ループ文アクセス・パターン解析における畳込み演算

朝倉宏一† 渡邉豊英†

ループ文の並列実行には，ループ文中で配列データがどのように操作されるかの解析が重要である．現在までにも，ループ文全体の実行における配列データの操作領域を解析する手法が多く提案されてきた．しかし，これらの手法では操作する領域を解析できるが，領域中の配列要素の操作順序などを解析することはできない．我々は，ループ文中の配列操作の順序を表現可能なアクセス・パターン表現を提案している．アクセス・パターンはループ文全体の実行で操作される配列データの操作領域だけでなく，ループ文の実行にともない操作領域がどのように変形，移動するかを表現可能としている．本稿では，アクセス・パターン解析アルゴリズムについて述べる．ネストしたループ文においてアクセス・パターンを解析するための畳込み演算を定義する．2つのアクセス・パターンが畳込み可能である条件を示し，畳込みによって得られるアクセス・パターンの計算方法について述べる．畳込み演算を定義することで，ネストしたループ文に対するアクセス・パターンの解析が可能となり，ループ文の漸進処理を適用することができる．

Convolution Calculation for Access Pattern Analysis in Nested Loops

Koichi Asakura† and Toyohide Watanabe†

For loop parallel processing, it is very important to analyze array manipulations in loops. Until today, many analysis methods have been proposed, in which the access region of array data in the whole loop execution is calculated. However, these methods can achieve only the access region analysis but do not take the access order of array elements into account. We have already proposed an access pattern representation in order to describe the access order of array manipulations in loops. With the access pattern representation, the movement and transformation of access region of array data according to value change of a loop variable can be analyzed correctly. In this paper, we propose an algorithm for calculating the access patterns. In order to calculate access patterns correctly in nested loops, a convolution calculation is introduced. We show conditions for application of the convolution calculation to two access patterns, and algorithm for calculating combined access patterns for nested loops. By introduction of the convolution calculation, we can apply the incremental parallel processing to nested loops appropriately.

1. はじめに

プログラム並列化処理では，プログラム中のループ文を対象にする手法が一般的である．これは，プログラム並列化処理の対象となるプログラムにおいては，プログラムの実行に時間を要する部分はループ文であることが多く，ループ文を並列化することで効率良い並列処理を達成できることが多いからである．ループ文を効率良く並列実行させるため，現在までに様々なループ文並列化手法が提案されている[1)~3)]．我々は先に，ループ文並列化手法としてループ文の漸進処理手法を提案した[4),5)]．ループ文の漸進処理は，同じ配列データを共有している複数のループ文をパイプライン的に並列実行させる手法である．複雑なループ運搬依存を有するループ文など，従来の手法では並列化が困難であったループ文を並列実行させることができる．

ループ文の漸進処理では，ネストしたループ文において，最外ループ文のイタレーションを並列処理の最小単位としている．そして，ループ文に対して漸進処理を適用するためには，ループ文における配列データの操作領域の解析だけでなく，ループ文がその配列データをどのような順序で操作しているかの解析が必要となる．配列データの操作順序を解析することによって，同じ配列データを類似した操作順序で操作する複数のループ文を，漸進処理の適用によりパイプライン的に並列実行可能となる．現在までにも，ループ文で操作される配列データに関する解析として様々な手法やモデルが提案されている．伝統的な手法としては，ループ文全体を実行したときに操作される配列デー

† 名古屋大学大学院工学研究科情報工学専攻
Department of Information Engineering, Graduate School of Engineering, Nagoya University

タの要素を不等式で表す手法がある[6]．不等式表現では，操作される配列要素が存在する領域を，空間を囲む不等式の集合で表現することで，ループ文における配列データの操作の解析を幾何問題に帰着している．また，複雑な配列添字式を持つ配列操作を解析可能なAccess Region[7]やLMAD（Linear Memory Access Descriptor)[8]なども提案されている．しかし，これらの手法はすべて配列データの操作領域のみの解析を目指したものであり，ループ文の実行にともなう操作領域の移動，変形を解析の対象としてはいない．もちろん，配列データのプライベート化やループ文間依存解析などに対しては，ループ文全体の実行における配列データの操作領域の解析で十分である．しかし，我々が提案したループ文の漸進処理では，ループ文における配列データの操作領域の解析はもちろんのこと，最外ループ文のイタレーションの実行において配列データの要素をどのような順序で操作しているかの解析が必要である．つまり，従来の手法で得られる解析情報は，ループ文の漸進処理に対して十分でない．

我々は，ループ文における配列データの操作領域や操作順序を解析しモデル化するために，アクセス・パターン表現を提案している[9]．アクセス・パターンについては次の2点を考慮しなければならない．1つは，アクセス・パターンで表現される情報を用いることにより，漸進処理をどのように適用できるかである．もう1つは，アクセス・パターンをどのように計算するか，またその計算手法のプログラムに対する適用範囲である．文献5)では，漸進処理の適用可能性解析に必要なイタレーション依存比などの情報を，アクセス・パターンより計算する手法を示した．本稿では，ループ文のアクセス・パターンを計算する手法について述べる．特に，漸進処理において重要となるネストしたループ文のアクセス・パターンを解析するための，アクセス・パターンの畳込み演算について述べる．ネストしたループ文に対してアクセス・パターンを解析するためには，ネストした個々のループ文のアクセス・パターンを計算し，それらを融合する処理が必要となる．本稿では，ネストした2つのループ文のアクセス・パターンが畳込み可能な条件，および畳込み後のアクセス・パターンの計算アルゴリズムについて述べる．

2. アクセス・パターン解析

アクセス・パターンは，ループ文における配列データの操作状況をモデル化したものである．ループ文の実行中に配列データのどのような領域をどのような順序で操作しているかを解析することを指向している．本章では，アクセス・パターンを解析するために必要な畳込み演算について述べる．まず，アクセス・パターンにおける配列操作のモデル化方法について述べる．そして，アクセス・パターンを計算するための，アクセス・パターン間の演算について述べる．

2.1 配列操作モデル

アクセス・パターンでは，Access RegionやLMADでの手法と同様に，ループ文における配列データの操作位置を表す記法として，配列データの先頭要素からのオフセット値を用いる．配列データの要素の位置を先頭要素からのオフセット値で表現することにより，解析対象である配列データの次元数に関係なく統一的に操作位置を指示することができる．

一般に，ループ文における配列データの操作位置は，ループ文の実行に従って移動する．操作領域の移動をモデル化するとき，移動をとらえる粒度とモデルの複雑さがトレード・オフの問題として発生する．つまり，個々の配列操作に対する移動をモデル化するか，イタレーションなどある一定回数の操作に対して移動をモデル化するかにより，表現されるアクセス・パターンの精密性と複雑性が変化する．たとえば，個々の配列操作をモデル化すれば，アクセス・パターンとして表現される情報の精密性は向上するが，モデルは複雑になる．また，Access RegionやLMADでは，ループ文実行中の操作領域の移動をモデル化しないことでモデルを簡潔化しているが，そのため操作領域の移動などの情報は得られない．

我々のアクセス・パターンでは，ループ文の漸進処理における解析のために配列操作をモデル化することが目的である．ループ文の漸進処理では，配列データを共有する複数のループ文を，最外ループ文のイタレーション単位でパイプライン的に並列処理する[5]．そこで，我々のアクセス・パターンでは，最外ループ文の1イタレーションの実行における配列データの操作領域と，その操作領域がループ文の実行にともないどのように移動するか，の情報によりループ文における配列データの操作をモデル化する．

2.2 アクセス・パターン表現

我々は，図1のように正規化されたプログラムにおけるアクセス・パターンを以下のように定義する[9]．

$$(Var, Start, Stride, Num, Velocity, Accuracy)_{ub}$$

ここで，

Var ⋯ 最外ループ文のループ変数，
$Start$ ⋯ ループ文の実行で最初に操作される位置，
$Stride$ ⋯ 操作位置の間隔，
Num ⋯ イタレーションで操作する要素の個数，

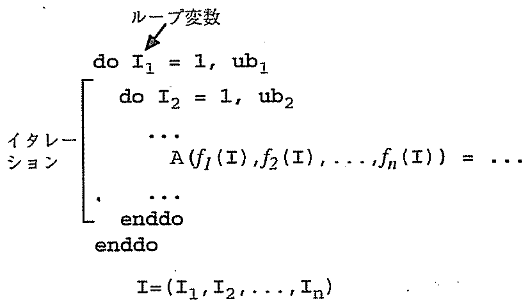


図 1　解析対象のループ文の形式
Fig. 1　Example of target loop.

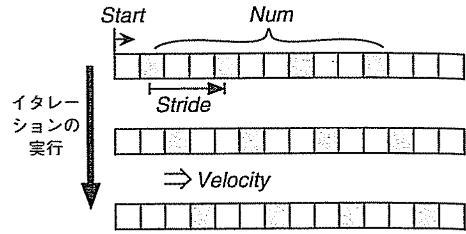


図 2　アクセス・パターンの模式図
Fig. 2　Brief example of access pattern.

Velocity ･･･　操作領域の移動速度，
Accuracy ･･･　操作の確実性，
ub ･･･　ループ変数の最大値，
である☆．ただし，配列データの先頭アドレスを 0 としている．最外ループ文の 1 イタレーションの実行において操作される配列要素は，*Stride* と *Num* でモデル化されている．1 イタレーションで操作される配列要素の個数を *Num* で，配列要素の規則的な並びを *Stride* でそれぞれ表現し，1 イタレーションで操作される配列データの操作領域を規定する．そして，*Stride* と *Num* で規定された操作領域が，ループ文の実行に従って *Velocity* で示される速度で移動するとして，ループ文全体の操作領域をモデル化している．また，*Accuracy* は *Stride*, *Num*, *Velocity* で表現される操作領域の正確性を表すパラメータであり，*MAY* か *MUST* のどちらかの値をとる．通常は *MUST* をとるが，ループ文中で実行される配列を操作する文が if 文などにより不確定である場合，*MAY* とすることでその不確定さを表現可能としている．

図 2 にアクセス・パターンにおける各パラメータの意味を模式的に示す．図中の正方形は配列要素を表しており，正方形の横の並びが解析対象になっている配列データを表している．ハッチがかかっている配列要素はイタレーション中で操作されていることを表している．縦方向が時間の経過を表しており，ループ文の実行が進むにつれて操作される配列要素が変化することを表している．最初に操作される配列要素が *Start* で，操作要素間の間隔が *Stride* で，操作要素数が *Num* で，それぞれ示される．そして，ループ文の実行による操作領域の移動は，各イタレーションにおいて最初に操作される操作要素間の間隔として，*Velocity* パラメータで特徴づけられることが分かる．

ループ文においてアクセス・パターンが計算されると，アクセス・パターンのパラメータよりループ文の漸進処理の適用可否を判断することができる．漸進処理では 2 つのループ文において，それぞれのイタレーションをどのような割合で実行できるかを表すイタレーション依存比の計算が重要である[5]．前ループ文，後ループ文のアクセス・パターンをそれぞれ $A_{pred}(Var_p, St_p, Str_p, Num_p, Vel_p, Acc_p)_{ub_p}$, $A_{succ}(Var_s, St_s, Str_s, Num_s, Vel_s, Acc_s)_{ub_s}$ とすると，2 つのループ文のイタレーション依存比は

$$Vel_s : Vel_p$$

と計算することができる．そして，他のパラメータより漸進処理開始点までのイタレーション実行回数 i_{wait} を計算することで，漸進処理の適用可否の判断，および同期タイミングを決定することができる．

2.3　アクセス・パターン間の演算

アクセス・パターンを計算するとき，ネストしたループ文の扱いや，複数の配列操作の扱いが重要である．単純なプログラム例を除けば，一般に実プログラムにおけるアクセス・パターンの計算は難しい．ネストしたループ文に対してアクセス・パターンを計算するとき，内側ループ文のアクセス・パターンをまず計算し，それを元に外側ループ文のアクセス・パターンを順次計算することができれば，アクセス・パターンの計算が容易になる．また，1 文中に複数の配列操作が存在する場合も，それぞれの配列操作に対してアクセス・パターンを計算し，それらを合成できれば，アクセス・パターンの計算が容易になる．このようなアクセス・パターン間の演算を定義することで，ネストの深さや配列操作の個数にかかわらず，つねにアクセス・パターンが解析可能となる．我々は，ネストしたループ文において，内側ループ文のアクセス・パターンを外側ループ文のアクセス・パターンに融合し，外側ループ文のアクセス・パターンを計算することを，畳込み演算と定義する．また，複数の配列操作に対してそれ

☆ 文献 5) においてもアクセス・パターンを提案したが，二次元配列に特化した形式であった．本稿のアクセス・パターンは様々な配列データに対する配列操作を表現可能なように改良したものである．

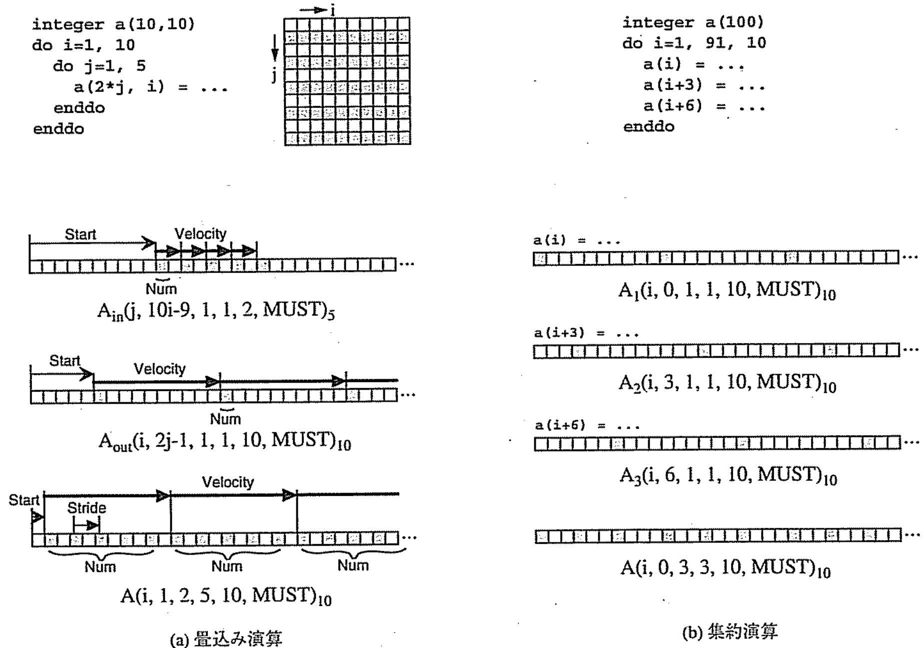


図 3　アクセス・パターン間の演算
Fig. 3　Calculations between access patterns.

ぞれのアクセス・パターンを合成し，1つのアクセス・パターンを計算することを集約演算と定義する．

図 3 にそれぞれの演算の例をあげる．図 3 (a) は二重ループ文で配列を操作しているプログラムである．このプログラムの配列添字式より，配列データの操作位置を先頭アドレスからのオフセット値で表すと $2 \times j + 10 \times (i-1) - 1 = 10i + 2j - 11$ となる．ここで，内側ループ文だけに注目してアクセス・パターンを解析する．ループ文の最初の実行で操作される要素は，配列添字式に $j = 1$ を代入することで得られ，$10i - 9$ となる．また，イタレーション内で実行される文は 1 文のみなので，1 イタレーションで操作される要素は 1 つのみである．さらに，配列添字の j の係数より，内側ループ文の実行にともなう操作領域の移動が 2 であることが分かる．したがって，内側ループ文のみに注目したアクセス・パターンは A_{in} のように表される．同様に，外側ループ文だけに注目し，内側ループ文が存在しないとして計算されたアクセス・パターンは A_{out} のように表される．畳込み演算は，2 つのアクセス・パターン A_{in}, A_{out} から，二重ループ文に対するアクセス・パターン A を計算するための演算である．同様に図 3 (b) に複数の配列操作が存在するループ文を示す．配列操作はそれぞれ A_1，A_2，A_3 のようなアクセス・パターンを有する．集約演算は，複数のアクセス・パターンより，ループ文のアクセス・パターン A を計算するための演算である．

アクセス・パターンの計算には上記 2 つの演算が重要であるが，本稿ではアクセス・パターンの畳込み演算に焦点を当て，畳込み演算の適用可否判定やアクセス・パターンの計算法についてのべる．アクセス・パターンの集約演算は今後の課題である．

3. アクセス・パターンの畳込み演算

2 章でも述べたように，我々のアクセス・パターンではイタレーションで操作される配列データを *Stride* と *Num* で，またその移動を *Velocity* でモデル化している．ネストしたループ文のアクセス・パターンを考えるとき，外側ループ文の 1 イタレーションの実行での配列データの操作領域は，内側ループ文全体の実行における操作領域となる☆．したがって，内側ループ文のアクセス・パターンが外側ループ文のアクセス・パターンに畳込み可能となるためには，内側ループ文のアクセス・パターン全体で操作される配列データの領域を，外側ループ文のアクセス・パターン中の *Stride* と *Num* の 2 つのパラメータで記述可能でなければならない．すなわち，内側ループ文全体の実行で操作される配列データの要素が，*Stride* と *Num* の 2 つ

☆ 簡単化のため，ここではパーフェクト・ループ文を対象としている．

のパラメータで表現可能となるよう，規則的に並んでいなければならない．そこで，ネストしたループ文においてアクセス・パターンを解析するとき，2つのアクセス・パターンが畳込み可能であるか否かの判定が必要となる．

以下，内側ループ文のアクセス・パターンが外側ループ文のアクセス・パターンに畳込み可能となる条件を示し，ネストしたループ文に対するアクセス・パターンの計算アルゴリズムについて述べる．

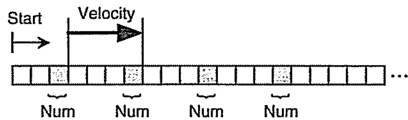
3.1 畳込み演算アルゴリズム

アクセス・パターンが畳込み可能となる条件，すなわちアクセス・パターンで表現される操作領域中の配列データ要素が，*Stride* で表すことのできる一定の間隔で並んでいる条件は以下の3通りである．

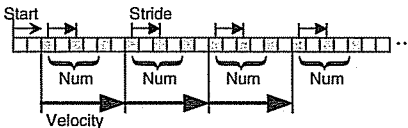
(1) 内側ループ文の1イタレーションで操作される配列要素が1つだけのとき．

(2) 内側ループ文で $Velocity \geq Stride$ のとき，*Velocity* が *Stride* の整数倍となっており，かつ領域が連続する十分な配列要素を操作しているとき．

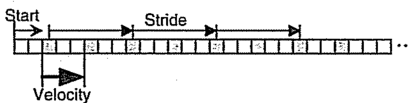
(3) 内側ループ文で $Velocity < Stride$ のとき，*Stride* が *Velocity* の整数倍となっており，かつ領域が連続する十分な配列要素を操作しているとき．



(a) Num=1 のとき



(b) Velocity ≧ Stride のとき



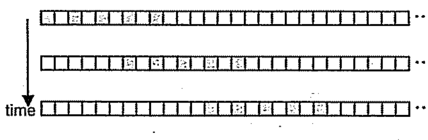
(c) Velocity ＜ Stride のとき

図4 畳込み可能条件
Fig. 4 Possible conditions for convolution calculation.

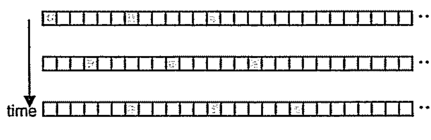
条件(1)は自明である．図4(a)で示すように，1イタレーションで操作される配列要素が1つだけのとき，ループ文全体の実行で操作される配列要素は *Velocity* の間隔で位置している．また，そのときの配列データの要素数は，ループ変数の上限値 *ub* となる．したがって，*Velocity* の値を *Stride* とし，*ub* の値を *Num* とすれば，ループ文全体で操作される配列データの要素集合を *Stride* と *Num* で表現できる．

次に，図4(b)により条件(2)を考える．1イタレーションの実行で *Stride* の間隔で並ぶ *Num* 個の配列要素が操作される．そして，ループ文の実行が進むにつれ，*Velocity* だけ操作位置が移動する．このような操作領域が *Stride* と *Num* の2つのパラメータで表現可能となるためには，*Velocity* が *Stride* で割り切れ，すべての配列要素が等間隔で並んでいる必要がある．また，*Velocity* が *Stride* で割り切れたとしても，配列要素が隙間なく並んでいなければならない．そのため，$Stride \times Num \geq Velocity$ という関係が成立していなければならない．この関係が成立していないループ文では，操作される配列要素が等間隔とならない．また，このときループ文全体で操作される配列要素数は，図5(a)で示すとおりに計算できる．つまり，ループ変数が1から $ub-1$ までの部分での要素数は $\frac{Velocity}{Stride}$ であり，ループ変数が *ub* の部分での要素数は *Num* である．したがって，ループ文全体で操作される配列要素数は

$$\frac{Velocity}{Stride} \times (ub-1) + Num$$



(a) Velocity ≧ Stride のとき



(b) Velocity ＜ Stride のとき

図5 操作される配列要素の個数
Fig. 5 Number of manipulated array elements.

と計算することができる．

条件(3)は，条件(2)において $Velocity$ と $Stride$ の大小関係が異なるだけであるので（図4(c) 参照），畳込み可能となる条件は条件(2)と同様である．また，操作される配列要素数も同様に

$$\frac{Stride}{Velocity} \times (Num - 1) + ub$$

と計算できる（図5(b) 参照）．

これらより，畳込み演算アルゴリズムは以下のように定義される．

[アクセス・パターンの畳込み演算アルゴリズム]
二重ループ文において，内側ループ文と外側ループ文のアクセス・パターンをそれぞれ，

内側： $A_{in}(Var_1, St_1, Str_1, Num_1, Vel_1, Acc_1)_{ub_1}$
外側： $A_{out}(Var_2, St_2, Str_2, Num_2, Vel_2, Acc_2)_{ub_2}$

とする．このとき，内側ループ文のアクセス・パターンが外側ループ文のアクセス・パターンに畳込み可能である条件，および畳込みの結果計算されるアクセス・パターン A は以下のようである．

(1) $Num_1 = 1$ であるとき．

$$A(Var_2, St'_2, Vel_1, ub_1, Vel_2, Acc'_2)_{ub_2}$$

(2) $Vel_1 \geq Str_1$ で，Vel_1 が Str_1 の整数倍であり，かつ $Str_1 \times Num_1 \geq Vel_1$ であるとき．

$$A(Var_2, St'_2, Str_1, \tfrac{Vel_1}{Str_1} \times (ub_1 - 1) + Num_1, Vel_2, Acc'_2)_{ub_2}$$

(3) $Vel_1 < Str_1$ で，Str_1 が Vel_1 の整数倍であり，かつ $Vel_1 \times Num_1 \geq Str_1$ であるとき．

$$A(Var_2, St'_2, Vel_1, \tfrac{Str_1}{Vel_1} \times (Num_1 - 1) + ub_1, Vel_2, Acc'_2)_{ub_2}$$

ここで，

$$St'_2 = St_1 + \begin{cases} Str_1 \times (Num_1 - 1) & (Str_1 < 0) \\ 0 & (\text{それ以外}) \end{cases} + \begin{cases} Vel_1 \times (ub_1 - 1) & (Vel_1 < 0) \\ 0 & (\text{それ以外}) \end{cases}$$

で $Var_1 = 1$ を代入した値であり，

$$Acc'_2 = \begin{cases} MUST & (Acc_1 = Acc_2 = MUST) \\ MAY & (\text{それ以外}) \end{cases}$$

である．

3.2 畳込み演算の例

前節に述べた畳込み演算アルゴリズムの動作例を図6にあげる．図6(a)は二次元配列データを全走査するプログラムである．内側ループ文のアクセス・パターンにおいて $Num = 1$ となるので，この内側ループ文のアクセス・パターンは外側ループ文のアクセス・パターンに畳込み可能である．アルゴリズムに従ってアクセス・パターンを計算すると，$A(i, 0, 1, 10, 10, MUST)_{10}$ となり，プログラムでの操作順序が正確に解析できていることが分かる．図6(b)のように操作順序が異なったプログラムに対してもアクセス・パターンは解析可能である．どちらのプログラムも同じ操作領域を有するが，操作順序の違いがパラメータにより正確に表現されている．

複雑な配列操作のプログラム例として，波頭（wave-front）型に配列を操作するプログラムを図6(c)に示す．これも内側ループ文のアクセス・パターンにおいて $Num = 1$ であるので，条件(1)により畳込み可能である．図6(d)は三次元配列データを操作する三重ループ文のプログラム例である．内側の二重ループ文に関しては他のプログラム例と同様に条件(1)により畳込み可能となり，内側二重ループ文のアクセス・パターンが $A_{kj}(j, 100i - 99, 2, 5, 10, MUST)_{10}$ と解析されている．そして，最外ループ文への畳込みに関しては，条件(2)を満たしており畳込み可能である．したがって，最終的にアクセス・パターンが $A(i, 1, 2, 50, 100, MUST)_{10}$ と正確に計算されている．

以上に示したように，我々が提案した畳込み演算アルゴリズムによって，ネストしたループ文に対するアクセス・パターンが正確に解析可能であることが分かった．最後の例のように，内側ループ文から順次畳込み演算を施すことで，ネストの数に関係なく様々なループ文においてアクセス・パターンを計算することができる．

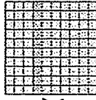
4. 評価実験

本章では，アクセス・パターンの畳込み演算が実プログラム中に現れる配列操作に対してどの程度適用可能かを評価するために実施した実験について述べる．

4.1 実験内容

本実験は NAS ベンチマーク・プログラム逐次版（NPB2.3-serial）に含まれる配列操作に対してアクセス・パターンを解析した．畳込み演算アルゴリズムの評価のため，本実験ではそれぞれの配列操作が独立のループ文中に存在するとした．すなわち，図7に示すように複数の配列操作が存在するプログラムの場合，

```
integer a(10,10)
do i=1,10
  do j=1,10
    a(j,i) = ...
  enddo
enddo
```



内側アクセス・パターン

$A_j(j, 10i\text{-}10, 1, \underline{1}, \underline{1}, MUST)_{10}$ ← 畳込み可能

外側アクセス・パターン

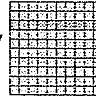
$A_i(i, j\text{-}1, 1, 1, 10, MUST)_{10}$

アクセス・パターン

$A(i, 0, 1, 10, 10, MUST)_{10}$

(a) 二重ループ文

```
integer a(10,10)
do i=1,10
  do j=1,10
    a(i,j) = ...
  enddo
enddo
```



内側アクセス・パターン

$A_j(j, i\text{-}1, 1, \underline{1}, \underline{10}, MUST)_{10}$ ← 畳込み可能

外側アクセス・パターン

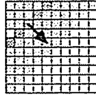
$A_i(i, 10j\text{-}10, 1, 1, 1, MUST)_{10}$

アクセス・パターン

$A(i, 0, 10, 10, 1, MUST)_{10}$

(b) 異操作順序の二重ループ文

```
integer a(10,10)
do i=1,10
  do j=1,i
    a(i-j+1,j) = ...
  enddo
enddo
```



内側アクセス・パターン

$A_j(j, i\text{-}1, 1, \underline{1}, \underline{9}, MUST)_i$ ← 畳込み可能

外側アクセス・パターン

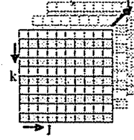
$A_i(i, 9j\text{-}9, 1, 1, 1, MUST)_{10}$

アクセス・パターン

$A(i, 0, 9, i, 1, MUST)_{10}$

(c) 波頭ループ文

```
integer a(10,10,10)
do i=1,10
  do j=1,10
    do k=1,5
      a(2*k,j,i) = ...
    enddo
  enddo
enddo
```



k ループ文アクセス・パターン

$A_k(k, 100i\text{+}10j\text{-}109, 1, \underline{1}, \underline{2}, MUST)_5$ ← 畳込み可能

j ループ文アクセス・パターン

$A_j(j, 100i\text{+}2k\text{-}101, 1, 1, 10, MUST)_{10}$

内側二重ループ文アクセス・パターン

$A_{kj}(j, 100i\text{-}99, \underline{2}, \underline{5}, \underline{10}, MUST)_{10}$ ← 畳込み可能

i ループ文アクセス・パターン

$A_i(i, 10j\text{+}2k\text{-}11, 1, 1, 100, MUST)_{10}$

アクセス・パターン

$A(i, 1, 2, 50, 100, MUST)_{10}$

(d) 三重ループ文

図 6 畳込み演算の例
Fig. 6 Examples of convolution calculation.

独立なループ文にそれぞれの配列操作が存在したプログラムであるととらえ，アクセス・パターンの畳込み演算を適用した☆．また，プログラムにはループ文内の定数伝搬処理，変数の展開処理などを事前に施した．アクセス・パターンの解析には，我々が現在開発中の並列化コンパイラ・ツールキットを用いた．並列化コンパイラ・ツールキットは SUIF[10] を元に開発されており，並列化コンパイラの開発を容易にするためプログラム中の文間の依存関係の解析などが可能となっている[11]．本実験では，並列化コンパイラ・ツールキッ

☆ したがって，本実験は NPB2.3-serial に対する提案手法の直接の有効性を示すものではない．実プログラムによく出現する配列操作に対して提案手法がどの程度適用可能かを示すために，例として NPB2.3-serial に出現する配列操作を利用した実験である．

```
integer a(10,10)

do i=1, 10
  do j=1, 10
    a(j,i) = a(j,i) + a(i,j)
  enddo
enddo
```

⇒

```
integer a(10,10)

do i=1, 10
  do j=1, 10
    a(j,i) = ...
  enddo
enddo

do i=1, 10
  do j=1, 10
    ... = a(j,i)
  enddo
enddo

do i=1, 10
  do j=1, 10
    ... = a(i,j)
  enddo
enddo
```

図 7 プログラムの変換
Fig. 7 Transformation of program.

表 1　実験結果
Table 1　Experimental result.

プログラム名	総配列操作数	解析成功数	完全解析	部分解析	解析失敗数	while ループ	添字式規定外	上限・下限値	可変サイズ
BT	1698	1698 100.0%	439	1259	0 0.0%	0	0	0	0
CG	54	52 96.3%	52	0	2 3.7%	0	0	2	0
EP	10	10 100.0%	10	0	0 0.0%	0	0	0	0
IS	53	40 75.5%	40	0	13 24.5%	0	3	0	10
LU	1648	1581 95.9%	1102	479	67 4.1%	0	0	0	67
MG	247	122 49.4%	122	0	125 50.6%	49	6	0	70
SP	1392	1392 100.0%	513	879	0 0.0%	0	0	0	0
合計	5102	4895 95.9%	2278 46.5%	2617 53.5%	207 4.1%	49	9	2	147

注釈
whileループ … whileループ文中の配列操作でループ変数が存在しない
添字式規定外 … 配列添字式が複雑で解析対象外
上限・下限値 … ループ文の上限値，あるいは下限値が不定
可変サイズ … 配列サイズが不定

表 2　部分解析結果の詳細
Table 2　Detailed results of partially analyzed loops.

BT	1	2	3	4	5
1	43				
2	0	39			
3	0	1202	320		
4	0	15	27	37	
5	0	3	6	6	0

LU	1	2	3	4	5
1	50				
2	0	877			
3	0	406	145		
4	0	44	24	30	
5	0	0	5	0	0

SP	1	2	3	4
1	43			
2	0	132		
3	0	678	301	
4	0	195	6	37

（網掛け）… 完全解析
□ … 部分解析

トのプログラム解析部を用いて，アクセス・パターンを解析するプログラムを実装した．

実験では，アクセス・パターンの解析率について調査した．プログラムに畳込み演算を適用するとき，3 種類の結果が生じる．すなわち，最外ループ文に対するアクセス・パターンが解析できる場合（完全解析）と，途中のネストまでアクセス・パターンが解析できる場合（部分解析）と，アクセス・パターンがまったく解析できない場合（解析失敗）である．プログラム中の全配列操作に対して上記の場合の割合を調査することで，畳込み演算アルゴリズムの適用可能性を評価した．

4.2 実験結果

表 1 に実験結果を示す．NPB2.3-serial の各プログラムに対するアクセス・パターンの解析結果を示している．表では，各プログラム内の配列操作の総数，解析成功数，解析失敗数が示されている．また，解析成功数の内訳として完全解析数と部分解析数が，解析失敗数の内訳として失敗の原因が，それぞれ示されている．解析失敗数，すなわちアクセス・パターン解析の対象外となる配列操作は 4.1%であり，ほとんどの配列操作がアクセス・パターンを解析可能であることが分かる．解析失敗となったのは，ループ文の上限・下限値が変数で不定であったり，配列のサイズが不定であったりするといったコンパイル時の静的解析では対処できないものや，while ループ文など解析の対象外であるものがほとんどである．配列添字式が複雑で解析できなかったものは，`a(b(i))` のような添字式に配列が存在するもののみであった．このことより，静的解析で対処可能な配列操作はほぼ解析可能であることが分かる．解析成功の中では，BT，LU，SP 以外のプログラムではすべて完全解析されている．部分解析の存在するプログラムに対する解析結果の詳細を表 2 に示す．表 2 ではプログラム中のネスト・ループ文において解析できたネスト数が示されている．縦方向の数字がループ文が何重ネストであるかを，横方向の数字が解析できたネスト数を，それぞれ表している．対角線部分の数字は完全解析となったループ文の数を，それ以外は部分解析となったループ文の数を，それぞれ表している．これを見ると，まったく畳込みが適用できない，すなわち横方向の数字が 1 である欄に属する部分解析となった配列操作文は存在せず，1 重ループ文を除くすべての配列操作で畳込み演算が適用できたことが分かる．また，この部分解析の多くは 5.2 節で述べる *Accuracy* パラメータを用いた対処により完全解析可能であり，これらの実験結果より，アクセス・パターンの畳込み演算はほとんどの配列操作に対して適用可能であることが分かる．

5. 議　論

5.1 関連研究

ループ文において操作される配列データに対する解析として，我々のアクセス・パターンに類似したものに LMAD がある[8]．LMAD は複雑な添字式を有する配列操作を持つループ文を実行したときの操作領域を解析するために開発された．ループ文全体で操作される領域を解析することで配列データのプライベート化を図り，ループ文間の並列性を抽出することを目的としている．したがって，ループ文の全体の実行中の

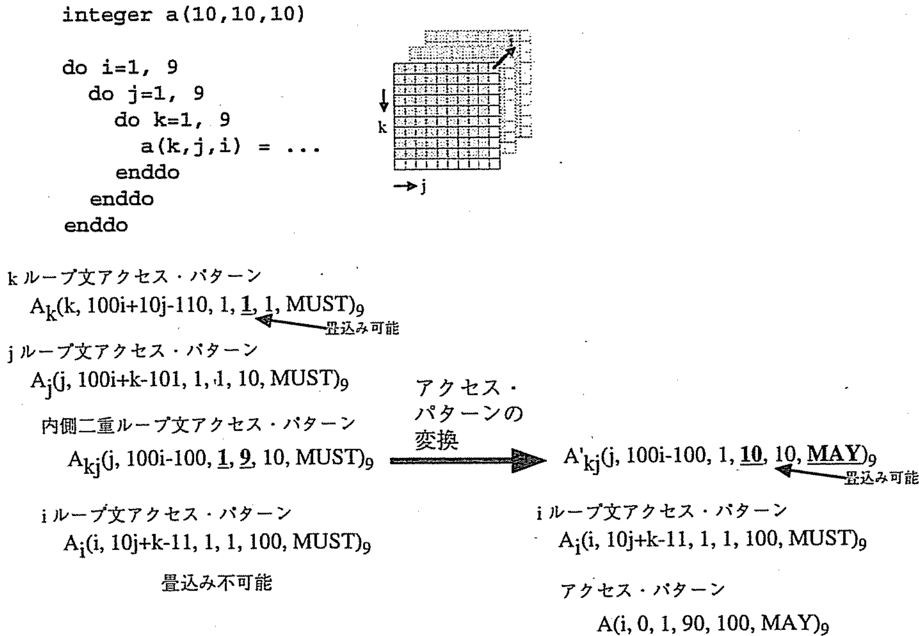


図 8　Accuracy パラメータを用いたアクセス・パターン変換
Fig. 8　Access pattern transformation with *Accuracy* parameter.

操作領域のみをモデル化し，その配列データ要素の操作順序はモデル化の対象としていない．LMAD では，*span* と *stride* のパラメータの組を複数用いることで，配列データの操作領域をモデル化している．*span* により領域の幅，すなわち操作領域の両端間の距離を表し，*stride* によりその領域内の操作される配列要素の間隔を表している．これは，我々のアクセス・パターンにおいて 1 イタレーションで操作される領域を *Stride* と *Num* で表現しているのと同様な考え方である．多くの場合，

$$span = Stride \times (Num - 1)$$

と表現することができ，我々のアクセス・パターンは LMAD と同様の表現力を有している．

LMAD は操作領域内の操作順序をモデル化していない．すなわち，ループ文中での操作順序に関係なく，操作する領域が同じであれば，そのループ文に対する LMAD は同値となる．たとえば，図 6 (a)，(b) のようなループ文の場合，LMAD はそれぞれ $A_{10}^{1}\,{}_{100}^{10}+0$，$A_{100}^{10}\,{}_{10}^{1}+0$ となる．このとき，同値の定義によりこれら 2 つの LMAD は同一のものとして計算可能となっている．このように LMAD では配列操作を簡略化してモデル化している．この簡略化のため，LMAD では我々のアクセス・パターンの畳込み演算と同様な coaleseable 演算のほか，集約演算も定義可能となっている．これに対し，アクセス・パターンでは操作領域内の配列データの操作順序を表現するための *Velocity* パラメータが導入されており，同様な配列領域を持つループ文でも様々なアクセス・パターン表現が考えられる．そのため，畳込み可能な条件も LMAD と比較すると複雑になっている．しかし，畳込み演算の定義により，LMAD と同様にネストしたループ文に対する解析が可能となっている．

つまり，まとめると，LMAD はループ文全体の実行における配列データの操作領域を解析するために開発され，*span* と *stride* のパラメータの組により構成されている．シンプルなモデルなので表現力は弱いが，様々な演算が定義されており，複雑な配列添字式が解析可能となっている．それに対して，我々のアクセス・パターンはループ文の実行にともなう配列データの操作領域の移動を解析可能なように拡張されている．そのため，演算は複雑になっているが，アクセス・パターンの解析に最低限必要な畳込み演算が定義されており，ループ文中の配列操作をより柔軟にモデル化することができる．

5.2 部分解析に対する対処

実験においては，解析成功数のうち約 50%が部分解析であった．部分解析とは，途中のネストまで畳込み演算が適用できたが，最外ループ文のアクセス・パターンは計算できなかったものを指す．これら部分解析となった配列操作の多くは，配列データの一部分の

みを操作し，配列領域が連続しないことが原因であった．つまり，図8に示すようにループ文の上限値や下限値が配列データ全体を操作しないプログラムの場合，畳込み演算適用判定において $Str \times Num \geq Vel$ や $Vel \times Num \geq Str$ が成立せず，アクセス・パターンを計算することができないことが，部分解析の主な原因であった☆．

ここで，アクセス・パターンの *Accuracy* パラメータを用い，アクセス・パターンを概略的に表現することで，アクセス・パターンが計算可能となる．すなわち，アクセス・パターンに対して以下の変換規則を用いることで畳込み演算を適用することが可能となる場合がある．

- $A(Var, St, Str, Num, Vel, Acc)_{ub}$
  $\downarrow$
  $A'(Var, St, Str, Num', Vel, MAY)_{ub}$
  ただし，$Num' = Num + \alpha (\alpha > 0)$

同様に，*Stride*, *Velocity* パラメータに対しても以下の変換規則が定義可能である．

- $A(Var, St, Str, Num, Vel, Acc)_{ub}$
  $\downarrow$
  $A'(Var, St, Str', Num', Vel, MAY)_{ub}$
  ただし，Str'は Str の約数,$Num' = Num \cdot \frac{Str}{Str'}$
- $A(Var, St, Str, Num, Vel, Acc)_{ub}$
  $\downarrow$
  $A'(Var, St, Str, Num, Vel', MAY)_{ub'}$
  ただし，Vel'は Vel の約数,$ub' = ub \cdot \frac{Vel}{Vel'}$

これらの演算により，部分解析となった配列操作に対してアクセス・パターンを解析することが可能となる．図8の例のように，内側のアクセス・パターンを

$$A_{kj}(j, 100i - 100, 1, 9, 10, MUST)_9$$
$$\downarrow$$
$$A'_{kj}(j, 100i - 100, 1, 10, 10, MAY)_9$$

と変換することで，最外ループ文のアクセス・パターンが $A(i, 0, 1, 90, 100, MAY)_9$ と計算できる．

しかし，これらの変換規則を適用することによりアクセス・パターンの正確性が減少するので，変換規則の適用は状況により判断する必要がある．*Accuracy* パラメータを用いたアクセス・パターンの変換と，その適用範囲については今後の課題として十分に検討する必要がある．

☆ 図のような配列操作は，配列の境界と内部での計算方法が異なる場合や，袖領域を用いた場合などによく現れる．

6. おわりに

本稿では，ループ文の漸進処理適用のためのアクセス・パターンの計算手法として畳込み演算アルゴリズムを示した．アクセス・パターンはループ文における配列データの操作領域だけでなく，ループ文実行中の操作領域の移動をモデル化している．畳込み演算は，ネストしたループ文中の内側ループ文のアクセス・パターンを外側ループ文のアクセス・パターンに融合し，ネストしたループ文におけるアクセス・パターンを計算する演算である．アクセス・パターンが畳込み可能である条件を示し，畳込みの結果得られるアクセス・パターンの計算アルゴリズムについて述べた．畳込み演算を定義したことで，ネストしたループ文でのアクセス・パターンが解析可能となった．

2.3 節で述べたように，畳込み演算のほか，アクセス・パターンの解析において重要な演算に集約演算がある．集約演算は，同一ループ文内に複数存在する配列操作やループ文のアクセス・パターンを融合したアクセス・パターンを計算する演算である．本稿では，簡単化のためパーフェクト・ループ文を対象としたが，並列化の対象となるプログラムではループ文中に複数のループ文が存在することが多い．集約演算を定義することで，パーフェクト・ループ文でないループ文のアクセス・パターンも解析することができ，より広範囲のプログラムに対して柔軟な解析が可能となる．実際に実プログラムにおいて漸進処理を適用するためには，ループ文中に複数存在する同じ配列データに対する配列操作をまとめて1つのアクセス・パターンとして表現しなければならないので，集約演算が必要となる．したがって，アクセス・パターンに対して集約演算を定義することが今後の課題である．また，我々のアクセス・パターンは配列添字式としてループ変数の一次多項式を対象としているが，LMAD は複雑な配列添字式を解析対象としている．複雑な配列添字式を対象とするためのアクセス・パターンの拡張も今後の課題としてあげられる．

謝辞　日頃よりご指導いただいている本学大学院工学研究科・稲垣康善教授，鳥脇純一郎教授，ならびに中京大学大学院情報科学研究科・福村晃夫教授，名城大学理工学部・杉江昇教授に深く感謝いたします．また，熱心に討論していただいた研究室の皆様に感謝いたします．さらに，有用な助言をいただいた査読者の方々に感謝いたします．

参考文献

1) Padua, D.A. and Wolfe, M.J.: Advanced Compiler Optimizations for Supercomputers, *Comm. ACM*, Vol.29, No.12, pp.1184–1201 (1986).
2) Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company (1991).
3) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
4) Sanda, K., Asakura, K. and Watanabe, T.: Access Patterns Analysis between Intertask-dependent Arrays, *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol.VI, pp.2746–2752 (1999).
5) 三田勝史，朝倉宏一，渡辺豊英：配列データを共有したループ文の並列実行のための漸進処理手法，情報処理学会論文誌，Vol.42, No.4, pp.847–859 (2001).
6) Banerjee, U.: *Loop Transformations for Restructuring Compilers — The Foundations*, Kluwer Academic Publishers (1993).
7) Paek, Y., Hoeflinger, J. and Padua, D.: Access Regions: Toward a Powerful Parallelizing Compiler, Technical Report 1508, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev. (1996).
8) Paek, Y., Hoeflinger, J. and Padua, D.: Simplification of Array Access Patterns for Compiler Optimizations, *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pp.60–71 (1998).
9) 朝倉宏一，渡辺豊英：ループ文アクセス・パターン表現による配列操作モデル，2001 年度電気関連学会東海支部連合大会，p.294 (2001).
10) Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S.-W., Tseng, C., Hall, M., Lam, M. and Hennessy, J.: SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers, *SIGPLAN Notices*, Vol.29, No.12, pp.31–37 (1994).
11) 朝倉宏一，渡辺豊英：並列化コンパイラ・ツールキットにおけるプログラム並列化処理とタスク生成処理について，情報処理学会第 62 回全国大会論文集，Vol.1, pp.169–170 (2001).

(平成 14 年 1 月 30 日受付)
(平成 14 年 5 月 21 日採録)

朝倉 宏一（正会員）

1970 年生．1992 年名古屋大学工学部情報工学科卒業．1994 年同大学院工学研究科情報工学専攻博士・前期課程修了．1995 年同大学院工学研究科情報工学専攻博士・後期課程中途退学．同年同大学院工学研究科情報工学専攻助手．計算機クラスタ環境，並列計算機環境におけるシステム・ソフトウェアに興味を持つ．

渡邉 豊英（正会員）

1948 年生．1972 年京都大学理学部卒業．1974 年同大学院工学研究科数理工学専攻修士課程修了．1975 年同博士課程中途退学．同年京都大学大型計算機センター助手．1987 年名古屋大学工学部情報工学科助教授．現在同大学大学院工学研究科情報工学専攻教授．京都大学工学博士．統合化環境，分散協調環境，データベース環境，データベースの高度インタフェース，知的 CAI，文書理解，地図理解に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，人工知能学会，ACM，IEEE Computer Society，AAAI 各会員．