数理解析研究所講究録
436 巻 1981 年 67-81

PARALLELISM IN ALGEBRAIC COMPUTATION
and
PARALLEL ALGORITHMS FOR SYMBOLIC LINEAR SYSTEMS

Tateaki Sasaki* and Yasumasa Kanada†

*) The Institute of Physical and Chemical Research
Wako-shi, Saitama 351, Japan
†) Institute of Plasma Physics, Nagoya University
Chikusa-ku, Nagoya 464, Japan

ABSTRACT

Parallel execution of algebraic computation is descussed in the first half of this paper. It is argued that, although a high efficiency is oftained by the parallel execution of divide-and-conquer algorithms, the processor turnover ratio is still small. Parallel processing will be most successful for the modular algorithms and many algorithms in linear algebra. In the second half of this paper, parallel algorithms for symbolic determinants and linear equations are proposed. The algorithms manifest a very high parallelism and will give a very high efficiency in a simple parallel processing scheme. These algorithms are well usable in also the serial processing scheme.

Key Words and Phrases: algebraic computation, divide-and-conquer algorithm, modular algorithm, parallel processing, symbolic determinant, symbolic linear system, minor expansion, Cramer's method.

§1. Introduction

A remarkable advancement has been attained in parallel processing recently, [1,2,3] and many parallel algorithms were developed in the area of numeric computations. This paper is concerned with parallel processing of algebraic computations, in particular, with parallel algorithms for symbolic linear systems.

There are two reasons for studying parallel algorithms for symbolic linear systems. First, the algebraic calculations which are performed most frequently in the applications are calculations of symbolic determinants and solving symbolic linear equations, and the users are requesting efficient routines for these calculations. Second, calculations for symbolic linear systems are well suited for parallel processing, and a very high efficiency will be obtained by parallel processing as we shall show in this paper.

In the below, the term "task" is used as a full calculation to be executed by a computer, and the term "task unit" as a part of the calculation which is executable by one of the parallel processors of the computer.

There are basically two problems in any scheme of parallel processing. One is the cost of communications among different processors working parallelly, and the other is how to divide a task into task units which are executable highly parallelly. Regarding to the first problem, the ratio of the communication cost to the total computation cost decreases as the size of each task unit increases. On the other hand, if the size of each task unit is increased, parallelism among the task units is decreased in general. Due to this reason, many researchers are concentrating their attention mainly upon low level parallelism, such as parallelism among basic arithmetic operations.

In this paper, we consider only high level parallelism, that is, the parallelism among task units of large sizes. We cannot always find such high level parallelism in a general algebraic algorithm. We can, however, find high level parallelisms in many algorithms for symbolic linear systems. For example, we can reduce the main part of the determinant calculation to a set of large task units of almost the same size which are executable parallelly. The communication cost is negligible in such algorithms, and we can obtain a very high efficiency.

§2. Various parallelisms in symbolic/algebraic algorithms

Parallel processing of LISP has been discussed by many authors.[4,5,6] Among various parallelisms in processing lists, the parallelism which is quite obvious and seems to be quite effective for speeding up the processing is the parallel execution of the function EVLIS. That is, we evaluate a set of arguments in each procedure parallelly. This scheme will be effective if the arguments themselves are large task units of almost the same size. However, the most LISP programs are composed of many procedures of quite different sizes, and they refer to each other many times going into deep recursion levels. Furthermore, the number of arguments in a procedure is not many: the number is only 2 or 3 on an average. Therefore, the efficiency of computation will not increase so much even if we use many parallel processors.

This point was clarified by Yasui et al. of Osaka University recently.[7] These authors simulated the parallel execution of EVLIS on five common LISP programs, and found that the efficiency was increased by a factor of $4 \sim 5$ on only one program and no remarkable increase of the efficiency was found on other four programs. Furthermore, the efficiency increase was almost saturated at eight processors.

A high efficiency will be obtained if we execute divide-and-conquer algorithms parallelly. In many divide-and-conquer algorithms, a task is usually divided into two almost equal subtasks which are executable parallelly. Each subtask itself is divided into two smaller subtasks, and so on. Therefore, we have 2^k subtasks at the k-th recursion level, and we can execute these subtasks parallelly.

Suppose we have a task of size n which can be divided into two subtasks of size n/2, and let f(n) be the number of basic operations necessary to divide the task into the subtasks and unify the results of subtasks to get the answer. Let $T_s(n)$ be the number of basic operations necessary to execute this task in the conventional serial processing scheme. Then, we have the following recursive relation:

$$(1) \qquad T_s(n) = 2 \cdot T_s(n/2) + f(n).$$

If $f(n) \propto n \log_2^m n$, which is the case for many fast algorithms, the above relation gives

$$(2) \qquad T_s(n) \propto \frac{1}{m+1}\, n \log_2^{m+1} n.$$

On the other hand, in the parallel processing scheme mentioned above, we have the following recursive relation for the time complexity function $T_p(n)$:

$$(3) \qquad T_p(n) = T_p(n/2) + f(n).$$

If $f(n) = cn$, where c is a constant, we have

$$(4) \qquad T_p(n) = 2cn.$$

Here, we assumed we had an enough number of processors. Similarly, if $f(n) = cn \log_2 n$, we get

$$(5) \qquad T_p(n) = 2cn \log_2(n/2),$$

and if $f(n) = cn \log_2^2 n$, we obtain

$$(6) \qquad T_p(n) = 2cn(\log_2^2 n - 2\cdot\log_2 n + 3).$$

For example, $f(n) = (3/2)n$ for the fast Fourier transform. Hence, $T_s(n) = (3/2)n \log_2 n$ steps are necessary in the serial computation, while the computation will be finished within $T_p(n) = 3n$ steps in the parallel processing.

Probably only one defect of the above parallel execution of divide-and-conquer algorithms is that high parallelism is attained only at deep recursion levels. At the k-th recursion level, 2^k processors are working while only one processor is working at the top level where the size of the task unit is largest. Therefore, the processor turnover ratio is about $(\log_2 N)/N$ if we have N processors ($N \gg 1$).

The third type of the parallelism, which is the main theme of this paper, is on an algorithm which can be divided into parallelly executable large task units of almost the same sizes. A typical example is the modular algorithm. For example, let us consider the modular algorithm for coupled linear equations of integer coefficients. The main

part of the algorithm is to solve the equations by the Gaussian elimination over Galois fields $GF(p_i)$, $i=1,\cdots,k$, where p_1, $\cdots$, p_k are mutually distinct prime numbers. Therefore, the original task is divided into k almost the same task units which are executable parallelly and a task unit which is to interpolate the answers over $GF(p_i)$, $i=1,\cdots,k$.

The architecture of a machine which executes the modular algorithms parallelly is quite simple: a set of processors having their own work memories and being coupled loosely to a large common memory under a supervising processor. In fact, Yoshimura et al. of Toshiba Corporation realized such a machine and demonstrated that the machine was quite useful for solving coupled linear equations of integer coefficients by a modular algorithm.[8)]

§3. Data-driven computation and a minor expansion algorithm

One of the most attractive and prospective schemes of parallel computation is the data-driven computation.[2)] In this scheme, the computation is done as follows: When a task unit has been executed, the result is transmitted to all the task units that use the result as their inputs. When a task unit is given all necessary inputs, it is activated to be executable irrespectively of the state of other task units. That is, the flow of data controls the computation and the processor turnover ratio is made as large as possible. Therefore, we will obtain a large throughput in principle.

Many algorithms for symbolic linear systems can be well executed by the data-driven computation scheme. A typical example is the minor expansion algorithm by Griss[9)] and Wang.[10)] In this algorithm, all different minors that are necessary to expand the determinant recursively are firstly replaced by temporal variables generated by the system. When the determinant is expanded completely, the minors are evaluated successively and substituted for the temporal variables. The computation time is mostly spent at the second step in this algorithm when executed serially. This step can, however, be executed highly parallelly in the data-driven scheme.

In order to get a better insight into the computation, let us consider the following 4×4 matrix:

$$(7) \qquad M_4 = \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{pmatrix}.$$

In the first step of the algorithm, the determinant of this matrix is expanded in temporal variables as follows:

$$\begin{aligned}
&G0001 := c_3d_4 - c_4d_3; \\
&G0002 := b_3d_4 - b_4d_3; \\
&G0003 := b_3c_4 - b_4c_3; \\
&G0004 := b_2G0001 - c_2G0002 + d_2G0003; \\
&G0005 := a_3d_4 - a_4d_3; \\
(8)\quad &G0006 := a_3c_4 - a_4c_3; \\
&G0007 := a_2G0001 - c_2G0005 + d_2G0006; \\
&G0008 := a_3b_4 - a_4b_3; \\
&G0009 := a_2G0002 - b_2G0005 + d_2G0008; \\
&G0010 := a_2G0003 - b_2G0006 + c_2G0008; \\
&|M_4| := a_1G0004 - b_1G0007 + c_1G0009 - d_1G0010;
\end{aligned}$$

The minors being assigned to temporal variables G0001, G0002, G0003, G0005, G0006 and G0008 can be evaluated parallelly, and after these evaluations, G0004, G0007, G0009 and G0010 can be evaluated parallelly.

It is interesting to point out that the procedure (8) is quite similar to a program written in the single assignment language.[11] This language was designed to manifest the flow of data maximumly, and it is considered to be the simplest language for data-driven computation.

We can observe from the above example that the size of a task unit increases and the degree of parallelism decreases as the processing proceeds in the above algorithm. This point is undesirable for parallel processing, although the processor tunrover ratio is much larger in the above algorithm than in parallel processing of a divide-and-conquer algorithm.

§4. New determinant algorithm with high parallelism

Let M be an n×n matrix:

$$M = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}. \tag{9}$$

Following Laplace's expansion formula, we can calculate the determinant of M as

$$|M| = \sum_{(i_1,\cdots,i_{n_1})} |M^{(i_1,\cdots,i_{n_1})}| \cdot |\tilde{M}^{(j_1,\cdots,j_{n_2})}|, \tag{10}$$

$$n_1 + n_2 = n,$$
$$1 \leqq i_1 < i_2 \cdots < i_{n_1} \leqq n,$$
$$1 \leqq j_1 < j_2 \cdots < j_{n_2} \leqq n,$$
$$\{i_1,\cdots,i_{n_1},j_1,\cdots,j_{n_2}\} = \{1,2,\cdots,n\},$$

where $M^{(i_1,\cdots,i_{n_1})}$ is the $n_1 \times n_1$ submatrix constructed from left n_1 columns and $i_1, \cdots, i_{n_1}$ rows of M without changing their order, $\tilde{M}^{(j_1,\cdots,j_{n_2})}$ is the $n_2 \times n_2$ submatrix constructed from right n_2 columns and $j_1, \cdots, j_{n_2}$ rows of M multiplied by the sign factor, and the summation is made over all possible sets $(i_1,\cdots,i_{n_1})$. The number of summands in (10) is ${}_nC_{n_1} = {}_nC_{n_2}$. In the below, we write $M^{(i_1,\cdots,i_{n_1})}$ and $\tilde{M}^{(j_1,\cdots,j_{n_2})}$ as, respectively, $M^{(i)}$ and $\tilde{M}^{(i')}$ in short.
For example, the determinant of the 4×4 matrix (7) is calculated as

$$|M_4| = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \times \begin{vmatrix} c_3 & d_3 \\ c_4 & d_4 \end{vmatrix} - \begin{vmatrix} a_1 & b_1 \\ a_3 & b_3 \end{vmatrix} \times \begin{vmatrix} c_2 & d_2 \\ c_4 & d_4 \end{vmatrix} + \begin{vmatrix} a_1 & b_1 \\ a_4 & b_4 \end{vmatrix} \times \begin{vmatrix} c_2 & d_2 \\ c_3 & d_3 \end{vmatrix}$$

$$- \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix} \times \begin{vmatrix} c_1 & d_1 \\ c_4 & d_4 \end{vmatrix} + \begin{vmatrix} a_2 & b_2 \\ a_4 & b_4 \end{vmatrix} \times \begin{vmatrix} c_1 & d_1 \\ c_3 & d_3 \end{vmatrix} - \begin{vmatrix} a_3 & b_3 \\ a_4 & b_4 \end{vmatrix} \times \begin{vmatrix} c_1 & d_1 \\ c_2 & d_2 \end{vmatrix},$$

if we choose $n_1 = n_2 = 2$.

The conventional minor expansion algorithms for determinant calculation use the formula (10) recursively by setting $n_1 = 1$ and $n_2 = n-1$, i.e., by expanding the determinant w.r.t. a row or a column (cf., the expansion (8)). The algorithm we propose in this paper uses the formula

(10) recursively by setting $n_1 \simeq n/2$, i.e., by splitting the matrix M into two matrices of nearly the same orders. Then, as the above example shows, the major part of the determinant calculation is divided into ${}_nC_{n_1}$ subcalculations, i.e., evaluation of minors $|M^{(i)}|$ and $|M^{(i')}|$ and their product. These subcalculations are of nearly the same sizes and executable parallelly. Furthermore, the summation in (10) can be made highly parallelly by the binary summation method:

$$(11)\qquad \sum_{i=1}^{\ell} A_i = (\cdots(((A_1+A_2) + (A_3+A_4)) + \cdots + ((A_{\ell-3}+A_{\ell-2}) + (A_{\ell-1}+A_\ell)))\cdots).$$

When n is small, the ratio ${}_nC_{n_1}/n!$, which is the ratio of the number of summands in (10) to the number of terms of the determinant expanded completely, is rather large. Hence, it is better to use a conventional minor expansion algorithm if n is small. (In the following algorithms, we employ the conventional minor expansion algorithm if n is less than six.) Taking this notice into account, we obtain the following algorithm:

```
Parallel algorithm DET.SPLIT
%Calculate the determinant of order n by splitting it
%  into minors of orders n1 = [n/2] and n2 = n-n1 ;
  Input : an n×n matrix M with polynomial entries;
  Output: a polynomial D = |M|;
    if n ≦ 5 then return DET.MINOR(M);
    n1 ← [n/2];  n2 ← n - n1;
    [parallel execution w.r.t. index i, i=1,···,nCn1]:
      begin
        construct M(i) and M̃(i');
        D(i) ← DET.SPLIT(M(i))×DET.SPLIT(M̃(i'));
      end;
    [parallel binary sum w.r.t. index i, i=1,···,nCn1]:
      D ← sum D(i);
    return D;
```

Here, DET.MINOR in the above procedure is a determinant-evaluating procedure using the conventional minor expansion method.

The algorithm DET.SPLIT manifests a very high parallelism. In fact, if we use ${}_nC_{n_1}$ processors, we can evaluate $D^{(i)}$, $i=1,\cdots,{}_nC_{n_1}$, all at once. A high parallelism comparable to that in the above algorithm can be found only in modular algorithms, so far as algebraic algorithms are concerned. Furthermore, it is worthwhile to note that the algorithm DET.SPLIT is executable by such a simple machine as was referred to at the end of §2.

It is possible to improve the above algorithm considerably. The DET.SPLIT evaluates the same minor many times when $n \geq 4$. The first improvement is, hence, to avoid duplicate evaluation of minors. This can be achieved mostly by dividing $|M^{(i)}|$ and $|\tilde{M}^{(i')}|$, $i=1,\cdots,{}_nC_{n_1}$, into groups so that all determinants in each group contain as many same minors as possible and by evaluating all these groups parallelly. Then, we can avoid duplicate evaluation of the same minor in each group easily by such a method as mentioned in §3. This improvement will be quite effective if n is quite large. However, for many practical applications where $n \simeq 10$, the redundancy due to duplicate evaluation of minors in DET.SPLIT will not be so serious. For example, for the determinant of order 10, the duplicate evaluation is done only for minors of order less than or equal to 4, while minors of order less than or equal to 8 are multiply evaluated in the conventional recursive minor expansion algorithm.

The second improvement is for evaluating sparse determinants. The improvement is quite simple in our algorithm. We make reordering of rows and columns of M so that the nonzero elements are gathered around the diagonal line of M. Then, the number of summands in (10) decreases much. For example, suppose n = 10 and all elements M_{ij}, i=9 and 10, j=1,2,··,5, become zero after a reordering of rows and columns. This decreases the number of summands in (10) from ${}_{10}C_5 = 252$ to ${}_8C_5 = 56$.

§5. Parallel algorithm for symbolic linear equations

Using the same idea as was used in algorithm DET.SPLIT, we can construct efficient parallel algorithms for solving symbolic linear

equations. Consider the following coupled linear equations of n unknowns:

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1, \\ \vdots \qquad\qquad & \vdots \quad \vdots \\ a_{n1}x_1 + \cdots + a_{nn}x_n &= b_n, \end{aligned} \tag{12}$$

where, we assume the coefficient matrix $M = (a_{ij})$ is not singular, i.e., $|M| \neq 0$. Let M_k be the following $n \times n$ matrix:

$$M_k = \begin{pmatrix} a_{11} & \cdots & a_{1,k-1} & b_1 & a_{1,k+1} & \cdots & a_{1n} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & \cdots & a_{n,k-1} & b_n & a_{n,k+1} & \cdots & a_{nn} \end{pmatrix}, \quad k=1,\cdots,n. \tag{13}$$

Then, Cramer's formula gives the solution of (12) as

$$\begin{aligned} &x_k = D_k/D, \quad k=1,\cdots,n, \\ &\text{with} \quad D_k = |M_k| \quad \text{and} \quad D = |M|. \end{aligned} \tag{14}$$

From the viewpoint of parallel processing, D and D_k, $k=1,\cdots,n$, can be computed parallelly. The degree of parallelism in this scheme is only n+1. We can, however, calculate D and D_k highly parallelly as follows. Let $n_1 = [n/2]$ and $n_2 = n-n_1$, as before. According to formula (10), we can calculate D as

$$D = \sum_{i=1}^{{}_nC_{n_1}} |M^{(i)}| \cdot |\tilde{M}^{(i')}|, \tag{15}$$

where $M^{(i)}$ is the matrix of order n_1, $\tilde{M}^{(i')}$ is the matrix of order n_2, and the summation is made over ${}_nC_{n_1}$ different submatrices. The D_k, $k \leqq n_1$, is given by

$$D_k = \sum_{i=1}^{{}_nC_{n_1}} |M_k^{(i)}| \cdot |\tilde{M}^{(i')}|, \quad 1 \leqq k \leqq n_1, \tag{16}$$

where $M_k^{(i)}$ is the submatrix of order n_1 constructed from M_k by the same way as the construction of $M^{(i)}$ from M. Similarly, for $n_1 < k \leqq n$, we have

$$(17)\qquad D_k = \sum_{i=1}^{{}_nC_{n_1}} |M^{(i)}| \cdot |\tilde{M}_k^{(i')}|, \quad n_1 < k \leq n.$$

Note that the same minor $|\tilde{M}^{(i')}|$ appears in (15) and (16), and the same minor $|M^{(i)}|$ in (15) and (17). Therefore, if we save the expressions $|M^{(i)}|$ and $|\tilde{M}^{(i')}|$, $i=1,\cdots,{}_nC_{n_1}$, which are obtained by the evaluation of D, we can use them for the evaluation of $D_1, \cdots, D_n$. Thus, we obtain the following algorithm:

Parallel algorithm CRAMER.SPLIT

%Solve the coupled linear equations $\sum_{j=1}^{n} a_{ij}x_j = b_i$, $i=1,\cdots,n$,
% by Cramer's method, where a_{ij} and b_i are polynomials;
Input : an n×n nonsingular matrix $M = (a_{ij})$ and
an n-dimensional vector $\vec{B} = (b_1,\cdots,b_n)$;
Output: polynomials $D = |M|$ and $D_k = |M_k|$, $k=1,\cdots,n$;

```
D ← DET.SPLIT(M), and
   save d^(i) ← |M^(i)| and d̃^(i) ← |M̃^(i')|, i=1,···,nCn1;
for k ← 1 until n1 step 1 do
  begin
     [parallel execution w.r.t. index i, i=1,···,nCn1]:
        D^(i) ← d̃^(i)×DET.SPLIT(M_k^(i));
     [parallel binary sum w.r.t. index i, i=1,···,nCn1]:
        D_k ← sum D^(i);
  end;
for k ← n1+1 until n step 1 do
  begin
     [parallel execution w.r.t. index i, i=1,···,nCn1]:
        D^(i) ← d^(i)×DET.SPLIT(M̃_k^(i'));
     [parallel binary sum w.r.t. index i, i=1,···,nCn1]:
        D_k ← sum D^(i);
  end;
return D, D1, ···, Dn;
```

If we have many processors, parallel evaluation of $D_1, \cdots, D_n$ is also possible.

The above algorithm manifests a very high parallelism as DET.SPLIT

does, and it is executable by such a simple machine as was referred to at the end of §2.

The CRAMER.SPLIT will be quite efficient if $n < 10$ or so. If n becomes quite large, however, the overhead due to duplicate minor evaluation becomes serious. This overhead can be decreased much by the same method described for algorithm DET.SPLIT. In addition to this improvement, we have the following attractive scheme for decreasing the overhead.

Let $n_1' = [n/4]$, $n_2' = [n/2] - [n/4]$, $n_3' = [3n/4] - [n/2]$, and $n_4' = n - [3n/4]$. Let us divide the n columns of M into four groups and define four matrices M_I, M_{II}, M_{III} and M_{IV} of forms $n \times n_1'$, $n \times n_2'$, $n \times n_3'$ and $n \times n_4'$, respectively:

$$
\begin{aligned}
M_I &= (M_{ij}), \quad i=1,\cdots,n, \quad j=1,\cdots,[n/4], \\
M_{II} &= (M_{ij}), \quad i=1,\cdots,n, \quad j=[n/4]+1,\cdots,[n/2], \\
M_{III} &= (M_{ij}), \quad i=1,\cdots,n, \quad j=[n/2]+1,\cdots,[3n/4], \\
M_{IV} &= (M_{ij}), \quad i=1,\cdots,n, \quad j=[3n/4]+1,\cdots,n.
\end{aligned}
$$

We first evaluate ${}_nC_{n_1'}$ minors of order n_1' which are constructed from M_I without changing the order of the elements. Let these minors be in class I. Similarly, we evaluate ${}_nC_{n_2'}$, ${}_nC_{n_3'}$, and ${}_nC_{n_4'}$ minors constructed from M_{II}, M_{III}, and M_{IV}, respectively. Let these minors be in classes II, III, and IV, respectively. Second, we evaluate $|M^{(i)}|$, $i=1,\cdots,{}_nC_{n_1}$, by using minors in classes I and II. Let these minors be in class I+II. Similarly, we evaluate $|\tilde{M}^{(i')}|$, $i=1,\cdots,{}_nC_{n_1}$, by using minors in classes III and IV. Let these minors be in class III+IV. Then, we can evaluate D_k, $k=1,\cdots,[n/4]$, by using minors in classes II and III+IV. Similarly, we can evaluate D_k, $k=[n/4]+1,\cdots,[n/2]$, by using minors in classes I and III+IV, and so on.

It is clear that the number of groups into which the n columns of M are divided may be another number than 4. In addition to the above improvement, preordering of rows and columns of M and $\vec{B}$ will reduce the total amount of computations much when the matrix M is sparse, as was explained for DET.SPLIT.

§6. Empirical study

We have not tested our parallel algorithms yet but tested only a serial computation version of the determinant algorithm. We compared three algorithms, DET.MINOR, DET.EMINOR, and DET.SPLIT'. The DET.MINOR is the well-known recursive minor expansion algorithm, and DET.EMINOR is the efficient minor expansion algorithm being implemented in REDUCE by Griss. This algorithm uses backet hashing to avoid duplicate minor evaluation. The DET.SPLIT' is the serial computation version of DET.SPLIT presented in §4.

The test problem is the following matrix of order n-1:

$$M = \begin{pmatrix} a_{n-1} & 2a_{n-2} & \cdots & (n-1)a_1 \\ & a_{n-1} & \cdots & (n-2)a_2 \\ & & \cdots & \\ & & & a_{n-1} \end{pmatrix} \times \begin{pmatrix} a_1 & & & \\ 2a_2 & a_1 & & \\ \cdots & \cdots & \cdots & \\ (n-1)a_{n-1} & (n-2)a_{n-2} & \cdots & a_1 \end{pmatrix}$$

$$- \begin{pmatrix} na_n & (n-1)a_{n-1} & \cdots & 2a_2 \\ & na_n & \cdots & 3a_3 \\ & & \cdots & \\ & & & na_n \end{pmatrix} \times \begin{pmatrix} na_0 & & & \\ (n-1)a_1 & na_0 & & \\ \cdots & \cdots & \cdots & \\ 2a_{n-2} & 3a_{n-3} & \cdots & na_0 \end{pmatrix}.$$

The determinant of M gives the discriminant of equation of degree n except for a numeric factor.[13)]

The results are given in Table I.

order of M	DET.MINOR	DET.EMINOR	DET.SPLIT'
4	0.169	0.165	0.177
5	1.096	0.848	1.230
6	9.52	5.65	9.60
7	95.5	42.5	94.6

Table I: Computing times for dense determinant by three algorithms; times in seconds, and GBC times excluded. Memory = 11.2 MB.

Table I shows that our algorithms are, when executed serially, not so efficient as the efficient minor expansion algorithm but as efficient as the recursive minor expansion algorithm. (Note that DET.MINOR and DET.SPLIT' are the same for determinants of order less than or equal to five.) Therefore, our algorithms will be quite efficient when executed parallelly.

It is rather dangerous to say much about the serial computation version of our algorithms from only the data in Table I. However, we may well expect that, when improvements on our algorithms mentioned in §4 and §5 will be done, the algorithms will become quite efficient even in the serial computation scheme. In particular, our algorithm for symbolic linear systems will be useful because we can systematically avoid duplicate evaluation of $|M^{(i)}|$ and $|\tilde{M}^{(i')}|$ in the calculation of D_k, $k=1,\cdots,n$. It is worthwhile to note that our algorithms are space-efficient when executed serially. This is an important feature in applications, because the determinant calculation is often necked not by time but by space.

References

[1] Enslow, P.H., "Multiprocessor Organization - A Survey," Computing Surveys, Vol.9, No.1 (1977).

[2] Misunas, D.P., "Workshop on Data Flow Computer and Program Organization," Comp. Arch. News (ACM SIGARCH), Vol.6, pp.6-22 (1977).

[3] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," CACM Vol.21, pp.613-641 (1978).

[4] Friedman, D.P. and Wise, D.S., "Aspects of Applicative Programming for Parallel Processing," IEEE Trans. Comp. Vol.c-27, pp.289-296 (1978).

[5] Prini, G., "Explicit Parallelism in LISP-like Languages," Proc. LISP Conf., pp.13-18 (1980).

[6] Marti, J.B., "Compilation Techniques for a Control-Flow Concurrent LISP System," Proc. LISP Conf., pp.203-207 (1980).

[7] Yasui, H., Saito, T., Mitsuishi, A. and Miyazaki, Y., "Architecture of EVLIS Machine and Dynamic Measurements of Parallel Processing in LISP" (in Japanese), working paper of Kigoshori-Kenkyukai, IPS of Japan, (Dec. 1979); abstract in JIP. Vol.2, p.232 (1980).

[8] Yoshimura, S., Mizutani, H. and Shibayama, S., "A Parallel Processing Machine" (in Japanese), Collected Papers on Pattern Processing System - Natl. R&D Program, Agency of Industrial Sci. and Tech. of Japan, pp.211-222 (1980).

[9] Griss, M.L., "Efficient Expression Evaluation in Sparse Minor Expansion, Using Hashing and Deferred Evaluation," Proc. Hawaii Intl. Conf. on System Sciences, pp.169-172 (1977).

[10] Wang, P.S., "On the Expansion of Symbolic Determinants," Proc. Hawaii Intl. Conf. on System Sciences, pp.173-175 (1977).

[11] Tesler, L.G. and Enea, H.J., "A Language Design for Concurrent Processors," Proc. AFIPS Conf., pp.291-293 (1968).

[12] Griss, M.L., "The Algebraic Solution of Sparse Linear Systems via Minor Expansion," ACM TOMS Vol.2, pp.31-49 (1976).

[13] Sasaki, T., Kanada, Y. and Watanabe, S., "Calculation of Discriminants of High Degree Equations," Tokyo J. Math. (to appear).