Committed-Choice 型言語 Fleng における静的粒度最適化

荒木 拓也, 田中 英彦
東京大学 工学系研究科
東京都文京区本郷 7–3–1

Committed-Choice 型言語 Fleng は, データフロー同期の機構を用いることにより, プログラムに内在するすべての並列性を抽出することが可能である. しかし, 実行の粒度が非常に細かいため, オーバーヘッドが大きい. 並列度を低下させない程度にプログラムの粒度を大きくすることができれば, オーバーヘッドを低減することができる. しかし, Fleng において粒度をプログラムの意味を変えずに大きくすることは容易ではない. 本研究は, Committed-Choice 型言語 Fleng において, プログラムの意味を変えずにその粒度を最適化することを目的とし, アルゴリズムの提案, 実装を行なった. また, 並列計算機上で評価を行なうことによりその有効性を確かめた.

A static granularity optimization method of a committed-choice language Fleng

Takuya ARAKI, Hidehiko TANAKA
School of Engineering, the University of Tokyo
7–3–1 Hongo, Bunkyo-Ku, Tokyo

A committed-choice language Fleng can extract maximum parallelism from any programs using dataflow synchronization. But there is large overhead because the granularity of execution is very fine. If granularity of a program is coarsened, such overhead can be reduced; but it is not easy to make a Fleng program coarse because it may change behavior of a program. In this paper, we propose a granularity optimization algorithm of Fleng programs. We implemented this algorithm and evaluated it on a parallel computer.

1 序論

Committed-Choice 型言語 Fleng[2] は, 単一代入変数と, データフロー同期の機構を用いることにより, ゴール (関数) 間に渡る並列性を含め, プログラムに内在するすべての並列性を抽出することが可能である. したがって, 従来存在する手法に比較して, 容易に大量の並列性を抽出できる.

しかし, 実行の粒度が非常に細かいため, ゴールのフォーク, スレッド間のコンテクストスイッチ, 同期, 通信といった, 並列処理に由来するオーバーヘッドが大きくなる. したがって, このオーバーヘッドの削減が, 効率的な実装の鍵となる.

オーバーヘッドの主な原因は, 実行の粒度が細か過ぎることにある. したがって, 並列度を低下させない程度にプログラムの粒度を大きくすることができれば, オーバーヘッドを低減することができる. しかし, Fleng において, 粒度をプログラムの意味を変えずに大きくすることは容易ではない. 安易にプログラムを変更すると, デッドロックを引き起こすためである.

本研究は, Committed-Choice 型言語 Fleng において, プログラムの意味を変えずにその粒度を最適化することを目的とし, アルゴリズムの提案, 実装, 評価を行なった.

2 Committed-Choice 型言語 Fleng

Fleng は論理型言語を祖先とする並列記号処理言語である. シンタックスは Prolog のものと良く似ているが, バックトラックを行なわないという点で, セマンティクスは大きく異なる. Committed-Choice 型言語には, 他に GHC, KL1 などがある.

Fleng は,

- すべてのゴールを並列に実行する.
- 単一代入変数を用いたデータフロー同期をとる.

ことにより, 細粒度, 高並列なプログラムの実行が可能である. この点が Fleng の大きな特徴になっている. 以下に例をあげながら説明する.

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

このプログラムは R = (A + 1) * 2 を実行するものである. ':-' の左側を**ヘッド**, 右側を**ボディ**といい, 全体で**定義節**と呼ぶ. Fleng のプログラムはこのような定義節の集まりである. Fleng における計算の単位は**ゴール**と呼ばれる. このプログラムの場合, foo(A,R) という初期ゴールが与えられると, add(A,1,B), mul(B,2,R) という 2 つのゴールに書き換えられ, それぞれが並列に実行される. しかし, この場合, mul の方は B の値が決定するまで実行することはできない. このような場合, add の実行が終了し B の値が決まるまで, mul は実行を中断 (**サスペンド**) し, add の実行が終了し B の値が決定すると, 実行を再開 (**アクティベイト**) する. この機構を矛盾なく実現するため, 変数は単一代入であり, 書き換えることはできない. 変数は値が決まっていないか決まっているかの 2 つの状態をもち, 一度決まってしまえば, その値が変わることは無い. 変数の値を決めることを**具体化**するという.

分岐は次のように表す.

```
foo(true,R):- R = 1.
foo(false,R):- R = 0.
```

このプログラムは第一引数が true ならば R = 1, false ならば R = 0 を実行するプログラムである. この場合, さきほどと同様に, 第一引数がきまるまでサスペンドし, 値が決まったところでアクティベイトされ, その値によって分岐を行なう.

また, 算術演算は

```
add(#A,#B,R):- compute(+,A,B,R).
```

のよう定義されている. ここで '#' のついた変数は具体化されるまで待つことを表す. また, 'compute' はサスペンドせずに実行し, ほぼアセンブリ言語の add 命令にコンパイルされるようなものである. したがって, compute を用いる際は # によって値が具体化されていることを保証しなければならない.

3 ゴール融合

プログラムの粒度を大きくする手法として, ゴール融合を用いる. ゴール融合とは, 複数のゴールを 1 つのゴールにまとめることによって粒度を大きくする手法である. 以下にゴール融合の手法について述べる.

3.1 ゴール融合の問題点

ゴール融合を行なう場合は, それを適用することによってプログラムの意味が変化しないことを保証しなければならない. この判定はそれほど容易なことではない. つぎに例をあげる.

```
foo(U,V,R,S):- add(U,U,R), mul(V,V,S).
```

このプログラムにおいて, foo から呼び出されている add と mul を融合し,

```
foo(U,V,R,S):- add_mul(U,V,R,S).
add_mul(#U,#V,R,S):-
  compute(+,U,U,R), compute(*,V,V,S).
```

とすることは**できない**.

`foo(1,Tmp,Tmp,S)` という呼び出しを行なった場合, もとののプログラムでは, `add` は `Tmp = 2` を出力し, `mul` は `Tmp` を入力として `S = 4` という結果を返す. しかし, 変更後のプログラムでは `add_mul(1,Tmp,Tmp,S)` が `Tmp` の値が決まるのを待ち続けるためデッドロックする.

したがってゴールを融合することによってプログラムの意味を変えてしまうので, この変換は誤りである. ゴール融合を行なう場合は, このようなプログラムの意味を変えてしまう変換を避けなければならない.

このプログラムにおいて, `add` と `mul` の融合がデッドロックを引き起こすのはなぜかを考えてみよう.

元のプログラムのデータフローグラフを図 1の左側に, 融合した場合のデータフローグラフを図 1の右側に示す.

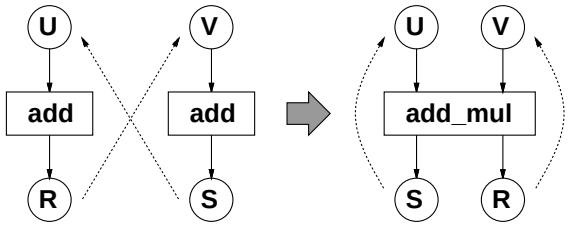

図 1: データフローグラフ

ここで, U, V, R, S はヘッドを通じて引数として外に見えている変数である. したがってV が R に, U が S に依存する可能性が存在する. これを文献 [3] の用語を用いて**潜在的な依存関係**と呼び, 点線であらわす.

融合後のデータフローグラフを見ると, 潜在的な依存関係を通って, U から S を通って U へ, あるいは V から R を通って V へという**サイクリックな依存関係**が存在する. サイクリックな依存関係があると, 自分の出力に依存するわけであるから, デッドロックを引き起こす. もとのプログラムでは, サイクリックな依存関係は存在しないため, デッドロックが起こらない.

3.2 ゴール融合のアルゴリズム

デッドロックの原因はサイクリックな依存関係であるから, ゴール融合を行なう際には, サイクリックな依存関係を作らないようにすればよい. したがって基本アルゴリズムは次のようになる.

> ある 2 つのゴールが融合可能である条件は, その 2 つのゴールの間に間接的な依存関係が存在しないことである.

これは, 2 つのゴール間に間接的な依存関係が存在する場合は, 融合するとかならずサイクリックな依存関係を作り出してしまうためである.

例を用いて説明する. 図 2のようなデータフローグラフであらわされるプログラムを考える.

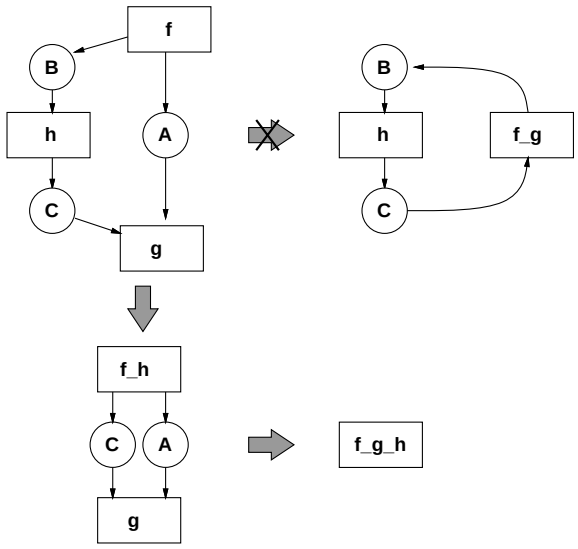

図 2: ゴール融合のアルゴリズム

円は変数, 四角はゴールをあらわす. このプログラムにおいて, ゴール f と g が融合できるかどうかを考えると, g は h を介して f に間接的に依存しているため f と g は融合することはできない. 実際融合した場合, サイクリックな依存関係を作ってしまう. このようなサイクリックな依存関係は, 2 つのゴールが間接的に依存する限り, 必ず生ずる. 融合によって, 同じゴールから出て同じゴールに入る依存を作るからである.

つぎにゴール f と h が融合できるかを考えると, h は f に直接依存しているので, 融合可能である. f と h を融合した f_h と g は先ほどと同じように融合可能である. 結局 f, g, h は 1 つのゴールに融合可能である.

3.3 依存関係の解析

ゴールの融合が可能かどうかを判定するには, 2 つのゴール間に間接的な依存関係が存在しないことを保証しなければならない. これは誤った変換しないよう, 安全に行なう必要がある. かといって, 存在しない依存関係まで存在するとしてしまうと, 可能な融合も不可能であると判定してしまう. すなわち, 依存関係の解析の正確さが, ゴール融合の性能に大きな影響を与える.

本節では依存関係の正確な解析法について述べる.

3.3.1 モード推論

Fleng は関数型言語などと異なり, 変数が入力か出力かシンタックスからは判断できない. 依存関係を解析するには変数の入出力を知らなければならないが, 入力か

出力か判断できない変数については, 安全側に倒して判断しなければならず, 依存関係を解析する際に性能を低下させてしまう. ある引数が入力か出力かを知るには, モード推論を行なう必要がある.

文献 [4] では well-moded な GHC プログラムについて構造データも含めたモード推論を行なっているが, 本研究で行なったモード推論は, 変数単位の単純なモード推論である. また, モード推論の考え方は文献 [4] のものを借用している.

アルゴリズムは

1. ある変数があるゴールの出力なら, その変数を共有している他のゴールでは入力である.
2. ある変数が 1 つを除いたすべてのゴールで入力なら, 残りの 1 つのゴールでは出力である.

とした.

3.3.2 存在し得る依存関係の検出

ボディー部の中で依存関係を作り得るものについて以下に述べる.

ゴールフォーク 安全に依存関係を検出するには, すべての入力の可能性のある変数から, すべての出力の可能性のある変数へ依存関係があるとしなければならない. そのゴール自体では依存関係がなくても, そのゴールから呼ばれるサブゴールの中で依存関係があるかもしれないからである.

ユニファイ 変数とシンボル, 数字などとのユニファイの場合は依存関係は考えなくて良い. 変数同士の場合は, お互いに依存関係があるとすればよい. 例えば A = B の場合は A は B に, B は A に依存しているとすればよい.

これらに対して変数と構造データのユニファイの場合は注意が必要である. 例えば `A = [B|C]` というユニファイの場合, 本質的にはこれらの変数 A, B, C の間には何の依存関係も存在しない. しかし, 仮想的に A から B, C へ, B, C から A へという依存関係が存在するとしないと, 本来存在する依存関係を検出できないことがある. 例えば,

```
foo(B,C,Z):-
  A=[B|C],bar(A,A1),A1=[X|Y],baz(X,Y,Z).
bar([B|C],A1):-
  add(B,B,X),mul(C,C,Y),A1=[X|Y].
baz(X,Y,Z):- sub(X,Y,Z).
```

というプログラムの場合, foo の中の X は B に, Y は C に依存しているが, これは B, C から A へ, A1 から X, Y へという依存関係が存在するとしないと検出できない.

ヘッド ヘッドは, 最初にあげた融合できない例のように, 呼び出し側で引数同士の依存関係を作り出してしまっている場合がある. これはモードの分からないボディーゴールと全く同じ状況である. したがって, モード推論, 依存関係の解析において, ヘッドはボディーゴールと全く等価に扱う. この場合, ヘッドのモードは外から見た場合のモードとちょうど反対になる.

3.3.3 矛盾する依存関係の排除

前節で存在し得る依存関係の求め方を示したが, その中にはあり得ない依存関係も存在する. 例えば

```
foo:- bar(Y,X), add(X,X,Y).
```

の bar において, Y から X という依存関係は存在し得ない. なぜならば `add` によって Y は X に依存していることがわかるので, もし Y から X という依存関係が存在すれば, デッドロックするからである. 与えられたプログラムにバグが無く, 正常に動作するものであると仮定すると, このような依存関係は存在しない (図 3).

本研究では, 文献 [3] に述べられている方法を借用し,

確実な依存関係に矛盾する潜在的な依存関係は排除する

という方針をとる. ここで潜在的な依存関係とは, ゴール融合を安全に行なうため, 存在するものとして扱わなければならないが, 確実に存在するわけではない依存関係である. 最初にあげた融合できないプログラム例で, ヘッドを介して存在する可能性のあった依存関係が潜在的な依存関係である.

潜在的な依存関係が確実な依存関係と矛盾する場合は, 潜在的な依存関係の方が, 実は存在しないことが保証できるわけである.

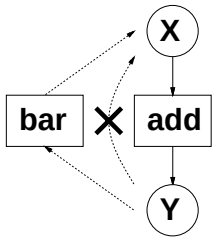

図 3: 矛盾する依存関係の排除

3.4 ゴール間解析

これまではゴール内での解析について述べた. 本節ではゴール間での解析について述べる.

情報を受け渡す方向には, 被呼び出し側から呼び出し側 (Bottom-Up) と呼び出し側から被呼び出し側 (Top-Down) の二種類があり, ゴール間で受け渡す情報には入出力モードと変数間の依存関係の二種類がある.

解析はトップゴールを与え, そこから呼び出されるゴールをたどることによって行なう. これは, 深さ優先で呼び出し木をたどればよい. たどったゴールは記憶しておき, 再帰があった場合はそこで解析を止める.

被呼び出し側から呼び出し側 (Bottom-Up) 次のような例を考える.

```
foo:- bar(U,V,S,R),...
bar(U,V,S,R):- add(U,U,S), mul(V,V,R).
```

bar で解析した情報をfoo に伝えることを考える. bar の入出力モードは先ほど述べたモード推論を行なうことによって知ることができる. ただし, モード推論によって得られる得られるモードはボディー側から見たモードのため, 反転する必要がある. この場合, U, V が入力, S, R が出力である.

つぎに依存関係であるが, これはヘッドを除いて上で述べた依存関係の解析を行なえば良い. 得られる依存関係が bar の中での依存関係である. ここでは, U から S, V から R という確実な依存関係が得られる.

また, ここで注目すべきは, U から R, V から S という依存関係が存在しないことが分かる点である. モード情報のみではこれは分からない.

呼び出し側から被呼び出し側 (Top-Down) 本研究における実装では, 呼び出し側から被呼び出し側への情報の受渡しは複雑さを嫌って採用していない. 被呼び出し側が複数の場所から呼ばれる場合は, 複数の場所で得られる情報を安全にまとめて伝えなければならず, また, プログラムの外部から呼ばれる可能性がある場合は情報を伝えることはできないからである. このような複雑さはあるが, 手法は本質的に被呼び出し側から呼び出し側への情報の受渡しと同じである.

4 実装と評価

コンパイラのプリプロセッサとして実装した. Fleng プログラムを入力とし, 粒度を大きくした Fleng プログラムを出力する. 約 6000 行の Fleng プログラムである. また, 粒度をできるだけ大きくできるようにするため, ゴール内で分岐を扱えるよう, コンパイラを拡張した. この分岐は (Cond --> Then ; Else) のようにあらわす. また, サスペンドしないゴールはできるだけインライン展開するようにした. 以下に変換例として, 絶対値を求めるプログラムをあげる.

```
abs(A,R):- greater(A,0,IsGt),abs1(IsGt,A,R).
abs1(true,A,R):- R = A.
abs1(false,A,R):- sub(0,A,R).
```

このプログラムを処理すると以下のようになる.

```
abs(A,B):- C = 0, greater_abs1(A,C,B).
greater_abs1(#A,#B,C) :-
  compute(>,A,B,D),
  ((D == true)-->
    C = A
  ;
    E = 0, compute(-,E,A,C)
  ).
```

評価は並列推論エンジン PIE64[1] の上で行なった. PIE64 は Fleng の高速実行を目的に設計された並列計算機である. 要素プロセッサ数は 64 台で, マルチコンテクスト処理をサポートする. また, 相互結合網は自動負化分散の機能を持つ. 実行時間の測定値は 3 回実行した値の平均値を用いた. ガーベジコレクションの時間は含まない.

対象プログラムは ベンチマークプログラムとして N-queens 問題を解く qu(55 行) を, 実用規模のプログラムとして Fleng のマクロを展開するプログラム fme (1175 行) を選んだ.

また, 粒度最適化部自身はプログラムの粒度を指定できるように実装しているが, 今回の評価では粒度をできるだけ大きくするよう変換した. これは, 関数型言語 Id のコンパイラでも指摘されているように [3], できるだけ大きな粒度にしても最適な粒度より細かい粒度にしかならないためである.

粒度最適化プログラムの処理時間は, プログラムをコンパイルし, ワークステーション (Sun Ultra 1) 上で実行した時のものである.

qu 粒度最適化プログラムを通すことにより, コンパイルしたバイナリのサイズは 8925 バイトから 7918 バイトに減少した. これは, ゴール融合やインライン展開により無駄なコードが削除されたためであると考えられる. 粒度最適化部の処理時間は約 5 秒であった.

図 4に 9 queens を実行した場合の速度向上を示す. 横軸はプロセッサ台数, 縦軸は粒度最適化を行なわなかった時の 1 台の時の速度を 1 とした相対速度である. プロセッサ台数に関わらず, 全領域で 8 倍以上の速度向上を示している.

また, qu 以外の他のベンチマークプログラムでも同様に数倍の速度向上を示している.

fme 実用規模のプログラムとして fme を評価した. fme は Fleng のマクロを展開するプログラムである. ここでは, 入力としてマクロ展開前の qu(43 行) を用いた. た

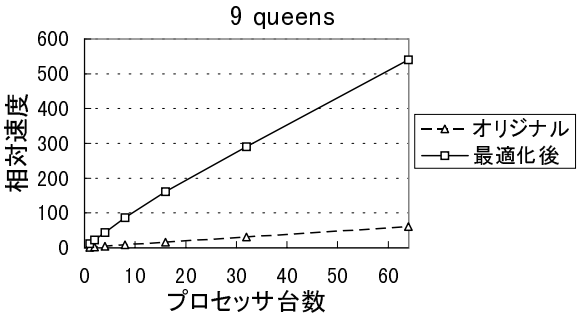

図 4: 9 queens の速度

だし, PIE64 とホスト計算機との通信時間を省くため, 実際の入出力は行なわなかった. すなわち, 入力は定義節の形でプログラムに埋め込み (粒度最適化プログラムには通さない), 出力は行なわないようにした.

また, 粒度最適化プログラムは, 定義節の大きさが大きいと処理が重くなる. 今回は一定義節内に変数が一定 (20 個) 以上あると処理を行なわないようにした.

粒度最適化プログラムを通すことにより, バイナリのサイズは 194110 バイトから 170054 バイトに減少した. 処理時間は約 12 分であった. 実行速度を図 5に示す. qu と同様, 横軸はプロセッサ台数, 縦軸は粒度最適化を行なわなかった時の 1 台の時の速度を 1 とした相対速度である.

最適化により, 1.2 倍程度の速度向上が見られた. 小規模なプログラムほどではないが, 実用規模のプログラムでも効果があることが示された.

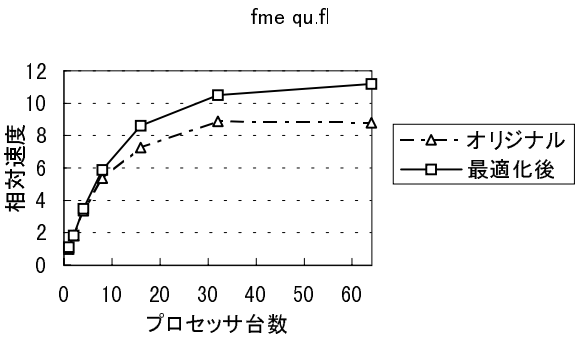

図 5: fme の速度

5 関連研究

Id のような Lenient な関数型言語と Committed-Choice 型言語の実行モデルは良く似ており, 本研究で行なったような静的粒度最適化は, 関数型言語の分野でもさかんに研究されている. 関数型言語におけるスレッド生成の最新のアルゴリズムは, separation constraint partitioning[3](以後 SCP) であり, 本研究も大きな影響を受けている. 表現は異なるが本研究の基本アルゴリズムは SCP と本質的には同じものであると考えられる.

SCP は入力としてデータフローグラフを前提としているので, Fleng のような入出力がプログラムの字面から判定できない言語では, 直接は適用できない. また, 本研究で提案したアルゴリズムは, 融合を途中でやめても正しい結果が得られる, という点で SCP と異なる.

6 結論

本研究では, Committed-Choice 型言語 Fleng において, プログラムの意味を変えずにその粒度を最適化することを目的とし, アルゴリズムの提案, 実装, 評価を行なった.

本研究では, プログラムの粒度を大きくするためにゴール融合という手法を用いた. ゴール融合が可能かどうかの判定を行なうアルゴリズムを提案し, 提案したアルゴリズムに基づき Fleng プログラムの粒度を最適化するプログラムを実装し, 評価を行なった. 粒度最適化により, N-queens 問題では 8 倍以上, 実用規模のプログラムでも 1.2 倍程度の高速化が達成された.

実用規模のプログラムで性能向上率が低いのは大域的な解析が十分に行なわれなかったためであると考えられる. 解析が困難な場合でも粒度を大きくすることが今後の課題としてあげられる.

参考文献

[1] Takuya ARAKI, Yasuo HIDAKA, Hidemoto NAKADA, Hanpei KOIKE, and Hidehiko TANAKA. System integration of the parallel inference engine PIE64. In *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, Dec. 1994.

[2] Martin NILSSON and Hidehiko TANAKA. Fleng prolog - the language which turns supercomputers into prolog machines. In *LNCS264*. Springer-Verlag, 1989.

[3] K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation constraint partitioning – a new algorithm for partitioning non-strict programs into sequential threads. In *POPL '95*, pp. 259–271. ACM, 1995.

[4] Kazunori Ueda and Masao Morita. Message-oriented parallel implementation of moded flat GHC. In *Proc. of Int. Conf. on Fifth Generation Computer Systems*, 1992.