計算機アーキテクチャ　143−2
（２００１．５．１１）

Genetic Programmingを利用したループ変換手法の組み合わせ

神戸和子 *　加古富志雄 †

* 奈良女子大学人間文化研究科
† 奈良女子大学理学部

概要

自動並列化コンパイラにおいて，プログラムの並列性を改善するために多くの変換手法が提案されてきた．しかしこれらの手法の適用について，最適な組み合わせと順序を求める事は今もなお大きな課題である．本研究では，進化論的手法である遺伝的プログラミング（Genetic Programming：GP）を利用した，ループを並列化する際のオーバーヘッドを最小に抑えたうえでプログラム全体としての並列性が最大になるような，変換手法の組み合わせと適用順序の決定について調べた．プログラム全体ではなく，プログラムの各コードセグメントに応じた変換手法と順序を求められるように考慮した．

Ordering of Loop-based Transformations using Genetic Programming

Kambe Kazuko *　Fujio Kako †

* Graduate School of Human Culture, Nara Women's University
† Faculty of Science, Nara Women's University

Abstract

A number of transformation techniques in parallelizing compilers have been proposed. It is still a great difficult problem to obtain the best combination and ordering of these transformations. In this paper, we applied Genetic Programming (GP: one of Evolutionary Computation) for determining a combination and ordering of transformations that minimize overhead of loop parallelization and maximize parallelisms of a source program. We can get a transformation sequence that operates each code segment of a program by different transformation techniques.

1　はじめに

逐次原始プログラムから，より多くの並列性を得るために，またその並列性を改善するために，さまざまな変換手法が提案されてきた．これらの手法をどのような場合にも最適な効果が得られるように，適用の組み合わせと順序を決定することはきわめて困難な問題である．

一般的には，原始プログラムレベルから機械語レベルへと変換は行われる．しかし同一レベルにおいてさえ，どの変換をどのように組み合わせて適用するかを決定することは簡単な問題ではない．また適用の回数も１回とは限らない．複数回適用したほうが効果的な場合もある．

各手法の間には適用について依存関係が存在する．１つの手法に対して，変換後に適用可能になる手法も存在するし，逆に不可能になる手法も存在する．組み合わせによって効果が得られる手法もある．

また，プログラム全体ではなく各コードセグメントに対して変換の組み合わせを求めることにより，よりよい結果が得られる．

たとえば，Perfect BenchMarks の FLO52 のプログラムコードについて考えてみる．ただし本稿の趣旨にとって冗長な部分は省略している．

```
L11:     DO 20 j = 2,jl
L2:        DO 10 i = 2,il
             xy = .5 * (x(i,j,1) - x(i - 1,j - 1,1)
             yy = .5 * (x(i,j,2) - x(i - 1,j - 1,2)
             qsi(i) = (yy * w(i,j,2) - xy * w(i,j,3))
             qsj(i) = (yy * w(i,j,3) - xy * w(i,j,2))
   10      CONTINUE
L31:       DOALL 366 i = 2,il
             radi(i,j) = abs(qsi(i))
   366     CONTINUE
L32:       DOALL 365 i = 2,il
             radj(i,j) = abs(qsj(i))
   365     CONTINUE
   20    CONTINUE
L12:     DOALL 367 j = 2,jl
L33:       DOALL 364 i = 2,il
             dtl(i,j) = vol(i,j) / (radi(i,j) + (radj(i,j))
   364     CONTINUE
   367   CONTINUE
L41:     DOALL 35 j = 2,jl
           dtl(1,j) = dtl(il,j)
           dtl(i2,j) = dtl(2,j)
   35    CONTINUE
L42:     DOALL 352 j = 2,jl
           radi(1,j) = radi(il,j)
           radi(i2,j) = radi(2,j)
   352   CONTINUE
```

図 1: 全体に一様に変換を適用した例

```
L1:   DO 20 j = 2,jl
L2:     DO 10 i = 2,il
          xy = .5 * (x(i,j,1) - x(i - 1,j - 1,1)
          yy = .5 * (x(i,j,2) - x(i - 1,j - 1,2)
          qsi(i) = (yy * w(i,j,2) - xy * w(i,j,3))
          qsj(i) = (yy * w(i,j,3) - xy * w(i,j,2))
  10    CONTINUE
L3:     DO 20 i = 2,il
          radi(i,j) = abs(qsi(i))
          radj(i,j) = abs(qsj(i))
          dtl(i,j) = vol(i,j) / (radi(i,j) + radj(i,j))
  20  CONTINUE
L4:   DO 35 j = 2,jl
        dtl(1,j) = dtl(il,j)
        dtl(i2,j) = dtl(2,j)
        radi(1,j) = radi(il,j)
        radi(i2,j) = radi(2,j)
  35  CONTINUE
```

このプログラムコードのループに一様に（多重ループについては内側ループから外側に）ループ分配を適用した後で並列化した結果が図 1 である．ループを細かく分割しすぎたために却ってループ並列化のオーバーヘッドが大きくなる結果となっている．しかし，L4 は分割せず L3 を分割した後，ループを融合し，L1 に対してループ分配を適用すると図 2 となる．こちらのほうがオーバーヘッドを抑えることができ，かつ並列性を高めることができる．このようにプログラムの部分ごとに，変換手法を適用できることは重要である．

```
L11:     DO 20 j = 2,jl
L2:        DO 10 i = 2,il
             xy = .5 * (x(i,j,1) - x(i - 1,j - 1,1)
             yy = .5 * (x(i,j,2) - x(i - 1,j - 1,2)
             qsi(i) = (yy * w(i,j,2) - xy * w(i,j,3))
             qsj(i) = (yy * w(i,j,3) - xy * w(i,j,2))
   10      CONTINUE
L31:       DOALL 366 i = 2,il
             radi(i,j) = abs(qsi(i))
             radj(i,j) = abs(qsj(i))
   366     CONTINUE
   20    CONTINUE
L12:     DOALL 350 j = 2,jl
L32:       DOALL 364 i = 2,il
             dtl(i,j) = vol(i,j) / (radi(i,j) + radj(i,j))
   364     CONTINUE
   350   CONTINUE
L4:      DOALL 35 j = 2,jl
           dtl(1,j) = dtl(il,j)
           dtl(i2,j) = dtl(2,j)
           radi(1,j) = radi(il,j)
           radi(i2,j) = radi(2,j)
   35    CONTINUE
```

図 2: 部分的に変換を適用した例

本研究の目的は，原始プログラムレベルにおいて，ループを並列化する際のオーバーヘッドを最小に抑えたうえで，プログラム全体としての並列性が最大になるような，かつ原始プログラムの特性に応じて部分的に適用可能な変換手法の組み合わせと順序を求める事である．このために，GP を利用した．GP は，最適化問題を解くために生物進化の過程をモデル化した遺伝的アルゴリズム（Genetic Algorithm：GA）の一手法で，遺伝子の記述性が高いという特長を持っている．

本稿の構成は以下の通りである．第 2 章で GP について簡単に概説し，第 3 章で本研究で実装したプログラムとその実行結果について，第 4 章では関連研究について述べる．最後に第 5 章でまとめる．

2 遺伝的プログラミング

GP は，プログラム生成や学習，推論，概念形成等への応用を目指し，GA の遺伝子を構造的な表現

が扱えるように拡張した手法である [1, 2]．個体の遺伝子としてプログラムを扱えるように木構造をしている．したがって，GA に比べて，遺伝子の記述性が高く条件分岐や繰り返しなどの表現も可能である．遺伝的操作については，突然変異は節を変更することであり，交叉は部分木を取り換えることである．次の 5 つの基本要素を設計することで，さまざまな応用例に適用できる．

- 終端記号
  定数，変数など．
- 非終端記号
  関数である．関数は 1 つ以上の引数と 1 つの返値をとる．
- 適応度
  各遺伝子の優劣をつけるための評価で個体ごとに求める．
- パラメータ
  交叉，突然変異の起こる確率や集団の個体数で，GP の実行を制御する役割を持っている．
- 終了条件
  適用する問題に依存するが，たとえば個体群の中の最大の適応度が設定された閾値を越えた時，または世代数があらかじめ設定された値を超えた時などである．

GP の応用例は数多くあるが，ここでは代表的なものを 2 例挙げておく．

1. 法則発見
   訓練データを与えて，その訓練データから何らかの法則を導き出したり，近似理論式を求めようというものである．遺伝子型が表す理論式の計算結果と訓練データの値の差を適応度として評価する．
2. 蟻の捕食行動
   効率的な餌の探索行動をプログラミングする．領域内に数個のえさをおいて，領域の左上端からスタートし，エネルギーのあるうちに捕食できたえさの数で適応度を表す．各遺伝子は蟻の領域内での行動を表している．

3 変換手法の組み合わせ問題

3.1 実装

我々はプログラムコード中のどのループにどの変換手法を適用するかといった順序を表現するために，蟻の捕食行動問題をモデルとする．終端記号，非終端記号および適応度を以下のように定義する．

- 終端記号
  ループの固有番号とする．
- 非終端記号（関数）
  - Up，Down，Pre，Post
    ループの階層構造を移動する関数である．左から順に，現在のループから上の階層へ移動, 下の階層に移動, 同じ階層内で前方のループに移動, 後方に移動を表す．
  - Distribute，Fusion，Motion，Interchange
    ループ変換手法を表す．左から順に，ループ分配，ループ融合，ループ交換，ループ不変なコードの移動である．
- 適応度
  総実行ステップ数とする．ループについては，並列化できないループはループ内のステートメント数とループ繰り返し数（ただし現在は定数に固定している）を掛けたものを，並列化できるループはループ内のステートメントとループ並列化のオーバーヘッドとして設定した定数を足し合わせたものをループの実行ステップ数として計算する．ステップ数が小さいものほど適応度が高いとみなす．
- 終了条件
  世代が指定された値を越えた時とする．

入力は，原始プログラムと GP に必要なパラメータ（木の生成方法，初期集団の木の最大の深さ等）である．原始プログラムはパージングされ，ループについて階層構造を持つ中間表現に変換される．ループの階層構造を移動する関数によってループ間を移動しながら，ループ変換を表す関数により中間表現を変換する．変換実行に際しては，変換を適用できる条件を調べている．各個体について，それぞれの移動と変換が終了した後の中間表現から，総実行ステップ数を求めて適応度としている．

出力は，指定された世代数を実行した後の最も適応度の高い個体であり，その遺伝子はたとえば

(Distribute (Pre (Up 2nd-loop)))

といったものである．これは原始プログラムにおいて 2 番目のループの直接上の階層のループ（すなわち親ループ）に，次に前方のループに移動し，そのループに対してループ分配を適用するという順序を表している．この出力結果にしたがって原始プログラムを変換すればよいことになる．

表 1: 実行結果

サブルーチン	減少率 (%)	適用順序
STEP	0.036	Distribute(Up(Fusion(Distribute(3rd-loop))))
EFLUX	0.098	Fusion(Post(Distribute(6th-loop)))
DFLUX	18.125	Distribute(Post(Post(Down(Pre(Pre(Pre(Pre(Pre(Distribute(Up(Distribute(Down(Pre(Distribute(Up(Distribute(22nd-loop)))))))))))))))))
ADDX	0.091	Distribute(Down(Pre(Distribute(Post(Fusion(Post(Post(Distribute(6th-loop)))))))))
BCWALL	6.982	Distribute(3rd-loop)
PSMOO	0.474	Distribute(Pre(Distribute(8th-loop)))

3.2 実行結果

表 1 に，Perfect BenchMarks の FLO52 の各サブルーチンに対する実行結果を示す．減少率とは，出力の変換手法の適用順序に従って原始プログラムを変換し並列化した場合と，原始プログラムを変換せず並列化した場合の総ステップ数の比率である．集団の個体数は 2000 から 3000 個体，世代数は 100 から 200 世代とした．

表 1 の適用順序は無駄な変換や移動を省略したものである．たとえばサブルーチン ADDX に対する出力は，実際は，

(Up(Pre(Motion(Distribute(Down(Pre(
Distribute(Down(Up(Distribute(Down(
Pre(Distribute(Post(Up(Fusion(Post(
Distribute(Post(Fusion(Distribute(
6th-loop)))))))))))))))))))))))

であった．

今回は本研究の第 1 ステップであるので実装した変換手法が 4 種類と少なかったが，ループ分配とループ融合に関してはよい結果が得られた．これらは互いに相反する変換手法であり，頻繁に衝突することが予想されたが，本システム適用により得られた結果は，無駄のない組み合わせになっている．

ループ交換とループ不変コードの移動は実際に適用される機会はなかった．ループ交換については、現在のところループの繰り返し数を固定として適応度を計算しているため，交換を行ってもそのメリットが適応度に反映されないことが原因とみられる．適応度を計算する際のループの繰り返し数の決定に関しては，実際の実行に近づけるように再考が必要である．

3.3 GP の問題点

今回の実装では，条件分岐や引数を 2 つ以上持つ関数を用意しなかった．したがって遺伝子は木構造ではなく，GP の特長を十分活かせていない．

GP は GA よりも遺伝子の記述性に優れているという長所はあるが，木構造を扱うことはひじょうに計算負荷が大きく，個体の遺伝子の構造が世代を重ねるごとに複雑になってしまう．実際に移動と変換の関数を終端記号として，他に引数を複数持つ関数を作成して実行した結果，計算時間が長くなり，遺伝子の構造が表 1 のものよりもはるかに複雑になってしまった．木構造ではなく線形構造を遺伝子として持つほうが好ましいと考える．

いっぽう，遺伝子の記述性も重要である．ループ交換（Interchange 関数）は，現在では内側ループに交換可能なループが複数存在する場合については考慮していない．複数存在する場合にどのループかを選択できる必要がある．または今後他の変換手法の実装を進める場合において，たとえばループ展開で展開数を指定できるなど，記述面における拡張性が必要である．

Ryan らは，バッカス記法で記述した文法規則に従ったプログラムをバイナリーの遺伝子型にマッピングする手法（Grammatical Evolution：GE）[3] を提案している．遺伝子の記述性を高めかつ個体の遺伝子長を短くして，遺伝子操作の負荷を軽減できる．今後は GE の利用を検討したい．

4 関連研究

変換手法の組み合わせについて，最適な適用順序を求めようとする研究はこれまでもいろいろなされてきた．

Whitfield らは，各変換手法を適用するために必要な事前条件と適用後に成り立つ事後条件を記述する言語 Gospel と，Gospel による記述から変換手法を生成する Genesis を提供することで，変換手法の実装にかかる負荷を軽減し，さまざまな変換を実験できる環境作りを目指した．事前条件と事後条件の関係から適用順序の推奨案も提案している [4, 5, 6]．ベンチマークセット HOMPACK に 15 種類の変換手法を適用した結果から，いくつかの変換手法は互いに変換できる機会を打ち消しあうことが報告されている．彼らは，ユーザがインタラクティブに適用順序を決定できる機能も提供しているが，上記のような問題を解決するためにはユーザの熟練した経験が必要である．

Cooper らは，GA を利用して，ILOC と呼ばれるアセンブラレベルの中間表現に対して，コード数が最小になるように，定数伝播，死んだコードの除去など 10 種類の最適化手法の適用順序を求める実験を行った [7]．プログラムコードへの部分的な適用は考慮されていない．ほとんどのベンチマークテストにおいて 1000 世代を実行するために約 1 日が費やされるほど計算負荷が大きかったことが報告されている．

Ryan と Neil はループ内のステートメントの並列性を抽出するために GP を応用した．非終端記号を ”並列化（PAR）”と ”逐次（SEQ）”とし，終端記号をステートメントとして，各ステートメントの関係を並列実行可能と逐次実行という関係に分別した [8]．適応度は GP の実行で得られたプログラムの実行（計算）結果が，逐次でループを数回実行した計算結果に一致するほど高くなる．これは GP の観点による斬新な試みではあるが，実際のプログラムを試行した結果を訓練データとせざるを得ないことから，現実のコンパイラへの応用には疑問の余地がある．

5 まとめ

本稿では，ループ並列化のオーバーヘッドを考慮しつつ並列性を最大にできるように，変換手法の適用の組み合わせと順序を求めた．プログラムコードのそれぞれの部分に応じて，異なる組み合わせと異なる順序で実行できる変換の適用順序を記述するために，GP を利用した．

今後は，変換手法をさらに増やし，実験を進めていく予定である．また，GE を利用して変換手法の適用順序をプログラムとして記述する可能性について検討したい．

参考文献

[1] Koza, R. J.: Genetic Programming II (Automatic Discovery of Reusable Programs), MIT Press (1994).

[2] 伊庭 斉志: 遺伝的アルゴリズムの基礎ー GA の謎を解くー, オーム社 (1994).

[3] Ryan, C., Collins, J. J. and Neill, O. N.: Grammatical Evolution: Evolving Programs for an Arbitrary Language, *EuroGP'98 Proceedings*, Vol. 1391, pp. 83–95 (1998).

[4] Whitfield, L. D. and Soffa, L. M.: An Approach to Ordering Optimizing Transformations, *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2nd PPOPP'90), SIGPLAN Notices*, Vol. 25, pp. 137–146 (1990).

[5] Whitfield, L. D. and Soffa, L. M.: Automatic generation of global optimizers, *ACM SIGPLAN Notices*, Vol. 26, No.6, pp. 120–129 (1991).

[6] Whitfield, L. D. and Soffa, L. M.: An Approach for Exploring Code Improving Transformations, *ACM Trans. Programming Languages and Systems*, Vol. 19, No. 6, pp. 1053–1084 (1997).

[7] Cooper, D. K., Schielke, J. P. and Devika, S.: Optimizing for Reduced Code Space using Genetic Algorithms, *ACM SIGPLAN Notices*, No. 7, pp. 1–9 (1999).

[8] Ryan, C. and Walsh, P.: Automatic conversion of programs from serial to parallel using Genetic Programming - The Paragen System, *ParCo '95 Proceedings*, Springer-Verlag, (1995).