysis will be very difficult, except for numerical coefficents. Our interest will be how well the above analysis matches practical result. As shown in the next section, the result is negative, and we investigate a good macroscopic measure for computing times in the next section.

4 Empirical Study

In general with algebraic computation, it is difficult to estimate the cost of computation in detail and to perform meaningful complexity analysis, because the structures and the sizes of treated data, symbolic expressions of formulas, vary during computation. To grasp the behavior of the computing cost and complexity of the algorithms, we have been doing experiments with various expressions and investigating qualitative characteristics of computing times [9, 8, 11, 10, 12]. We have such a prospect that because the cost of multiplication of symbolic expressions is much more than that of additive operations, the fast matrix multiplication algorithm with less elementwise multiplications will take much effect. In what follows, we shall show some typical results from our various experiments that characterize the computational behavior, and explain our pragmatic view. Note that the examples include the cases with which the fast algorithm reveals (incredibly) remarkable speedup. We show that some quantity, obtained through our experiments, is closely related with computing times, and mention that the quantity or its estimate can be used to measure the computational complexity.

4.1 Implementation in Risa/Asir and Experimental Data

To investigate the algorithms empirically, we use an experimental general-purpose computer algebra system Risa/Asir[5]. In Risa/Asir, as in other computer algebra systems, matrix is represented by two-dimensional array and the standard algorithm has been the only algorithm implemented for matrix multiplication. We implemented Strassen-Winograd algorithm in C, and incorporated it into the original implementation of the standard algorithm so that recursive call terminates to use the standard algorithm when the size of submatrix gets smaller than some threshold. While a submatrix of a matrix is represented as a portion of the array of the original matrix, every time when arithmetic operation is performed, a new matrix of the result is created.

Computing time is measured by using the functions `tstart()` and `tstop()` of Risa/Asir. All the timing data below are taken on FreeBSD 5.2 running on AMD Athlon™ XP 3200+ with 512MB memory.

In this paper, we use two types of square matrices of Table 2. Case I is quite simple that the

Table 2: Experimental data: expression of matrix elements

Case I	$a_{ij} = (i+1)(j+1)(x^5 + x^4 + x^3 + x^2 + x)$ $b_{ij} = (i+1)(j+1)(y^5 + y^4 + y^3 + y^2 + y)$
Case II	$a_{ij} = x^{(2*i+1)} + x^i + x^{(j+1)} + i * j$ $b_{ij} = x^{(2*i+1)} + x^{(i+1)} + x^j + i + j$

additive operations never change the structure of element expressions, and the costs for multiplications of a_{ij} and b_{kl} and for additive operations of their products are equivalent. Therefore, we can expect that the fast matrix multiplication algorithm reveals much better performance than the standard algorithm with Case I matrices. Therefore, it is expected that computing time will well reflect the number of operations given by Eq.'s (2) or (3). On the other hand, with Case II, it is difficult to tell the overall behavior of computing complexity. In the standard algorithm, the cost t_* for elementwise multiplication is equivalent for all elements, but the cost t_+ for addition in the inner-product calculation (1) will increase as k increases. In the case of the fast algorithm, the sizes, the number of terms more precisely, of element expressions in the intermediate submatrices change, and the cost t_* of multiplication is much more than that of inner product case.

4.2 Timings

First, we observe the behavior and the dependence of the computing times with respect to the matrix size(order) n, and compare two algorithms. Table 3 summarizes actual computing(CPU) times in seconds. Also, the growth ratio of the computing times and the ratio of the computing times of the two algorithms are given in Table 4, for the purpose of comparison with our theoretical analysis. In the case of Case I, the costs for additions and multiplications are almost fixed and as its result, speedup of the fast algorithm is achieved; Strassen-Winograd algorithm($O(n^{\log_2 7})$-algorithm) is much faster than the inner-product type algorithm($O(n^3)$-algorithm). Furthermore, it can be observed that while the computing times of the inner-product algorithm reveal stronger dependency on n than $O(n^3)$, those of SW algorithm do weaker dependency than $O(n^{\log_2 7})$. With respect to Case II, computing times are almost comparable between SW algorithm and the inner-product algorithm. In either case, or in general, the usually stated time complexity $O(n^3)$ and $O(n^{\log_2 7})$ of the algorithms do not agree with the behavior of actual computing times and almost meaningless.

Table 3: Computing times (CPU time, unit: second)

k	Size n	inner-product algorithm			Strassen-Winograd algorithm		
		CPU($t_k^{(in)}$)	GC	total	CPU($t_k^{(SW)}$)	GC	total
Case I							
3	8	0.008180	0.004957	0.01351	0.008301	0.005341	0.01408
4	16	0.05496	0.03483	0.09006	0.04470	0.02531	0.07020
5	32	0.4802	0.3173	0.8023	0.2416	0.1718	0.4143
6	64	4.068	3.177	7.288	1.215	1.014	2.237
7	128	35.76	29.08	65.12	6.007	4.866	10.94
8	256	307.4	311.2	622.2	28.88	28.76	58.06
Case II							
3	8	0.005243	0.002421	0.008109	0.005653	0.002519	0.008416
4	16	0.05067	0.02646	0.07772	0.04965	0.02825	0.07849
5	32	0.5060	0.2568	0.7791	0.4887	0.2456	0.7359
6	64	5.382	3.623	9.046	4.821	3.174	8.040

4.3 Amount of Arithmetic Operations and Space Complexity

Our question is what a factor affect on computing time most, and whether there is a good quantity which well reflects the computational complexity. The number of elementwise operations actually performed, shown in Table 5 for reference, may have deep relation with the complexity, but, needless to say, is almost useless as symbolic computation is concerned. Computing cost depends on the sizes or the structures of element expressions in general, but the number does not.

Table 4: Growth and ratios of CPU times

k	n	$t_k^{(in)}/t_{k-1}^{(in)}$	$t_k^{(SW)}/t_{k-1}^{(SW)}$	$t_k^{(in)}/t_k^{(SW)}$
Case I				
3	8			0.985
4	16	6.718	5.385	1.230
5	32	8.737	5.405	1.988
6	64	8.471	5.029	3.348
7	128	8.791	4.944	5.953
8	256	8.596	4.808	10.64
Case II				
3	8			0.927
4	16	9.664	8.783	1.021
5	32	9.986	9.843	1.035
6	64	10.64	9.865	1.116

Table 5: Number of operations on matrix elements

Size	inner product			Strassen-Winograd		
	add/sub	mult	total	add/sub	mult	total
8	448	512	960	624	448	1072
16	3840	4096	7936	5520	3136	8656
32	31744	32768	64512	43248	21952	65200
64	258048	262144	520192	321168	153664	474832
128	2080768	2097152	4177920	2321904	1975648	4297552
256	16711680	16777216	33488896	16548240	7529536	24077776

For more preciseness, we consider the number of term-wise operations. We limit our concern to the cases with matrices of polynomial entries for simplicity, and count how many terms are processed in all the arithmetic operations performed while matrix multiplication. The number of terms in the sum for addition and the product of the number of terms of two factors for multiplication will be good estimates for this count. If the form of polynomials of matrix elements is almost fixed, it will be possible to give an upper bound for the count. Table 6 summarizes this count actually obtained from our experiments. The total number of operations seems to have closer relation with actual computing time.

Let's get into more detail. For further preciseness, we have to consider the cost of coefficient calculations, which might be well represented by the size of numeric coefficients generated during calculation. To check this, we measured the amount of memory space allocated and exhausted in the numeric coefficient calculations, as shown in Table 7. This amount is the space requirement for numeric coefficients, which corresponds to space complexity. Finally, to confirm the correctness

Table 6: Number of operations

Size n	inner product			Strassen-Winograd			A/B
	add/sub	mult	total(A)	add/sub	mult	total(B)	
Case I							
8	12800	12800	25600	14640	11200	25840	0.99
16	102400	102400	204800	99360	62400	161760	1.27
32	819200	819200	1638400	579120	302400	881520	1.86
64	6553600	6553600	13107200	3080880	1339200	4420080	2.97
128	52428800	52428800	104857600	15442800	5572800	21015600	4.99
256	419430400	419430400	838860800	74336080	22161600	96497680	8.69
Case II							
8	8178	6666	14844	9005	5921	14926	0.99
16	122191	59036	181227	102764	48792	151556	1.20
32	1871637	497472	2369109	1046161	413030	1459191	1.62
64	29209204	4085384	33294588	10140399	3634367	13774766	2.42

Table 7: Total memory amount used for coefficients (unit:byte)

Size n	inner product			Strassen-Winograd		
	add/sub	mult	total	add/sub	mult	total
Case I						
8	1204875	44725	1249600	1783230	45950	1829180
16	11976300	395700	12372000	16675580	342325	17017905
32	121810500	4271100	126081600	134629575	2632175	137261750
Case II						
8	179598	20065	199663	381834	19904	401738
16	1893566	179799	2073365	8859643	188212	9047855

of our assertion, we compute the ratio of the amount of memory to the computing time. Table 8 gives the natural logarithms of the ratios. As can be seen in the table, the ratio is almost constant. Therefore, we insist that the space requirement is an important factor which have a close relation with computing time, and space complexity will be a good measure.

Table 8: Ratio of memory amount to computing time

Size n	inner product			Strassen-Winograd		
	add/sub	mult	total	add/sub	mult	total
Case I						
8	7.950	6.520	7.966	8.103	6.514	8.114
16	8.124	7.055	7.884	8.376	6.688	8.385
32	8.181	6.726	8.196	8.512	6.803	8.520
Case II						
8	7.345	6.393	7.391	7.657	6.374	7.679
16	7.386	6.364	7.426	8.053	6.380	8.062

5 Implementation for Sparse Matrix and Empirical Study

Efficient use of memory space or reducing the amount of memory use often leads to speedup in general. The result in the previous section insists, in a sense, that redundant expansion of memory use may degrade processing speed. We often use sparse matrices in practice rather than dense ones, and it will be the case with treating sparse matrices.

In Risa/Asir, there is prepared only one canonical matrix representation using two-dimensional arrays, and it always requires memory space proportional to matrix size, even for zero matrices. This representation is easy to understand and treat, however, saving of memory is not considered and not possible.

Another problem with this representation is that every time arithmetic operation is performed, even zero entries must be treated, usually for nothing, and its computing cost, although negligible, can hardly be measured and thus predicted. To sharpen the estimate the computing cost based on the hypothesis in the previous section, we ought to discard zero entries from matrix representation for sparse matrices. We implement a new data structure for sparse matrices, which requires memory space proportional to the number of non-zero elements, independently of the matrix size.

5.1 Index-type Representation of Matrix

A new matrix representation is designed with policy of use of less memory. It is a list of non-zero elements, which is same as polynomial representation in Risa/Asir. The following portion of C program describes the definition, and Figure 1 depicts a simple example. A matrix is represented by struct `oIndMat`, and each element, represented by struct `oIndEnt`, is stored in struct `oIndMatC` of chunk of memory.

We consider a method for fast access to each element. Let matrix size be $n \times m$, stored in the struct `oIndMat` as `row` and `col`, and let the index of an element a_{ij} be

$$(i-1) \times m + j,$$

which is stored in the field `cr` of struct `IndEnt`. For this reason, we call this new matrix representation "index type" matrix. The field `clen` of struct `oIndMat` contains the number of non-zero elements, and we can discriminate a zero matrix by checking the field. Notice again that with index

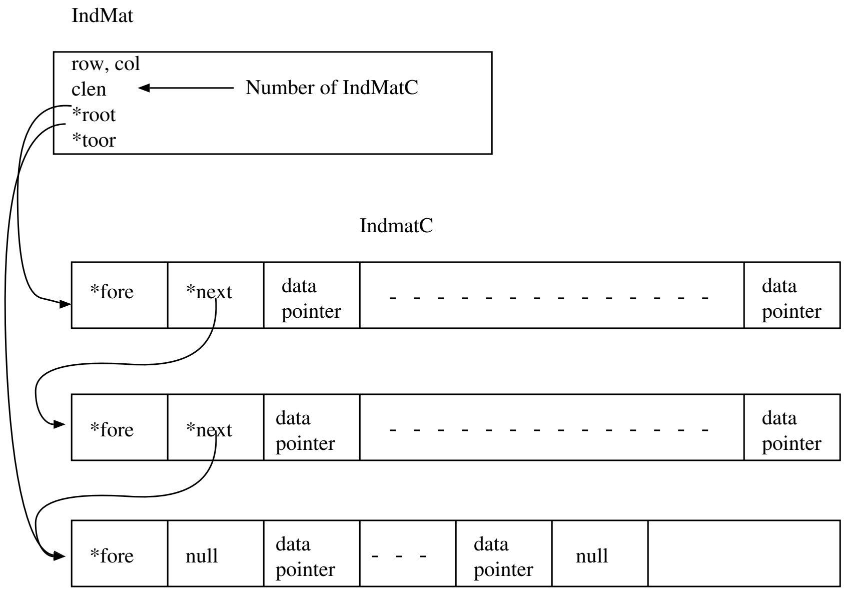

Figure 1: Diagram of Index-type Representation

type matrix, access to zero elements never occurs because the representation does not contain zero entries.

5.2 Comparison of Matrix Representation by Timings

We measure timings of matrix multiplications of sparse matrices in canonical matrix representation and index type matrix, and compare these two matrix representations. We use the matrix of Case I in Table 2, with some randomly-chosen elements of each of A and B replaced by zero, and measure timings for various ratios of the number of zeros, matrix sparseness. Throughout this section, we use this matrix for experiments. Table 9 summarizes the result of our experiment, with ratio of the number of zero elements 1/2 to 31/32.

When matrices are dense or not sparse, there is no distinct difference of speed between two types of representations. However, as matrices get sparse, the index type matrix representation gets more speeded and become faster than the canonical matrix representation.

5.3 Comparison of Algorithms in Index-type Matrix Representation

We also implemented Strassen-Winograd algorithm for index type matrix, and compared with the inner-product algorithm. Table 10 shows timings for sparse matrices, where the matrices used are the same as in the previous subsection. The results in Table 10 indicate that SW algorithm is much slower than the inner-product algorithm and seems almost useless, at least as sparse matrices represented as index type matrix are concerned.

```
#define IndMatCH 64       /* chunk size */
typedef struct oIndMat {  /* matrix structure */
        short id;
        int row, col;     /* matrix size,
                             the number of rows and columns */
        int clen;         /* the number of structure IndMatC's */
        pointer *root;    /* pointer to the first chunk data */
        pointer *toor;    /* pointer to the last chunk data */
} *IndMat;
typedef struct oIndMatC { /* structure to store chunk data */
        pointer *fore;    /* pointer to the previous chunk */
        pointer *next;    /* pointer to the next chunk */
        IndEnt ient[IndMatCH];
} *IndMatC;
typedef struct oIndEnt {  /* structure to store real data */
        int cr;           /* index of this element */
        int row, col;     /* position at matrix */
        pointer *body;    /* real data of an element */
} IndEnt;
```

5.4 Sparseness, Representation and Algorithm

We are investigating a good method to treat sparse matrices. In the previous subsections, we observed that the index type representation reveals better performance than the canonical representation if we use the inner-product algorithm, and that for algorithm comparison in the index type representation, Strassen-Winograd algorithm never be better than the inner-product algorithm, unlike the results of dense cases in Section 4. We wonder the latter result; isn't the index type representation suited for SW algorithm, or isn't SW algorithm suited for sparse matrices? With matrix sizes being fixed as 64×64, we measure every possible combinations of representations and algorithms, and observe how computing times change as the ratio increases from 0.

Table 11 summarizes the timings of our final experiment. It shows that computing times are affected by the choice of algorithm much more than that of matrix representation, SW algorithm is useless for sparse matrices, and with the inner-product algorithm, the index type representation becomes much faster than the canonical representation as matrices get sparse. Notice that for dense matrices, the computing times in both matrix representations are equivalent. From these facts, we may insist that our new matrix representation, index type representation, is suited for computer algebra, rather than two-dimensional array representation. Also notice that the the inefficiency of SW algorithm for sparse matrices will be caused by the violation of sparseness by additions and subtractions of submatrices, which might be justified by the fact that SW algorithm get fast when matrices are extremely sparse.

Table 9: Computing times in canonical representation and index-type representation (unit:μsec)

ratio of 0's	Size	canonical type matrix CPU	GC	total	index type matrix CPU	GC	total
$\frac{1}{2}$	8	1073		1073	1045		1044
	16	12570	6776	19370	12190	6805	19060
	32	117600	74080	192100	113400	63890	177600
	64	967100	631400	1607000	919200	499900	1427000
	128	8052000	5227500	13340000	7342000	2078000	9462000
$\frac{3}{4}$	8	392		390	372		373
	16	2055		2055	1981		2017
	32	21450	11360	32850	23520	12000	35780
	64	231500	113900	345900	209900	81170	291700
	128	1957000	1101000	3069000	1766000	647600	2423000
$\frac{7}{8}$	8	101		100	69		68
	16	561		560	403		403
	32	5104		5113	3765		3801
	64	46530	11230	57810	36840	14010	50920
	128	475300	128200	609400	356300	175400	537700
$\frac{15}{16}$	8	46		45	25		24
	16	272		271	103		102
	32	2370		2377	995		996
	64	17770		17830	7307		7318
	128	154600	23210	178100	68900	22340	91370
$\frac{31}{32}$	8	45		42	17		17
	16	215		213	39		35
	32	1579		1575	273		270
	64	12260		12270	1862		1861
	128	102000		102100	15750		15790

6 Conclusion

To investigate and find a relation between computing time in practice and the complexity obtained by theoretical analysis in the matrix multiplication algorithms, we have repeated experiments and examined their results in detail. Throughout this empirical study, we found that computing time reveals close connection with space complexity, in the case of matrix multiplication with polynomial elements. Based on this fact, we proposed a new matrix representation. Empirical tests using our simple implementation of matrix arithmetics indicated satisfactory results; in the case of dense matrices, the computing speed in the new representation is comparable with the usual canonical representation, and becomes much faster as matrices get sparse. So, to conclude, we state that the new matrix representation is useful and we need to consider much more about matrix representation

Table 10: Computing time of index type matrix multiplication (unit:μsec)

ratio of 0's	Size	inner product			Strassen-Winograd		
		CPU	GC	total	CPU	GC	total
$\frac{1}{2}$	8	1045		1044	3812	2853	6760
	16	12190	6805	19060	35500	21130	56780
	32	113400	63890	177600	322600	209200	532800
	64	919200	499900	1427000	2457000	1954000	4430000
	128	7342000	2078000	9462000	18310000	17560000	36120000
$\frac{3}{4}$	8	372		373	1784	2440	4227
	16	1981		2017	16630	11280	27950
	32	23520	12000	35780	206000	120900	327600
	64	209900	81170	291700	1888000	1363000	3269000
	128	1766000	647600	2423000	15470000	12380000	28020000
$\frac{7}{8}$	8	69		68	271		271
	16	403		403	3140	3339	10770
	32	3765		3801	97620	51600	149500
	64	36840	14010	50920	1269000	853300	2131000
	128	356300	175400	537700	12010000	9545000	21710000
$\frac{15}{16}$	8	25		24	120		119
	16	103		102	3140		3140
	32	995		996	42790	14500	57390
	64	7307		7318	612000	345600	959300
	128	68900	22340	91370	7884000	6280000	14280000
$\frac{31}{32}$	8	17		17	32		30
	16	39		35	966		965
	32	273		270	17130	6751	23940
	64	1862		1861	240000	79500	320300
	128	15750		15790	4148000	3260000	7490000

for computer algebra.

References

[1] D. Bini and V.Y. Pan. *Polynomial and Matrix*, Vol. 1 of *Progress in Theoretical Computer Science*. Birkhäuser, 1994.

[2] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

Table 11: Comparison of computing speed by type of matrix unit:sec)

ratio	inner product		Strassen-Winograd	
of 0's	canonical	index	canonical	index
0	7.264	7.324	2.244	2.113
1/2048	7.367	7.256	2.288	2.199
1/1024	7.268	7.256	2.266	2.280
1/512	7.289	7.061	2.298	2.337
1/256	7.235	7.038	2.490	2.480
1/128	7.237	6.982	2.654	2.665
1/64	7.113	6.974	2.981	3.099
1/32	6.845	6.652	3.450	3.484
1/16	6.424	6.242	3.929	3.960
1/8	5.589	5.392	4.453	4.453
1/4	4.082	3.859	4.813	4.766
1/2	1.607	1.427	4.360	4.430
3/4	0.3459	0.2917	3.274	3.269
7/8	0.05781	0.05092	2.024	2.131
15/16	0.01783	0.007318	0.8046	0.9593
31/32	0.01227	0.001861	0.2494	0.3203
63/64	0.01102	0.0005422	0.1004	0.1289
127/128	0.01064	0.0001462	0.03898	0.05984
255/256	0.01057	0.00009203	0.03317	0.03170
511/512	0.01045	0.00002599	0.03125	0.02442
1023/1024	0.01040	0.00001597	0.02871	0.01264
2047/2048	0.01036	0.00001597	0.02700	0.0006859

[3] M. L. Griss. The algebraic solution of sparse linear systems via minor expansion. *ACM Transactions on Mathematical Software*, Vol. 2, No. 1, pp. 31–49, 1976.

[4] E. Horowitz and S. Sahni. On computing the exact determinant of matrices with polynomial entries. *Journal of the ACM*, Vol. 22, No. 1, pp. 38–50, 1975.

[5] M. Noro and T. Takeshima. Risa/Asir — a computer algebra system. In P. S. Wang, editor, *Proceedings of ISSAC '92*, pages 387–396, Berkeley, CA, July 27–29 1992.

[6] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[7] S. Winograd. On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4:381–388, 1971.

[8] 兵頭 礼子, 村尾 裕一, 齋藤 友克. Risa/asir の matrix 演算の検討. 数式処理, 9:10–11, 2002. 第 11 回大会報告.

[9] 兵頭 礼子, 村尾 裕一, 齋藤 友克. Risa/asir の matrix 演算の新しい実装について. 講究録 *1295* 「Computer Algebra – Algorithms, Implementations and Applications, 2001」, pages 213–219. 京都大学数理解析研究所, 2002.

[10] 兵頭 礼子, 村尾 裕一, 齋藤 友克. 数式処理のための行列演算の効率的な実装法について. 数式処理, 10:18–19, 2003. 第 12 回大会報告.

[11] 兵頭 礼子, 村尾 裕一, 齋藤 友克. 多項式表現と行列演算の改良. 講究録 *1335* 「Computer Algebra – Algorithms, Implementations and Applications, 2002」, pages 28–32. 京都大学数理解析研究所, 2003.

[12] 兵頭 礼子, 村尾 裕一, 齋藤 友克. 行列計算と基本線形演算の実装法について. 講究録 *1395* 「Computer Algebra – Algorithms, Implementations and Applications, 2003」, pages 218–223. 京都大学数理解析研究所, 2004.