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

三田勝史[†] 朝倉宏一[††] 渡邉豊英[††]

ループ文の並列実行では，イタレーション単位の並列実行が基本となっている．しかし，複雑なループ伝搬依存が存在するループ文では，イタレーション間の並列性を抽出することは困難である．また，計算機クラスタ環境では，タスク間通信オーバヘッドなどによりイタレーション単位の細粒度並列処理は適さない．本稿では，ループ文の並列実行方式である漸進処理を提案する．漸進処理では，同一配列データを操作する複数のループ文をパイプライン的に実行させる．ループ文はそれぞれ並列実行されるが，各ループ文内のイタレーションは逐次的に実行されるので，複雑なループ伝搬依存が存在しても適用可能である．また，イタレーションでなく，ループ文全体がタスク生成の単位となるので，タスク起動のオーバヘッドを削減でき，計算と通信のオーバラップにより通信オーバヘッドを減少させることができる．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要素の操作順序を解析する必要がある．我々はループ文での操作順序をアクセス・パターンとして定義し，漸進処理を効率的に行うためにアクセス・パターンの類似性を解析する漸進処理適用判定アルゴリズムを開発した．また，様々なループ文に対して漸進処理適用判定アルゴリズムを適用し，アルゴリズムの有効性についても調査した．

Incremental Processing for Parallel Execution of Loops with Shared Array

Katsushi Sanda,[†] Koichi Asakura[††] and Toyohide Watanabe[††]

Traditionally, in parallel execution of loops, iterations are executed in parallel. However, we can not extract parallelism among iterations for the loop which has complex loop-carried dependences. In addition, iteration-based parallel execution is not suitable to the computer cluster environment because of task granularity. In this paper, we propose the incremental processing for parallel execution of loops. In the incremental processing, plural loops which access same array data are executed in parallel with the pipeline method. We can apply incremental processing to loops with complex loop-carried dependences since incremental processing makes full use of parallelism among loops and iterations in loops are executed sequentially. Furthermore, loops are executed in parallel by means of overlap between computation and communication, which reduces communication overhead among tasks. For incremental processing, analysis of the access order on array elements in the loop is very important. We introduce the access pattern which defines the access order in loops, and develop the algorithm for analyzing the similarity of access patterns. We also examine the effectivity of our algorithm for parallel execution of loops with shared arrays.

1. はじめに

近年，様々なプログラム並列化手法が提案されているが，その多くは Fortran の DO ループ文や C 言語の for ループ文などの繰返し構文に対する並列化手法である．イタレーション間に依存関係が存在しないループ文を並列実行する doall 並列処理[1)]，ループ伝搬依存が存在するループ文を並列実行する doacross 並列処理[1),2)]，doalong 並列処理[3)]，波頭（wave front）計算により並列処理を可能とするためのループ傾斜変換（loop skewing）[4)] など，様々な手法が提案されている．また，ループ交換（loop interchange）やループタイル化（loop tiling）により，イタレーションをパイプライン的に制御し，計算と通信のオーバラップにより効率良く並列実行する手法も提案されている[5)]．

しかし，複雑なループ伝搬依存が存在し，並列化できないループ文も依然として存在する．また，上記のようなループ文並列化手法では，イタレーション単位の並列実行が基本となっている．このようなイタレーション単位の並列実行は，通常の並列計算機環境では問題ないが，近年普及している計算機クラスタ環境に対しては並列実行の粒度が細かく，タスク起動オーバヘッドにより効率良く実行できないという問題がある．

† 株式会社豊田中央研究所
Toyota Central R&D Laboratory Inc.
†† 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

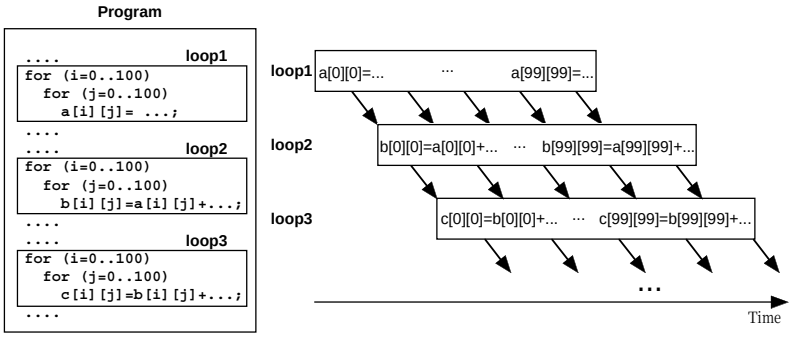


図 1 漸進処理
Fig. 1 Incremental processing.

特に，イーサネットなどの一般的なネットワークにより接続されたワークステーションや PC で構成された NOW（Network of Workstation）システム[6] や，Class I の Beowulf システム[7] などの計算機クラスタ環境にとって，イタレーション単位の並列実行は通信オーバヘッドが大きく，効率的ではない．

複雑な依存関係により並列実行できないループ文や，イタレーション単位の並列実行が適さない計算機環境においても効率良くループ文を並列実行するために，我々はイタレーション単位ではなく，ループ文単位で並列実行を実現する漸進処理を提案する．漸進処理では，同一配列データを操作する複数のループ文を独立のタスクとして生成し，パイプライン的に並列実行させる．ループ文はそれぞれ並列実行されるが，各ループ文内のイタレーションは逐次的に実行されるので，複雑なループ伝搬依存が存在しても適用可能である．また，イタレーションでなく，ループ文全体がタスク生成の単位となるので，タスク起動オーバヘッドの削減と，計算と通信のオーバラップによる通信オーバヘッドの削減が可能となり，複数のループ文を適切に並列実行させることができる．

ループ文の漸進処理で計算と通信のオーバラップによる並列処理を実現するためには，各ループ文が配列データのどの要素を参照，更新するか，またその要素をどのような順序で参照，更新するかを解析しなければならない．そして，その参照，更新順序の類似性により，ループ文の漸進処理に基づいた並列実行が有効か否かを判定しなければならない．現在までにも，ループ文が配列データのどの要素を操作するかを解析するアルゴリズムが多く提案されている[8)~10)]．しかし，これらのアルゴリズムでは，各ループ文が操作する配列データの要素に重なりがあるか否かを解析するのみである☆．漸進処理の適用可否を判断するためには，操作する要素に重なりがあるか否かだけでなく，ループ文の実行経過に従って，要素をどのような順序で操作するかの解析が必要になる．本稿では，ループ文における配列データの要素の操作順序をアクセス・パターンとして定義する．アクセス・パターンは，ループ文のループ変数で構成される式で定義され，ループ文で操作される配列データの要素が，時間の経過に従いどのように遷移するかを表現する．前述のアルゴリズムにより，2 つのループ文が同じ配列データの要素を操作するか否かを解析し，同じ要素を操作する場合は，アクセス・パターンを解析し，漸進処理を適用する．これにより，従来並列実行されなかったループ文を並列実行させることが可能となる．

2. アプローチ

2.1 配列データを共有するループ文の漸進処理

漸進処理では，ループ文のイタレーション間ではなく，プログラムのループ文間で並列性を抽出する．ループ文をタスク生成の単位とし，タスク間で複数回データを授受しながら計算と通信をオーバラップさせる．漸進処理の概念を**図 1** に示す．漸進処理では，ループ文全体がタスクとして実行される．ループ文が実行されるとき，実行の経過に従ってタスク間で必要なデータが通信され，それぞれのタスクがパイプライン的に並列実行される．このとき，実行させるイタレーション回数を調整し，計算と通信を最大限オーバラップさせ，効率的な並列実行を実現する[11),12)]．

漸進処理はイタレーション間に複雑な依存関係が存在するループ文に有効である．たとえば，**図 2** (a) に示すような漸化関係が存在するループを考える．イタレーション間に存在するループ伝搬依存によりすべてのイタレーションは通常逐次的に実行されなければな

☆ 要素に重なりがなければ依存関係が存在しないので，ループ文は完全に並列実行可能であるが，そうでなければ並列実行されないとして解析している．

```
for (i=2; i<10000; i++)
  a[i] = a[i-1] + a[i-2];
```

(a) 漸化関係にあるループ文

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    a[i][j] = a[i][j] + a[b[i]][c[j]];
```

(b) インデックスに配列が使用されているループ文

図 2 複雑な依存関係を有するループ文
Fig. 2 Example of loops with complex loop-carried dependency.

らず，並列実行できない．また，図 2 (b) のように配列データのインデックスに配列データが使用されているループ文の場合も，依存関係が複雑でイタレーション間の並列性を抽出できず，イタレーション単位の並列実行が不可能になる．

ループ伝搬依存が複雑でなく，イタレーション単位で並列実行が可能なループ文であっても，対象とする計算機環境によっては，効率良く実行できない場合がある．イタレーション単位の並列実行では，通常バリア同期などを用いて，すべてのイタレーションの実行終了までプロセッサの実行を停止させなければならない．すなわち，すべてのプロセッサの実行が一番処理時間の長いプロセッサの実行に影響され，無駄なアイドル状態が発生する．多くのプロセッサが使用可能で，高速な通信機構，同期機構を備えた計算機環境であれば，各プロセッサの処理のばらつきも小さくなり，同期処理も高速に行われるので，アイドル状態の発生は少ない．しかし，我々の対象としている計算機クラスタ環境の場合，並列実行の粒度とバリア同期のオーバヘッドの問題のため，イタレーション単位の並列実行は効率的ではない．複数のイタレーションをまとめて 1 つのタスクとすることで粒度の問題を回避可能な場合もあるが，イタレーションをまとめることによりプロセッサ間の処理のばらつきが大きくなる可能性があり，一番処理の遅いプロセッサに引きずられる形で，他のプロセッサのアイドル状態がより多く発生し，効率的な並列実行を達成できない．

これに対し，我々の提案する漸進処理は，イタレーション間の並列実行ではなく，同じ配列データを操作するループ文間の並列実行を実現する．各ループ文のイタレーションは通常の逐次処理と同じ順序で実行されるので，ループ伝搬依存を考慮する必要がない．また，ループ文全体が 1 つのタスクとして生成されるので，計算機クラスタ環境における粒度の問題も回避できる．もちろん，漸進処理を適用すると，タスク間で複数回データ通信が発生するが，同期タイミングを計算し，イタレーションの実行回数とデータ通信を調整することで，計算と通信のオーバラップを最大限活用し，並列実行を実現することができる．

以上のように，漸進処理は以下の状況に適した並列実行手法である．

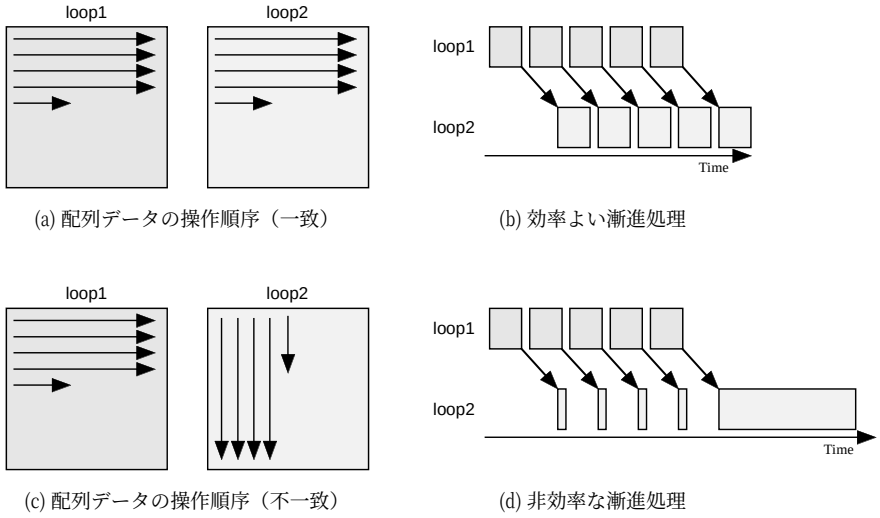
- ループ伝搬依存が存在し，イタレーション間の並列実行が不可能なループ文を効率良く制御したいとき．
- 命令，演算，文単位の細粒度並列処理が適さない計算機クラスタ環境においてループ文を効率良く制御したいとき．

2.2 漸進処理適用のアプローチ

漸進処理では，ループ文で共有される配列データの各要素に対する操作が，その操作順序に従い部分的，漸進的に実行される．したがって，漸進処理の効果は，それぞれのループ文における配列データの操作順序の類似性に大きく依存する．例として，**図 3** (a) のように配列データの操作順序が一致している 2 つのループ文を考える．ここで，図 3 (a) の正方形は二次元配列データを表し，その内部の矢印が各要素の操作順序を表している．`loop1` のループ文が更新した配列データを，`loop2` のループ文が参照しているとする．以下，`loop1` を前ループ文，`loop2` を後ループ文と呼ぶ．図 3 (a) のように 2 つのループ文の操作順序が一致しているとき，ループ文のイタレーションの実行とループ文間の通信がうまくオーバラップされ，効率良く並列実行できる（図 3 (b) 参照）．しかし，図 3 (c) のようにループ文間で配列操作順序が著しく異なるとき，効率の良い漸進処理を行うことができない（図 3 (d) 参照）．したがって，効果的に漸進処理を行うためには，ループ文の配列データの操作順序を解析し，その類似性を評価しなければならない．

我々はループ文の配列データに対する操作順序をアクセス・パターンとして定義し，ループ文間のアクセス・パターンの類似性により，漸進処理を適用したときの並列実行性を評価する．ループ文における配列データの操作順序は，一般にループ変数の値の変化に従って変化する．したがって，我々はループ文のループ変数に着目し，ループ変数の値の変化によりアクセス・パターンを定義する．ループ文のアクセス・パターンの解析方法，類似性の評価方法については 3 章で述べる．

効率的な漸進処理を実現するためには，ループ文間の通信頻度が重要となる．すなわち，何イタレーションごとにループ文間で通信するかを決定する必要があ



(a) 配列データの操作順序（一致）　(b) 効率よい漸進処理

(c) 配列データの操作順序（不一致）　(d) 非効率な漸進処理

図 3 配列操作順序の違いによる漸進処理の効果
Fig. 3 Efficiency of incremental processing.

る．通信頻度を高くするとオーバヘッドの増大により処理効率が低下する．逆に，通信頻度を低くすると，並列実行されるイタレーションが少なくなり，並列実行の効果が得られない．したがって，適切なタイミングで通信処理を発生させ，計算と通信のオーバラップを実現することが重要となる．この漸進処理による実行効率の向上と，通信オーバヘッドの増加による実行効率の低下のトレードオフを解決するため，我々はイタレーションの実行時間より，漸進処理を適用したときのループ文全体の実行時間をモデル化し，最適な通信頻度を与えるアルゴリズムを提案する．このアルゴリズムでは，アクセス・パターンを評価し，ループ文に対して漸進処理を適用する場合の通信タイミングが計算される．この計算結果により，漸進処理の適用可能性，適用する場合の最適な通信頻度，すなわち各通信処理間のイタレーションの実行回数が評価される．このアルゴリズムについては 4 章で述べる．

3. アクセス・パターン

本章では，ループ文における配列データの操作順序を規定するアクセス・パターンの解析方法について述べる．2 つのループ文間で操作する配列データの要素に重なりがあるか否かを解析し[8)~10)]，重なりが存在する場合にアクセス・パターンを解析し，漸進処理適用の可否を判断する☆．

3.1 対象プログラム

並列処理の対象となる数値計算プログラムには，二次元配列を基本とした配列操作が多く見られる．これは，流体力学や熱力学などに対する数値解析法では，問題の対象領域をメッシュに分割し，メッシュ上の格子点間での計算により全体の解析処理を達成するアルゴリズムが多いからである．配列データはループ文中で操作され，配列添字式はループ文の実行とともに規則的に変化するプログラムが多い．本稿では，**図 4** (a) に示すような二次元配列データに対する操作を記述した二重のパーフェクト・ループ文を漸進処理の対象とする．パーフェクト・ループ文とは，ループ文中で実行される文がすべて内側ループ文内に存在するものである．また，ループ変数の初期値，最終値，増分値を表す $lb_{i,j}$，$ub_{i,j}$，$st_{i,j}$ はともに定数であるとする．さらに，配列添字式はループ変数の線形結合で表現されており，図中の c_i，d_i $(i = 0, \cdots, 2)$ も定数であるとする．このような二重ループ文に対して，それぞれのループ変数の初期値が 0，増分値が 1 となるように正規化処理を施し，図 4 (b) のようなプログラムに変換する☆☆．ここで，定数 off_1，off_2 は配列添字式の初期値，すなわちループ文で最初に操作される配列データの要素を表す．また，定数 mov_{i1}，mov_{j1}，mov_{i2}，mov_{j2} はループ文の実行が進行し，ループ変数の値が変化することにより，配列添字式がどのように変化するかを表す．

以下のアクセス・パターンの議論では配列データを操作する文がループ文内に 1 カ所のみ存在する正規化

☆ 操作する配列データの要素に重なりがなければ，2 つのループ文間には依存関係が存在しないので，漸進処理を適用しなくても並列実行が可能である．

☆☆ ループ文の正規化処理に関しては様々な手法が提案されている[1),2),13)]．

```
for(i= lb_i; i<= ub_i; i=i+ st_i) {
  for(j= lb_j; j<= ub_j; j=j+ st_j) {
    ... a[c_0 + c_1*i + c_2*j][d_0 + d_1*i + d_2*j] ...
  }
}
```

(a) 対象プログラム

```
for(i'=0; i'<= ub_1; i'=i'+1) {
  for(j'= 0; j'<= ub_2; j'=j'+1) {
    ... a[off_1 + mov_i1*i' + mov_j1*j'][off_2 + mov_i2*i' + mov_j2*j'] ...
  }
}
```

(b) 正規化後のプログラム

図 4 対象とするプログラム例
Fig. 4 Example of target program.

後のループ文を対象にしている．配列データを操作する文が複数存在するループ文の扱いについては 6 章で述べる．また，前ループ文では配列データに対する更新操作に対して，後ループ文では配列データに対する参照操作に対して，それぞれアクセス・パターンが解析される☆．

3.2 アクセス・パターン表現

上記のように正規化されたループ文を構成する定数 ub_1, ub_2, off_1, off_2, mov_{i1}, mov_{j1}, mov_{i2}, mov_{j2} により，我々は以下のようにアクセス・パターンを定義する．

アクセス・パターン

配列データの操作順序を定義するアクセス・パターン $\mathbf{A}$ は，以下の三つ組で表現される．

$$\mathbf{A} = (\mathbf{S},\ \mathbf{R},\ \mathbf{V})$$

ここで，開始点 $\mathbf{S}$ はループ中で最初に操作される配列要素に対応する添字式値であり，

$$\mathbf{S} = (s_1,\ s_2) = (off_1,\ off_2)$$

である．範囲 $\mathbf{R}$ は二重ループにおける内側ループ（inner loop），外側ループ（outer loop）それぞれのループの繰返し回数であり，

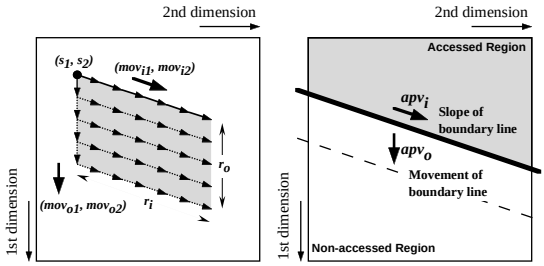
$$\mathbf{R} = (r_i,\ r_o) = (ub_1,\ ub_2)$$

である．アクセス・パターン・ベクトル対 $\mathbf{V}$ は内側，外側ループ文のイタレーションごとに発生する配列添字式値の変化であり，

$$\mathbf{V} = \{\overrightarrow{apv_i},\ \overrightarrow{apv_o}\}$$
$$= \{(mov_{j1},\ mov_{j2}),\ (mov_{i1},\ mov_{i2})\}$$

である．ベクトル $\overrightarrow{apv_i}$, $\overrightarrow{apv_o}$ をそれぞれ，内側，外側ループに対するアクセス・パターン・ベクトルと呼ぶ．□

☆ 図 2 (b) のようなループ文の場合，前ループ文であればアクセス・パターンを解析できるが，後ループ文であればアクセス・パターンを解析できないので，漸進処理の対象にはならない．



(a) 操作される要素領域　(b) 要素の操作順序の規則性

図 5 アクセス・パターンの捉え方
Fig. 5 Access pattern.

図 5 (a) は二次元配列空間上でループ文により操作される配列要素の集合を摸式的に表現している．図中の黒丸は，ループ文の実行で最初に操作される配列要素を表す．ループ文の実行中に操作される要素の集合は，2 つのベクトル $\overrightarrow{apv_i}$, $\overrightarrow{apv_o}$ で囲まれる平行四辺形領域で表現され，図中の網掛け部分に対応する．平行四辺形の各辺の長さは，各ループの繰返し回数 r_i, r_o に基づいて決定される．

図 5 (a) で表されるループ文では，まず最初に開始点 $\mathtt{A}[s_1][s_2]$ から内側ループに対するアクセス・パターン・ベクトル $\overrightarrow{apv_i}$ の方向へ操作箇所が遷移する．内側ループの実行が終了すると，外側ループのループ変数の値の変化により，開始点が $\overrightarrow{apv_o}$ だけ移動した要素から，$\overrightarrow{apv_i}$ の方向に操作箇所が遷移する．ここで，内側ループのイタレーションで操作される要素，つまり $\overrightarrow{apv_i}$ の傾きを持つ直線上に配置された要素を集合として考えると，この要素集合は外側ループの実行に従って $\overrightarrow{apv_o}$ の方向に遷移すると捉えることができる．したがって，配列データの傾き $\overrightarrow{apv_i}$ の直線に対して，

$\overrightarrow{apv_o}$ の方向はループ文の実行にともない，これから操作する可能性のある領域を表し，$-\overrightarrow{apv_o}$ の方向はループ文で今後操作する可能性がない領域を表していると捉えることができる．すなわち，2 つのアクセス・パターン・ベクトルを用いて配列要素の操作順序を図 5 (b) のように捉えることができる．二次元配列上で網掛けされた部分は今後のループ文の実行で操作されない領域であり，この領域を既操作領域（accessed region）と呼ぶ．また，配列空間内の白塗り部分は今後操作される可能性のある領域であり，この領域を未操作領域（non-accessed region）と呼ぶ．これら 2 つの領域は，図 5 (b) の太線で表される傾き $\overrightarrow{apv_i}$ の境界線により分割される．ループ文の実行による操作領域の変化は，外側ループのイタレーションごとに発生する境界線の移動方向 $\overrightarrow{apv_o}$ で表現できる．この外側ループの実行にともなう境界線の移動を境界線遷移と呼ぶ．

このように，ループ文で操作される配列データの要素集合と操作順序をアクセス・パターンで表現することで，配列操作順序の規則性を解析する．

3.3 アクセス・パターンの類似性

本節では，ループ文間でのアクセス・パターンの違いが漸進処理に与える影響として，後ループ文の実行遅延と，ループ文間での配列操作の進行具合について考察する．以下，外側ループ，内側ループのイタレーションを，それぞれ外側イタレーション，内側イタレーションと表記する．また，議論の簡単化のため，アクセス・パターンの類似性解析では，開始点が等しいループ文を対象にする．開始点の違いについては 6 章で議論する．さらに，コード生成時の複雑さの回避のため，漸進処理は前，後ループ文の外側イタレーション単位で行われるとする．

図 6 のようなアクセス・パターン・ベクトルの異なるループ文間での漸進処理を考える．このとき，後ループ文の各外側イタレーションは，各イタレーションで操作する配列データの要素すべてが前ループで操作済みでないと実行することができない．すなわち，後ループ文の外側イタレーションが操作する配列データの要素のうち，前ループ文で最後に操作される要素が存在し，その要素が前ループ文で操作された後，後ループ文の外側イタレーションが実行可能となる．図 6 における黒丸，白丸がそのような配列データの要素を表している．これらの要素は後ループ文の各外側イタレーションの実行を規定する．これらを実行トリガ要素と呼ぶ．特に，黒丸の要素は後ループ文の実行開始を規定する要素であり，これを漸進処理開始点と呼ぶ．また，これらの要素が存在する図中のハッチ部の領域を実行トリガ領域と呼ぶ．

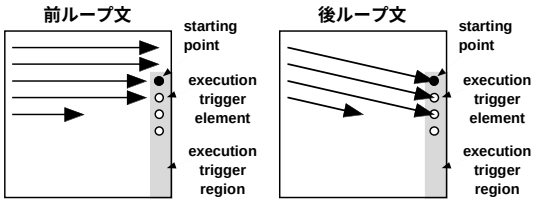


図 6 アクセス・パターンが異なる例
Fig. 6 Different access patterns.

3.3.1 後ループ文の実行遅延

後ループ文は，前ループ文で漸進処理開始点が操作されるまで実行されない．したがって，漸進処理開始点を操作する前ループ文の外側イタレーションを求めることで，漸進処理における後ループ文の実行遅延の程度を表すことができる．すなわち，前ループ文の i 番目の外側イタレーションが漸進処理開始点を操作するとすると，後ループ文の実行前に，前ループ文は $i-1$ 番目までの外側イタレーションを実行しなければならない．この前ループ文の外側イタレーションの実行回数 $i-1$ を i_{wait} として表すと，この i_{wait} は以下のように計算することができる．

前，後ループ文のアクセス・パターン $\mathbf{A}_p$，$\mathbf{A}_s$ が，それぞれ以下のようであるとする．

$\mathbf{A}_p=(\mathbf{S_p},\ \mathbf{R_p},\ \mathbf{V_p})$　　$\mathbf{A}_s=(\mathbf{S_s},\ \mathbf{R_s},\ \mathbf{V_s})$

$\mathbf{S_p}=(s_{p1}, s_{p2})$,　　$\mathbf{S_s}=(s_{s1}, s_{s2})$,

$\mathbf{R_p}=(r_{pi}, r_{po})$,　　$\mathbf{R_s}=(r_{si}, r_{so})$,

$\mathbf{V_p}=(\overrightarrow{apv_{pi}}, \overrightarrow{apv_{po}})$.　　$\mathbf{V_s}=(\overrightarrow{apv_{si}}, \overrightarrow{apv_{so}})$.

このとき，後ループ文の各外側イタレーションで操作される要素は，傾き $\overrightarrow{apv_{si}}$ の直線上に配置され，漸進処理開始点は境界線の端に配置される．つまり，最初の外側イタレーションで操作される要素の中で最初か最後に操作される要素，すなわち

(s_{s1}, s_{s2})　または，

$(e_{s1}, e_{s2}) = (s_{s1}, s_{s2}) + (r_{si}-1)\cdot\overrightarrow{apv_{si}}$

のいずれかが漸進処理開始点となる☆．(s_{s1}, s_{s2})，(e_{s1}, e_{s2}) が参照可能となるまでの前ループ文の外側イタレーションの実行回数を求めるために，前ループ文のアクセス・パターン・ベクトルを用いてそれぞれ

☆ ここでは，後ループ文の外側イタレーションで参照する要素は，前ループ文のそれと等しいか少なく，漸進処理開始点が後ループ文の境界線に存在するとしたときの計算法を示している．そうでない場合は，前ループ文のアクセス・パターンを用いて，$(e_{s1}, e_{s2}) = (s_{s1}, s_{s2}) + \frac{s_{p2}+(r_{pi}-1)\cdot mov_{pj2}-s_{s2}}{mov_{sj2}}\cdot\overrightarrow{apv_{si}}$ と表すことができる．ただし，$\overrightarrow{apv_{pi}} = (mov_{pj1}, mov_{pj2})$，$\overrightarrow{apv_{si}} = (mov_{sj1}, mov_{sj2})$.

の点を表現すると，以下のようになる．

$$(s_{s1}, s_{s2}) = (s_{p1}, s_{p2}) + j' \cdot \overrightarrow{apv_{pi}} + i' \cdot \overrightarrow{apv_{po}}$$
$$(e_{s1}, e_{s2}) = (s_{p1}, s_{p2}) + j'' \cdot \overrightarrow{apv_{pi}} + i'' \cdot \overrightarrow{apv_{po}}$$

ここで，i'，i'' はそれぞれの要素を操作するのに必要な外側ループのイタレーション回数を表している．$\overrightarrow{apv_{pi}}$ と $\overrightarrow{apv_{po}}$ が一次独立であれば，i' や i'' が一意に定まり，i_{wait} は，

$$i_{wait} = \max(i', i'') - 1$$

で表される．したがって，後ループ文を実行可能とするためには，前ループ文の外側イタレーションを i_{wait} 回実行しなければならない．

3.3.2 ループ文間での配列操作の進行具合

前ループ文の外側イタレーションが i_{wait} 回実行された後，前，後ループ文の外側イタレーションをある一定の割合で漸進的に実行させることができる．しかし，この並列実行性はアクセス・パターンの類似性に強く依存する．同一のアクセス・パターンを持つループ文の場合，前，後ループ文の外側イタレーションで操作される配列要素が完全に一致するので，前ループ文の外側イタレーションが 1 回実行されると，後ループ文の外側イタレーションが 1 回実行され，理想的な漸進処理を実現できる．しかし，アクセス・パターンに差異がある場合は，前，後ループ文の外側イタレーションで操作される要素が異なるので，前ループ文の一定回数の外側イタレーションの実行が終了するまで，後ループ文の外側イタレーションが実行できない状況が生じる．このような配列操作の依存関係を表現するために，前ループ文の外側イタレーション実行後に，後ループ文の外側イタレーションの実行可能な回数をイタレーション依存比として定義する．イタレーション依存比は，漸進処理により得られる並列性を特徴付ける重要な指標である．

後ループ文の各外側イタレーションは，それぞれの実行トリガ要素を操作する前ループ文の外側イタレーションが実行された後に，実行可能となる．したがって，各実行トリガ要素を操作する前ループ文の外側イタレーションの実行間隔を求めれば，それがイタレーション依存比となる．

イタレーション依存比は，実行トリガ領域上における境界線遷移，つまり外側ループのアクセス・パターン・ベクトルにより特徴付けられる．前，後ループ文の外側ループのアクセス・パターン・ベクトルの実行トリガ領域方向成分の大きさを，それぞれ a，b とすると，前ループ文の外側イタレーションを b 回実行することで，後ループ文の外側イタレーションが a 回実行可能となる．したがって，イタレーション依存比は

前ループ文：後ループ文 $= b : a$

と表される．このイタレーション依存比は，2 つの実行トリガ要素を操作する間の外側イタレーションの実行回数を計算することで得られる．すなわち，実行トリガ要素を (t_1, t_2)，(t'_1, t'_2) として，

$$(t_1, t_2) = (s_{p1}, s_{p2}) + j_p \cdot \overrightarrow{apv_{pi}} + i_p \cdot \overrightarrow{apv_{po}}$$
$$(t'_1, t'_2) = (s_{p1}, s_{p2}) + j'_p \cdot \overrightarrow{apv_{pi}} + i'_p \cdot \overrightarrow{apv_{po}}$$
$$(t_1, t_2) = (s_{s1}, s_{s2}) + j_s \cdot \overrightarrow{apv_{si}} + i_s \cdot \overrightarrow{apv_{so}}$$
$$(t'_1, t'_2) = (s_{s1}, s_{s2}) + j'_s \cdot \overrightarrow{apv_{si}} + i'_s \cdot \overrightarrow{apv_{so}}$$

で得られる i_p，i'_p，i_s，i'_s を用いて，

$$a = i'_p - i_p$$
$$b = i'_s - i_s$$

と計算できる．

4. アクセス・パターン類似性と同期タイミング

配列データを共有するループ文は，上で述べたイタレーション依存比で表される依存関係に基づき漸進処理可能である．たとえば，イタレーション依存比が

前ループ文 : 後ループ文 $= dep_p : dep_s$

のとき，前ループ文の外側イタレーションが dep_p 回実行されると，後ループ文の外側イタレーションが dep_s 回実行可能であることを示している．ここでは，前ループ文の dep_p 回，後ループ文の dep_s 回の外側イタレーションを漸進処理適用時に同期処理を挿入する最小単位とし，これを漸進処理単位と呼ぶ．つまり，前ループ文の漸進処理単位が 1 つ実行されれば，後ループ文の漸進処理単位も 1 つ実行可能となる．

イタレーション依存比とともに，ループ文間での並列実行性を特徴付ける要素として漸進処理単位の実行時間があげられる．前，後ループ文の外側イタレーションの実行時間をそれぞれ $iter_p$，$iter_s$ とすると☆，漸進処理単位の実行時間比は

$$t_p : t_s = iter_p \times dep_p : iter_s \times dep_s$$

で求められる．ここで，t_p，t_s はそれぞれ前，後ループ文の漸進処理単位の実行時間に対応する．

漸進処理で同期処理のタイミングを決定するとき，各漸進処理単位間で実行遅延状態が発生しないこと，および余分な同期処理を削減することが重要となる．このための同期処理を挿入するタイミングの計算方法を以下に述べる．

4.1 同期タイミング計算方法

後ループ文の最初の漸進処理単位を実行する時点において，すでに前ループ文で x 個の漸進処理単位が実

☆ コンパイラにおけるプログラムの実行時間推定は非常に重要であるが，困難な問題である．ここでは，実行ステップ数などから推定できるとしている．

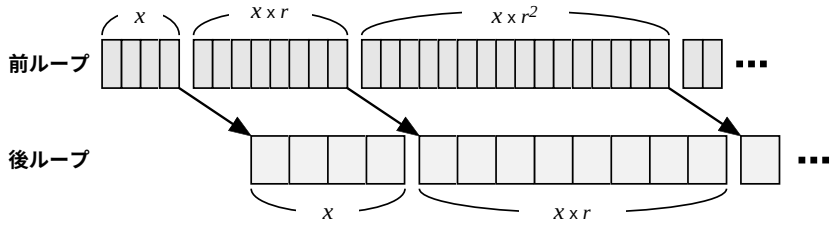


図 7 同期処理を挿入するタイミング
Fig. 7 Synchronization processing.

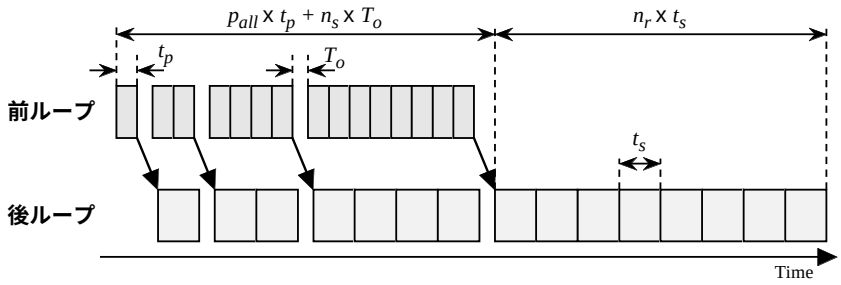


図 8 配列操作時間のモデル化
Fig. 8 Formulation of execution time for array computation.

行されているとする．また，前，後ループ文の漸進処理単位の実行時間をそれぞれ t_p，t_s とする．前ループ文で x 個の漸進処理単位が実行されると，後ループ文でも同様に x 個の漸進処理単位が実行可能となる．漸進処理単位の実行時間比を考慮すると，後ループ文で x 個の漸進処理単位を実行するのに $x \times t_s$ 時間必要となり，その間に前ループ文では $\lfloor x \times \frac{t_s}{t_p} \rfloor$ 個の漸進処理単位が実行可能である．ここで，

$$r = \frac{t_s}{t_p}$$

とすれば，その後同様に，後ループ文が $\lfloor x \times r \rfloor$ 個の漸進処理単位を実行可能であるのに対して，前ループ文では $\lfloor x \times r^2 \rfloor$ 個の漸進処理単位が実行可能となる．したがって，**図 7** に示すように，実行遅延状態が発生しないように漸進処理を適用するためには，前ループ文の $(x \times r^i)$ 個の漸進処理単位ごとに同期処理を挿入すればよい．ここで，図中の長方形は漸進処理単位に対応している．このように計算される同期タイミングおよび同期回数に基づき，ループ文間の配列操作時間をモデル化し，ループ文全体の実行時間を最小にする x を求めればよい．

4.2 漸進処理適用判定アルゴリズム

図 8 に示すように，前，後ループ文の漸進処理単位の実行時間をそれぞれ t_p，t_s とし，前ループ文に含まれる漸進処理単位数を p_{all} とする．また，1 回の同期処理で発生する通信オーバヘッドを T_o とする．これらのパラメータを用いて，漸進処理適用後のループ文全体の実行時間 t_{inc} は以下のように表現できる．

$$\begin{aligned} t_{inc} &= \text{前ループ文の操作時間} + \text{同期処理時間} + \\ &\quad \text{前ループ文実行終了後の後ループ文の} \\ &\quad \text{操作時間} \\ &= p_{all} \cdot t_p + n_s \cdot T_o + n_r \cdot t_s \end{aligned}$$

ここで，式中の p_{all}，n_s，n_r は以下のように計算できる．

まず最初に，前ループ文の全漸進処理単位数 p_{all} は，

$$p_{all} = ub_1/dep_p$$

である．前ループの漸進処理開始点までに実行される漸進実行単位数を p_{pre} とすると，実際に漸進的に実行される前ループ文の漸進処理単位数は

$$p'_{all} = p_{all} - p_{pre}$$

となる．図 7 に示したように漸進処理適用時には $x \times r^i$ 個の漸進処理単位ごとに同期処理が発生するので，$r \neq 1$ のとき

$$p'_{all} = \sum_{i=0}^{n_s} x \times r^i = \frac{x(r^{n_s} - 1)}{r - 1}$$

が成り立ち，漸進処理適用時に挿入される同期処理回数 n_s は，

$$n_s(x) = \log_r \{ \frac{p'_{all}(r-1)}{x} + 1 \} \quad (r \neq 1)$$

と求められる．$r = 1$ のときは，漸進処理単位 x 個ごとに同期処理が発生する．したがって，まとめると，

$$n_s(x) = \begin{cases} \log_r\{\frac{p'_{all}(r-1)}{x} + 1\} & (r \neq 1) \\ \frac{p'_{all}}{x} & (r = 1) \end{cases}$$

となり，x の関数である．最後に n_r であるが，前ループ文で最後に同期処理が発生するまでに，後ループで実行された漸進処理単位数は

$$\sum_{i=0}^{n_s-1} x \times r^i = \frac{x(r^{n_s-1}-1)}{r-1}$$

なので，前ループ文の配列操作が終了した時点において，後ループ文で操作されていない漸進処理単位数 n_r は，

$$n_r(x) = \begin{cases} ub_1/dep_s - \frac{x(r^{n_s-1}-1)}{r-1} & (r \neq 1) \\ ub_1/dep_s - x \cdot (n_s - 1) & (r = 1) \end{cases}$$

となり，これも x の関数である．以上より，ループ文全体を漸進処理したときの実行時間 t_{inc} は，

$$t_{inc}(x) = p_{all} \times t_p + n_s(x) \times T_o + n_r(x) \times t_s$$

のように x の関数としてモデル化することができる．この t_{inc} を最小にする x により，タスク間で同期処理を行えばよい．この式を解析的に解くのは困難であるが，x は $1 \leq x \leq p'_{all}$ の範囲の整数値であるので，$t_{inc}(x)$ を最小にする x を計算することは可能である．ここで，$x = p'_{all}$ はループ文の実行に同期処理を一度も挿入しないとき，実行時間が最短となることを表す．すなわち，対象とするループ文は漸進処理を適用しても効果が得られないことを表している．したがって，この式により漸進処理適用の可否も判断可能となる．

漸進処理適用判定アルゴリズムを**図 9** に示す．アルゴリズムの入力は，前，後ループ文の外側イタレーション実行時間 $iter_p$，$iter_s$，イタレーション依存比 $dep_p : dep_s$，外側ループ文のループ変数の値の最大値 ub_1，および漸進処理開始点までの外側イタレーションの実行回数 i_{wait} である．これにより，ループ文に漸進処理を適用するか否か，適用する場合の同期タイミングを出力する．

```
ALGORITHM: INCREMENTAL_PROCESSING
Input:
    dep_p:dep_s    イタレーション依存比
    iter_p, iter_s 前，後ループ文のイタレーション実行時間
    ub_1           前ループ文の外側ループのループ変数の最終値
    i_wait         漸進処理開始点までのイタレーション実行回数
Output:
    漸進処理を適用しない場合は，「漸進処理適用せず」と出力
    漸進処理を適用する場合は，同期タイミングを規定する x を出力

begin
  t_p := iter_p * dep_p;
  t_s := iter_s * dep_s;
  p_all := ub_1 / dep_p;
  p'_all := p_all - i_wait / dep_p;
  t_min := MAXVALUE;
  foreach x = 1 .. p'_all
    t := tinc(x, t_p, t_s, p_all, p'_all);
    if t < t_min then
      t_min := t;
      x_min := x;
    endif
  endfor
  if x_min = p'_all then
    output("漸進処理を適用せず");
  else
    output(x_min);
  endif
end

FUNCTION tinc(x, t_p, t_s, p_all, p'_all)
begin
  r := t_s / t_p;
  if r ≠ 1 then
    ns := log(r, p'_all * (r - 1) / x + 1);
    nr := ub_1 / dep_s - x * (r**(ns - 1) - 1)/(r - 1);
  else
    ns := p'_all / x;
    nr := ub_1/dep_s - x * (ns - 1);
  endif
  tinc := p_all * t_p + ns * T_0 + nr * t_s;
end
```

図 9 漸進処理適用判定アルゴリズム
Fig. 9 Algorithm for adaptation of incremental processing.

5. 評価実験

本章では，アクセス・パターン表現の有効性，同期タイミング計算法の妥当性を確認するために行った評価実験について述べる．アクセス・パターンの類似性が並列実行性を表現しているか否かを，様々な配列添字式のアクセス・パターンの比較により調査した．

5.1 対象とする配列添字式

実験対象とする配列添字式として，**図 10** を使用した．これらはすべて正規化された後の配列添字式である．図 10(a)，(b) は配列データを順序良く操作する単純な配列添字式であり，それぞれ操作方向が異なっている．図 10 (c)〜(f) は i と j の一次式になっている配列添字式を表している☆．図中の網掛け領域がループ文で操作する要素の集合を表している．また，領域内の矢印は要素の操作順序を表している．そして，領域外の矢印が外側ループのアクセス・パターン・ベクトルを表している．

これらのループ文をそれぞれ前ループ文，後ループ文として，漸進処理適用判定アルゴリズムを適用した結果を**表 1** に示す．イタレーション依存比，漸進処理開始点までのイタレーション回数 i_{wait}，漸進処理適用判定アルゴリズムの出力をそれぞれ示している．ここでは，イタレーションの実行時間を 10 ステップ，同期オーバヘッドを 100 ステップとして計算した．表を見ると，漸進処理の適用可否の判断が適切であることが分かる．たとえば，漸進処理開始点までの外側イタレーションの実行回数 i_{wait} の値が大きいループ文間には漸進処理の適用は指摘されていない．これは，i_{wait} の大きいループ文間では依存関係により並列実行できないことを表している．

また，漸進処理を適用すると判定されたときの同期タイミングの計算結果であるが，イタレーション依存比 1:1，$i_{wait} = 0$ のときの分割数について考察する．最適分割数が 37 という計算結果に基づき，分割数を

☆ 各イタレーションの繰返し回数はすべて一定とし，操作される配列 a は十分な大きさを持っているとする．

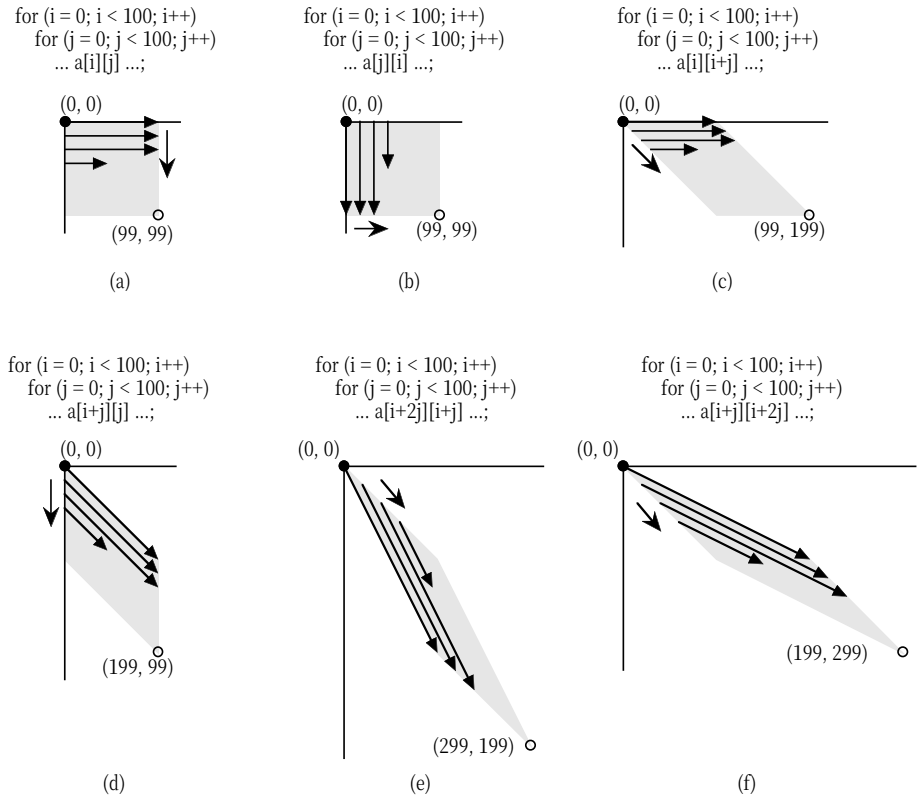


図 10 例題配列添字式
Fig. 10 Examples for access pattern analysis.

表 1 実験結果
Table 1 Experimental result.

前 / 後	(a)	(b)	(c)	(d)	(e)	(f)
(a)		依存比 1:100 i_{wait} 99 適用せず	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37
(b)	依存比 1:100 i_{wait} 99 適用せず		依存比 1:1 i_{wait} 0 分割数 37	依存比 1:100 i_{wait} 99 適用せず	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37
(c)	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:100 i_{wait} 99 適用せず		依存比 1:100 i_{wait} 0 分割数 1	依存比 1:1 i_{wait} 0 分割数 37	依存比 1:1 i_{wait} 0 分割数 37
(d)	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず		依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず
(e)	依存比 1:100 i_{wait} 99 適用せず	依存比 1:1 i_{wait} 49 分割数 20	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず		依存比 1:1 i_{wait} 0 分割数 37
(f)	依存比 1:1 i_{wait} 49 分割数 20	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 99 適用せず	依存比 1:100 i_{wait} 0 分割数 1	依存比 1:1 i_{wait} 0 分割数 37	

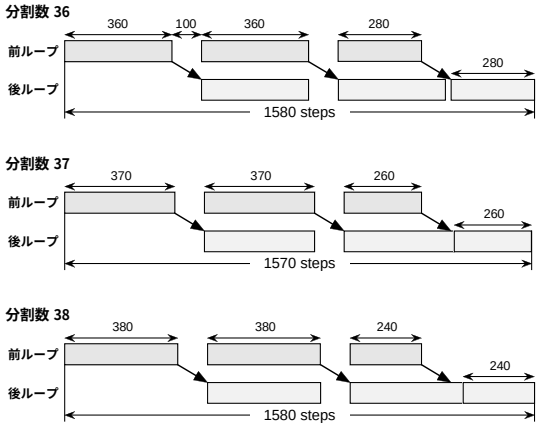


図 11 実行時間の比較
Fig. 11 Comparison of execution steps.

36，37，38 としたときの実行経過を**図 11** に示す．図を見ると分かるように，分割数 36 の場合では同期回数の増加が，分割数 38 の場合では分割数の増加が発生し，これにより漸進処理の効率低下が表れている．したがって，分割数を 37 とするときが効率的な漸進処理を実現可能なことが分かる．

また，前ループ文が (d) で，後ループ文が (c) あるいは (f) のとき，前ループ文の最初のイタレーションが実行された後は依存関係が存在せず，並列実行が可能である．実験結果を見ると，分割数が 1 となっており，前ループ文の漸進実行単位が 1 個実行終了した後，後ループ文の漸進実行単位 100 個が実行可能となることを表している．すなわち，前ループ文の外側イタレーションが 1 回実行終了後，後ループ文がすべて実行可能であることを表しており，ループ文間の並列性が適切に抽出されている．

以上の結果より，我々の漸進処理適用判定アルゴリズムにより，漸進処理の適用可否，および漸進処理適用時の最適分割数が適切に計算されていることが分かった．

6. 様々なループ文への対応

本章では，アクセス・パターンの議論中で導入したループ文に対する制限について検討し，本稿で提案したアクセス・パターンと漸進処理の適用可能性について述べる．6.1 節では，配列データを操作する文がループ文中に複数存在する場合のアクセス・パターンの計算法について述べる．6.2 節では，前，後ループ文の開始点の違いについて述べる．

6.1 複数の配列操作文の扱い

3 章でのアクセス・パターン解析では，配列データを

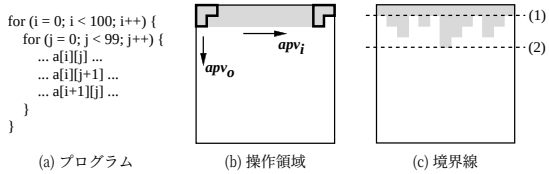


図 12 複数の配列操作文
Fig. 12 Multiple statements.

操作する文が 1 文であるとしていた．しかし，多くのループ文では，内側ループ内に複数の文が存在し，配列データの異なる要素を操作している．このとき，内側イタレーションの 1 回の実行で操作される配列データの要素は複数であり，領域として表される．そして，その領域が内側ループの実行に従って，内側ループのアクセス・パターン・ベクトル方向に移動する．例として，**図 12** (a) のようなループ文を考える．このループ文の場合，内側イタレーションの 1 回の実行では 3 点からなる領域を操作し，この領域が内側ループの実行に従い $\overrightarrow{apv_i}$ 方向に遷移する．その結果，外側イタレーションの 1 回の実行では，網掛け領域が操作される．そしてこの領域が外側ループの実行に従い，$\overrightarrow{apv_o}$ の方向へ遷移する（図 12 (b) 参照）．

このような複雑な操作領域を持つループ文の場合，以下のようにアクセス・パターンを近似して対処する．

- 前ループ文のとき，既操作領域に操作していない配列データの要素が含まれない箇所を境界線として，アクセス・パターンを計算する．
- 後ループ文のとき，未操作領域に操作した配列データの要素が含まれない箇所を境界線として，アクセス・パターンを計算する．

境界線の設定を図 12 (c) に示す．外側イタレーションの実行により操作される領域が図のハッチ部で表されるとき，前ループ文の場合は (1) を，後ループ文の場合は (2) を，それぞれ境界線としてアクセス・パターンを解析することで，並列実行性は減少するが，前，後ループ文間の依存関係を阻害せず漸進処理を適用することができる．

6.2 開始点が異なるループ文の扱い

前，後ループ文で開始点が異なることで問題となるのは，i_{wait} の計算と，漸進処理の適用可否の判断である．後ループ文の開始点が，前ループ文の開始点から見てどの方向に存在するかにより，**図 13** (a)〜(d) の 4 つの場合に分けることができる．図中のハッチ部が実行トリガ領域を，黒丸が漸進処理開始点を，それぞれ表している．このとき，図 13 (a)，(b) の場合の漸進処理開始点は，後ループ文の最初の外側イタレー

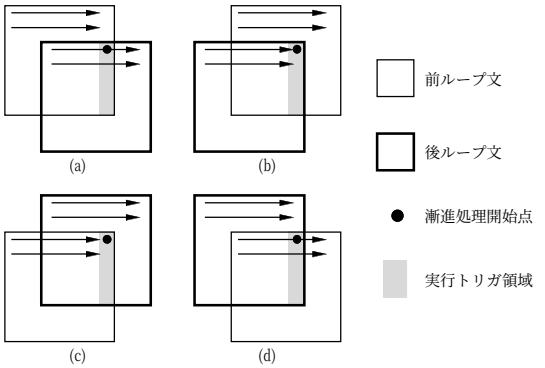


図 13 開始点の異なるループ文
Fig. 13 Loops with different starting points.

ションで操作する要素に含まれるので，3 章で述べたアクセス・パターンの解析法が適用可能である．しかし，図 13 (c)，(d) のように，前ループ文の実行に先立ち実行可能な外側イタレーションが後ループ文に存在する場合，漸進処理開始点は後ループ文の最初の外側イタレーションが操作する要素に含まれない．したがって，i_{wait} の計算法をそのまま適用することができない．

しかし，このような後ループ文の外側イタレーションを，前ループ文に先立ち実行可能か否かで分割し，漸進処理を適用することは可能である．したがって，前処理により後ループ文を適切に変形できれば，図 13 (c)，(d) のような場合も，本稿の議論を適用することができる．

また，漸進処理開始点が端点に表れないようなループ文も存在する．このようなループ文に対する i_{wait}，イタレーション依存比の計算方法は今後の課題である．

7. おわりに

本稿ではループ文の並列実行手法として漸進処理を提案した．漸進処理では，イタレーション単位の並列実行ではなく，ループ文単位の並列実行が行われる．それぞれのループ文は逐次実行時と同じ実行順序でイタレーションが実行される．したがって，複雑なループ伝搬依存が存在し，イタレーション間の並列性が抽出できないループ文に対しても適用可能である．また，イタレーション単位の細粒度並列処理が適さない計算機環境においても，ループ文全体をタスクとする粗粒度並列処理によりループ文を並列実行できる．ループ文に漸進処理を適用するためには，ループ文間の並列データの操作順序に類似性が必要となるが，その類似性を表現するため，アクセス・パターンを定義した．アクセス・パターンはループ文の実行経過に従って配列データがどのような順序で操作されるかをモデル化している．ループ文が操作する配列データの領域を表現するだけでなく，時間軸方向における操作部分の遷移を表現可能としている．また，ループ文間での配列操作の開始時期のずれやアクセス・パターンの違いを考慮し，漸進処理適用時の同期処理を適切に挿入する手法を提案した．このアルゴリズムにより，漸進処理の適用可否と，適用するときの最適な同期タイミングが計算される．この同期タイミングに従い，ループ文を計算と通信のオーバラップにより実行することで，効率の良い並列実行を実現する．

本稿では，ループ文のみに注目し，最適な同期タイミングの計算方法について述べた．しかし，実際のプログラムの並列実行においては，すべてのループ文に対して漸進処理を適用する必要はない．すなわち，漸進処理を適用することでプログラム全体の処理効率が向上するループ文のみに漸進処理を適用しなければならない．したがって，プログラムから漸進処理の適用が必要なループ文を抽出しなければならない．我々は，プログラムを並列実行したとき，プログラム全体の実行時間を規定するタスクを抽出し，そのタスク間にループ文の漸進処理を適用するアルゴリズムを開発中である．漸進処理を適切なループ文に適用し，プログラム全体の処理効率を向上させる手法を開発することが今後の課題である．

また，本稿では対象外となったループ文に対するアクセス・パターンの解析，漸進処理の適用も今後の課題である．特に，開始点の違いにより解析できないループ文や，一次元配列データ，あるいは三次元以上の配列データを扱うループ文，非線形な配列添字式を持つループ文などに対処する必要がある．

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) Wolfe, M.: *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company (1991).
2) Padua, D.A. and Wolfe, M.J.: Advanced Compiler Optimizations for Supercomputers, *Comm. ACM*, Vol.29, No.12, pp.1184–1201 (1986).

3) 丸島敏一，村岡洋一：ループ並列処理方式 doalong，電子情報通信学会論文誌，Vol.J71-D, No.8, pp.1511–1517 (1988).
4) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
5) 石崎一明，小松秀昭：分散メモリ並列計算機のためのコンパイラによる通信遅延隠蔽アルゴリズム，情報処理学会論文誌，Vol.38, No.9, pp.1849–1858 (1997).
6) Culler, D., Dusseau, A., Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C. and Wong, F.: Parallel Computing on the Berkeley NOW, *Proc. JSPP'97* (1997).
7) Sterling, T., Becker, D., Savarese, D., Dorband, J., Ranawake, U. and Packer, C.: Beowulf: A Parallel Workstation for Scientific Computation, *Proc. Int'l Conf. on Parallel Processing*, Vol.1, pp.11–14 (1995).
8) Pugh, W.: A Practical Algorithm for Exact Array Dependence Analysis, *Comm. ACM*, Vol.35, No.8, pp.102–114 (1992).
9) 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).
10) Hoeflinger, J., Paek, Y. and Padua, D.: Region-based Parallelization Using the Region Test, Technical Report 1514, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev. (1996).
11) 三田勝史，朝倉宏一，渡邉豊英：配列データ分割による漸進的並列処理手法の適用，並列処理シンポジウム JSPP'98, p.147 (1998).
12) 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).
13) Zima, H. and Chapman, B.: *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing Company (1991).

(平成 12 年 8 月 28 日受付)
(平成 13 年 2 月 1 日採録)

三田 勝史（正会員）

1975 年生．1998 年名古屋大学工学部情報工学科卒業．2000 年同大学院工学研究科情報工学専攻博士・前期課程修了．現在，(株) 豊田中央研究所勤務．在学中，計算機クラスタ環境，並列計算機環境における並列処理の研究に従事．

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．ワークステーション・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．