プログラミング－言語・基礎・実践－ 18－4
（1994. 7. 21）

線形依存ベクトルのループの並列性抽出法

北須賀 輝明†, 城 和貴†, 福田 晃†, 荒木 啓二郎†

†:奈良先端科学技術大学院大学 情報科学研究科

概要

本稿では, 剰余類による分割を用いた, ループ並列化手法と通信を考慮したイタレーションのプロセッサ割当手法を提案する. 本手法の対象とするループは依存ベクトルが定数ではなく線形となる場合である. 既存の手法では依存ベクトルを定数に変換することで並列化されている. そのため, 並列性が犠牲にされ, プロセッサ間通信に対する考慮がなされていない. 本手法では, 剰余類を用いてループを分割し, これを単位として並列化とプロセッサ割当を統一的に行う. 剰余類による分割により, イタレーション数 n に対して $O(log(n))$ の並列化および通信を意識したプロセッサ割当が可能となる.

A Loop Parallelization Technique for Linear Dependence Vector

Teruaki KITASUKA†, Kazuki JOE†, Akira FUKUDA†, Keijiro ARAKI†

†:Graduate School of Information Science,
Nara Institute of Science and Technology
8916–5, Takayama-cho, Ikoma-shi, Nara 630–01, Japan
E–mail: {teruak–k, kazuki–j, fukuda, araki}@is.aist–nara.ac.jp

ABSTRACT

In this paper, we propose loop parallelization technique and iteration alignment technique using cosets. The targeted loops are with linear dependence vector, rather than with constant vector. Early work of loop parallelization with linear dependence vector sacrifices parallelism and do not consider interprocessor communication. The technique we propose is based on cosets. It extracts more parallelism and considers interprocessor communication. The loops of n iterations are parallelized with $O(\log(n))$ steps, which are allocated to processors to reduce interprocessor communication cost.

1 はじめに

近年, 並列計算機に関する研究が盛んになり, 商用の並列計算機も普及しつつある. 並列計算機において, その性能を最大限に引き出すためには, プログラムの並列化が必須である. 並列化コンパイラを用いて並列化の作業を自動化することで, 現在までの逐次プログラム資産を有効利用でき, アーキテクチャに依存したプログラミングを避けることができる.

本稿では, 並列化で最も重要であるループレベルの並列処理に関して, 以下の 2 つの手法を提案する.

- イタレーションを単位とするループの並列化手法
- プロセッサ間通信を抑えるイタレーションのプロセッサ割当

この 2 点は超並列計算機を対象としたプログラムの並列化において重要である. 一般に, 大規模な並列性はプログラムの繰り返し構造によって表現されている場合が多く, 特に科学技術計算では, この部分で実行時間の大部分を費している. また, 並列計算機ではデータ参照の局所性が問題となる. 参照するデータが必要とするプロセッサから離れれば離れるほど, そのアクセスに必要な時間は増加する. これはメッセージパッシング型計算機において顕著であるが, 共有メモリ型に関してもキャッシュなどの点から同様のことが言える. 以上から繰り返し構造の代表としてループを取り扱い, その並列化とデータ参照の局所性を上げる方法を提案する.

本稿で扱うループおよび配列は線形なディオファントス方程式 (2.1.1 節 参照) で依存関係が表されるものとする. ただし, ループイタレーション間の依存ベクトルが定数であるものは除く. 依存ベクトルが定数となる場合についてはすでに多くの研究がなされている [1, 3].

以下, 2 章で依存解析の基本事柄と剰余類の定義を述べる. 3 章以降で本稿で提案する並列化およびプロセッサ割り当て手法について述べる. 3 章では簡単な例を用いて本手法の基本的な考え方を説明し, 4 章で並列化アルゴリズムとプロセッサ割当アルゴリズムを述べる. 5 章でより一般的な例を示し, 6 章で関連研究との比較を行う. 7 章でまとめと今後の課題を述べる.

2 背景

2.1 節で, ループレベルの依存解析の基本的な手法である, ディオファントス方程式と GCD テストについて説明し, 2.2 節では, 以降用いる剰余類の定義を述べる.

2.1 依存解析 [3]

本稿では, 依存関係としてイタレーション間のデータ依存のみを考慮する[1]. よって, 依存解析はループのイタレーション間のデータ依存を調べることになる. 例えば以下のプログラムの配列 A の参照に注目する.

```
      do i = 1,1000
        do j = 1,1000
S :       A(2j+1,i+3) = B(...) - 1
T :       C(...) = A(i+j+3,2i+1)
        enddo
      enddo
```

配列 A は文 S と文 T で参照されている. 配列 A の各要素毎に,

1. 文 S で書き込まれた後, 文 T で読み込まれる配列要素 (フロー依存)
2. 文 T で読み込まれた後, 文 S で書き込まれる配列要素 (逆依存)
3. 文 S で書き込まれた後, 改めて文 S で書き込まれる配列要素 (出力依存)

などの依存関係 (順序関係) があり得る. これらの依存関係を壊すと誤った動作をすることになる. よって, 並列化する際には依存関係の解析が必要である.

また, イタレーション (i,j) で S が参照する (書き込む) A の配列要素と, イタレーション (i',j') が参照する (読み込む) A の配列要素が等しい時,

$$(i'-i, j'-j)$$

を依存ベクトルと呼ぶ. 本稿で扱う線形依存ベクトルは, 依存ベクトルが i, j, i', j' の線形結合で表される依存ベクトルとする.

2.1.1 ディオファントス方程式

ディオファントス方程式とは変数, 係数が共に整数の (連立) 方程式である. 依存解析においては, ループのイタレーション間の依存の存在を調べるために用いられる. 上記のプログラムの例では, 文 S, T で配列 A を参照しており, 同一の配列要素を参照する可能性がある. 同一要素を参照するイタレーションの対 $\{(i,j), (i',j')\}$ は,

$$2j+1 = i'+j'+3,\ i+3 = 2i'+1$$

を満たす. この連立方程式がディオファントス方程式である.

[1] 以降, 依存をデータ依存と同義として使う.

2.1.2 GCD テスト

GCD テストでは, ディオファントス方程式の各変数が整数であることを利用して, この方程式に整数解が存在するか否かを調べる. 整数解が存在しなければ, 依存関係は存在しない. 変数の係数の最大公約数を d, 定数項を c とするとき, d が c を割り切れば整数解が存在し, イタレーション間依存関係が存在し得る[2]. d が c を割り切らなければ整数解は存在せず, 依存関係も存在しない. 先の例ではディオファントス方程式は,

$$2j - i' - j' = 2,\ i - 2i' = -2$$

である. 各方程式について各変数の係数の最大公約数を求めると, $gcd(2,-1,1) = 1$, $gcd(1,2) = 1$ である. 定数項 2, -2 ともに 1 で割り切れるので, 整数解が存在し, 依存関係が存在し得る.

2.2 剰余類

この節では以降で用いる剰余類の定義を述べる.

整数集合 $\mathbf{Z}$ において, $m \in \mathbf{Z}$ を法とする a の剰余類を,

$$a + (m) = \{a + mc \mid c \in \mathbf{Z}\}$$

と定義する. すべての剰余類は以下のいずれかと一致し, 以下のいずれの 2 つも共通部分を持たない.

$$0 + (m),\ 1 + (m), \ldots, m - 1 + (m)$$

それぞれを順に, 以下のように記述する[3].

$$\begin{array}{rl} & m\mathbf{Z},\ m\mathbf{Z} + 1,\ \ldots,\ m\mathbf{Z} + m - 1 \\ \text{または,} & 0(m), 1(m),\ \ \ldots,\ m - 1(m). \end{array}$$

3 基本的なアイデア

この章では, 本稿で提案する並列化手法とプロセッサ割当手法を簡単な例を用いて説明する. この手法は, イタレーション空間を剰余類に分割し, 剰余類間の依存関係を用いる. 例として以下のプログラムを考える.

```
      do i = n, 1, -1
S :       A(i) = A(i) + A(2i)
      enddo
```

[2] ループ範囲を満たす整数解が存在すれば, 依存関係が存在する.

[3] 一般に, m を法として同じ剰余類に属する整数 a, b を $a \equiv b(m)$ と記述する.

3.1 並列化

上記のプログラムを例として, 剰余類を用いた並列化を説明する. まず A の参照についてディオファントス方程式,

$$i = 2j$$

を解くと i, j は整数 t の関数で表される.

$$(i, j) = (g_1(t), g_2(t)) = (2t, t) \qquad t \in \mathbf{Z}$$

これはある t に対して, イタレーション $i = g_1(t) = 2t$ における $A(i)$ と, $j = g_2(t) = t$ における $A(j)$ が同じ配列要素の参照であることを意味する. 解とループ範囲から, 配列 A に関する依存はすべてフロー依存であることが分かる.

ここで, $t \in \mathbf{Z}$ に注意すると, $(i, j) \in (2\mathbf{Z}, \mathbf{Z})$ である. よって以下が成り立つ.

$$\{(i, j)\} \cap (2\mathbf{Z} + 1, \mathbf{Z}) = \Phi$$

つまり, イタレーション $i \in 2\mathbf{Z}+1$ における $A(i)$ への代入は, このループ中で読み込まれない. よって, この条件を満たすイタレーションの集合 $2\mathbf{Z} + 1$ はこのループの最後に実行すればよい. また, I_n に属すイタレーション間には依存関係が存在しないので, すべて並列に実行できる. $I_n = 2\mathbf{Z} + 1$ とする.

では, $g_2(t) = I_n$ を満たす依存関係は, $t \in 2\mathbf{Z} + 1$ から,

$$\{(i, j)\} \supset (4\mathbf{Z} + 2, 2\mathbf{Z} + 1)$$

よって, イタレーション集合 $I_{n-1} = 4\mathbf{Z} + 2$ は I_n の直前に実行すればよいイタレーション集合である. また, I_n 同様に I_{n-1} に属すイタレーションは並列に実行可能である. 以下, 再帰的に集合 I_{n-k} を求めることができる.

$$\begin{array}{c} \vdots \\ I_{n-k} = 2^{k+1}\mathbf{Z} + 2^k \\ \vdots \\ I_{n-1} = 4\mathbf{Z} + 2 \\ I_n = 2\mathbf{Z} + 1 \end{array}$$

それぞれの集合 I_k は, $l < k$ なるすべての I_l の実行が終了した時点で実行を開始できるイタレーションの集合である. ループの総イタレーション数 n に対して, $O(\log_2(n))$ 個の集合に分割される. 各集合のイタレーションの実行が終了する毎に同期を挿入すると仮定した場合, ループを $O(\log_2(n))$ ステップで実行可能である. **doall** を用いると以下のように記述できる.

do $l = \lceil \log_2(n) \rceil, 1, -1$
  doall $i = 2^{l-1}, n, 2^l$
S : $A(i) = A(i) + A(2i)$
  enddoall
enddo

3.2 プロセッサ割当

この節では, 3.1 節と同じプログラム例に対して, プロセッサ間通信を抑えるようにプロセッサ割当を行う. 前節では, 2 の冪乗の剰余類でイタレーションを分割した. 同一の剰余類に属すイタレーションは並列実行できた. よって各剰余類をより大きい法の剰余類でさらに分割し, それをプロセッサ割当の単位とする.

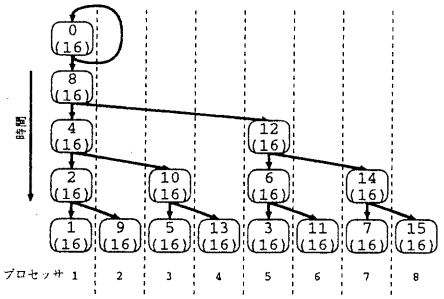

図 1: 法 16 による分割とプロセッサ割当

まず, イタレーション空間全体を $2^4 = 16$ の剰余類を用いて分割すると, 図 1 のようになる. 図の下段から順に $2\mathbf{Z}+1, 4\mathbf{Z}+2, \ldots$ を分割した剰余類の集合となっている. ただし, 0(16) は $32\mathbf{Z}+16, 64\mathbf{Z}+32, \cdots$ の和集合である. また図 1 は, プロセッサ数 8 として, 各剰余類をプロセッサ割り当ての単位と仮定した場合の, 実行時間/プロセッサ間通信ともに最小となる割当を示している.

プロセッサ間の通信量は, 異なるプロセッサに割り当てられ, 互いに依存関係にあるイタレーション対の数とみなせる. 種々の仮定をおくと, 平均の通信量はイタレーション数 n に対して,

$$(1+2+4)\cdot\frac{1}{2}\cdot\frac{1}{16}n = \frac{7}{32}n$$

である.

では, 同じくプロセッサ数 8 でさらに通信を抑えた分割はできないであろうか? 実は可能で, 図 1 において, $1, 9, \ldots, 15(16)$ の列 (最下段) は各プロセッサに分散して割り当てられている. これらのそれぞれと依存関係にある剰余類を求めると, 左から 2, 18, 10, 26, 6, 22, 14, 30(32) であり, 図 1 の下から 2 段目の $2, 10, 6, 14(16)$ と等しいイタレーション集合である. さらにこれらと依存関係にある剰余類を求めると $4, 36, \ldots, 26(64)$ である. ここで, 割り当てられていないイタレーション集合 $8\mathbf{Z}$ を, 64 を法とする剰余類に分割する. この剰余類間の依存関係を求め, 先ほどと同様のプロセッサ割当を行う. この時のプロセッサ割当を図 2 に示す. この場合の平均の通信量は,

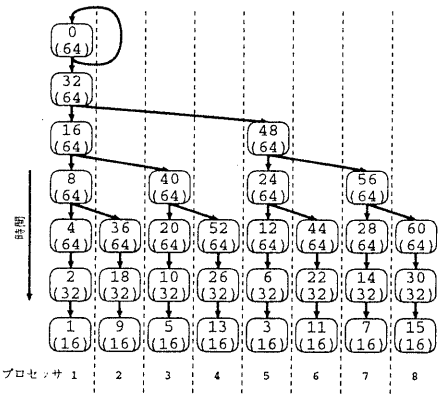

図 2: 16 分割のプロセッサ割当 (その 2)

$$(1+2+4)\cdot\frac{1}{2}\cdot\frac{1}{64}n = \frac{7}{128}n$$

であり, 図 1 の分割に比べて 4 分の 1 の通信量である. このように再分割を k 回することで, 通信量は $1/2^k$ になる. コンパイル時にイタレーション回数 n がわかれば,

$$m^k > n$$

を満たす m^k を法とする剰余類になるまで分割することで, プロセッサ間通信を必要とするイタレーション対の数は 0 となる.

4 アルゴリズム

4.1 並列化アルゴリズム

並列化アルゴリズムを述べる. 次のようなループを対象とし, 条件を満たせばイタレーション数 n に対して $O(\log(n))$ の並列化を行う.

do $i_1 = \cdots$
  do $i_2 = \cdots$
    $\vdots$

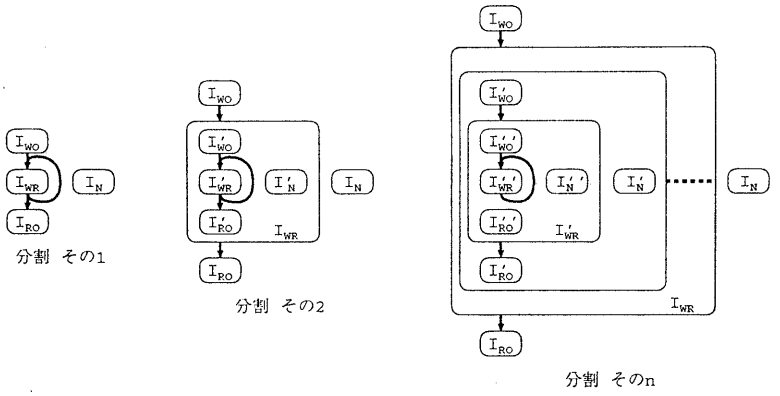

図 3: 並列化のための分割

do $i_n = \cdots$
S : $A(f_W(i)) = \cdots$
T : $\cdots = \cdots A(f_R(i)) \cdots$
enddo
$\vdots$
enddo
enddo

ただし, $i = (i_1, i_2, \ldots, i_n)^t \in \mathbf{Z}^n$ とする. また, A を m 次元配列として, f_W, f_R を i の線形関数でかつ, m 次元整数関数とする. このループに対して, 以下のアルゴリズムを適用する. 分割の様子を図 3 に示す.

1. ディオファントス方程式

$$f_W(i_W) = f_R(i_R)$$

を解く. 解は, $n'(2n - m \leq n' \leq 2n - 1)$ 次元ベクトル $t \in \mathbf{Z}^{n'}$ を媒介変数として,

$$i_W = i_W(t),\ i_R = i_R(t)$$

と表される. 以降, 文 S, T 間の依存関係がすべてフロー依存であることが確認されたものと仮定する[4].

2. $t \in \mathbf{Z}^{n'}$ とすると, i_W, i_R それぞれの集合は,

$$I_W = \{i_W(t)\} = \begin{pmatrix} a_1\mathbf{Z} + c_{W1} \\ a_2\mathbf{Z} + c_{W2} \\ \vdots \\ a_n\mathbf{Z} + c_{Wn} \end{pmatrix}$$

$$I_R = \{i_R(t)\} = \begin{pmatrix} b_1\mathbf{Z} + c_{R1} \\ b_2\mathbf{Z} + c_{R2} \\ \vdots \\ b_n\mathbf{Z} + c_{Rn} \end{pmatrix}$$

(ただし, $a_i, b_i, c_{Wi}, c_{Ri} \in \mathbf{Z}$) と表せる. I_W は, I で読み込まれる配列要素に書き込むイタレーションの集合である. I_R は, I で書き込まれる配列要素に読み込むイタレーションで集合ある.

ここで, m_0 を $a_1, \ldots, a_n, b_1, \ldots, b_n$ の最小公倍数とする. また, $I = \mathbf{Z}^n, m = m_0$ とする. $m_0 = 1$ ならばこのアルゴリズムで分割できないので終了.

3. $I_W, I_R \subset I$ をもとに, I を以下の 4 つに分割する.

$$I_N = \overline{I_W} \cap \overline{I_R}$$
$$I_{WO} = I_W \cap \overline{I_R}$$
$$I_{WR} = I_W \cap I_R$$
$$I_{RO} = \overline{I_W} \cap I_R$$

I_N に属すイタレーションは, I のどのイタレーションとも依存関係にない. よって, I の実行時の任意の時点で実行できる. I_{WO} に属すイタレーションは, I_R のイタレーションで読み込まれる配列要素に書き込む. よって, 依存関係がフロー依存であることから, I_{WO} は I_{WR}, I_{RO} に先立って実行される必要がある. 同様に I_{RO} は I_{WO}, I_{WR} の実行終了後に実行されねばならない. I_{WR} のイタレーション間は, 依存関係が存在するので, このままでは, 逐次的にしか実行で

[4] S, T の間がすべて逆依存の場合にも同様にこのアルゴリズムを適用できる. フロー依存と逆依存が混在する場合には適用できない. これらの違いはループ変数の範囲によって起こる.

きない. 以上の順序関係を 図 3 (分割 その 1) に示す.

4. I_{WR} のイタレーションをさらに分割する. I_{WR} のイタレーションのうち I_{WO}, I_{RO} と依存関係にあったイタレーションは, 3. で依存関係を保証されたので, その依存関係は考慮する必要がない. よって, I_{WR} 内で依存関係があるイタレーションを求める.

$$
\begin{aligned}
m &\leftarrow m \times m_0 \\
I &\leftarrow I_{WR} \\
I_W &\leftarrow \{i_W(t) \mid i_R(t) \in I_{WR}\} \cap I_{WR} \\
I_R &\leftarrow \{i_R(t) \mid i_W(t) \in I_{WR}\} \cap I_{WR}
\end{aligned}
$$

として 3. へ行く. 再帰の様子を図 3 に示す.

上記の式で, I_W, I_R を求める際の注意点として, I_{WR} を各次元の法が m の剰余類の集合として扱う.

このアルゴリズムによって, イタレーション空間 $\mathbf{Z}^n$ が分割され, その間の依存関係が得られる. 総イタレーション数を n とすると, $O(\log_{m_0}(n))$ 個の集合に分割される. 実行は, 依存関係にしたがって集合間に同期をはさみながら, $O(\log_{m_0}(n))$ で実行される. 同じ集合に属するイタレーションは互いに依存関係がないため, 並列に実行可能であることに注意されたい.

このアルゴリズムの 3. 4. の繰り返しは I_{WR} に含まれるイタレーション数が与えられた数を下回った時点で終了するものとする. この際に最後の I_{WR} に属すイタレーションは逐次的に実行する必要がある.

4.2 プロセッサ割当アルゴリズム

通信を意識したプロセッサ割当アルゴリズムを述べる. このアルゴリズムは前節の並列化アルゴリズムによる分割を基に, 相互に依存関係のないイタレーション集合に分割することを目的とする.

1. 並列化アルゴリズム 1. 2. 3. を用いると, m_0 が定まり, I が $I_{WR}, I_{WO}, I_{RO}, I_N$ に分割される. $m = m_0$ とする. I_N の割当については特に考慮しない.

2. イタレーション部分空間

$$I_{WO},\ I_{RO}$$

をそれぞれ m を法とする剰余類に分割して, それぞれをプロセッサ割当の単位とする. I_{WO} のイタレーション間には依存関係がないので, これを分割した各剰余類間にも依存関係はない. よって, 各剰余類をプロセッサ割当の単位とすると, プロセッサ間通信なしに I_{WO} のイタレーションを実行できる. I_{RO} についても同様.

分割した剰余類の個数がプロセッサ数に満たなければ, 法を m_0 倍して 2. をやり直す.

3. 2. で分割した各剰余類について, 依存関係にある剰余類を求める[5]. 剰余類 $I_w \subset I_{WO}$ と依存関係にある剰余類 I_{wr} は 式 (1) で求められる. $I_r \subset I_{RO}$ についても同様 (式 (2)).

$$I_{wr} = \{i_R(t) \mid i_W(t) \in I_w\} \cap I_{WR} \quad (1)$$

$$I_{rw} = \{i_W(t) \mid i_R(t) \in I_r\} \cap I_{WR} \quad (2)$$

I_w と I_{wr} を同じプロセッサに割り当てる. これにより, I_w, i_{wr} 間の依存関係はプロセッサ内で起こり, プロセッサ間通信を起こさない. I_r と I_{rw} についても同様.

$$
\begin{aligned}
I_w &\leftarrow I_{wr} \\
I_r &\leftarrow I_{rw} \\
I_{WR} &\leftarrow I_{WR} \cap \overline{I_{rw}} \cap \overline{I_{wr}}
\end{aligned}
$$

として, 3. を再帰的に実行する.

並列化アルゴリズムとの違いは, 並列実行できる剰余類の集合をさらにプロセッサ数まで分割することである. これにより, 依存関係をプロセッサ内に閉じ込めて, プロセッサ間通信を削減できる.

3. の再帰は, 並列化アルゴリズム同様, I_{WR} に含まれるイタレーション数が充分小さくなった時点で終了するものとする. 実行時には I_{WR} の実行前後でのみプロセッサ間通信 (同期) が必要になる.

5 例

以下のプログラムを例にとり, 上記アルゴリズムを適用する. ただし, イタレーション間の依存関係は配列 A に関してのみ存在し, すべてフロー依存とする.

```
      do i = ···
         do j = ···
S           A(i+j+6, 2i-j+2) = ···
T           ··· = A(2i+j, i+j)
         enddo
      enddo
```

並列化.1. 配列 A の参照 $S(i,j)$ と $T(i',j')$ に関するディオファントス方程式は,

$$
\begin{aligned}
i + j + 6 &= 2i' + j' \\
2i - j + 2 &= i' + j'
\end{aligned}
$$

[5]法は限定しない.

であり, その解は,

$$i_W(t) = \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix},$$
$$i_R(t) = \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} -t_1 + 2t_2 + 4 \\ 3t_1 - 3t_2 - 2 \end{pmatrix}$$

並列化.2. $t \in \mathbf{Z}^2$ より, $I_W = \{i_W(t)\}, I_R = \{i_R(t)\}$ は,

$$I_W = \begin{pmatrix} \mathbf{Z} \\ \mathbf{Z} \end{pmatrix}, \quad I_R = \begin{pmatrix} \mathbf{Z} \\ 3\mathbf{Z}+1 \end{pmatrix}$$

よって以降では, $m = 3$ の冪乗を法とする剰余類を用いて並列化およびプロセッサ割当をする.

並列化.3-1. I_W, I_R より, I を分割する.

$$I_{WO} = \left\{ \begin{pmatrix} \mathbf{Z} \\ 3\mathbf{Z} \end{pmatrix}, \begin{pmatrix} \mathbf{Z} \\ 3\mathbf{Z}+2 \end{pmatrix} \right\}$$
$$I_{WR} = \begin{pmatrix} \mathbf{Z} \\ 3\mathbf{Z}+1 \end{pmatrix}$$
$$I_N = I_{RO} = \Phi$$

イタレーション全体の 2/3 は I_{WO} に属し, 残る 1/3 は I_{WR} に属す. この分割による実行順序は, まず最初に I_{WO} のイタレーションをすべて並列に実行でき, 次に I_{WR} のイタレーションを逐次実行することになる. 図 4 左参照 (濃い部分の剰余類が I_{WR} に属す).

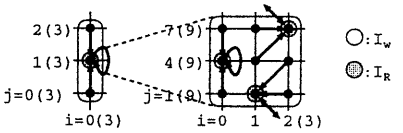

図 4: 並列化

並列化.4-1. I_{WR} から並列化実行できる部分を取り出すべく, さらに分割する. まず, I_{WR} を各次元の法を $3(= m)$ とする剰余類に分割する.

$$I_{WR} = \left\{ \begin{pmatrix} 3\mathbf{Z} \\ 3\mathbf{Z}+1 \end{pmatrix}, \begin{pmatrix} 3\mathbf{Z}+1 \\ 3\mathbf{Z}+1 \end{pmatrix}, \begin{pmatrix} 3\mathbf{Z}+2 \\ 3\mathbf{Z}+1 \end{pmatrix} \right\}$$

これを基に, 新たな I, I_W, I_R を求める.

$$m \leftarrow m \times m_0$$

$$I \leftarrow \left\{ \begin{pmatrix} 3\mathbf{Z}+a \\ 3\mathbf{Z}+1 \end{pmatrix} \middle| a \in \{0,1,2\} \right\}$$
$$I_W \leftarrow \left\{ \begin{pmatrix} 3\mathbf{Z}+a \\ 3\mathbf{Z}+1 \end{pmatrix} \middle| a \in \{0,1,2\} \right\}$$
$$I_R \leftarrow \left\{ \begin{pmatrix} 3\mathbf{Z}-a \\ 9\mathbf{Z}+3a+4 \end{pmatrix} \middle| a \in \{0,1,2\} \right\}$$

並列化.3-2. I_W, I_R より,

$$I_{WO} = \text{省略.}$$
$$I_{WR} = \left\{ \begin{pmatrix} 3\mathbf{Z}-a \\ 9\mathbf{Z}+3a+4 \end{pmatrix} \middle| a \in \{0,1,2\} \right\}$$
$$I_N = I_{RO} = \Phi$$

I_{WO} は I のイタレーションの 2/3, つまりイタレーション全体の 2/9 が第 2 ステップに並列実行できることになる (図 4 右参照.).

以下これを繰り返すと, イタレーション数を n とするとき, 各ステップ $i(> 0)$ で $(2/3^i)n$ のイタレーションが実行できる. よって, ループ全体は $O(\log_3(n))$ ステップで実行可能となる.

プロセッサ割当.1. および 2. $3(= m_0)$ を法とする剰余類に I_{WO}, I_{RO} を分割する.

$$I_{WO} = \left\{ i_w(a,b) \middle| \begin{matrix} a \in \{0,1,2\} \\ b \in \{0,2\} \end{matrix} \right\}$$
$$= \left\{ \begin{pmatrix} 3\mathbf{Z}+a \\ 3\mathbf{Z}+b \end{pmatrix} \middle| \begin{matrix} a \in \{0,1,2\} \\ b \in \{0,2\} \end{matrix} \right\}$$
$$I_{RO} = \Phi$$

I_{WO} の 6 つの剰余類が並列実行可能である. 簡単のため, プロセッサ数 6 の場合を考える.

プロセッサ割当.3. I_{WO} の剰余類 $I_w(a,b)$ と依存関係にある剰余類 $I_{wr}(a,b)$ は,

$$I_{wr}(a,b) = \{g_R(t) \mid g_W(t) \in I_w(a,b)\}$$
$$= \begin{pmatrix} 3\mathbf{Z}-a+2b+4 \\ 9\mathbf{Z}+3a-3b-2 \end{pmatrix}$$

であり, いずれも I_{WR} に属す. a, b が等しい $I_w(a,b)$, $I_{wr}(a,b)$ を同じプロセッサに割り当てることで, この間の通信をプロセッサ内通信とすることができる. 実際に a, b を代入すると,

$a =$	0	1	2
$b = 0$	(1(3),7(9))	(0(3),1(9))	(2(3),4(9))
$b = 2$	(2(3),1(9))	(1(3),4(9))	(0(3),7(9))

となる. この剰余類それぞれについて, 法を 9 とする剰余類に分割して, さらに依存関係にある剰余類を求める. ここまでのプロセッサ割当を図 5 に示す. I_{WR} はプロセッサ 1 で逐次的に実行される.

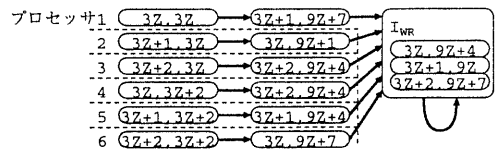

図 5: プロセッサ割当

6 関連研究

本稿と同種のイタレーション間依存を扱った論文として, T.Z.Tzen [2] らの研究がある. この研究では, ループ変数の線形関数で表される依存ベクトルを定数ベクトルの和の形式に変換することで, 並列化を行う. 以下のプログラムの変換過程を図 6 に示す.

do $i = 1, 1000$
  do $j = 1, 1000$
S : $A(i+j, 3i+j+3) = \cdots$
T : $\cdots = A(i+j+1, i+2j+4) \cdots$
  enddo
enddo

この手法を適用すると, ウェーブフロント手法などを用いて, 総イタレーション数 n の 2 重ループを $O(n^{1/2})$ の時間で実行できる.

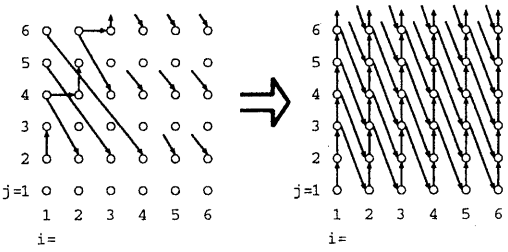

図 6: イタレーション間の依存グラフの変換

この変換の長所は既存の定数依存ベクトルの並列化手法の適用が可能なことである. 短所は内在する並列性を犠牲にしている点, およびプロセッサ間通信が考慮されていない点である. この 2 つの短所は実際のプログラムの実行時間に大きく影響するものと思われる.

7 おわりに

本稿では, 剰余類を用いて,

- 処理時間をイタレーション数 n に対し, $\log(n)$ オーダとするループ並列化手法と,
- 通信を削減するプロセッサ割り当て

について述べた. この手法が適用できる条件は以下の 3 つである.

- ディオファントス方程式が線形方程式である. ただし, ディオファントス方程式の解を媒介変数ベクトル t の関数として与えた場合, 解が t の多項式で表されるものについては, 自然な拡張が可能と思われる.
- ディオファントス方程式の係数のうち同一ではないものが存在する.
- 依存関係がすべてフロー依存, またはすべて逆依存である.

ことである. 今後の課題として, 以上の条件の緩和と以下の 3 つが挙げられる.

- 複数の依存関係を持つループを扱う.
- 依存ベクトルが定数の場合に対する従来の方法との親和性の検討.
- 複数のループネストをまたがったイタレーションの分割.

謝辞

貴重な討論を頂いた Dr. Utpal Banerjee に感謝致します.

参考文献

[1] E.H.D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Trans. on Parallel and Distributed Systems*, 3(4):465–476, July 1992.

[2] T.H.Tzen and L.M.Ni. Dependence uniformization : A loop parallelization technique. *IEEE Trans. on Parallel and Distributed Systems*, 4(5):547–558, May 1993.

[3] U.Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic, 1988.