記　号　処　理 71－3
（1993．11．19）

# SIMD 型超並列計算機による並列数式処理系

大野 智保　高橋 岳之　斉藤 制海　湯浅 太一

豊橋技術科学大学 知識情報工学系

REDUCE や Mathmatica を始めとする汎用数式処理系は様々な分野に急速に普及しつつあり、それに伴い機能や処理速度の向上が求められている。本稿では、現在筆者らが開発を進めている SIMD 型超並列計算機およびその上で稼働する並列 Common Lisp による高速並列数式処理系について報告する。

# Parallel Computing Algebra by using SIMD Parallel processor

OHNO Tomoyasu　TAKAHASHI Takeyuki　SAITO Osami　YUASA Taiichi

Toyohashi University of Technology

According as the general formula manipulation system , such as REDUCE and Mathematica , become the popular computing tools in various fields , the higher performance and faster computing speed are required .

This paper deals with the development of high speed formula manipulation system by using the SIMD parallel processor and parallel Lisp.

# 1 はじめに

現在では、REDUCE や Mathematica など種々の汎用数式処理系がリリースされている。それらは工学、理学などを始め様々な分野で利用され、その有用性はだれしもが認めるところである。筆者らも REDUCE をシステム制御工学の分野に応用し、新しいタイプの制御系 CAD 等を提案するなど一定の成果を挙げることが出来た。しかし、最近活発に研究が進められている多次元システム理論等に応用すると、従来の汎用数式処理系では、機能や処理速度の点で不満が残る。そこで本研究では、数式処理を並列化することにより機能の向上と処理速度の高速化をはかる。

数式処理の並列化の研究は従来からも見られるが、これらは既存の数式処理系が稼働する複数台の計算機に分散処理させるものである。それに対し、ここでは超並列アーキテクチャー、その上で稼働する並列処理言語及び並列アルゴリズムと一貫して並列化を考え、独自の並列数式処理系を開発する。すなわち、筆者の一人が開発した SIMD 型超並列計算機、及び記号処理に不可欠な並列 Lisp を基に並列数式処理系の開発を進める。

# 2 SIMD 型超並列計算機と並列 Lisp

本研究で提案する並列数式処理系の開発は、本学湯浅研究室で開発された超並列計算機 SM-1、及びその上で稼働する並列拡張版 Kyoto Common Lisp(以下 TUPLE と記す) を用いて開発した。ここでは SM-1 と TUPLE について概要を述べる。SM-1 は、SIMD 型並列アーキテクチャを採用している。すなわち、制御装置 (Front End、以下 FE と記す) から 1024 台の演算装置 (Processing Element、以下 PE と記す) へ同一の命令を同時に伝え、各 PE はその命令を並列に実行する。

SM-1 は FE として既存のワークステーションを用い、OS として UNIX を採用している。1024 台の PE は、32 個のレジスタ、1Mbyte のローカルメモリー、マイクロコードからなるマクロ命令セットを持っている。各 PE は自分自身のローカルメモリーだけではなく、PE 間通信を利用して他の PE のローカルメモリーも参照できる。この PE 間通信のためのいくつかのメモリー参照モードを持っている。これは SM-1 のアーキテクチャの大きな特徴である。また FE から PE のローカルメモリーの参照も PE-FE 間通信を用いて可能である。SM-1 のアーキテクチャの詳細については文献 [1] を参照されたい。

開発言語として使用した TUPLE は、SIMD 型並列計算機用の拡張版 Common Lisp であり、並列処理のための関数が用意されている。TUPLE の特徴は、並列リスト処理が可能であることと、並列計算を行なう場合に Common Lisp と同様の表記方法により記述できることである。

TUPLE では、FE 部、PE 部で取り扱う関数をそれぞれ FE 関数、PE 関数と区別している。FE 関数とはいわゆる一般的な Lisp と同様なものであり、PE 関数とは PE 部において並列計算を行なうためのものである。PE 関数は、多数の PE で同時に処理されているということ以外は一般の Lisp と同様である。TUPLE の詳細については文献 [2] を参照されたい。

# 3 並列数式処理のアルゴリズム

既存の汎用数式処理系は、多項式の各種演算、記号式を含む行列の演算、初等関数の微分、積分などの機能 (関数) を持ち、実用に十分耐え得るシステムである。これらの関数のうちで並列処理が可能なものがかなり見受けられる。例えば多項式の無平方分解やそれを基にした有理式の不定積分などがあげられる。

本研究では、既存の汎用数式処理系全般の並列化を対象とするのではなく、筆者らが現在興味を持っている、多項式環上でのシステム制御理論への数式処理の応用を念頭にいれた並列化を考える。すなわち、多項式を要素とする行列の演算や、行列方程式に関する並列数式処理系の開発を行なう。

本章において、SIMD 型並列計算機の概念モデルとして用いられる PRAM(Parallel RandomAccess Machine) により並列数式アルゴリズムの設

計を行なう。PRAMとは、多数のプロセッサ(PE)としてランダムアクセス機械(RAM)を使用し、それらは共有メモリによって結合される。本研究では2章で述べたSIMD型並列計算機SM-1及びTUPLEにより並列化を行なうため、PRAMに以下のルールを追加する。

1. 個々のPE間にはある種のネットワークが存在し、各PEの局所的な記憶装置である記憶レジスタは他のPEからの参照が可能である。
2. 複数PEでの共有メモリからの同時読み出しについては許可するが、同時書き込みについては許可しない。
3. 一旦計算待機となったPEは、現在計算中の全てのPEが終了するまで計算待機となる。
4. それぞれのPEは、レジスタ pn(processor number) にPEを指定する固有の番号を持つ。

以下使用したアルゴリズムを紹介する。

### 3.1 行列、逆行列の並列アルゴリズム

行列式を求める場合、ラプラスの展開法はガウスの消去法に比べて計算量が多くなるため数値計算の分野においては用いられることはなかった。しかし、行列の要素として多変数多項式を含む行列式の計算においては、ラプラス法は数式の中間結果の爆発が回避できるなど優れた点がある。ラプラスの展開法は周知のように、n × n行列Aの行列式 $|A|$ は、k行で展開(以下ではことわりのないかぎり1行で展開する)すると、

$$|A| = \sum_{l=1}^{n} a_{kl}(-1)^{k+l}|A^{(kl)}| \qquad (1)$$

となる。ここで $A^{(kl)}$ は、行列Aからk行$l$列を除いて得られる(n-1) × (n-1)小行列である。つまりn × n行列式の計算は、n個の(n-1) × (n-1)行列の行列式を並列計算することにより求められる。この展開法を(n-1) × (n-1)行列に適用すればさらに分割が進み、以下これを順次続けていけば最終的にn!個のスカラーからなる行列の行列式の並列計算となる。この後逆に結果を順次統合していけば最終的に $|A|$ が求められる。ラプラス展開法は、各PEが異なる行列の値で、行列式を計算するという同一の命令を実行すればよいのでSIMD型並列計算機向きのアルゴリズムであるといえる。

ラプラス法を並列化した場合、各PEがどのように命令を実行していくのかを3 × 3行列を例に模擬的に示す。3 × 3行列のラプラス展開を以下のように表す。

$$\begin{aligned} &|A| && \text{lev.1}\\ &= A_1 + A_2 + A_3 && \text{lev.2}\\ &= A_{11} + A_{12} + A_{21} + A_{22} + A_{31} + A_{32} && \text{lev.3} \end{aligned}$$

$A_i$、$A_{ij}$は、それぞれ $A_i = (-1)^{1+i}a_{1i}|A^{(1i)}|$、$A_{ij} = (-1)^{(i+j)}a_{1i}a_{2\alpha}|(A^{(1i)})^{(1j)}|$ であり、$\alpha = 3 \times ((i+j)\ floor\ 3) + (i+j)\ mod\ 3$ である。上式からlev.3で6個のスカラ行列式に帰着されるので、この計算には6個のPEを必要とする。lev.3で求められた結果は、lev.2の2 × 2小行列の行列式に統合され、さらにこの結果はlev.1の3 × 3行列の行列式に統合される。lev.3の6個のPEに渡される行列の分割及び結果の統合はFEで処理することも考えられるが、ここでは図3.1に示すように全ての処理をPE部で行なう。

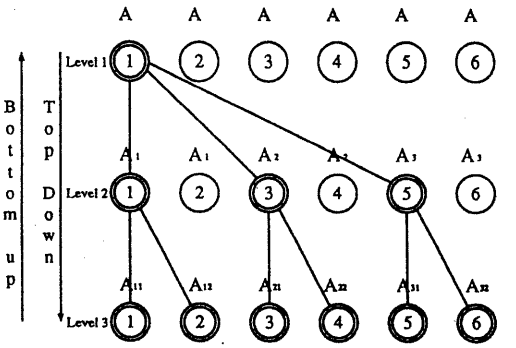

図3.1 アルゴリズムの計算例

図中の円はPEを、その中の数字はpnを表わしている。図のように最初FE-PE間通信により6個全てのPEに3 × 3行列Aを渡す。このとき親PEを1とし、他のPEは親と同じ命令を実行する。lev.2の段階で、行列Aをラプラス法により$A_1$、$A_2$、$A_3$の3つに展開される。このとき6個のPEは2個ずつ3つのグループに分かれる。1、

3、5をlev.2の親PEとし、1と2、3と4、5と6はそれぞれ$A_1$、$A_2$、$A_3$を持つ。つづいて1、2のPEにおいて$A_1$は2つのスカラー行列$A_{11}$、$A_{12}$に分割される。同様に$A_2$は3、4で、$A_3$は5、6で分割される。この時点で6個のPE全てが親になる。このようにトップダウン(分割)過程が終了すると、次はボトムアップ(解の統合)に入る。lev.2の各グループの親PEはlev.3の結果をPE間通信により読み出し、$A_i = \sum_{j=1}^{2} A_{ij}$、$i \in \{1,2,3\}$を実行する。同様に、lev.1で$|A| = \sum_{i=1}^{3} A_i$が実行される。

以上のラプラス法は一般的には図3.2(a)のように問題をトップダウン式に細分化し、ついで図3.2(b)のように各PEの結果をPE間通信を用いてボトムアップ的に統合することにより得られる。

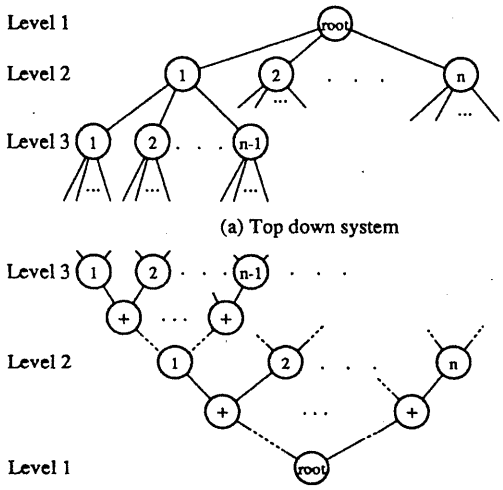


図3.2 問題の細分化と再構築

図3.1(a)のトップダウンの過程では、親PEが計算結果を得るために複数の子PEに作業を分担する必要がある。従って図3.1のように、必要となるn!個の全てのPEに最初n×nの行列を渡し、段階ごとに同じ小行列を扱うPEを一つのグループとし、最終的に全てのPEの扱うデータがスカラーとなったところでボトムアップの実行に入る。以下は、用いたアルゴリズムである。

アルゴリズム：MinorDet(A, n, RPN, CC)

入力：Aは正方行列、nは行列Aの大きさ、RPNは、同一の小行列計算をするPEグループの親PEのpn、1行CC列を取り除いて小行列を作成

出力：$|A|$

```
Polynomial Adj ;
Integer GroupNum, NextRPN, NextCC ;

if n=1 then return a11 ;
GroupNum ← pn - RPN ;
NextRPN ← pn-(GroupNum mod (n-1)!) ;
NextCC ← GroupNum floor (n-1)! + 1 ;
Adj ← a1CC × MinorDet(A^(1CC), n-1,
                      NextRPN, NextCC) ;
if a1CC ≠ 0 then
 if CC is even then Adj ← - Adj ;
else
 Adj ← 0 ;
return BottomUp(Adj).
```

初期値として、各PEを以下に設定する。

MinorDet(A, n, 1, ((pn-1) floor (n-1)!)+1)

アルゴリズム中の関数BottomUpは、図3.2(b)のボトムアップを行なう。図3.2(b)は、レベル3のあるグループがレベル2の1番の解を構成し、レベル2の1からn番までが一つのグループとなりレベル1の解を構成する。ここでレベル1を親PEとした場合レベル2を子PE、レベル2を親PEとした場合レベル3を子PEと呼び、レベルnまで同様に定義する。ボトムアップの過程においては、子PEの値の二項演算の結果が親PEの値を形成することになり、この動作を繰り返すことによって最終的なrootの結果を求める。このようなボトムアップの計算は、関数BottomUpで行なう。関数BottomUpは、引数として二項演算を行なうための各PEの変数を持ち、本アルゴリズムでは二項演算として加算を行なう。

次にn×n行列の逆行列計算の並列アルゴリズ

ムについて述べる。数値計算における逆行列はガウス方が一般的であるが、ここでは並列性を考慮して余因子行列を用いた。

$$A^{-1} = \frac{1}{|A|} adjA \qquad (2)$$

(2)式には$n^2$個の(n-1)×(n-1)余因子行列の行列式計算が含まれるが、前述の並列行列式計算を適用する。従って、この並列計算には$n^2 \times (n-1)!$個のPEを必要とする。この問題も行列式と同様問題の分割と統合を行なわなくてはならないが行列式の場合に類似しているので割愛する。

(2)式の分母、分子には共通因子が存在することがあるので、各行列要素毎に両者の最大公約式を導出する必要がある。そこで要素毎にユークリッドの互除法を用いて並列にGCD計算を行なう。

## 3.2 グレブナー基底の並列アルゴリズム

計算機代数学の分野で近年注目を浴びているグレブナー基底は、連立代数方程式や、行列代数方程式の数式計算アルゴリズムの強力な手法である。ここではグレブナー基底の導出の並列化について述べる。

グレブナー基底を説明する前に、多変数$(x_1, \ldots, x_n)$多項式Fの M- 簡約およびS-多項式について述べる。多項式Fを単項式の和で表すと、

$$F = c_1 T_1 + c_2 T_2 + \cdots \qquad (3)$$
$$c_i \in K(\text{数体}), T_i = x_1^{e_{i1}} \cdots x_n^{e_{in}}$$

指数の組$(e_{i1}, \cdots, e_{in})$に適当な全順序を導入し、多項式中の任意の各項を一意的に順序づけする。(3)式に$T_1 \succ T_2 \succ \cdots$という項順序を導入するとき、$c_1T_1$を頭項(head term)、$c_1$を頭係数(head coefficient)、$T_1$を頭べき積(head power product)といい、それぞれht(F)、hc(F)、hpp(F)と表す。

$G = c_1'T_1' + c_2'T_2' + \cdots$、$c_i' \in K$、$ht(G) = c_1'T_1'$なる多項式とする。(3)式のある単項cTに対し、Tが$T_1'$で割り切れるとき、

$$F - \frac{c \cdot T}{ht(G)} \cdot G = F' \qquad (4)$$

で求められる多項式$F'$は、$F \succ F'$となる。これをFのGによるM-簡約といい、$F \longrightarrow\!\!\!\rightarrow_G F'$と表わす。FをGで可能なかぎり簡約し、どの項もGでM-簡約できなくなった式をM-既約であるという。同様に、Fを多項式イデアル$\Gamma = \{G_1, ..., G_s\}$でM-既約化した式を$\tilde{F}$とするとき、

$$F \longrightarrow\!\!\!\rightarrow_\Gamma \tilde{F} \qquad (5)$$

と表わす。二つの多項式$F_1$、$F_2$に対して関数S-POLを、

$$\text{S-POL}(F_1, F_2) = hc(F_2)\frac{lcm(hpp(F_1), hpp(F_2))}{hpp(F_1)} F_1 - hc(F_1)\frac{lcm(hpp(F_1), hpp(F_2))}{hpp(F_2)} F_2 \qquad (6)$$

と定義し、得られる結果をS-多項式と呼ぶ。ここでlcmは二つの多項式の最小公倍式である。

多項式集合$\{F_1, ..., F_r\}$からグレブナー基底を構成する方法としてブーフバーガーの算法が知られている。すなわち、

(i) $\Gamma \leftarrow \{F_1, ..., F_r\}$

(ii) $F_i \longrightarrow\!\!\!\rightarrow_{\Gamma - \{F_i\}} 0$ ならば$\Gamma \leftarrow \Gamma - \{F_i\}$, $i \in \{1, ..., r\}$。最終的に$\Gamma$の要素数が1であるならば$\Gamma$を返す。そうでなければ$\Gamma' \leftarrow \Gamma$。

(iii) $\Gamma$の要素のあらゆる対$(F_i, F_j)$に対してS-多項式$S_{ij}$を計算し、$\Gamma'$によってM-既約された結果を$\tilde{S}_{ij}$とする。もし$\tilde{S}_{ij} \neq 0$ならば$\tilde{S}_{ij}$を$\Gamma'$に追加する。

(iv) もし$\Gamma = \Gamma'$ならば$\Gamma$はグレブナー基底となっている。そうでなければ$\Gamma \leftarrow \Gamma'$とし、(iii)へ行く。

上記アルゴリズムのステップ (ii)、および(iii)(iv)が並列化可能であり、以下に説明する。

ステップ (ii) の並列化を考える。多項式集合$\Gamma = \{F_1, ..., F_r\}$が、$F_i \succ F_j, i < j$のように頭項の優先順位順に並んでいるとする。この条件によりステップ(ii)は、多項式$F_i \in \Gamma$を多項式集合$\Gamma - \{F_1, ..., F_i\}$でM-簡約し、0になるかどうかを調べればよい。逐次計算では、この作業を1からrまで繰り返さなければならないが、ここではr個のPEを用いてM-簡約を並列的に行なう。すなわちpn番目のPEは、$F_{pn}$を$\{F_{pn+1}, ..., F_r\}$でM-簡約を行なう。そのアルゴリズムは、

アルゴリズム：PeStepII($\Gamma$)

入力：$\Gamma = \{F_1, ..., F_r\}$、$F_i \succ F_j$、$i < j$

出力：多項式集合Γ

```
Polynomial F ;

if F_pn ⟶→_{Γ-{F_1,...,F_pn}} 0 then F ← NIL
else F ← F_pn ;
return BottomUp(F).
```

関数BottomUpは、引数として与えられた多項式(あるいは多項式集合)を要素とする多項式集合を生成する。生成された多項式集合の多項式は、頭項の優先順位の高い順に並んでいるとする。関数BottomUpに関する新しい定義は以下のアルゴリズムにも適用される。

次にステップ(iii)、(iv)の並列化を考える。グレブナー基底を求める場合、ある条件を満たすまで前回の計算結果をフィードバックし、繰り返し計算を行なわなければならない。このように、後になるに従い要素数が増加するので行列式計算のように必要となるPE数を最初に把握することができない。従って、十分に多いPE数で各PE毎に多項式集合Γの全ての対を割り当て、S-多項式およびそのM-簡約を並列に計算する。各PEからの値から構成される多項式集合がNILの場合にはΓを解とし、それ以外である場合には、求められた集合をΓに追加し、再度計算を行なう。アルゴリズムで表すと以下のようになる。

アルゴリズム：GroebnerBasis(Γ)
入力：$\Gamma = \{F_1, ..., F_r\}$、$F_i \succ F_j$、$i < j$
出力：多項式集合Γ

```
ArrayOfInteger Pair[2] ;
Polynomial F ← NIL ;
PolynomialSet Γ' ;

Pair ← MakeSPolyPair(length(Γ), pn) ;
if Pair is not NIL then
 F ← Mreduction(SPoly(F_Pair[1], F_Pair[2]),Γ) ;

Γ'← GetRegisterData(1,BottomUp(F)) ;
if Γ' is NIL then reutrn Γ
else return GroebnerBasis(Γ + Γ').
```

上記アルゴリズムに新しくlength、MakeSPolyPair、SPoly、Mreductionl、GetRegisterDataの5つの関数が登場する。lengthは、引数として与えられた集合の要素数を返す関数であり、SPoly、Mreductionはそれぞれ S-多項式、M-既約された式を返す関数である。GetRegisterData($arg_1$,$arg_2$)は、$arg_1$と等しいpnを持つPEの$arg_2$で指定された値(この場合は多項式集合)を取り出すものである。MakeSPolyPair($arg_1$,$arg_2$)は、(i,j)、$i < j$、$i \in \{1, \ldots, arg_1\}$で表される全ての対のいずれかを$arg_2$で与えられる数値により返す。もし与えられた$arg_2$が大きく、相当するペアが存在しない場合にはNILが返される。また、関数MakeSPolyPairは、アルゴリズムGroebnerBasis自身の再帰によって何回も呼び出される可能性があるが、なんらかの方法で記憶しており同じ組み合わせは再度生成されないとする。アルゴリズムGroebnerBasisでは、十分に多いPEを用意し計算を行なう必要があるが、SM-1の用意しているPE数は十分であると考えられる。

以上のGroebnerBasisによって求められた多項式集合は、集合内の多項式どうしにより M-簡約が可能であるので、求められた多項式集合Γを以下のアルゴリズムにより M-簡約する。

アルゴリズム：MRed(Γ)
入力：$\Gamma = \{F_1, ..., F_r\}$、$F_i \succ F_j$、$i > j$
出力：グレブナー基底Γ

```
Polynomial F ;

F ← Mreduction(F_pn,Γ) ;
return BottomUp(F).
```

アルゴリズムMRedは、一回行なっただけではM-既約化されない場合があるので、入力した多項式集合と出力された多項式集合が等しくなるまで繰り返し行なう必要がある。

グレブナー基底は、以上のような3つのアルゴリズムで順次計算することにより求めることができる。

# 4 並列アルゴリズムの実現と実行例

前節で提案したアルゴリズムは、必要とするPE数が必ず存在するという前提の元に成り立っている。しかし、SM-1のPE数は1024個という制限があり、そのまま前節のアルゴリズムを適用することはできない。以下で行列式計算アルゴリズムを例にとり、TUPLEによるSM-1上への実現方法とその実行例を紹介する。

## 4.1 並列アルゴリズムの実現

前節で提案したアルゴリズムは、PE数の制限を考慮していないため、限定された問題にしか適用できない。例えば7×7の行列の場合には5040個のPEを必要とする。3.1の行列式計算アルゴリズムMinorDetによりn×n行列の行列式を求める場合、レベルnまで細分化が進むことになる。そこで、それ以前のレベルiにおいて細分化するために必要なPE数が足りなくなった場合、渡されたi×i小行列の行列式を各PEが逐次アルゴリズムによって求めることにより、有限PE数の問題を解決する。次数iによる逐次アルゴリズムへの移行は、

- ${}_nC_{(i-1)} > 1024$
- $i = 2$

が成立したときにi×i小行列の行列式を逐次アルゴリズムにより求め、以後ボトムアップを開始する。

逆行列計算に関しても条件を、

- $n \times {}_nC_{(i-1)} > 1024$

とすることで実現できる。

## 4.2 数式処理系による実行例

以上により行列式及び逆行列を計算する並列処理系をTUPLEを用いてSM-1上に実現することができた。以下その実行例を示す。

図4.1は4×4行列

$$\begin{pmatrix} v^3+3w+2 & w+5 & w^2+v^2 & 7 \\ w^2+5v+3 & w+v+3 & 16 & w^2-5 \\ v^3+w^2+4w & 7 & 16 & w^3-27 \\ 9 & 5w+8v+7 & 9w-9 & 2w^2+3v^2 \end{pmatrix}$$

の行列式を求めた結果である。

```
KTerm
Skarabe (Kyoto Common Lisp) 2.0

skb[1] > pedet(\(\(v^2+3*w+2,w+5,w^2+v^2,7\),
                \(w^2+5*v+3,w+v+3,16,w^2-5\),
                \(v^3+w^2+4*w,7,16,w^3-27\),
                \(9,5*w+8*v+7,9*w-9,
                  2*w^2+3*v^2\)\)) ;
answer > (-3) * V^8 + ((-3) * W + (-9)) * V^7
       + (3 * W^2 + (-40)) * V^6
       + ((-11) * W^2 + 11 * W + 358) * V^5
       + (6 * W^4 + (-43) * W^3 + (-40) * W^2
          + 75 * W + (-8)) * V^4
       + ((-5) * W^5 + (-39) * W^4 + 70 * W^3
          + 533 * W^2 + 408 * W + (-4011)) * V^3
       + ((-5) * W^6 + (-56) * W^5 + 83 * W^4
          + 238 * W^3 + 1452 * W^2 + (-1231) * W
          + 651) * V^2
       + ((-8) * W^7 + (-19) * W^6 + (-8) * W^5
          + 819 * W^4 + 179 * W^3 + 61 * W^2
          + (-16792) * W + 7762) * V
       + (-5) * W^8 + 5 * W^7 + 21 * W^6
          + 198 * W^5 + 477 * W^4
       + (-512) * W^3 + (-4997) * W^2
       + (-17003) * W + 21436
skb[2] >
```

図4.1 行列式計算の実例

図中のpedetは行列式並列計算関数である。本並列数式処理系では行列は、\(\(a_{11},a_{12}\),\(a_{21},a_{22}\)\)と表わす。

次に逆行列の計算例を挙げる。図4.2は、4×4行列

$$\begin{pmatrix} x+3 & 5 & y & 2 \\ 6 & 3 & 8 & 2x+4 \\ y & 3 & 3 & x^2 \\ 2 & y & 8 & x \end{pmatrix}$$

の逆行列を求めたものである。

多変数多項式行列の逆行列を求める場合、多項式GCDを計算する必要がある。ここでは、多項式GCDを計算するアルゴリズムとして擬剰余を用いた方法を採用している。

## 4.3 並列計算と逐次計算の比較

ここでは行列式計算における並列計算、逐次計算の実行時間の比較を行なう。並列計算、逐次計算とも行列式を計算するのにラプラスの展開法を用いた。並列計算は並列計算機SM-1上の本数式処理系で行い、逐次計算はSun Sparc Station上のKCLにより行った。

図4.3は、縦軸に並列計算実行時間を逐次計算

```
KTerm
>Skarabe (Kyoto Common Lisp) 2.0

skb[2] > inverse(\(\(x+3,5,y,2\),\(6,3,8,2*x+4\),
                  \(y,3,3,x^2\),\(2,y,8,x\)\)) ;
answer >
mat(((((8 * Y + (-24)) * X^2 + ((-6) * Y + 33) * X
      + (-12) * Y + 96) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
      + (-16) * Y^2 + (-136) * Y + 180)) ,
    ((((-1) * Y^2 + 40) * X^2 + (3 * Y + (-15)) * X
     + 6 * Y + (-48)) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
      + (-16) * Y^2 + (-136) * Y + 180)) ,
    (((2 * Y^2 + (-3) * Y + (-40)) * X + 4 * Y^2
     + (-16) * Y + (-112)) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
      + (-16) * Y^2 + (-136) * Y + 180)) ,
    (((3 * Y + (-40)) * X^2 + ((-6) * Y + 30) * X
     + (-12) * Y + 90) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
      + (-16) * Y^2 + (-136) * Y + 180)) ,
    (((32 * X^2 + ((-8) * Y + (-6)) * X + (-32) * Y
     + 24) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
      + (-16) * Y^2 + (-136) * Y + 180)) ,
    (((-8) * X^3 + (2 * Y + (-21)) * X^2
               + ((-1) * Y^2 + 9) * X + 16 * Y
     + (-12)) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
      + (-16) * Y^2 + (-136) * Y + 180)) ,
    ((8 * X^2 + (2 * Y + 56) * X + (-8) * Y + 32) /
     ((8 * Y + (-24)) * X^3 + ((-6) * Y^2 + 24 * Y
               + 121) * X^2 + (2 * Y^3 + (-3) * Y^2
               + (-64) * Y + 165) * X + 4 * Y^3
```

図 4.2 逆行列計算の実例

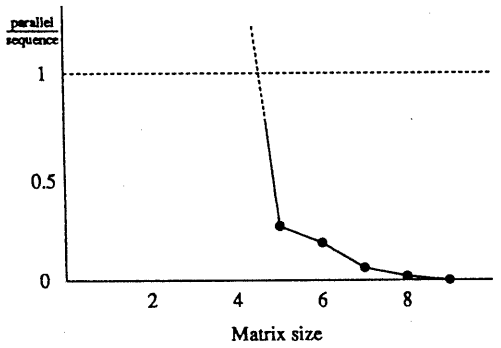


図 4.3 並列、逐次計算実行時間の比率

実行時間で割った値、横軸に行列の大きさを取ったものである。図より、行列が大きくなるほど逐次計算の場合に比べて並列の計算時間が短くなっているのがわかる。表 4.1 は行列式を求めるときに使用した PE の数をまとめたものである。

表 4.1 使用 PE 数

| 行列の大きさ | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|
| PE 数 | 60 | 360 | 840 | 336 | 504 |

表によれば 7 × 7 行列のとき使用 PE 数が極大になっている。なお、グレブナー基底の実行例は、紙数の都合により割愛する。

## 5 おわりに

並列数式処理系の開発を SIMD 型並列計算機 SM-1、および並列 Lisp で行なった。数式処理系により、多項式に関する基本的な演算やいくつかの並列化した関数を利用できる。

さらに並列化することにより実行時間の短縮が考えられるアルゴリズムやシステム制御理論へのさらなる応用を検討中である。

## 参考文献


[1] 松田, 湯浅:SIMD 型超並列計算機 SM-1(仮称)の概要, 計算機アーキテクチャ,1984.

[2] Yuasa,T. :TUPLE:An Extended Common Lisp for Massively Parallel SIMD Architecture,In Proc. of the DPRI Symposium,1992.

[3] 岩間:PRAM 上の並列アルゴリズム, 情報処理 vol.33 No.9, ,1992.

[4] 佐々木:情報処理, 社団法人 情報処理学会, オーム出版,1980.

[5] Buchberger,B.:Gröbner Basis:An Algorithmic Method in Polynomial Ideal Theory,in Multidimensional Systems Theory(ed. Bose,N.K.), D.Reidel Publ.Comp.,1985.