バルク回帰並列処理：依存のあるループの並列実行方式

川端英之[†] 谷口宏美[††] 津田孝夫[†]

本論文では，ループ運搬依存を含むループの効率的な並列化手法を提案する．依存のあるループの並列化においては，ループ中で繰り返し実行される命令列（ループボディ）全体の計算処理を prefix 計算における要素演算と見なしてアルゴリズム変換によって並列化する方法がある（回帰演算の並列化手法の適用）．回帰演算の並列化手法は，pipeline 法や doacross 法の適用が困難なループに対しても有効であるなどの特徴を持つが，その単純な適用は総計算量の大幅な増加を引き起こし，プロセッサ数が少ない並列計算機では必ずしも高速化につながらない．これに対し，本論文で提案する手法は，依存のあるループへの回帰演算の並列化手法の適用において「ループ運搬依存の量」に着目したループ変換を導入する．これにより並列化にともなう計算量の増加を抑え，数台〜数十台程度のプロセッサから成る並列計算環境において効率の良い並列実行を可能にする．疎結合計算機や SMP 計算機を用いた実測では，pipeline 法や doacross 法が適用できないループに対してプロセッサ数に比例した高速化が可能であることが確かめられた．

Bulk Recurrent Parallel Processing: A Method of Parallel Execution for Non-doall Loops

Hideyuki Kawabata,[†] Hiromi Taniguchi[††] and Takao Tsuda[†]

In this paper, we propose an efficient method to parallelize non-doall loops. Parallelization techniques for linear recurrences are applicable for many non-doall loops which are not amenable to such methods as pipeline method and doacross method. However, naive application of parallelization for recurrences to non-doall loops can cause additional volumes of computation for parallel execution of the loops, so that speedup on computers with small number of processors is not necessarily guaranteed. In order to tackle this problem, we introduce a loop transformation technique into the parallelizing method for recurrences. The transformation is based on the idea of reducing "the amount of loop-carried dependence," and suppresses the growth of the amount of computation. Experimental results show the method is efficient for loops which can not be parallelized by doacross method on both shared-memory SMP systems and distributed-machine environments.

1. はじめに

プログラムの高速実行のための最適化手法として，並列化技術の向上に対する要求は尽きない．プログラムの並列化は様々な粒度において行われるが，特にプログラムの実行時間を大きく削減できる可能性が高いといわれるループの並列化に対しては，多くの取組みがなされている[1),2)]．並列処理による高速化においては，並列化率をいかに向上させられるかが重要であり，ループ運搬依存があるループに対しても，様々な並列化手法が検討されている．

ループ運搬依存のあるループを複数のプロセッサ（以下，PE と略す）で並列処理させるためには，各 PE の協調によって命令実行順序制約を守りつつ計算を進める必要がある．最も簡単な並列処理方式としては，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かかわる部分とを分離し，前者のみを doall 並列実行する方法（ループ分割）があげられる．しかし，依存関係のパターンや処理内容と PE 数との関係によっては必ずしもループ分割が有効とは限らない[3)]．

ループ運搬依存のあるループの並列実行方式として広く知られたものに，doacross 法および pipeline 法がある．doacross 法では，実行順序に制約のある命令間で同期をとりながら各 PE が並列に計算を遂行する．pipeline 法では，ループ中で繰り返し実行される命令

† 広島市立大学情報科学部
Faculty of Information Sciences, Hiroshima City University
†† 広島市立大学大学院情報科学研究科
Graduate School of Information Sciences, Hiroshima City University

列（ループボディ）を複数のステップに分割し，各々のループボディの処理の繰返し（イタレーション）をステップ単位で時間的にずらして行うことにより依存関係を保つ．doacross 法および pipeline 法は，PE への処理のマッピング方法には違いはあるが，いずれも，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．

しかしこれらの並列実行方式は以下のような欠点がある．doacross 法や pipeline 法では，並列実行に際して頻繁に同期をとる必要があるため，効率良い並列処理のためには同期処理がきわめて高速になされる必要があり，疎結合計算機のような同期コストが大きい計算機では高速化が容易ではない．また，いずれもループボディ内の処理を分割する並列化であるため，（ループ分割による並列化と同様に）各イタレーション内に閉じた参照の局所性を無駄にしてしまう．加えて，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して逐次処理するため，実行時間を左右するクリティカルパスに計算処理が含まれる．このため並列度の上限が低く抑えられる．特にループボディの末尾から先頭へのループ運搬依存がある場合には，これらの方法では並列化による高速化が望めない．

このほか，イタレーション間の依存距離に応じた並列度を得る方法[4]もあるが，一般に高い並列度は望めず，依存距離が 1 の場合には並列化できない．

以上あげた方法と対比されるものに，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ある．代表的なものとして，ループ中から総和計算や最大値検索などの reduction 計算や回帰演算に内在する prefix 計算を抽出してプログラム変換を施す並列化手法があげられる．これらの計算は，演算順序の変更を許すことによって大きな並列性を引き出すことが可能で，分割統治法，巡回縮約法などの様々な並列アルゴリズムが研究されている[5)~12)]．

しかしながら，多くの並列 prefix アルゴリズムは，多数個の PE を仮定して計算ステップ数を削減しようとするもので[5),11)]，たとえば N 個の要素の計算に対して N ステップ逐次アルゴリズムを $O(\log N)$ ステップで解くなど，著しい高速化の可能性を示唆するものの，プログラム変換によって一般に総計算量は増大する．このため，小規模な PE 数の計算機では高速化につながらない場合が多い[13)]．

prefix 計算の高速化については，ベクトルプロセッサを用いたハードウェアによるアプローチもある[14)]．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．一般のベクトル命令との比較ではその高速化率はわずかであるといわざるをえない．

以上のとおり，ループ運搬依存のあるループの並列化を考えるとき，広く一般に利用可能な数台〜数十台規模の並列計算機において効果的な並列実行を行うための枠組みは確立されているとはいい難い．これに対し本論文では，線形回帰演算の並列化手法を応用した，少ない PE 数からスケーラブルに高速化できるループ並列化手法を提案する．

本論文で並列化対象とするループは，ベクトル $\vec{a}_i$，$\vec{q}_i$ および行列 P_i を用いてループボディを

$$\vec{a}_i = P_i \cdot \vec{a}_{i-1} + \vec{q}_i$$

の形に書き換え可能なループである（バルク回帰ループと呼ぶ；3 章参照）．このループは各イタレーション内での計算全体を prefix 計算における一演算単位と見なして並列化できるが，本手法ではその変換の過程でループに最適化処理を施す．処理の中心は，「ループ運搬依存の量」すなわちイタレーション間で授受される値の個数を削減するループ再構成である．これにより，単純な回帰演算の並列化において問題となる計算量の大幅な増加を抑え，少数の PE からでも PE 数に比例した高速化を実現する．

本手法の有効性を確認するため，疎結合並列計算機および SMP 並列計算機において，いくつかのループについて実測を行った．その結果，PE 数 2〜32 での計測において，ループボディが小さいループでもその繰返し回数（ループ長）が大きければ PE 数に比例する高速化が可能であることが確かめられた．SR2201 における 8PE での実測では，単純な回帰演算並列化手法を適用した場合と比較して，3.4 倍の速度向上が得られる例も観測された．

なお本論文では，並列化対象とするループはループ運搬依存の依存距離が 1 であるものとし，ループ中に制御依存が存在するものは除外する．並列化はソースレベルのプログラム変換により行う．並列実行環境が共有メモリを有するか否かは問わない．

以下，まず 2 章では，本手法のベースとなる線形回帰演算の並列実行方式の概要について述べる．3 章では，バルク回帰ループの並列化手法とその最適化手法について述べ，4 章では実測評価に基づく考察を行う．5 章でまとめと今後の課題を述べる．

```
for (i = 1; i ≤ N; i++) {
    a_i = p_i · a_{i-1} + q_i;
}
```

図 1 線形一次回帰演算
Fig. 1 First order linear recurrence.

2. 準備：線形回帰演算の並列化手法

2.1 線形回帰演算における並列性

線形一次回帰演算は**図 1** に示すループで表現できる．ここで $p_i, q_i (i = 1, 2, \ldots, N)$ はループ不変変数である．このループの各イタレーション間には依存距離 1 のループ運搬フロー依存が存在しており，a_i の値を求める計算は個々の i についてまったく独立には実行することはできない．しかし，

$$\vec{a}_i \equiv \begin{pmatrix} a_i \\ 1 \end{pmatrix}, \quad P_i \equiv \begin{pmatrix} p_i & q_i \\ 0 & 1 \end{pmatrix}$$

と置くと，その回帰の様子は以下のように表せる．

$$\begin{aligned} \vec{a}_i &= P_i \cdot \vec{a}_{i-1} \\ &= P_i P_{i-1} \cdots P_1 \cdot \vec{a}_0 \end{aligned} \tag{1}$$

行列積は結合法則を満たす二項演算であり，式 (1) は図 1 の一次回帰演算が prefix 計算と見なせることを示している．

2.2 線形回帰演算の並列実行方式

ここで，数台〜数十台規模の並列計算機において効率的に実行できる線形回帰演算の並列実行方式として，計算量を $O(N)$ に抑えた単純な分割アルゴリズムを示す．次章で述べるバルク回帰並列処理手法も，ここで述べるアルゴリズムをベースとしている．

まず，

$$P_i^{(j)} \equiv P_i P_{i-1} \cdots P_{j+1} \quad (0 \le j < i \le N)$$

と置くと，式 (1) は次のように表現できる．

$$\vec{a}_i = P_i^{(j)} \cdot \vec{a}_j \tag{2}$$

式 (2) は，付加的な計算によって回帰演算の依存距離を引き延ばすことができる様子を表している．なお，$P_i^{(j)}$ の各成分は，

$$\tilde{p}_i^{(j)} \equiv \prod_{k=j+1}^{i} p_k$$

$$\tilde{q}_i^{(j)} \equiv \sum_{k=j+1}^{i} \{ (\prod_{l=k+1}^{i} p_l) \cdot q_k \}$$

を用いて以下のように置ける．

$$P_i^{(j)} = \begin{pmatrix} \tilde{p}_i^{(j)} & \tilde{q}_i^{(j)} \\ 0 & 1 \end{pmatrix}$$

```
/* step1 */
t = a_0;
for (i = 1;
     i ≤ m;
     i++) {
    t = p_i · t + q_i;
}

/* step2 */
send t to PE1;

/* step3 */
for (i = 1;
     i ≤ m;
     i++) {
    a_i=p_i a_{i-1} + q_i;
}
```

(a) code for PE0

```
/* step1 */
u = p_{km+1};
v = q_{km+1};
for (i = km + 2;
     i ≤ (k + 1) · m;
     i++) {
    u = p_i · u;
    v = p_i · v + q_i;
}

/* step2 */
receive t from PE k − 1;
t = u · t + v;
send t to PE k + 1;

/* step3 */
a_{km} = t;
for (i = km + 1;
     i ≤ (k + 1) · m;
     i++) {
    a_i=p_i a_{i-1} + q_i;
}
```

(b) code for PE k $(1 \le k \le p-2)$

図 2 線形回帰演算の並列実行コード
Fig. 2 Parallel code for linear recurrence.

表 1 各 PE への計算のマッピング
Table 1 Mapping of computation for each PE.

PE	step1	step3
0	a_m	$a_1, \ldots, a_m$
1	$\tilde{p}_{2m}^{(m)}, \tilde{q}_{2m}^{(m)}$	$a_{m+1}, \ldots, a_{2m}$
2	$\tilde{p}_{3m}^{(2m)}, \tilde{q}_{3m}^{(2m)}$	$a_{2m+1}, \ldots, a_{3m}$
...	...	...
$p-2$	$\tilde{p}_{(p-1)m}^{((p-2)m)}, \tilde{q}_{(p-1)m}^{((p-2)m)}$	$a_{(p-2)m+1}, \ldots, a_{(p-1)m}$
$p-1$	—	$a_{(p-1)m+1}, \ldots, a_N$

つまり式 (2) は次のように表現できる．

$$a_i = \tilde{p}_i^{(j)} a_j + \tilde{q}_i^{(j)} \tag{3}$$

式 (3) を用いて，PE 数が p である環境で図 1 のループによって $a_1, \ldots, a_N$ の N 個の値を計算する手順を述べる．この計算の並列実行は**図 2** に示す 3 ステップからなるコードで行うことができる．各 PE への計算のマッピングは**表 1** に示すとおりである．図 2 や表 1 において，$m \equiv N/p$ とする（簡単のため N は p の倍数であるとする☆）．たとえば PE1 は，step1 で $\tilde{p}_{2m}^{(m)}$ および $\tilde{q}_{2m}^{(m)}$ の値を計算し，step3 では $a_{m+1}, \ldots, a_{2m}$ の値を計算する．図 2 の (a) および (b) はそれぞれ

☆ 実際は各 PE の担当範囲は任意の区間にできる．

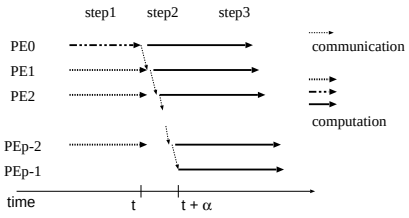


図 3 回帰演算の並列実行の様子
Fig. 3 Progression of parallel linear recurrence execution.

PE0 用，PE $k(1 \leq k \leq p-2)$ 用のコードである．PE $p-1$ は図 2 (b) のコードのうち step1 全体と step2 の send を行わない．変数 t，u，v はいずれも作業用の局所変数である．send や receive は，PE 間通信や共有メモリアクセスなどによる PE 間でのデータ授受を示す．配列変数は，共有メモリ環境では共有変数と見なせばよいが，分散メモリ環境であればあらかじめデータを分散しておく必要がある☆．**図 3** 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，横軸および縦軸はそれぞれ時間の経過および PE の番号に対応する．太実線矢印および太点線矢印は，それぞれ a_i の計算および $\tilde{p}_i^{(j)}$ および $\tilde{q}_i^{(j)}$ の計算の進行の様子を示し，細点線矢印は，PE 間通信の様子を示している．PE0 は step1 では配列要素へのストアをしないので，step3 よりも step1 の方が短時間で済ませられよう．また PE0 の step1 は他の PE の step1 より短時間で済むと予想できる．

図 2 や図 3 から分かるように，この並列処理方式では，頻繁な同期や通信は不要，ループ長はループ開始時に確定するならば並列化が可能，計算処理を複数のプロセッサでほぼ均等に分散できるため計算負荷に限れば原理的には並列性に上限がない，などの特徴を持つ．

ここで，この並列実行手順で必要とされる実行時間を見積もる．浮動小数点演算 1 回に要する時間を t_f，配列要素 1 つのロードやストアに要する時間を t_l とすると，図 1 の計算を逐次的に行う際の実行時間は $t_{seq} \equiv (2t_f + 3t_l)N$ と見なせる．一方 PE 数 p での並列実行に要する時間は，図 2 および図 3 から，およそ $t_p \equiv (5t_f + 5t_l)(N/p) + \alpha(p)$ である．ここで $\alpha(p)$ は図 3 中の α で，PE 間での同期やデータ授受に必要となる時間である．$t_l = \beta \cdot t_f \ \ (0 < \beta)$ と仮定し，$\alpha(p)$ が微小であるとすれば，高速化率 $S(p) \equiv t_{seq}/t_p$ はおよそ以下のようにまとめられる．

$$S(p) \equiv \frac{(2t_f + 3t_l) \cdot N}{(5t_f + 5t_l) \cdot (N/p) + \alpha(p)} \simeq \frac{2 + 3\beta}{5 + 5\beta} \cdot p \qquad (4)$$

つまり，$S(p)$ は p にほぼ比例し（比例定数 2/5〜3/5），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いえる．なお，$\alpha(p)$ は p の増加にほぼ比例して増えるので，最大の高速化率には限界がある．また N が十分に大きくない場合は，総実行時間に占める並列処理にまつわるオーバヘッドの割合が増え，高速化につながらない可能性が高い．

3. バルク回帰並列処理手法

本章では，ループ運搬依存のあるループに対する効率的な並列化手法について述べる．以下，並列化の対象とするループについて述べた後，線形回帰演算の並列化手法を単純に適用した場合についての考察を行い，続いて，少ない PE 数で高い台数効果を引き出すための最適化手法とその効果について述べる．

3.1 バルク回帰ループ

ループ内で行われる計算が式 (5) に示す回帰演算で表されるとき，そのループは（n 元 l 次）**バルク回帰**ループであると呼ぶことにする．

$$\vec{a}_i = f_i(\vec{a}_{i-1}, \vec{a}_{i-2}, \ldots, \vec{a}_{i-l}) + \vec{q}_i \qquad (5)$$

ここで $\vec{a}_i$ は n 個の変数から成るベクトル，f_i は i のみに依存して定まる係数による $\vec{a}_{i-1}, \vec{a}_{i-2}, \ldots, \vec{a}_{i-l}$ の線形結合，$\vec{q}_i$ は定数ベクトルである．ここでは簡単のため，$n \times n$ 行列 P_i によってループボディが式 (6) で表される n 元一次回帰の場合，すなわち，ループ運搬依存の依存距離がすべて 1 である場合に限定して話を進める．

$$\vec{a}_i = P_i \cdot \vec{a}_{i-1} + \vec{q}_i \qquad (6)$$

式 (6) で表されるバルク回帰ループは，2.2 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．

たとえば**図 4** (a) に示すループは，前進代入によって**図 5** のように書き換えることができる．ここで，

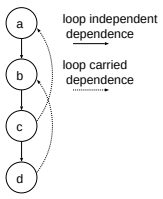
$$\vec{a}_i \equiv \begin{pmatrix} a_i & b_i & c_i & d_i \end{pmatrix}^T,$$
$$\vec{q}_i \equiv \begin{pmatrix} 0 & 0 & z_i & z_i w_i \end{pmatrix}^T$$

および

☆ 全配列要素を複写するのでないならば各 PE 用のコードにおける配列参照の添字式は局所メモリ参照用に修正する必要があるが，本論文で示すコード例では明記しない．また PE0 以外が receive によって得た値は図 2 では便宜上 step3 にて配列要素にストアしているように記述したが，実際はローカル変数に一時的に保持しておけばよい．

```
for (i = 1; i ≤ N; i++) {
    a_i = x_i · c_{i-1};    /* a */
    b_i = a_i + d_{i-1};    /* b */
    c_i = b_i · y_i + z_i;  /* c */
    d_i = c_i · w_i;        /* d */
}
```

(a) loop code



(b) dependence graph of (a)

図 4 ループ運搬依存のあるループ
Fig. 4 A non-doall loop.

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

図 5 図 4 (a) のループの変形後
Fig. 5 Modified non-doall loop.

$$P_i \equiv \begin{pmatrix} 0 & 0 & x_i & 0 \\ 0 & 0 & x_i & 1 \\ 0 & 0 & x_i y_i & y_i \\ 0 & 0 & x_i y_i w_i & y_i w_i \end{pmatrix} \tag{7}$$

と置くと，図 5 のループボディは式 (6) で表せる（4 元バルク回帰）．すなわち，図 4 (a) のループは，ループボディ全体を 1 つの演算単位とする線形回帰演算と見なした並列化が可能である．

3.2 バルク回帰ループへの単純な並列化の適用

ここで，2.2 節で述べた線形一次回帰演算の並列実行方式をバルク回帰ループに適用する場合の計算量を見積もる．まず図 2 の並列コードを n 元バルク回帰ループ用に修正すると**図 6** となる．図中，$\vec{a}_i$，$\vec{q}_i$ および $\vec{v}$ は n 次元ベクトル，P_i および U は $n \times n$ 行列である．また $m \equiv N/p$ とする．代入記号や演算子は適宜，行列やベクトルに対する代入あるいは演算と見なす．図 6 から，PE 数 p での並列実行に要する時間はおよそ $t_p \equiv (2n^2+3n)(nt_f+t_l)m + \alpha(p)$ と見なせる☆．式 (6) によるループの逐次実行に要する時間は $t_{seq} \equiv (2n^2 t_f + (n^2+2n)t_l)N$ であるので，2.2 節同様の仮定の下で，高速化率 $S_n(p) \equiv t_{seq}/t_p$ は以下のとおりとなる．

$$S_n(p) = \frac{(2n^2 t_f + (n^2+2n)t_l)N}{(2n^2+3n)(nt_f+t_l)(N/p) + \alpha(p)}$$

/* step1 */	/* step1 */
$\vec{t} = \vec{a}_0$;	$U = P_{km+1}$;
for ($i = 1$;	$\vec{v} = \vec{q}_{km+1}$;
$i \le m$;	for ($i = km+2$;
$i++$) {	$i \le (k+1)\cdot m$;
$\vec{t} = P_i\vec{t} + \vec{q}_i$;	$i++$) {
}	$U = P_i \cdot U$;
	$\vec{v} = P_i \cdot \vec{v} + \vec{q}_i$;
	}
/* step2 */	/* step2 */
send $\vec{t}$ to PE1;	receive $\vec{t}$ from PE $k-1$;
	$\vec{t} = U \cdot \vec{t} + \vec{v}$;
	send $\vec{t}$ to PE $k+1$;
/* step3 */	/* step3 */
for ($i = 1$;	$\vec{a}_{km} = \vec{t}$;
$i \le m$;	for ($i = km+1$;
$i++$) {	$i \le (k+1)\cdot m$;
$\vec{a}_i = P_i\vec{a}_{i-1} + \vec{q}_i$;	$i++$) {
}	$\vec{a}_i = P_i\vec{a}_{i-1} + \vec{q}_i$;
	}
(a) code for PE0	(b) code for PE k ($1 \le k \le p-2$)

図 6 バルク回帰ループの並列実行コード
Fig. 6 Parallel code for bulk recurrence.

$$\simeq \frac{2n+(n+2)\beta}{(2n+3)(n+\beta)} \cdot p \tag{8}$$

$S_1(p)$ は式 (4) の $S(p)$ と一致している．

式 (8) から，式 (6) によって表されるバルク回帰ループは並列実行によって PE 数 p にほぼ比例した高速化率が得られることが期待できる．しかしその際の比例定数の値は非常に小さい場合がある（β の値によるが，$S_2(p)$ では 2/7〜4/7，$S_4(p)$ では 2/11〜6/11）．β が一定のとき，n が増加するにつれて $S_n(p)$ は減少する（その減少の仕方は β の値が小さいほど急速である）．また，p は $n+2$ 以上の値でなければ $S_n(p)$ が 1 を超えず，n の値が大きいループの並列化による高速化は PE 数が少ない並列計算機では難しいということが予想される．

なお，図 4 (a) と図 5 の対比からも分かるとおり，オリジナルのループをバルク回帰の形にするためのコードの変更は「共通部分式削除」の逆操作であって，代入文間のループ独立な依存関係を緩和する代わりに演算量を増大させる．またこの変換過程で，オリジナルのループにはなかった係数どうしの計算処理が新たに必要となる☆☆．これらの演算量の増加は並列化後のコードの実行時間の増加に直接つながる．つまり，ループ

☆ n 次ベクトルの内積に要する演算回数を $2n-1$ とした．また，n の値は，図 6 中の U および $\vec{v}$ がキャッシュに十分収まる程度の大きさであると仮定している．

☆☆ たとえば図 5 のループ中の d_i への代入文における変数 c_{i-1} の係数である x_i，y_i，w_i の積を計算するコストは $S_n(p)$（式 (8)）の算出の際には考慮しなかった．

がバルク回帰であることが判明しても，本節で述べたような単純な並列化の適用では，上記 $S_n(p)$ で期待されるほどの高速化が達成できない可能性もある．

一方で，一般のループを n 元バルク回帰の式 (6) に変形した際に，P_i が密行列になるとは限らない．このとき，バルク回帰ループに変形後のコードおよびそれを並列化したコードの双方について，マシン命令列生成の際に演算量を大幅に削減できる可能性もある．その場合はあたかも n の値が減ったごとくに高速化率は $S_n(p)$ よりも大きい値となる可能性もある．

3.3 効率的な並列処理—バルク回帰並列処理手法

本節では，前節で考察したバルク回帰ループの並列実行における特性をふまえて，並列化による高速化をより大きく引き出すための並列処理手法をまとめる．

まず，オリジナルのループを式 (6) に変形した際の P_i に注目する．P_i に零列が含まれる場合は，図 6 の並列コードにおける U の対応する列もつねに零列，$\vec{v}$ の対応する要素はつねに零となる．つまり P_i の零列の数だけ元数 n を減じたバルク回帰ループの場合と同等な並列コードで記述できる☆．n の値の削減は $S_n(p)$（式 (8)）における p に対する比例定数の値の増加を意味し，高速化率向上に効果があると考えられる．

一方，P_i における非零列は依存グラフ中のループ運搬フロー依存エッジに対応する．たとえば図 4 (a) のループに対応する P_i（式 (7)）における非零列（第 3, 第 4 列）は，ベクトル $\vec{a}_i$ を求める際に変数 c_{i-1} および d_{i-1} の値しか必要としないことを意味し，それは図 4 (b) に示される依存グラフのノード c および d から出ているループ運搬依存エッジによってイタレーション間で授受される値に対応する．すなわち，イタレーション間で授受される値の個数（つまりは n の値）を削減するループの再構成をあらかじめ適用すれば，並列化による高速化がより効率的に行えると考えられる．

イタレーション間で授受される値の個数の削減は，software pipelining [15] に類似のループ再構成技法によって，ループ中の演算量を本質的には変更することなく行える．software pipelining はイタレーション内の命令レベル並列性の抽出が目的であって，ここで検討するループ再構成手法とは目的はまったく異なるが，変換結果が同等となる場合はある．

以上を考慮すると，バルク回帰ループの並列化手順は以下のようにまとめられる．本手法を，バルク回帰並列処理手法と呼ぶ．

☆ この最適化は並列ソースコード出力の際に明示的に行わなくても，マシン語コードの生成時の局所的な最適化（「0 乗算」の削除など）で十分になされる場合もあろう．

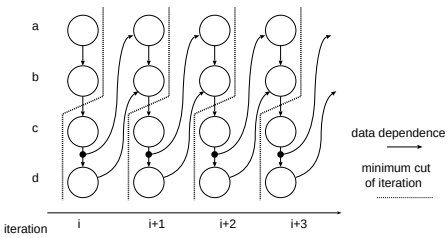


図 7 展開後の依存グラフとその分割
Fig. 7 Expanded dependence graph.

```
a1 = x1 · c0;              /* a' */
b1 = a1 + d0;              /* b' */
for (i = 1; i ≤ N − 1; i++) {
    ci = bi · yi + zi;     /* c */
    di = ci · wi;          /* d */
    ai+1 = xi+1 · ci;      /* a */
    bi+1 = ai+1 + di;      /* b */
}
cN = bN · yN + zN;         /* c' */
dN = cN · wN;              /* d' */
```

図 8 再構成後のループ
Fig. 8 Restructured loop.

(1) ループ運搬依存を削減するループ変換
(2) 回帰成分の抽出
(3) 回帰成分の並列化
(4) PE 内ループ融合

各々のステップについて以下で順に述べる．

(1) ループ運搬依存を削減するループ変換

ループの再構成によって，イタレーション間で授受される値の個数を最小化する．これは繰返し空間方向に展開した依存グラフを用いて行える．たとえば図 4 (a) に対する依存グラフ（図 4 (b)）を繰返し空間方向に展開すると**図 7** となる．このグラフをイタレーションの進行方向に等間隔に分割すればループを再構成できるが，このときの「カット」に含まれるエッジ数が最小となるようにすればよい．なお一般にデータ依存グラフにおいては各ノードの出次数が 1 とは限らず，同一データ値に対応するデータフローエッジが複数存在しうる．しかしカットに含まれるエッジ数の数え上げに際してそれらを区別する必要はない（図 7 では同一データ値のエッジを束ねて「枝分かれ」で表現している）．図 7 中の破線は，ループ運搬依存エッジ数を 1 とする分割の例である（ループ運搬依存エッジ数が最小となる分割は一意ではない）．図 7 の分割に基づき図 4 (a) のループを再構成した結果を**図 8** に示す．ループ中の計算量は，このループ変換によって本質的には変化しない．

(2) 回帰成分の抽出

手順 (1) の再構成を施したループから「回帰成分」

を抽出する．ここで回帰成分とは，ループ運搬フロー依存によりイタレーション間で値が受け渡される変数のみでベクトル $\vec{a}_i$ を構成したときの，式 (6) に対応するバルク回帰ループである．回帰成分の抽出は，依存グラフの分割の情報から行える．すなわち分割をまたぐエッジに対応する変数についての回帰的代入文を前進代入で抽出すればよい．図 7 に示す依存グラフの分割から抽出される回帰成分を**図 9** に示す．

(3) 回帰成分の並列化
前項で抽出した回帰成分に対し 3.2 節で述べた手法を適用し，並列化する．

(4) PE 内ループ融合
回帰成分を抽出して単独で並列化した後，残りのコードと融合して全体を並列化し，各 PE 用の並列コードを生成する．ここで残りのコードとは，イタレーション間で授受されない変数値を算出するステートメントなどである．図 9 の回帰成分に並列化を適用し融合処理を行った結果を**図 10** に示す．図中，p，q，u，v，t はいずれも作業用の局所変数である．

```
for (i = 1; i ≤ N − 1; i++) {
    b_{i+1} = (x_{i+1} y_i + w_i y_i) b_i + (x_{i+1} z_i + w_i z_i);
}
```

図 9 抽出された回帰成分
Fig. 9 Extracted recurrence.

オリジナルループにループ分割を適用することによって doall ループとして分離できる部分が含まれている場合には，その部分は手順 (2) において回帰成分からいったん除外され，手順 (4) において並列コードの step3 に融合される．すなわち，バルク回帰並列処理手法では，ループ分割によって doall ループが分離できる場合でも，ループ分割でイタレーション内のデータ参照局所性を破壊することなく，かつ，doall ループの並列性を犠牲にすることなく，ループ全体を並列処理できる．

ここで，本手法が計算量削減に直接影響を及ぼす点をまとめる．いずれもバルク回帰ループへ変換する必要のあるステートメント数の削減に関連している．

- 本手法は，手順 (2)〜(4) で依存のあるループから回帰成分のみを抽出して並列化し，その後にその他の部分と融合する．これはバルク回帰ループへ変換するループのステートメント数を回帰成分のステートメント数にまで削減する効果がある．
- 本手法は，手順 (1) でループの再構成を行い回帰成分自体を小さくする．これによりバルク回帰ループへ変換すべきループのステートメント数も削減される（ループ再構成による計算量の変動の詳細につ

```
/* step1 */
p = x_{i+1} y_i + w_i y_i;
q = x_{i+1} z_i + w_i z_i;
t = p · b_1 + q;
for (i = 1; i ≤ m; i++) {
    p = x_{i+1} y_i + w_i y_i;
    q = x_{i+1} z_i + w_i z_i;
    t = p · t + q;
}

/* step2 */
send t to PE 1

/* step3 */
a_1 = x_1 · c_0;          /* a' */
b_1 = a_1 + d_0;          /* b' */
for (i = 1; i ≤ m; i++) {
    c_i = b_i · y_i + z_i;    /* c */
    d_i = c_i · w_i;          /* d */
    a_{i+1} = x_{i+1} · c_i;  /* a */
    b_{i+1} = a_{i+1} + d_i;  /* b */
}
```

(a) PE0 用のコード

```
/* step1 */
u = x_{km+2} y_{km+1} + w_{km+1} y_{km+1};
v = x_{km+2} z_{km+1} + w_{km+1} z_{km+1};
for ( i = km + 2; i ≤ (k + 1)m; i++) {
    p = x_{i+1} y_i + w_i y_i;
    q = x_{i+1} z_i + w_i z_i;
    u = p · u;
    v = p · v + q;
}

/* step2 */
receive t from PE k − 1
t = t · u + v;
send t to PE k + 1

/* step3 */
b_{km} = t;
for (i = km + 1; i ≤ (k + 1)m; i++) {
    c_i = b_i · y_i + z_i;    /* c */
    d_i = c_i · w_i;          /* d */
    a_{i+1} = x_{i+1} · c_i;  /* a */
    b_{i+1} = a_{i+1} + d_i;  /* b */
}
```

(b) PE $k(1 \le k \le p-2)$ 用のコード

```
/* step1 */
    /* do nothing */

/* step2 */
receive t from PE p − 2

/* step3 */
b_{pm−1} = t;
for (i = pm; i ≤ N − 1; i++) {
    c_i = b_i · y_i + z_i;    /* c */
    d_i = c_i · w_i;          /* d */
    a_{i+1} = x_{i+1} · c_i;  /* a */
    b_{i+1} = a_{i+1} + d_i;  /* b */
}
a_N = x_N · c_N;          /* a' */
b_N = a_N + d_N;          /* b' */
```

(c) PE $p-1$ 用のコード

図 10 バルク回帰並列処理コード
Fig. 10 Bulk recurrent parallel processing code.

いては 3.4 節で考察する）．

- 図 6 に示す並列コードのうち step3 および PE0 用の step1 におけるループのボディには，バルク回帰ループへ変換したループではなくオリジナルのループボディ記述をそのまま用いることができる．これによって，バルク回帰ループへの変換の過程でいったん増加した計算量を部分的に削減できる．

4 章では，上の項目それぞれの効果に着目して実測結果をまとめる．

ここで，図 4 (a) のループと，それに対して上記の最適化手順によって並列化した結果である図 10 の並列コードとから，計算量の具体的な比較を行う．2.2 節で用いた記号を使う．図 4 (a) のループの計算時間は $(5+8\beta)t_f N$ である．一方，図 10 から，並列実行に要する時間はおよそ $(14+12\beta)t_f m$ と見積もることができる．すなわち，PE 数 p に対しておよそ $5p/14$ 倍〜$2p/3$ 倍の高速化が見積もられる．

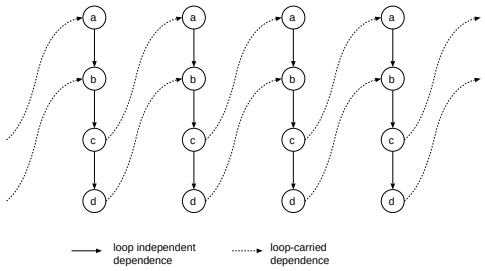
3.4 回帰成分抽出時の計算量増加

オリジナルのループをバルク回帰ループに変形すること，つまり前進代入の適用が演算量を増加させる点について 3.2 節で触れた．ところで，3.3 節で述べたバルク回帰並列処理手法では，コードの並列化の過程でループを再構成することにより，ループ運搬フロー依存によってイタレーション間で授受される変数の個数を削減することで，並列コードが要する計算量を削減する方法であった．しかし回帰成分の抽出の際には本手法でも前進代入を行っているのである．このときの演算量の増加について，ループ再構成を行わなわずに回帰成分を抽出する場合と比較して考察する．

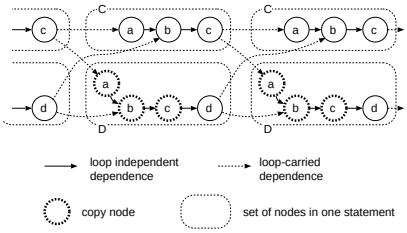
図 4 (a) のループを例にして回帰成分抽出の際の演算量増加の様子を考える．図 4 (b) に示す依存グラフを繰り返し空間方向に展開すると**図 11** (a) が描ける．ここで，図 4 (a) から図 5 への変形は，依存グラフ上では図 11 (b) のようにノードの複製を行っていることに相当する．図 11(b) 中，破線で記されたノードは演算の複写を示し，C および D とラベル付けしたノード集合は，抽出後の 2 元バルク回帰ループを構成する 2 つのステートメントをそれぞれ表している．

一方，3.3 節で述べたループ再構成手法を適用した後の依存グラフは**図 12** (a) である（図 11 (a) とはトポロジー的な差異はない）．またループ再構成後に抽出される回帰成分（図 9）に対応する依存グラフは，図 12 (b) で表される．図 11 (b) の場合と異なり，ノードの複写がまったく不要であることが分かる．

バルク回帰ループを構成するためには，ループボディの各ステートメント間にループ独立依存があってはならない．このため，バルク回帰ループの元数が多いほど，図 11 (b) における複製ノードで示されるような，計算量の増加をともなう前進代入が必要となる状況が増えると考えられる．すなわち，バルク回帰ループの元数を削減するためのループ再構成は，回帰成分抽出時の演算量増加も抑える働きがあるといえる．

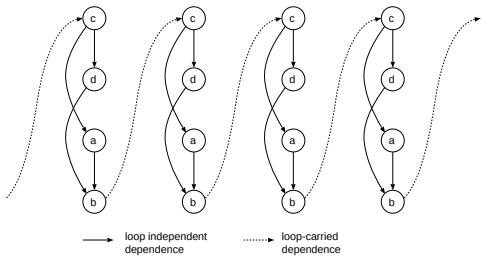


(a) オリジナルの依存グラフ

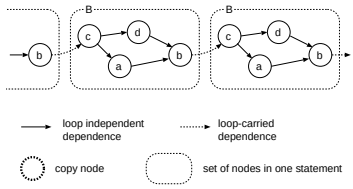


(b) 2 元バルク回帰ループの抽出結果

図 11 バルク回帰ループの依存グラフ
Fig. 11 Dependence graphs of bulk recurrence.



(a) ループ再構成後の依存グラフ



(b) 1 元バルク回帰ループの抽出結果

図 12 ループ再構成後のバルク回帰ループの依存グラフ
Fig. 12 Dependence graphs of restructured loop.

表 2 実測に用いた並列計算機の諸元
Table 2 Specifications of experimental environment.

機種名	日立 SR2201†	富士通 GP7000F‡
CPU	拡張版 PA-RISC	SPARC64-GP
L1 Cache	I16 KB/D16 KB	I64 KB/D64 KB
L2 Cache	I512 KB/D512 KB	8 MB
CPU 数	64（32 個使用）	24（16 個使用）
主記憶	256 MB/PE（疎結合）	24 GB（SMP）
ライブラリ	MPI	POSIX スレッド
OS	HI-UX/MPP	SunOS 5.8

† 日本原子力研究所計算科学技術推進センター設置
‡ 京都大学大型計算機センター設置

4. 実測および評価

本章では，バルク回帰並列処理手法をいくつかのループに適用して実測を行った結果を示し，本手法の評価について述べる．

実測に用いた計算機は，疎結合計算機の日立 SR2201 および SMP 計算機の富士通 GP7000F（以降，それぞれ SR2201 および SPARC と呼ぶ）の 2 種類である．諸元を**表 2** に示す．

表 2 に示したとおり，MPI および POSIX スレッド（以降，P スレッドと呼ぶ）を用いて実測コードを記述した．使用した言語は C である．

SR2201 上で MPI を用いた実測では，各 PE に必要なデータを割り付けてバリア同期をとった時点から，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．計時は `MPI_Wtime()` を用い，PE 間通信では `MPI_Isend()` および `MPI_Recv()` を用いた．

SPARC 上では P スレッドを用いて実測を行った．1 つのスレッドでデータの初期化を行った後に他のすべてのスレッドを生成するようにし，最初の `pthread_create()` の呼び出しの直前から最後の `pthread_join()` の呼び出しから戻るまでの時間を計測した．計時には `gettimeofday()` を用いた．PE 間のデータの授受は別個の共有変数によって行った．

いずれの測定でも，計測開始直前に，計算とは無関係の多量のデータをロード/ストアし，各 PE のキャッシュをクリアした．配列要素参照については，ロード後に局所スカラ変数へコピーして使用するなどのメモリアクセスの最適化をソースレベルで行った．演算はすべて倍精度実数（`double` 型）で行った．

コンパイラはいずれもシステム付属のものを用いた．使用した最適化オプションは以下のとおりである．

- SR2201：`mpicc +O4`
- SPARC：`fcc -Kfast_GP=2 -lpthread`

実測には，3.3 節であげた演算量の削減効果の比較のため，以下の 4 通りの並列コードを用いた．

type1 並列化するループをバルク回帰ループへ書き換え，それを単純に図 6 にあてはめたもの．

type2 type1 の並列コードに対し，step3 および PE0 の step1 のループを，オリジナルのループボディに戻したもの．

type3 3.3 節で示した手順のうち，(1) のループ再構成を行わなかったもの．

type4 3.3 節で示した手順をすべて適用したもの．

なお，PE0 以外の step1 における行列計算（図 6 参照）はすべてスカラ局所変数を用いて記述した．

4.1 Livermore Fortran Kernels 19 番ループ

Livermore Fortran Kernels[16] の 19 番ループ（LFK19）の主要部に対する type2 の並列コードの実測結果を**図 13**，**図 14**，**図 15** に示す．計測対象のループもあわせて図中に示す．

図 13 は SR2201 における実測結果で，ループ長と使用 PE 数を変化させたときの実行時間を示している．横軸はループ長，縦軸は経過時間（単位は秒）で，両軸とも対数目盛である．ループ長が大きくなるにつれて，使用 PE 数に応じた曲線が互いに等間隔で平行な直線に近づいている．すなわち，ループ運搬依存のあるループに対し，バルク回帰ループと見なした並列処理によって PE 数に比例する並列性を引き出せることが分かる（PE 数が 2 より大きい場合）．ループ長が小さい範囲では，横軸に平行で等間隔の直線に近づく様子が見られる．ループ長が小さいと PE 数に比例する量の通信オーバヘッドが所要時間を支配している．

なお，SPARC における P スレッドでの実測でも図 13 とほぼ同様な傾向が見られた．ループ長 2^{21}，PE 数 16 にて，実行時間は 23.5 ミリ秒であった．ただしループ長が小さいときには，実行時間に占めるスレッド起動オーバヘッドの割合が大きく，逆に同期待ちはほとんど生じていなかった．実際，1 つのスレッドが起動されて計算処理を開始するまでに 200 マイクロ秒以上，1 回の `pthread_create()` 呼び出しから戻るのに 70 マイクロ秒前後のオーバヘッドがあった．PE 数が 16 の場合には，最後のスレッドが計算処理を開始するまでには最初のスレッドが起動されてから 1 ミリ秒以上かかっていた．1 ミリ秒といえば単一 PE でループ長 2^{15} 程度の LFK19 の計算を終えてしまえる程度の時間である．バルク回帰ループの並列処理による高速化を行うには，LFK19 程度のループボディの演算量が少ない場合には，PE 数に見合う程度にループ長が十分大きいことが必要である．

図 14（SR2201 での実測結果）および図 15（SPARC）

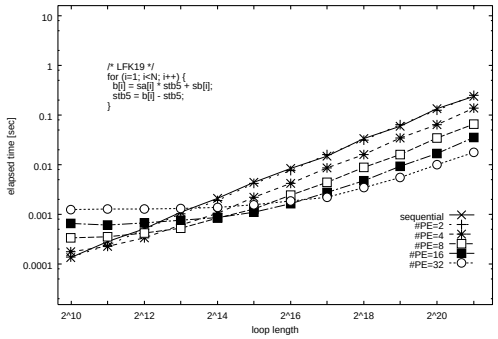


図 13 LFK19 の並列実行結果（SR2201）—実行時間
Fig. 13 Parallel execution of LFK19 (SR2201)—elapsed time.

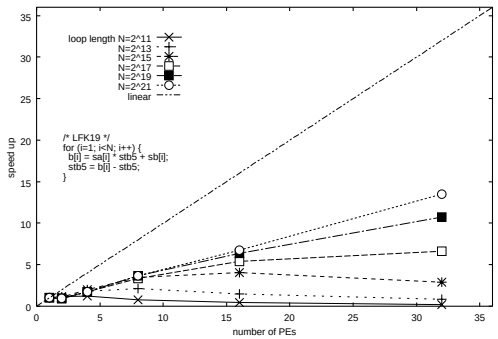


図 14 LFK19 の並列実行結果（SR2201）—高速化率
Fig. 14 Parallel execution of LFK19 (SR2201)—performance.

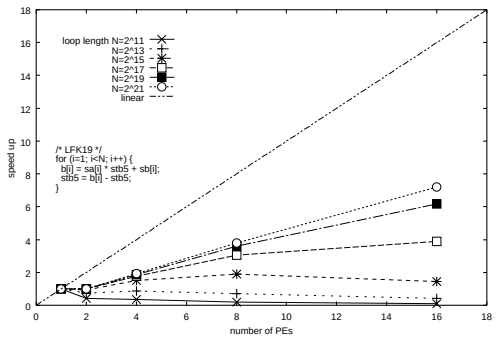


図 15 LFK19 の並列実行結果（SPARC）—高速化率
Fig. 15 Parallel execution of LFK19 (SPARC)—performance.

は，逐次コードの実行時間に対する使用 PE 数に応じた高速化率を，いくつかのループ長の場合についてプロットしたものである．横軸は使用 PE 数，縦軸は高速化率である．ループ長が十分大きい場合には，2 より大きな PE 数に対して線形に高速化可能であることが分かる．SR2201 および SPARC で，ループ長 2^{21}，PE 数 16 における高速化率はそれぞれ 6.8 および 7.2 であった．

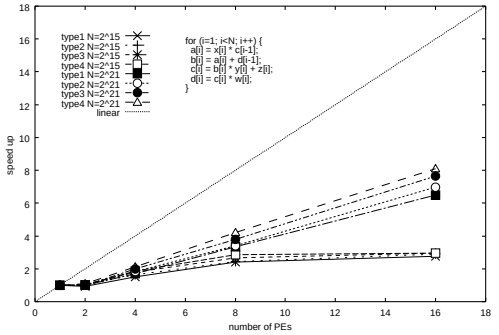


図 16 図 4 (a) の並列実行結果（SPARC）—高速化率
Fig. 16 Parallel execution of Fig. 4 (a) (SPARC)—performance.

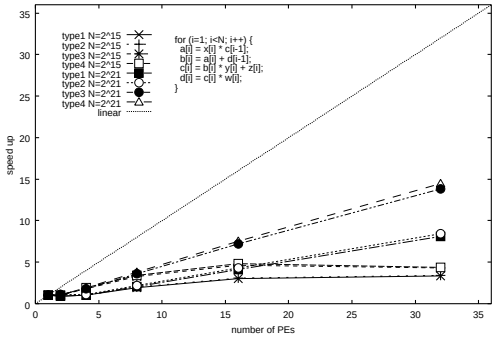


図 17 図 4 (a) の並列実行結果（SR2201）—高速化率
Fig. 17 Parallel execution of Fig. 4 (a) (SR2201)—performance.

なお LFK19 はループボディにおける演算量が少ないループであるため，type1〜type3 のコードの実測結果にはほとんど差が見られなかった．また，type3 と type 4 を差別化するループ再構成処理を適用する余地もない．

4.2 図 4 のループの並列処理

3 章におけるバルク回帰並列処理手法の説明の中で例に用いた図 4 (a) のループに対する実測結果について，PE 数と高速化率の関係を**図 16** および**図 17** に示す．図 16 は SPARC において P スレッドを用いた実測結果で，図 16 では，type1〜type4 のコードによる実行結果を，ループ長 2^{21} および 2^{15} の場合について重ねて示している．ループ長が 2^{21} の場合の高速化率は PE 数にほぼ比例しているといえ，その比例定数は，type1，type2，type3，type4 の順に大きくなっている．type 4 の場合の比例定数は約 0.52 で，ループ長 2^{21}，PE 数 16 のときの実行時間は 49.4 ミリ秒であり，これは逐次実行に対して 8.1 倍，type1 に対して 25%，type3 に対して 5.6%高速化された値である．ループ長 2^{15} では通信オーバヘッドの占める割合

が大きくなり，高速化率は PE 数に比例するには至っていない．ループ長 2^{15} における type 4 での PE 数 16 のときの実行時間は 2.07 ミリ秒であった．

図 17 は SR2201 での PE 数と高速化率の関係である．ループ長 2^{21}，PE 数 16 のときの実行時間は 67.3 ミリ秒であった．これは逐次実行に対して 7.5 倍，type1 および type 3 の実行時間と比較してそれぞれ 82%および 4.9% 高速化された値である．

図 16（SPARC）と図 17（SR2201）とを比較すると，SR2201 では type2 と type3 の間での性能差が非常に大きい．大雑把にいえば，type2，type3 および type4 のコードは，それぞれ 4 元，2 元および 1 元バルク回帰ループの並列実行を行っているのと同等であるので，type2 と type3 の計算量の比は type3 と type4 の比よりもソースコードの字面上でははるかに大きい．その点からすれば，SR2201 での実測結果は計算量の比がそのまま現れているといえる．しかし，type2（および type1）は多数のスカラ変数による行列積演算，すなわち多数の乗算と加算の繰返しでしかも 0 乗算や 0 加算を多く含むため，マシンコード生成の際の最適化によって実際の演算回数は大きく削減されうる．つまり，type2 と type3 の間の性能差が小さいという SPARC での実測結果は，使用したコンパイラの（指定したコンパイルオプションにおける）スカラ最適化の適用の程度が高かったことを示しているともいえる．

図 16 と図 17 の示す高速化率の変化率は，3.3 節での見積り（比例定数 $5p/14$〜$2p/3$）にあてはまる結果である．

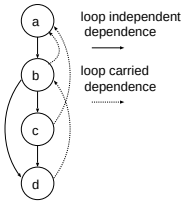
4.3 ループ再構成の適用効果の大きいループ

図 18 (a) に示すループ（対応する依存グラフは図 18 (b)）に対して，バルク回帰並列処理手法を適用して SR2201 で実測した結果を**図 19** および**図 20** に示す．図 19 は PE 数と高速化率の関係を示している．図 18 (a) のループは，3.3 節で述べたループ再構成手法を適用すると，イタレーション間で受け渡される変数の個数が 3 個から 1 個に減る．これは図 4 (a) の場合（2 個から 1 個への削減）に比べて大きい．このため，type3 と type4 の並列コードの実行時間の差が顕著に現れると予想された．図 20 は，ループ長 2^{21}，PE 数 8 の場合の，type1〜type4 のそれぞれの実行における各 PE の処理の進行の様子を示したもので，縦軸は時間の経過を示している．たとえば type1 において PE0 上のスレッドは計算開始後約 0.08 秒で step1 を終え，他の PE が step1 の処理を終わらせる前に開始後 0.18 秒の時点で step3 も終えている．図 20 の縦軸および横軸は，図 3 における横軸および縦軸に相当する．PE7 における step2 の処理時間が長いのは，step1 の処理がないために長時間待たされているためである（図 3 参照）．

図 20 より，type1 を type2 に最適化した効果が step3 および PE0 の step1 に現れていること，type2 から type3 への最適化は PE0 以外の step1 に大きく影響していることが確認できる．また，3.3 節で述べたループ再構成の適用の効果も大きく，ループ再構成が高速化率の向上に大きく寄与していることが分かる．type4 は，PE 数 16 のときに，逐次実行に対して 8.5 倍，type1 に対して 3.4 倍，type3 に対して 1.9 倍の

```
for (i = 1; i ≤ N; i++) {
  a_i = b_i + 3c_{i-1} + w_i;  /* a */
  b_i = a_i + 3d_{i-1} + 7;    /* b */
  c_i = x_i b_{i-1} + y_i;     /* c */
  d_i = z_i b_{i-1} + w_i;     /* d */
}
```

(a) loop code



(b) dependence graph of (a)

図 18 複雑なループの例
Fig. 18 A complicated loop.

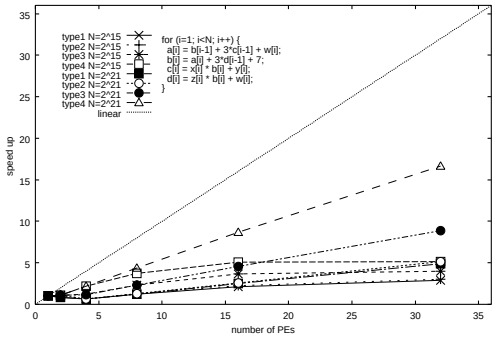


図 19 図 18 (a) の並列実行結果（SR2201）—高速化率
Fig. 19 Result of Fig. 18 (a) (SR2201)—performance.

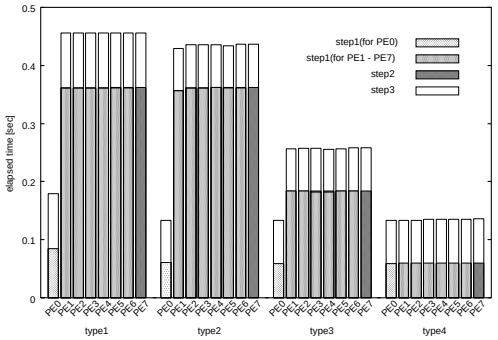


図 20 図 18 (a) の並列実行結果（SR2201）—処理の経過
Fig. 20 Result of Fig. 18 (a) (SR2201)—progression.

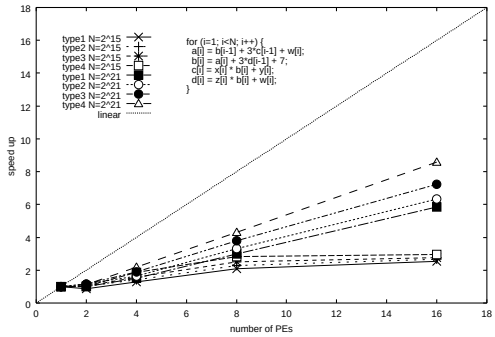


図 21 図 18 (a) の並列実行結果（SPARC）—高速化率
Fig. 21 Result of Fig. 18 (a) (SPARC)—performance.

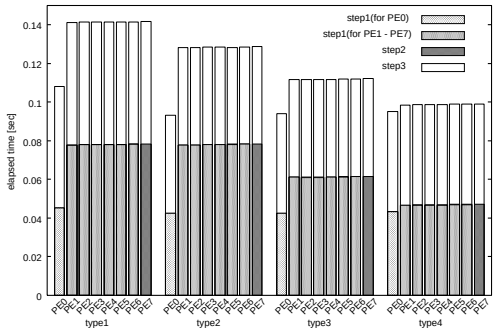


図 22 図 18 (a) の並列実行結果（SPARC）—処理の経過
Fig. 22 Result of Fig. 18 (a) (SPARC)—progression.

速さで実行された．

同じループの SPARC での実測結果は**図 21** および**図 22** に示す．SPARC 上では，PE 数 16 の場合，逐次実行に対して 8.6 倍の速さで実行され，type1 に対して 47%，type3 に対して 19%の速度向上が得られている．type1 から type4 への最適化の過程での処理時間の変化は SR2201 の場合とほぼ同様の傾向を示しているが，全体的に最適化の効果が小さい．この理由は明らかではないが，前節同様，type1〜type3 のコードに対するマシン語生成時の最適化の度合いの違いが原因の 1 つにあげられる．また，バルク回帰ループの高速化率を表す $S_n(p)$（式 (8)）によると，PE 数 p が一定のとき，演算コストに比べてロードやストアのコストが大きい場合（β が大きい場合）にはバルク回帰の元数 n の増加に対する高速化率 $S_n(p)$ の変化はより小さいことがいえる．つまり図 22 に示される最適化の効果の小ささは，$S_n(p)$ から，SMP 並列計算機である SPARC の方が SR2201 よりも，演算コストに対するメモリアクセスコストが相対的に高いことを示唆しているといえよう．

なお，同一 PE 数の際の対逐次実行の高速化率は，SR2201 よりも SPARC の方が，本節で述べたデータを含めこれまでに述べたほとんどの実測結果において大きい．これも，$S_n(p)$ から，SPARC のメモリアクセスコストの相対的な大きさに起因すると説明してもよいかもしれないが，より詳細な検討を要するだろう．

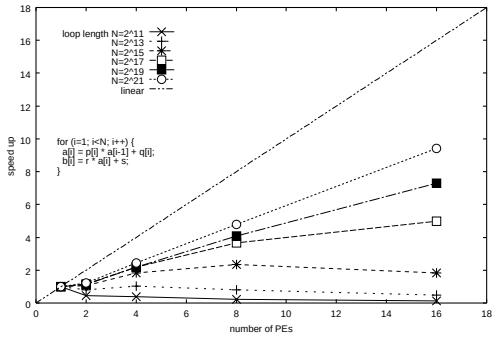


図 23 Doall 成分を含むループ（SPARC）—高速化率
Fig. 23 Doall inclusive loop (SPARC)—performance.

4.4 doall 成分を含むループ

図 23 に，doall 成分を含むループおよびその実測結果を示す．ループ長が十分に大きい範囲で PE 数に比例する高速化が可能であることが分かるが，その比例定数は図 15 や図 16 のものよりも大きく，ループ長 2^{21} においておよそ 0.61 である．なお，同じ環境による図 1 のループの実測では，比例定数は約 0.52 であった．バルク回帰並列処理手法は，doall 成分の並列性を減じることなく，ループ全体を効率良く並列化できるといえる．

5. おわりに

本論文では，バルク回帰並列処理手法と呼ぶ，ループ運搬依存のあるループの並列化手法の提案を行った．いくつかのループ運搬依存のあるループを用い，疎結合計算機および SMP 共有メモリ計算機上で実測を行った結果，いずれの計算機環境においても PE 数に比例した高速化が達成できることが分かった．SR2201 における 8PE での実測では，単純な回帰演算の並列処理手法を適用した場合に対して 3.4 倍に至る速度向上も観測された．

本論文で用いた並列処理アルゴリズムは PE 数に比例する回数の通信処理を行うものであるため，PE 数が増加した際の並列化率の向上には限界がある．通信処理の改良は，明白な改良の余地の 1 つである．

本論文で述べたバルク回帰ループの並列化手法は，もともと逐次的な依存関係のある処理単位間の依存距離を付加的な計算により拡大することによって並列性を抽出するものである．このため，wavefront 法などが適用可能な，インスタンス間の依存関係が疎である

多重ループなどに対しては，バルク回帰並列処理方式の効果は発揮しにくい可能性がある．これについての調査は今後の課題である．依存距離が 1 でない場合（多次バルク回帰）についての考察，より大きなアプリケーションにおける有効性の調査なども，今後の課題としてあげられる．

参 考 文 献

1) Zima, H. and Chapman, B.: *Supercompilers for Parallel and Vector Computers*, Addison-Wesley (1991).
2) Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley (1996).
3) 高畠志泰，本多弘樹，大澤範高，弓場敏嗣：Doacross ループの sandglass 型並列化方法とその評価，情報処理学会論文誌，Vol.40, No.5, pp.2037–2044 (1999).
4) Polychronopoulos, C.D.: Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design, *IEEE Trans. Comput.*, Vol.37, No.8, pp.991–1004 (1988).
5) Egecioglu, O., et al.: Prefix Algorithms for Tridiagonal Systems on Hypercube Multiprocessors, *Proc. 3rd conference on Hypercube concurrent computers and applications*, Vol.2, pp.1539–1545 (1988).
6) 津田孝夫：数値処理プログラミング，第 2 章，岩波書店 (1988).
7) Nicolau, A. and Wang, H.: Optimal Schedules for Parallel Prefix Computation with Bounded Resources, *Proc. 3rd ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp.1–10 (1991).
8) 梅尾博司：SIMD 上の並列アルゴリズム，情報処理，Vol.33, No.9, pp.1042–1055 (1992).
9) Lakshmivarahan, S. and Dhall, S.K.: *Parallel Computing Using the Prefix Problem*, Oxford Univ. Press (1994).
10) Lakshmivarahan, S. and Dhall, S.K.: *Analysis and Design of Parallel Algorithms: Arithmetic and Matrix Problems*, McGraw-Hill (1990).
11) Fisher, A.L. and Ghuloum, A.M.: Parallelizing Complex Scans and Reductions, *Proc. ACM SIGPLAN '94 Conf. Programming Language Design and Implementation* (*PLDI*), pp.135–146 (1994).
12) Ben-Asher, Y. and Haber, G.: Parallel Solutions of Simple Indexed Recurrence Equations, *IEEE Trans. Parallel and Distributed Systems*, Vol.12, No.1 (2001).
13) Dongarra, J.J., et al.: *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM (1991).
14) Wada, H., et al.: High-speed Processing Schemes for Summation Type and Iteration Type Vector Instructions on HITACHI Supercomputer S-820 System, *Proc. Intl. Conf. on Supercomputing*, pp.197–206 (1988).
15) Allan, V.H., et al.: Software Pipelining, *ACM Computing Surveys*, Vol.27, No.3 (1995).
16) Feo, J.T.: An Analysis of the Computational and Parallel Complexity of the Livermore Loops, *J. Parallel Computing*, Vol.7, pp.163–185 (1986).

(平成 13 年 5 月 9 日受付)
(平成 13 年 9 月 19 日採録)

川端 英之（正会員）

1992 年京都大学工学部情報工学科卒業．1994 年同大学大学院工学研究科修士課程修了．同年より広島市立大学情報科学部助手．高性能計算，自動並列化技術に関する研究に従事．ACM，IEEE-CS 各会員．

谷口 宏美

2001 年広島市立大学情報科学部卒業．現在同大学大学院情報科学研究科修士課程在学中．並列アルゴリズムに興味を持つ．

津田 孝夫（正会員）

1957 年京都大学工学部電気工学科卒業．1979 年より京都大学工学部情報工学科教授．1996 年より同大学名誉教授，広島市立大学情報科学部教授．工学博士．モンテカルロ法，自動ベクトル化/並列化コンパイラ，並列数値処理等に関する研究に従事.「モンテカルロ法とシミュレーション」（培風館),「数値処理プログラミング」（岩波書店）等の著書がある．昭和 63 年度および平成 3 年度本会論文賞授賞．ACM，SIAM 各会員．