計算機アーキテクチャ 142－23
ハイパフォーマンス
コンピューティング 85－23
（2001．3．9）

線形回帰演算の並列化手法の応用による DOACROSS ループの並列実行

川端英之† 谷口宏美† 津田孝夫†

本稿では、ループ運搬依存を含むループ（DOACROSS ループ）の並列化に、線形回帰演算の並列化手法を応用する方法について述べる。本手法は、ループのイタレーションの一つ一つを回帰演算の処理単位とみなして線形回帰演算の並列化手法を適用するものである。従来のループ並列化手法とは異なり、ループ全体から並列実行可能な成分を切り分けないため、ループインスタンス内に自然に存在しているデータ局所性が乱されない。アルゴリズム変換により総計算量が増加するものの、粒度は大きく保たれ、計算処理を全てのプロセッサが分担できることから台数効果を得やすい。実験では、細粒度並列処理でしか効果が上がらないと言われる微小ループでも、並列化により数倍程度の高速化が可能であることが確かめられた。

Bulk Recurrent Parallel Execution of DOACROSS Loops

HIDEYUKI KAWABATA,† HIROMI TANIGUCHI† and TAKAO TSUDA†

In this paper, we show a method of parallel execution of DOACROSS loops utilizing parallel execution techniques for linear recurrences. The method is able to make every processor in a system participate essential computation unlike such methods as normal DOACROSS execution and pipelining, which have *critical paths* in computation and cannot necessarily utilize all of the processors in a system. A drawback of the method shown in this paper is the increase of the amount of computation in order to cut serial chains of dependences. Experimental results show the method is feasible enough and efficient on both shared-memory systems and distributed-machine environments.

1. はじめに

プログラムの並列化は、様々な粒度において行なわれるが、自動並列化の対象として最も一般的なのはループの並列実行である。プログラム中のループは、その実行に費やされる時間がプログラムの総実行時間の大部分を占める場合が多く、ループを並列化の対象として重点的に最適化しようとするのは全く妥当なことである。しかしながら、ループは、その繰返されるループボディの実行インスタンス各々が他のイタレーションに依存している、すなわちループ運搬依存がある場合が多い。ループ運搬依存のあるループは、ループレベル並列化の際に、依存関係を保つための策を要する。

従来、ループ運搬依存のあるループに対する並列化においては、同期命令を挿入することによって順序関係を保持させることで並列実行を行なわせたり、ループ運搬依存の影響を受けない部分と本質的にループ運搬依存に関わる部分とを分離して（ループ分割）、DOALL 並列実行可能な部分のみを並列実行させることが一般的であった。しかし、前者の場合は、同期命令が頻繁に実行されるため、高速な同期ファシリティが必要とされるし、後者の場合にはキャッシュとの親和性が悪い。

これに対し、本稿では、ループインスタンスレベルの並列処理に線形回帰演算の並列化方式を応用するループ並列化手法を提案する。本手法では、ループ運搬依存のあるループにおける計算処理をバルク回帰と呼び、イタレーション全体を線形回帰演算における一演算単位とみなす。分割統治手法による並列化により、同期命令の数を抑え、粒度を高くし、キャッシュの利用効率を高めた並列実行が実現できる。

本手法は基本的には演算内容の把握によるアルゴリズム変換であるので、適用範囲と総計算量の増加が問題となるが、計算処理を複数のプロセッサでほぼ均等に分割できるため、原理的には並列性に上限がない。実験では、単純な線形回帰演算程度であっても、普及型の小規模並列計算機において充分に台数効果が得られることが分かった。

† 広島市立大学 情報科学部
Faculty of Information Sciences, Hiroshima City University

2. 依存のあるループの並列化 — 既存の手法

2.1 計算順序を変えない方法

ループ運搬依存のあるループの並列化手法としては、DOACROSS 法、Pipeline 法、DOALL の切り出し、が代表的なものとして挙げられる。

DOACROSS 法及び Pipeline 法は、並列実行方式（プロセッサへの処理のマッピング方法）には違いはあるが、いずれも、基本的には計算順序を変更することなく処理の途中で必要に応じて同期をとりながら計算を進める方式であるという点では類似のものである。

DOALL の切り出しと呼ぶ方法は、ループ運搬依存に絡む部位をループ分割によって切り出して逐次的に実行し、その他の部分を DOALL ループとして並列に実行する方式である。

ここに挙げた三者は、いずれも、ループ中で逐次的な処理を必要とする部分をそのまま残す方法であるため、計算処理を含むクリティカルパスが無視できず、並列度の上限が抑えられやすい。

2.2 アルゴリズム変換による並列化

前節の方法と対比されるものとして、ループに対してアルゴリズム変換を適用することにより並列性を高める方法が挙げられる。本稿で述べるループ並列化手法は、その代表例である線形回帰演算の並列化手法に基づくものである。一般には総演算量の増加を伴う並列化であり、全ての環境で高速化が達成できるとは限らないという問題はある。

典型的な回帰演算を高速に実行するためのハードウェアに関する研究もある。[4] その基本的なアイデアは、回帰演算の並列化のためのアルゴリズム変換に起因する演算量の増加分を、演算パイプラインや長命令語の空きスロットへ押し込める試みにある。またそれらハードウェアを有効に利用するためのコンパイル技術も研究されている。[3]

3. 線形回帰演算の並列化手法

本稿で提案するループ並列化手法は、線形回帰演算の並列化手法をもとにしたものである。本章では準備として線形回帰演算の並列化手法の概要を述べる。

3.1 線形回帰演算の並列化手法の基本原理

線形回帰演算は、以下のように表現できる。

$$a_i = p_i \cdot a_{i-1} + q_i \quad (i = 1, 2, 3, \ldots, N) \qquad (1)$$

このままでは、別々の i に対する a_i の値を求める計算を独立に（同時に）実行することはできない。しかし、

$$\begin{aligned} a_i &= p_i \cdot p_{i-1} \cdot a_{i-2} + p_i \cdot q_{i-1} + q_i \\ &\cdots \\ &= \underline{p_i \cdot p_{i-1} \cdot \ldots \cdot p_{j+1}} \cdot a_j \\ &\quad \underline{+ p_i \cdot \ldots \cdot p_{j+2} \cdot q_{j+1} + \cdots + p_i \cdot q_{i-1} + q_i} \end{aligned}$$

と式 (1) を変形すると、

$$\begin{cases} P_i^{(j)} \equiv p_i \cdot \ldots \cdot p_{j+1} = \displaystyle\prod_{k=i}^{j+1} p_k \\ Q_i^{(j)} \equiv p_i \cdot \ldots \cdot p_{j+2} \cdot q_{j+1} + \ldots + p_i \cdot q_{i-1} + q_i \\ \qquad = \displaystyle\sum_{k=i}^{j+1} \{ (\prod_{l=i}^{k-1} p_l) \cdot q_k \} \end{cases}$$

を用いて、以下のように記述できる。

$$a_i = P_i^{(j)} \cdot a_j + Q_i^{(j)} \quad (0 \le j < i \le N) \qquad (2)$$

式 (2) における $P_i^{(j)}$ 及び $Q_i^{(j)}$ は、i について独立に算出が可能であるので、任意の $j(<i)$ について a_j の値さえ判れば a_i の値を算出できる。

なお、$P_i^{(j)}$ 及び $Q_i^{(j)}$ は、次の式 (3) あるいは式 (4) の関係式から求めることができる。

$$\begin{cases} P_i^{(j)} = P_i^{(j+1)} \cdot p_{j+1} \\ Q_i^{(j)} = P_i^{(j+1)} \cdot q_{j+1} + Q_i^{(j+1)} \end{cases} \qquad (3)$$

$$\begin{cases} P_i^{(j)} = p_i \cdot P_{i-1}^{(j)} \\ Q_i^{(j)} = p_i \cdot Q_{i-1}^{(j)} + q_i \end{cases} \qquad (4)$$

3.2 並列実行方式

式 (2) の $P_i^{(j)}$ 及び $Q_i^{(j)}$ の算出の際の j と i の決め方や、$P_i^{(j)}$ 及び $Q_i^{(j)}$ 自体の求め方によって、cyclic elimination や cyclic reduction など、いくつかの手法に分類できる。

ここで、回帰演算の並列実行方式の例として、単純な分割統治による方法を示す。本稿で提案する手法の実測評価の際にも本並列実行方式を用いる。プロセッサ（以下では PE と表現）数が p であるとし、回帰演算によって新規に値を求める要素数を N とする。また便宜上 $M \equiv N/(p+1)$ と表す。このとき、式 (2) を用いた回帰演算の並列実行は、表 1 に示す PE–データのマッピングを仮定して、以下の 3 ステップで行なうことができる。

step1 表 1 の step1 欄に示す要素の値を計算：
- PE 0 は $a_1 \sim a_{M-1}$ を逐次的に計算する。
- PE k $(k = 1, \ldots, p-1)$ は $P_{M\cdot(k+1)-1}^{((M-1)\cdot k)}$ と $Q_{M\cdot(k+1)-1}^{((M-1)\cdot k)}$ を計算する。

step2 PE 間で値の授受：
- PE 0 は a_{M-1} の値を PE 1 に送信。
- PE k $(k = 1, \ldots, p-2)$ は PE $k-1$ から $a_{(M-1)\cdot k-1}$ の値を受け取り、この値と $P_{M\cdot(k+1)-1}^{((M-1)\cdot k-1)}$ と $Q_{M\cdot(k+1)-1}^{((M-1)\cdot k-1)}$ の値から式 (2) を用いて $a_{M\cdot(k+1)-1}$ の値を計算し、それを PE $k+1$ に送信。
- PE $p-1$ は、$a_{M\cdot p-1}$ の値を PE 0 に伝える。

step3 表 1 の step3 欄に示す要素の値を計算：
PE $0, 1, 2, \ldots, p-1$ は、それぞれが保持してい

る $a_{M\cdot p-1}, a_{M-1}, a_{M\cdot 2-1}, ..., a_{M\cdot(p-1)-1}$ の値を用いて、各々逐次的に表1の step3 欄に示す要素の値を計算する。

表1 各 PE への計算のマッピング †

PE	step1	step3
0	$a_1 \sim a_{M-1}$	$a_{M\cdot p} \sim a_N$
1	$P^{(M-1)}_{M\cdot 2-1}, Q^{(M-1)}_{M\cdot 2-1}$	$a_M \sim a_{M\cdot 2-1}$
2	$P^{(M\cdot 2-1)}_{M\cdot 3-1}, Q^{(M\cdot 2-1)}_{M\cdot 3-1}$	$a_{M\cdot 2} \sim a_{M\cdot 3-1}$
...	...	...
$p-1$	$P^{(M\cdot(p-1)-1)}_{M\cdot p-1}, Q^{(M\cdot(p-1)-1)}_{M\cdot p-1}$	$a_{M\cdot(p-1)} \sim a_{M\cdot p-1}$

†: $M \equiv N/(p+1)$

図1に、この方法による回帰演算の並列実行の様子を示す。図中、縦軸及び横軸はそれぞれ PE の番号及び時間の経過に対応する。太実線矢印及び太点線矢印は、それぞれ a_i の計算及び P^j_i 及び Q^j_i の計算に費やされる時間を示し、細点線矢印は、PE 間通信に要する時間（待ちを含む）を示している。通信は全体の計算の中で一度だけ行なわれているが、全 PE に渡って逐次的に行なわれるため、PE 数が増えると通信オーバヘッドがボトルネックとなることが分かる。

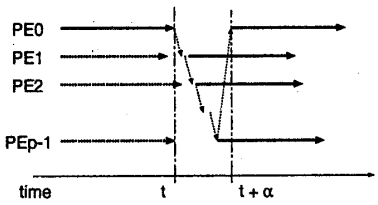


図1 回帰演算の並列実行の様子

ここで、この並列実行アルゴリズムで必要とされる実行時間の見積りを考える。この並列実行アルゴリズムでは、step1 及び step3 がいずれも M 回繰り返されるループである。浮動小数点演算（以下 flop と書く）一回に要する時間を n、浮動小数点データの一参照に要する時間を d と置くと、step1 のループの一インスタンスにおいて PE 0 では、load/store 命令 4 回、flop が 2 回行なわれるとみることができるが、次のインスタンスでの load のうちの一つは短時間に行なわれると考えて、load/store 命令 3 回、flop が 2 回であるとみなすことにし、ループ全体では $(2n+3d)\cdot M$ だけ時間がかかるとみなす。同様に、step1 では他の PE は $(3n+2d)\cdot M$、step3 ではいずれの PE も $(2n+3d)\cdot M$ だけの時間を必要とするとみなすことができる。PE 間での通信（共有メモリにおける同期）にかかる時間（図1中の α であり、「待ち」を含む）を $\alpha(p)$ であるとし、$M = N/(p+1)$ を代入すると、結局、各 PE の実行時間は、

- PE 0: $(6n+4d)\cdot(N/(p+1)) + \alpha(p)$
- その他の PE: $(5n+5d)\cdot(N/(p+1)) + \alpha(p)$

である。この仮定では、$n > d$ のときには step1 の終了は PE 0 が最も遅いことになり、PE 1 には PE 0 からのデータの受信待ち状態が生じることになる。

一方式 (1) による逐次実行で必要となる実行時間は $(3n+2d)\cdot N$ である。$d = \beta\cdot n$ と仮定し、$\alpha(p)$ が微小であり N が充分大きいとすれば、高速化率 S は、

- $n > d$ のとき:

$$S = \frac{(3n+2d)\cdot N}{(6n+4d)\cdot(N/(p+1)) + \alpha(p)} \simeq \frac{p+1}{2}$$

- $n < d$ のとき:

$$S = \frac{(3n+2d)\cdot N}{(5n+5d)\cdot(N/(p+1)) + \alpha(p)} \simeq \frac{3+2\beta}{5+5\beta}\cdot(p+1)$$

つまり、$\alpha(p)$ を無視すれば S は p にほぼ比例（比例定数 2/5 ～ 3/5）し、PE 数が 2 ～ 3 程度以上の環境では逐次実行よりも高速化できると言える。実際は $\alpha(p)$ は p にほぼ比例して増加するので最大の高速化率には限界がある。p が非常に大きい値のときや N が小さい値の時は、最大の高速化率は低く抑えられる。

4. バルク回帰とその並列化手法

本章では、ループ運搬依存のあるループに対する新たな並列化手法について述べる。本手法では、ループのイタレーションを回帰演算の要素演算とみなして前章で述べた線形回帰演算の並列化手法を適用する。このとき、単純な回帰演算でない場合に、並列化後の計算量をいかに削減するかが大きな問題となる。これに対する最適化手法及びプログラム変換の自動化方法についてまとめる。

4.1 バルク回帰とその検出

ループ内で行なわれる計算が式 (5) に示されるような回帰演算とみなすことができるとき、そのループは**バルク回帰**であると呼ぶことにする。

$$\vec{v}_i = f(\vec{v}_{i-1}, \vec{v}_{i-2}, ..., \vec{v}_{i-m}) \qquad (5)$$

ここで $\vec{v}_i$ は複数の変数から成るベクトルである。本稿では、簡単のため、式 (6) のように表される線形一次回帰の場合に限定して話を進める。

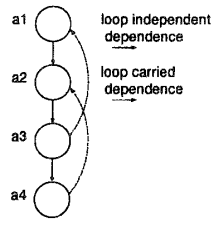
$$\vec{v}_i = \mathbf{A}_i \cdot \vec{v}_{i-1} + \vec{b}_i \qquad (6)$$

式 (6) で示されるバルク回帰であるループは、3.2 節で示した線形回帰演算の並列実行方式を適用してそのまま並列化することができる。なお m 次回帰を一次回帰に帰着させる方法は知られている。[1] また、非線形な回帰演算については、媒介数列の利用により、線形に帰着できる場合もある。[2]

ここで、図 2(a) に示すループを例に用いつつ、並列化手順を説明する。図 2(a) のループボディを構成する四つの代入文の間にはループ運搬依存が存在している。図 2(b) はその様子を示す依存グラフである。

```
for (i = 1; i <= N; i++) {
    a_i = x_i · c_{i-1};    /* a1 */
    b_i = a_i + d_{i-1};    /* a2 */
    c_i = b_i · y_i + z_i;  /* a3 */
    d_i = c_i · w_i;        /* a4 */
}
```

(a) ループ



(b) 依存グラフ

図 2 ループ運搬依存のあるループ

図 2(a) のループボディの四つの式は、以下のように書き換えることができる。

$$\begin{cases} a_i &= x_i \cdot c_{i-1} \\ b_i &= x_i \cdot c_{i-1} + d_{i-1} \\ c_i &= x_i \cdot y_i \cdot c_{i-1} + y_i \cdot d_{i-1} + z_i \\ d_i &= x_i \cdot y_i \cdot w_i \cdot c_{i-1} + y_i \cdot w_i \cdot d_{i-1} + z_i \cdot w_i \end{cases} \quad (7)$$

ここで、

$$\vec{a}_i \equiv \begin{pmatrix} a_i \\ b_i \\ c_i \\ d_i \end{pmatrix}, \mathbf{P}_i \equiv \begin{pmatrix} 0 & 0 & x_i & 0 \\ 0 & 0 & x_i & 1 \\ 0 & 0 & x_i y_i & y_i \\ 0 & 0 & x_i y_i w_i & y_i w_i \end{pmatrix}, \vec{q}_i \equiv \begin{pmatrix} 0 \\ 0 \\ z_i \\ z_i w_i \end{pmatrix}$$

と置くと、ループボディの式 (7) は

$$\vec{a}_i = \mathbf{P}_i \vec{a}_{i-1} + \vec{q}_i \quad (8)$$

と表せる。式 (8) は式 (6) であり、式 (1) と同様、一次回帰演算に他ならない。すなわち、図 2(a) のループは、ループボディ全体を一つの演算単位とする線形回帰演算とみなした並列化が可能であることが分かる。

4.2 最適化

前節で検出されたバルク回帰は、ループの並列化の可能性を示すものではあるが、3.2 節で示した並列実行方式をそのまま適用すると、逐次処理よりも大幅に計算量が増加してしまい、高速化が望めない。ここでは、計算量の削減のための手順について述べる。

3.2 節で示した並列アルゴリズムでは、step1 (PE 0 のみ) 及び step3 (全 PE) にて、各 PE は逐次的に各自に割り当てられた計算を行なう。この処理の中ではコードに並列性は不必要であるので、式 (7) に示される計算は不要で、図 2(a) のループボディをそのまま用いれば良い。

また、式 (8) における $\vec{a}_i$ の要素を全て独立に計算できるようにするための「下準備」は不要であることから、step1 における PE $1 \sim p-1$ の計算量を減らすことができる。$\vec{a}_{i-1}$ の各要素の値を用いて $\vec{a}_i$ の各要素の値を求めることを考えると、まず、a_{i-1} 及び b_{i-1} の値は不要である (式 (7) の右辺で引用されていない)。また d_{i-1} は c_{i-1} の値から直接計算できることが図 2(a) のループボディ中の代入文/*a4*/から分かる ($d_{i-1} = c_{i-1} \cdot w_{i-1}$)。結果として、step1 における PE $1 \sim p-1$ の処理は、式 (7) を変形して得られる式 (9) による変数 c_i についての回帰演算の並列化を補助すればよい。

$$c_i = (x_i + w_{i-1}) \cdot y_i \cdot c_{i-1} + z_i \quad (9)$$

ここで述べた最適化適用後の並列実行コードを図 3 に示す。図中の *send* や *receive* は、PE 間のデータ授受のための命令を意味し、主記憶を共有する計算機であればロック変数を用いた同期で実現できるし、疎結合マシンであれば送受信ライブラリコールを当てはめればよい。図 3 の通り、前章で用いたパラメータを用いると、各 PE の実行時間は、

- PE 0: $(10n + 16d) \cdot (N/(p+1)) + \alpha(p)$
- その他の PE:$(9n + 13d) \cdot (N/(p+1)) + \alpha(p)$

となり、step1 の処理は PE 0 が最も時間がかかるとみることができる。一方、逐次実行時間は $(5n+8d) \cdot N$ であり、高速化率 S は以下のようになる。

$$S = \frac{(5n+8d) \cdot N}{(10n+16d) \cdot (N/(p+1)) + \alpha(p)} \simeq \frac{p+1}{2}$$

最適化前と比較すると大幅に計算量が削減され、並列化による高速化が充分期待できることが分かる。

```
for (i=1;
     i<=M;
     i++) {
 a_i=x_i c_{i-1};
 b_i=a_i + d_{i-1};
 c_i=b_i y_i + z_i;
 d_i=c_i w_i;
}

send c_i to PE 1

receive c_i
 from PE p - 1

for (i=M·p;
     i<=N;
     i++) {
 a_i=x_i c_{i-1};
 b_i=a_i + d_{i-1};
 c_i=b_i y_i + z_i;
 d_i=c_i w_i;
}
```

PE 0

```
P=p_M;
Q=q_M;
for (i=M;
     i<= M · 2;
     i++) {
 P=p_i·P;
 Q=p_i·Q+q_i;
}

receive c_i
 from PE 0

c_i=P·c_i+Q;

send c_i to PE 2

for (i=M;
     i<= M · 2;
     i++) {
 a_i=x_i c_{i-1};
 b_i=a_i + d_{i-1};
 c_i=b_i y_i + z_i;
 d_i=c_i w_i;
}
```

PE 1

```
P=p_{M·2};
Q=q_{M·2};
for (i=M·2;
     i<= N · 3;
     i++) {
 P=p_i·P;
 Q=p_i·Q+q_i;
}

receive c_i
 from PE p - 2

c_i=P·c_i+Q;

send c_i to PE 0

for (i=M·2;
     i<= N · 3;
     i++) {
 a_i=x_i c_{i-1};
 b_i=a_i + d_{i-1};
 c_i=b_i y_i + z_i;
 d_i=c_i w_i;
}
```

PE $p-1$

(図中、 $p_i \equiv (x_i + w_{i-1}) \cdot y_i$, $q_i \equiv z_i$)

図 3 最適化適用後の並列実行コード

4.3 バルク回帰の並列化における自動最適化

前節で述べた最適化を機械的に行なう方法について、図 2(a) のループを例に用いて説明する。図 2(b) の依存グラフに対して、

- 各ノードの出次数を 1 とする (データ依存エッジが一本しか出ないようにする)
- 一つのノードから複数のデータ依存エッジが出ていた場合には、仮想的なノードを置いてデータ依存エッジを「枝分かれ」させる。ここで設ける仮想的なノードの出次数は限定しない。

という書き換えを行ない、繰返し空間方向に展開すると、図4が得られる。ここで、図4の依存グラフに対

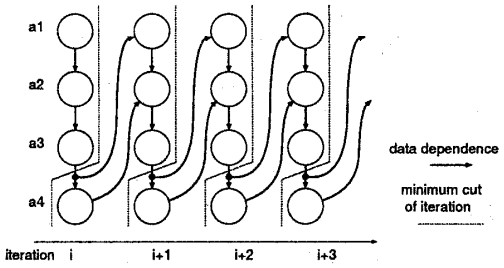


図4 繰返し空間における依存グラフ

してイタレーション方向に等間隔にグラフを分割することを考える。このとき、「カット」に含まれるエッジ数が最小となるように分割を行なうと、例えば図4中の破線で示す分割が得られる（一意的ではない）。この分割に従う形で図2(a)のループの各イタレーションを別々のPEに割り付けて並列実行するとすれば、イタレーション間で授受が必要となるデータ量は最小に抑えられる。

得られた分割を処理単位として3.2節で示した回帰演算の並列化を適用すれば、図3と同等なコードが得られる。図4より、ノードs3において定義される変数c_iの値のイタレーション間の授受のみを考慮すればよいことが分かるので、ノードs3に対応するコードを式(9)のように書き換えて、それに応じたコードを出力すればよい。

5. 広義線形バルク回帰

線形バルク回帰は、式(8)によって表される処理を指しており、式(1)が式(2)のように変形できることから、並列化可能であるということができる。しかし、並列化可能であることの条件は、緩和できる。

実数上で定義される二項演算$\otimes$及び$\oplus$による回帰演算(10)を考える。

$$a_i = p_i \otimes a_{i-1} \oplus q_i \quad (i = 1, 2, 3, \ldots, N) \tag{10}$$

ここで$\otimes$及び$\oplus$が以下の性質を満たすとする。

- $\otimes$及び$\oplus$はいずれも単位元を持つ。
- $\otimes$及び$\oplus$はいずれも結合法則を満たす。
- $\otimes$は$\oplus$について分配的である。

このとき、回帰演算(10)は、

$$\begin{cases} P_i^{(j)} \equiv \bigotimes_{k=i}^{j+1} p_k \\ Q_i^{(j)} \equiv \bigoplus_{k=i}^{j+1} \{ (\bigotimes_{l=i}^{k-1} p_l) \otimes q_k \} \end{cases}$$

を用いて、式(11)のように表現できる。

$$a_i = P_i^{(j)} \otimes a_j \oplus Q_i^{(j)} \quad (0 \le j < i \le N) \tag{11}$$

ここで、通常の意味の「乗算」を$\otimes$で、「加算」を$\oplus$で置き換えてできる「行列／ベクトルの積及び和」の表現を用いると、線形バルク回帰は式(12)のように表せる。

$$\vec{v}_i = \mathbf{A}_i \otimes \vec{v}_{i-1} \oplus \vec{b}_i \tag{12}$$

すなわち、$\otimes$及び$\oplus$で構成される線形バルク回帰も、並列化できる。

例えば、$\otimes$を通常の加算、$\oplus$をmaxあるいはmixとする線形バルク回帰は並列化できる。

6. 実測及び評価

本稿で提案するループ並列化手法の有効性の検証のため、いくつかのループについて実測を行なった。測定環境は表2に示す二種類を用いた。図5、6、7、8

表2 実測に用いた並列計算機の諸元

	測定環境1	測定環境2
機種名	富士通 GP7000F model 900	日立 SR2201
CPU	SPARC64GP	疑似ベクトル機構付き RISC
一次キャッシュ	命令 64KB/データ 64KB	命令 16KB/データ 16KB
二次キャッシュ	8MB	命令 512KB/データ 512KB
CPU数	24（うち8個のみ使用）	64（うち32個のみ使用）
主記憶	24GMB (SMP)	7GB/PE (疎結合)
並列ライブラリ	MPI, POSIX スレッド	MPI

に、四種類のループに対して本手法を適用して実測した結果を示す。いずれも横軸にPE数を、縦軸には逐次処理時と比較した高速化率をとった。計測した経過時間には初期化処理は含まれていない。疎結合計算機である測定環境2における実測では、各PEに必要なデータを割り付けた後、各PEがそれぞれの担当の計算を終了した時点までの時間を計測した。実測を行なったループは、それぞれのグラフの枠内に示した。

実測に際しては表2に示した通り、MPIあるいはPOSIXスレッドを用いてプログラムを記述した。MPIプログラムは、両方の測定環境で全く同じコードを用いた。コンパイラはいずれもシステム附属のものを用い、コンパイル時の最適化オプションは-Oのみを付加した。

図5は、3章で述べた線形回帰演算の並列化手法を、式(1)に相当するループに対して適用した結果である。いずれの測定環境においても、PE数が小さい間は理想的な高速化が達成されている。測定環境2においては、PE数の増加に伴って通信処理がネックになり、速度が低下している。

図6は、4章の図2(a)のループの実測結果である。コードの骨組みは図3に示したものである。ここでも図5と同様の振舞が見受けられるが、通信量が変わらず全体的な計算量が増加した分だけ高速化率が向上している。

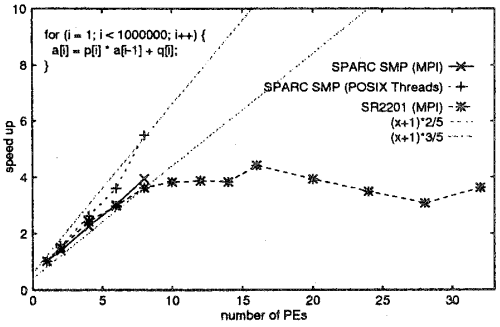


図 5　一次回帰演算

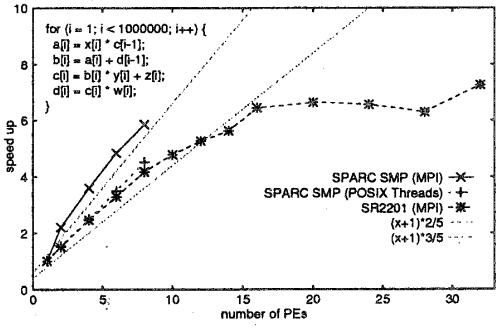


図 6　一次回帰演算

図 7 は、ループボディをより大きくしたもので、イタレーション間には多数のループ運搬依存が含まれる。測定環境 2 においては、図 6 よりもさらに高速化率が高まっているが、測定環境 1 において POSIX スレッドを用いた場合には逆に大幅に速度低下が起こっている。他のループにおいて MPI と POSIX スレッドとで経過時間に極端な差はみられない上に、図 7 中の MPI での結果にはこの現象が見られないため、共有メモリに対するアクセスの多さによる false sharing の頻発などが原因であると考えられる。

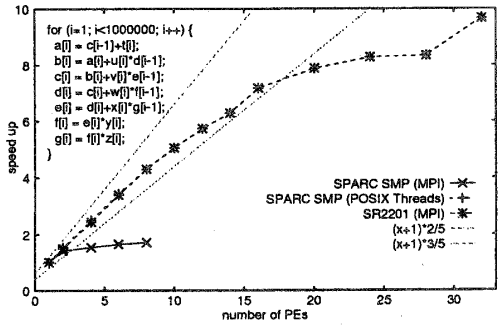


図 7　規模の大きいバルク回帰

図 8 は、ループ中に四則演算以外の演算が入っているものであるが、これも線形バルク回帰とみなすことができ、並列化できる。しかも、十数台規模の並列計算機においてはその性能を充分に生かした高速化が可能であることが分かる。

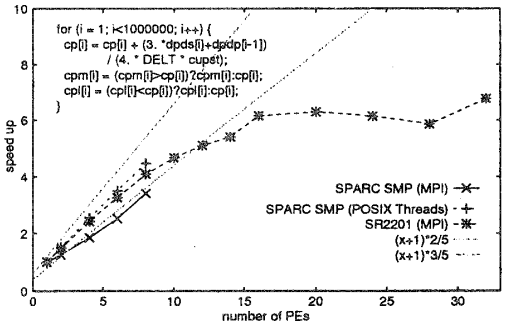


図 8　最大値最小値検索を含むループ

7. おわりに

本稿では、バルク回帰検出によるループの並列化手法について述べた。本手法は、従来細粒度並列でしか高速化が難しいとされていた DOACROSS ループに対して、高級言語レベルでの変換を施すことによって、様々な構成の並列計算機上の高速実行を可能にする。

本稿で示した実測結果は、プロセッサ間通信処理を最適化していない簡易型の分割統治法に基づくものであるが、従来は逐次ループとみなされていたループであっても、台数効果が得られることを示している。通信処理部の最適化を考慮した並列実行方式や、アンローリングなどの最適化手法との組合せの効果についての考察が、今後の課題である。

参考文献

1) S. Lakshmivarahan, S. K. Dhall: "Parallel Computing Using the Prefix Problem", Oxford Univ. Press (1994)
2) 津田孝夫: "数値処理プログラミング", 第 2 章, 岩波書店 (1988)
3) 中村素典, 津田孝夫: "自動ベクトル化コンパイラにおけるイディオム認識法," 情報処理学会論文誌, 第 32 巻, 第 4 号, pp.491–503 (1991)
4) H. Wada, K. Ishii, M. Fukagawa, H. Murayama, S. Kawabe: "High-speed Processing Schemes for Summation Type and Iteration Type Vector Instructions on HITACHI Supercomputer S-820 System," Proceedings of International Conference on Supercomputing, pp.197–206, (1988)