

超並列数式処理系の並列 LISP プログラム作成技法

清野 博之 斉藤 制海 湯浅 太一

豊橋技術科学大学 知識情報工学系

近年並列数式処理系の研究、開発が種々進められている。筆者らも SIMD 型超並列計算機上で超並列数式処理系の開発を進めている。本稿では、超並列数式処理系の処理速度の改善のために 2 つの技法について報告する。第 1 は並列処理において避けられない PE 間通信をセーブするには、SIMD 型計算機の PE 間通信アーキテクチャに依存したアルゴリズムを選択する必要があることを指摘する。第 2 は、並列処理速度を改良するための並列 LISP プログラムの作成技法について報告する。

Programming Technic of Parallel LISP for Massively Parallel Computer Algebra

SEINO Hiroyuki SAITO Osami YUASA Taiichi

Toyohashi University of Technology

Toyohashi 441 Japan

Recently development of parallel computer algebra system has been investigated from various aspects. The authors are also engaged in developing a massively parallel computer algebra system with SIMD parallel computer. This paper proposes two technics to improve the computing speed. The first is a choosing suitable algorithms for saving communication time between processing elements which cannot be avoided in parallel computing. The second is a programming technic for parallel LISP programming of computer algebra.

1 はじめに

近年数式処理系の並列化の研究が進められはじめている。一つは複数の既存の数式処理系をネットワークで結合したシステムで、これは並列というより分散処理システムといえる。これに対しパイプライン方式や MasPar 等の SIMD 型超並列計算機による超並列数式処理システムも提案されている。筆者らも SIMD 型超並列計算機 SM-1 による超並列数式処理系を提案し、そのプロトタイプの開発を進めてきた。[1][2] そこでは超並列アーキテクチャ、その上で稼働する超並列処理言語及び超並列アルゴリズムと一貫して超並列化を考えた超並列数式処理系を開発し一定の成果を得てきたが、処理速度は必ずしも充分とはいえない。その原因として多数の演算装置 (PE) 間の通信時間やプログラムの作成技法がボトルネックになっていることが推測できる。

本研究では、2つの視点から上述の問題の解決を図る。一つは、SM-1 の持つ PE 間通信ネットワークの多彩な機能を活用することである。並列処理には PE 間通信は避けられないが PE 間通信に時間がかかりすぎると並列処理の利点が活かせなくなる。ここでは SM-1 の持つ通信機能を考慮したアルゴリズムを採用し、PE 間通信時間の改善を図る。二つめは並列性を考慮したプログラム作成技法について提案する。並列処理においては、全ての PE が同時に処理を行なっているわけではないが、できるだけ多くの PE が処理しているのが望ましい。ここでは多くの PE が並列処理するためのプログラム作成技法を提案する。これらの提案を基に、SM-1 上に超並列数式処理系を実装し、これら 2つの提案の有効性を具体的に検証する。

2 SIMD 型超並列計算機と並列 Lisp

本研究で提案する超並列数式処理系は、超並列計算機 SM-1、及びその上で稼働する超並列拡張版 Kyoto Common Lisp (以下 TUPLE と記す) を

用いて開発している。以下 SM-1 と TUPLE について概要を述べる。[3][4]

SM-1 は、SIMD 型超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。

各 PE は 0~1023 の固有の番号を持つ。以下この PE 固有の番号を pn と表記する。また、 $pn=i$ の PE をあらわすのに PE_i と表記する。PE は自分自身のローカルメモリだけではなく、PE 間通信を利用して他の PE のローカルメモリも参照できる。この PE 間通信のためにいくつかのメモリ参照モードがある。これは SM-1 のアーキテクチャの大きな特徴である。

PE 間通信を行なうために各 PE は互いに接続されたネットワーク構造を持ち、これには、2次元メッシュ、縦横の OR バス、全 PE の OR バス、シャッフルエクスチェンジ結合の 4 種類がある。また FE から PE のローカルメモリの参照も FE-PE 間通信を用いて可能である。

開発言語として使用した TUPLE は、SIMD 型超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 関数はいわゆる一般的な Lisp と同様なものであり、PE 関数は 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。

PE 間通信としては 2次元メッシュによる通信、シャッフルエクスチェンジ結合による通信が実現されている。2次元メッシュによる通信にはその組み合わせも利用して、8 近傍 PE との通信、または pn を指定した通信が実現されている。図 1 は 2次元メッシュ結合の一部分を示している。円は

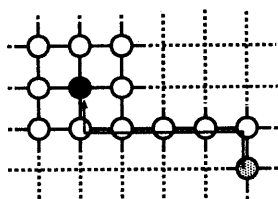


図 1: 2次元メッシュ結合

PE を、線は PE 間をつなぐ結線を意味する。今、黒色の PE が PE 間通信を利用して他の PE と通信を行なうものとする。8 近傍 PE とは 1~2 回の通信を行えばよい。pn により指定した灰色の PE とは何回か 2 次元メッシュを利用した通信を繰り返さなければならない。この時 2 次元メッシュのどの経路を使うかはシステムに依存する。通信を行なう PE が離れると通信時間は大きくなる。

図 2 はシャッフルエクスチェンジ結合を示している。円は PE を、その中の数字は pn をあらわ

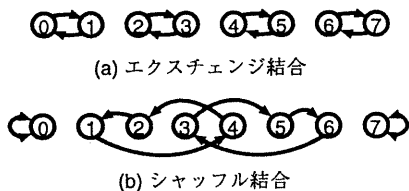


図 2: シャッフルエクスチェンジ結合

す。この通信は図の矢印の向きのメモリ参照を可能にする。例えば、シャッフル結合を利用すれば PE_1 は PE_2 からデータを受け取り、 PE_4 にデータを渡す。ただしこの図は説明のためにトポロジーをそのままに PE の数を 8 に減らしたものである。

各 PE にあるデータに何らかの処理をして 1 つにまとめることをリダクションといい、超並列処理では頻繁に用いる。シャッフルエクスチェンジ結合を利用することで直接結線される PE 間の通信のみを用いて高速に実現できる。

3 並列アルゴリズムと通信時間

筆者らは数式処理の、多項式環上でのシステム制御理論への応用を試み、多変数多項式を要素とする行列の演算や、行列方程式に関する超並列数式処理系の開発を行なっている。ここではそれらの基礎となる行列式の並列計算を取り上げ、並列処理では避けられない PE 間通信とアルゴリズムの関係を検討する。

3.1 ラプラス展開を利用した並列アルゴリズム

行列式の並列計算のアルゴリズムとして次のラプラス展開法を適用する。ラプラス展開法を用いて、 $n \times n$ 行列 A の行列式 $|A|$ を 1 行で展開すると、

$$|A| = \sum_{k=1}^n a_{1k}(-1)^{k+1}|A^{(k1)}|$$

となる。ここで $A^{(ij)}$ は、行列 A から i 行 j 列を除いて得られる $(n-1) \times (n-1)$ 小行列である。

つまり $n \times n$ 行列式は、 n 個の $(n-1) \times (n-1)$ 行列の行列式を計算することにより求められる。この展開法を $(n-1) \times (n-1)$ 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といえる。

高次の行列式をより低次の行列式に分解していく過程をトップダウン、求められた低次の行列式を高次の行列式へ統合していく過程をボトムアップと呼ぶことにする。以下このアルゴリズムを DL アルゴリズムと呼ぶことにする。

DL アルゴリズムを 3 次の場合について説明する。 $A_i = A^{(1i)}$, $A_{ij} = (A^{(1i)})^{(1j)}$ と書けば展開を繰り返すと次のようになる。

$$\begin{aligned} |A| & \text{lev.1} \\ &= a_{11}|A_1| + (-a_{12})|A_2| + a_{13}|A_3| \text{lev.2} \end{aligned}$$

$$\begin{aligned}
&= a_{11}(a_{22}|A_{11}| + (-a_{23})|A_{12}|) \\
&\quad + (-a_{12})(a_{21}|A_{21}| + (-a_{23})|A_{22}|) \\
&\quad + a_{13}(a_{21}|A_{31}| + (-a_{22})|A_{32}|) \quad \text{lev.3}
\end{aligned}$$

このとき、並列計算に必要な PE は $3! = 6$ 個である。6 個の PE がどのような計算をするかを図 3～6 を用いて模式的に示す。図中の白い円はアクティブな PE を、色のついた円はインアクティブな PE をあらわす。また、円内の数字は pn を、矢印は PE 間通信をあらわす。

図 3 にトップダウンの模式図を示す。まず lev.1

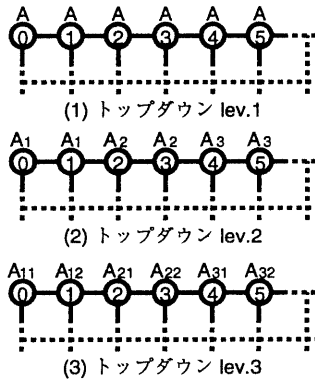


図 3: トップダウン模式図

では FE-PE 間通信により行列 A が $PE_{0\sim5}$ に渡される。この 3 次の行列式を求めるために lev.2 のようにラプラス展開する。ここで、 $|A_i|$ は 2 次の行列式なのでそれぞれ $2! = 2$ 個の PE を使って並列計算できる。したがって、6 個の PE を 2 個ずつのグループに分け、 $PE_{0,1}$ で $|A_1|$ を、 $PE_{2,3}$ で $|A_2|$ を、 $PE_{4,5}$ で $|A_3|$ を計算する。

同様に、lev.3 のようにさらにラプラス展開できる。この時点で各 PE は $|A_{ij}|$ を計算することになるが A_{ij} はスカラであるので展開は終了する。ここまでは、トップダウンの過程である。トップダウンでは最初に FE から PE に行列データを渡す以外には通信を全く必要としない。各 PE は pn のみを手がかりに必要なデータを作成していき、全ての PE がアクティブである。

次にボトムアップであるが、各 PE で作成された低次の行列式の値を利用してもとの行列式 $|A|$ を計算する。

最初に lev.3 まで展開された $|A_{ij}|$ を求める。(図 4(step.1)) これは 1 次であるのでその要素自体が行

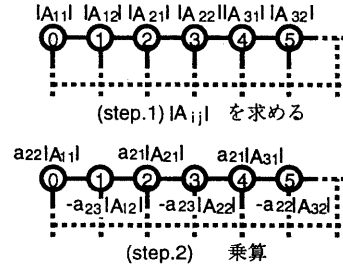


図 4: ボトムアップ lev.3 模式図

列式の値となる。 $PE_{0\sim5}$ はその $|A_{ij}|$ と展開に使った a_{kl} の積を計算結果として持つ。(図 4(step.2)) これらは lev.3 の括弧の中各项になっている。

次に lev.2 の $|A_i|$ を求める。各 PE の計算結

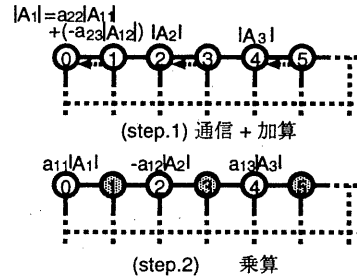


図 5: ボトムアップ lev.2 模式図

果 $a_{kl}|A_{ij}|$ に対応する適切な 2 個を PE 間通信によって 1 つの PE に集めその和をとることで $|A_i|$ を求めることができる。(図 5(step.1)) 例えば、 $PE_{0,1}$ の結果を使えば $|A_1|$ を求められる。それらを、 $PE_{0,2,4}$ に作っておく。このとき、 $PE_{1,3,5}$ はその計算結果を渡した時点で不必要となるのでインアクティブとなる。これら $|A_i|$ と展開に使

た a_{mn} の積を各 PE の新しい計算結果とする。(図 5(step.2))

最後に lev.1 の $|A|$ を求める。これは PE 間通信を利用して $PE_{0,2,4}$ の結果の和をとれば求められる。ただし、ここでは 3 つの多項式の加算を行わなければならない。したがって、通信・加算を 2 回行なう必要がある。(図 6(step1-1~2)) 一般に

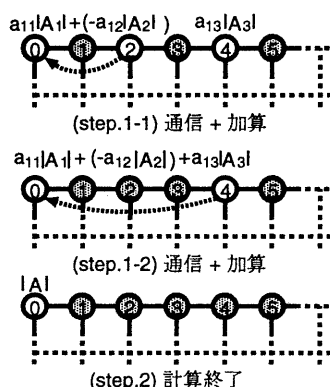


図 6: ボトムアップ lev.1 模式図

は、 n 個の PE にある多項式の和を求める場合には $\lceil \log_2 n \rceil$ 回だけ通信と加算を並列に実行する。

このボトムアップの過程では通信・加算という作業が繰り返される。この時各 PE は厳密に決められた他の PE と通信を行なう必要がある。したがって、通信は 2 次元メッシュネットワークを利用した pn を指定する方法を用いなければならない。

DL アルゴリズムによる超並列行列式計算の関数を SM-1 上に作成し、その処理速度及び通信時間を具体的に評価してみる。図 7 は DL アルゴリズムを並列処理と逐次処理で行ないその計算時間を比にしたグラフである。使用したデータは、数値、2 変数 2 次多項式、2 変数 4 次多項式である。評価に使用した行列の一例として、2 変数 2 次多項式を要素とする 3 次の行列を示す。

$$\begin{pmatrix} y^2 + 5x + 4 & 4xy + y + 1 & 2x^2 + 3x + 5 \\ xy + 6x + 4 & 7x^2 + x + 5 & 2xy + y \\ 7xy + 5y + 9 & 9x^2 + 7y & x^2 + y + 6 \end{pmatrix}$$

図 7 から分かるように並列化することにより高

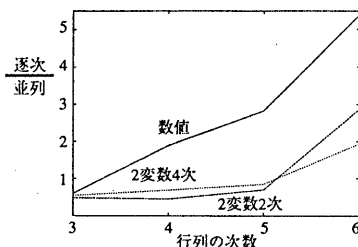


図 7: 逐次/並列実行時間比

速化がなされ、行列の次数が高くなるとその傾向は強くなる。しかし、複雑な多項式を要素とする場合には並列化による性能の向上は大きくない。

図 8 は行列要素を 2 変数 2 次の多項式にしたときの行列式の次数と並列計算時間及び PE 間通信に要する時間のグラフである。

図からも分かるように計算時間のほとんどが PE

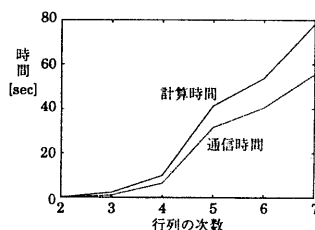


図 8: 2 変数 2 次多項式の計算時間と通信時間

間通信に費やされている。

通信時間が莫大になってしまうのは DL アルゴリズムでは、離れた位置にある指定された PE 間通信が行なわれなければならないからである。TUPLE ではリダクションする場合は通信時間をほとんど必要しないので、これを利用できれば通信時間を大幅に押え込むことができる。しかし、DL アルゴリズムは本質的にリダクションを用いることができないので通信時間を劇的に減少させ

ることはできない。そこで、リダクションを利用できる方法を考える。

3.2 定義式を利用した並列アルゴリズム

いま、ラプラス展開の代わりに行列式の定義式を考える。

$$|A| = \sum_{\text{all}(j_1 j_2 \dots j_n)} \varepsilon(j_1 j_2 \dots j_n) a_{1j_1} a_{2j_2} \dots a_{nj_n}$$

$(j_1 j_2 \dots j_n)$ は $1 \sim n$ までの整数の順列で、 $\varepsilon(j_1 j_2 \dots j_n)$ は順列の符号である。

この行列式の定義によれば、 $n \times n$ 行列式は n 個の行列要素の積であらわされる多項式を $n!$ 個加算することで求められる。この方法を逐次で使えば $n!$ 個の $\varepsilon(j_1 j_2 \dots j_n) a_{1j_1} a_{2j_2} \dots a_{nj_n}$ を求めなくてははいけない。しかし、それらはデータが違うだけで実行する命令は同じであり、SIMD 型超並列計算機で並列計算として実行できる。以下このアルゴリズムを DD アルゴリズムと呼ぶことにする。

この方法を超並列アルゴリズムとして利用するには次のようにする。 $n \times n$ 行列式を求めるのに必要な PE 数は作成される順列の数と等しく $n!$ 個である。 $n!$ 個の PE に FE-PE 間通信を利用して行列 A を渡す。各 PE は p_n から順列 $(j_1 j_2 \dots j_n)$, $1 \leq j_i \leq n, j_p \neq j_q$ を作成する。この時 p_n に固有の順列を生成する関数を定義しておく。こうしてできた順列と A をもとに $\varepsilon(j_1, \dots, j_n) a_{1j_1} a_{2j_2} \dots a_{nj_n}$ を並列計算する。

各 PE の計算結果の総和をとれば $|A|$ となる。この場合 $n!$ 個の多項式の和をとることになるが、多項式は可換環をなすのでその順序は任意である。したがって、この総和は多項式の和を求める関数を使ったリダクションを行えば良いことになり、DL アルゴリズムと比べて通信時間を大幅に改善できる。

図 9 は DL アルゴリズムと DD アルゴリズムの並列計算時間の比のグラフである。

行列の要素が数値から次数の高い多変数多項式までいずれの場合も計算が高速になっている。通信のボトルネックによる計算速度の低下は回避で

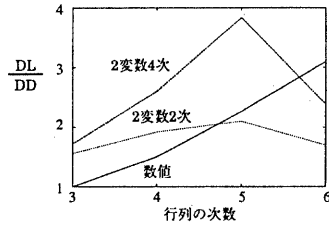


図 9: DL, DD アルゴリズムの計算時間比

きた。グラフを詳しく見ると、数値の場合は行列の次数のグラフが単調増加となるが、多項式の場合、行列の次数が 5 次の時を境に再び高速化の効果が薄れてしまう。

この理由は加算、乗算自体の計算時間が増加することに帰着する。特に、多数の PE が同時に高次の多項式の演算をするような場合に計算速度は急激に低下する。したがって、DD アルゴリズムというよりも加算、乗算関数を SIMD 型を考慮したものに改良する必要がある。

4 プログラム作法と並列効率

3 章では通信時間に着目して高速化を図ってきた。ここではアルゴリズムの実装法、すなわちプログラミング作法により並列関数の並列実行時間をいかにして短縮するかについて説明する。

多項式は以下のようなリストで表現される。

(*POLY* VAR ($E_1 \cdot C_1$) ... ($E_n \cdot C_n$))

VAR: 変数 E_i : べき指数 C_i : 係数

*POLY*はこのリストが多項式であることを示す識別子(タグ)である。 C_i はタグなしの多項式リストまたは数値である。指数 E_i は $E_1 < E_2 < \dots < E_n$ と整列されており、変数 VAR は辞書式順序になっている。例えば、 $x^3 + (y+2)x^2 + 2y^2 + 4$ は次のようになる。

```
(*POLY* X (3 . 1) (2 Y (1 . 1) (0 . 2))
           (0 Y (2 . 2) (0 . 4)))
```

行列式の計算に多用される多項式の加算、乗算関数は条件によって引数を変えて再帰あるいは関数呼出を行なうことが多々ある。しかし、それらはプログラム中の異なる位置に書かれているために同時に実行されることはない。それらを同時に実行できれば高速な処理ができる。すなわち、再帰・関数呼出の多重化を行なう。

例として一変数多項式の加算関数 ADD について説明する。図 10 は二つの多項式 $P = 3X^4 + 5X^3$ 、 $Q = 2X^3 + X^2$

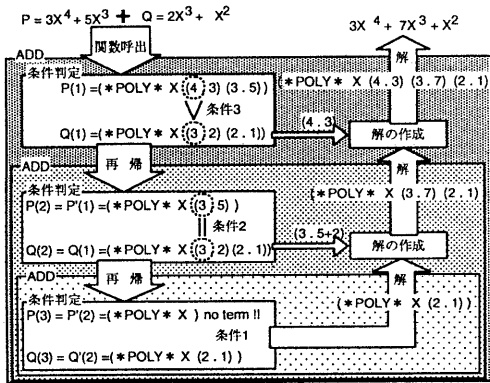


図 10: 加算処理

と $Q = 2X^3 + X^2$ の加算処理を図示したものである。この処理を実現する関数を以下に示す。多項式 P の次数を deg(P)、最高次数の項 (頭項, head term) を ht(P)、頭項の係数 (head coefficient) を hc(P)、 $P - ht(P) = P'$ と書けば、

並列関数: ADD(P, Q)

入力: 多項式 P, Q

出力: 多項式 Ans

```
switch{
  case (一方の項がない): /* 条件 1 */
    Ans=SELECT(P, Q); break;
  case (deg(P) == deg(Q)): /* 条件 2 */
    Ans=MAKEANS((deg(P).hc(P)+hc(Q)),
                  ADD(P', Q'));
    break;
  case (deg(P) > deg(Q)): /* 条件 3 */
    Ans=MAKEANS(ht(P), ADD(P', Q)); break;
  case (deg(P) < deg(Q)): /* 条件 4 */
    Ans=MAKEANS(ht(Q), ADD(P, Q'));};
return(Ans);
```

MAKEANS は一つの項と多項式から新しい多項式を作成する関数であり、図 10 の“解の作成”

の働きをする。SELECT は項が残っている方の多項式を選択する関数である。この関数では分岐中 3 つの場所で再帰が行なわれている。この並列関数を PE_i, PE_j, PE_k が同時に実行中で、それぞれ条件 1、条件 2、条件 3 を満たしているとする。この場合のタイムチャートは図 11 のようになる。矢

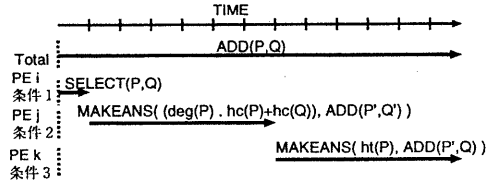


図 11: ADD タイムチャート

印は計算を実行している状態を示す。矢印がない間 PE は条件が成立していないのでインアクティブになり他の実行中の PE が全て計算終了するまで待機状態になる。再帰が 2 回行なわれているがそれらはプログラム中別の場所書いているので同時に実行することはできない。そこで、多重化を考慮してプログラムを書き換える、

並列関数: NEWADD(P, Q)

入力: 多項式 P, Q

出力: 多項式

flag = false;

```
switch{
  case (一方の項がない): /* 条件 1 */
    flag=true; break;
  case (deg(P) == deg(Q)): /* 条件 2 */
    Ht = (deg(P).hc(P)+hc(Q));
    P = P'; Q = Q'; break;
  case (deg(P) > deg(Q)): /* 条件 3 */
    Ht = ht(P);
    P = P'; break;
  case (deg(P) < deg(Q)): /* 条件 4 */
    Ht = ht(Q);
    Q = Q';}
if (flag == true)
  return(SELECT(P, Q));
return(MAKEANS(Ht, NEWADD(P, Q)));
```

こうするとタイムチャートは図 12 のように変る。多重化を考慮したプログラムの作成を行えば並列関数の並列計算量を小さくすることができる。関数呼出についても同様のことがいえる。

ADD と NEWADD を多変数多項式を扱えるように拡張し、実際に多重化をした並列関数を SM-1

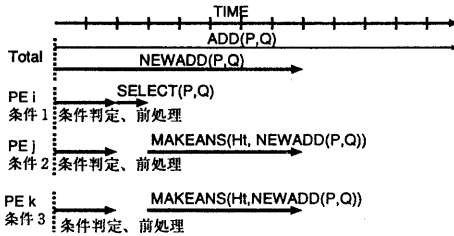


図 12: NEWADD のタイムチャート

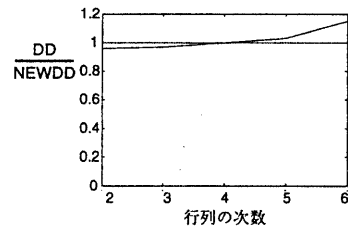


図 14: DD/NEWDD 実行時間比

上に作成した。図 13は 2 変数多項式の次数とアクティブな PE の数を変えて ADD と NEWADD の計算時間の比を求め、グラフにしたものである。

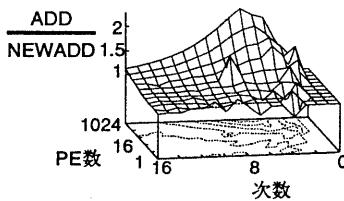


図 13: ADD/NEWADD - PE の数 - 次数

処理速度は多重化することにより改善されているが、DD アルゴリズムで問題となった、多項式が高次で、アクティブな PE 数が多いところでの計算時間は改善されていない。

しかし、この NEWADD は多項式の次数、アクティブな PE 数のほとんどの範囲で ADD よりも高速に計算を行なう。よって、この関数を利用することで行列式計算関数の多少の計算速度の改善ができる。結果を図 14 に示す。

5 おわりに

超並列数式処理系を開発する上で配慮すべきことを、通信、プログラム作法にしばって検証した。

通信についてはできる限り隣接する PE 間で行なえるようなアルゴリズムを採用すべきであり、プログラム作法については再帰や、関数呼出の多重化が効果的であることが明らかになった。

参考文献

- [1] 大野, 高橋, 斎藤, 湯浅: SIMD 型超並列計算機による並列数式処理系, 記号処理 71-3, 1993.11.19
- [2] Saito O., Ono T., Yuasa T., Takahashi T.: Parallel Computer Algebra for n-D Systems by Using SIMD Parallel Computer, Proceedings of the Asian Control Conference Tokyo, July 27-30, 1994
- [3] 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.
- [4] Yuasa, T.: TUPLE: An Extended Common Lisp for Massively Parallel SIMD Architecture, In Proc. of the DPRI Symposium, 1992.
- [5] 佐々木: 情報処理, 社団法人 情報処理学会, オーム出版, 1980.