Comitted-Choice 型言語へのループ記述構文の導入

荒木 拓也, 田中英彦
東京大学 工学系研究科

1 はじめに

Committed-Choice 型言語は単一代入変数によって同期をとりながらすべてのゴールを並列に実行することにより, 非定型的な問題においても 多くの並列度を抽出して実行可能な言語である. その一方, 配列を扱うデータ並列処理においては Fortran の並列化コンパイラなどに効率の面で及ばない. 本稿では, Committed-Choice 型言語の一種である Fleng に for ループ記述構文を導入することにより, データ並列性を持つプログラムを効率良く実行する手法について報告する.

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 などがあるが, Fleng は GHC からガードゴールを取り除いたものに相当する.

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てデータフロー同期をとる.

ことにより, プログラムの持つ並列性を大量に抽出可能である.

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. 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る. 書き換えられた add と mul は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 は実行を中断し, add の実行が終了し B の値が決定すると, 実行を再開する. 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.

このような機構を用いることで, Fleng は非定型的なプログラムからでも大量の並列性を抽出することができる. しかし, その一方で, Fleng のような単一代入変数の言語は一般に配列を扱うのが苦手である. この理由は,

- 配列の一部分を変更した配列が欲しい場合, 変更しない部分をコピーして新たな配列を作る必要がある.
- 並列実行モデルの違いにより, Fortran の並列化コンパイラなどのようにはデータ並列性を単純に取り出せない.

ことによる. これらの問題点の解決は, ループ記述構文を導入することにより解決が容易になる. 特に, ループからデータ並列性を抽出する研究は Fortran の並列化コンパイラの世界で非常に良く研究されており, この研究を直接利用することが可能となる.

3 ループ記述構文

次のような構文を持つ for ループ記述構文を Fleng に導入する.

```
sum(Vec,Size,R):-
  (for I = 1 to Size init Sum = 0, V = Vec do
    element(I,V,Elm),
    next Sum is Sum + Elm
  finally R = Sum).
```

このプログラムは, 配列 Vec とその大きさ Size を入力として, 配列の全要素の和を R に出力するものである. セマンティクスは直観的にわかる通りである. ここで, next 付きの変数は, Id などの関数型言語と同様で, 次の繰り返しでその名前で参照される変数であることを意味する. 値の初期化は init 内で行なう.

変数のスコープは, 基本的にループ内でローカルである. ループ外の変数をループ内で使用したい場合は, init 内で宣言する. 例えば上記の例では, init 内に V = Vec があるので, ループの外の Vec をループの中で V という変数としてアクセスできる. それ以外の変数, 例えば Elm はループの各イタレーションで別の変数として扱われる.

ループを抜けた後, finally 以下を実行し, next で修飾された変数の最後の値をループの外に受け渡す. 上記の例の場合, ループ内で計算した値 Sum をループ外の変数 R として渡している.

4 配列のコピー除去

Fleng は単一代入則を用いているため, 一部分を変更した配列が欲しい場合, 変更しない部分をコピーして新た

Introduction of loop syntax to committed-choice languages
Takuya ARAKI, Hidehiko TANAKA
School of Engineering, the University of Tokyo

な配列を作る必要がある．もしこれを行なわず，破壊代入による更新を行なった場合，誤った結果を出力する可能性がある．例えば，

```
Vec = {1,2,3},
element(1,Vec,Elm),
set_element(1,Vec,100,NewVec)
```

のようなプログラム片を考える．ここで，set_element (Pos,Vec,NewElm,NewVec) は，Vec の Pos 番目を NewElm にした新しい配列 NewVec を返すものとする．この set_element を破壊代入で実現した場合，element の実行が set_element より後であったときに更新後の値を取り出してしまい，誤った結果を出力してしまう．

ここで，更新操作を行なう時点でその配列を参照しているゴールが存在しなければ，破壊代入をしても構わない．そこで，上の例の場合，プログラムを以下のように変換する．

```
Vec = {1,2,3},
get_element(1,Vec,Elm,Vec1),
assign_element(1,Vec1,100,NewVec)
```

ここで，get_element(Pos,Vec,Elm,NewVec) は，Vec の Pos 番目を参照し，それが完了してから NewVec を Vec に束縛するものであるとする．したがって，set_element を実行する時点では，Vec を参照しているゴールは存在しないので，破壊代入しても構わない．そのため，set_element を破壊代入を行なう述語 assign_element に書き換えることができる．

ただし，このような変換はプログラムの逐次化を意味するため，デッドロックを引き起こす可能性がある．例えば，

```
element(Pos,V,Elm),     % (1)
set_elenet(1,V,2,V1),   % (2)
element(2,V1,Pos)       % (3)
```

のような場合，Pos を通して (3)→(1) という依存関係が存在する．ここで，さきほどと同様に (1) を get_element に変換すると，(1)→(2) という依存関係を作ってしまう．V1 を通して (2)→(3) という依存関係も存在するため，(1) → (2)→(3)→(1) という循環的な依存関係を作ってしまい，デッドロックする．

この手法を用いてコピー除去を行なう場合は，データ依存関係を解析し，このような循環的な依存関係を生じないことを保証する必要がある．

5　データ並列性の抽出

大きさ 100 の配列 Vec の要素を 2 倍した配列 Res を生成するような，以下のプログラム片を考える．このプログラムには前節で述べたコピー除去をすでに行なってある．

```
(for I = 1 to 100 init V = Vec do
  get_element(I,V,VE,V1), RE is VE * 2,
  assign_element(I,V1,RE,next V)
finally Res = V)
```

ここで，ループボディーの操作は効率のため，文献 [1] のような手法でインライン展開されると考える．この場合，このループの実行はすべて逐次に行なわれる．しかし，プログラムは配列の各要素を 2 倍するという意味なので，ループの各繰り返しは並列に実行可能なはずである．そこで，このプログラムを次のように 2 つに分割する．

```
(for I = 1 to 50 init V = Vec do
  get_element(I,V,VE,V1), RE is VE * 2,
  assign_element(I,V1,RE,next V)
finally Tmp1 = V),
(for I = 51 to 100 init V = Vec do
  get_element(I,V,VE,V1), RE is VE * 2,
  assign_element(I,V1,RE,next V)
finally Tmp2 = V),
(wait(Tmp1)-> Res = Tmp2)
```

このように分割することにより，I が 1 〜 50 までと 51 〜 100 までを並列に実行できる．ただし，それぞれを並列に実行可能にするため，分割したループ間では finally により出力される変数を渡していない．すなわち，後半のループでも Tmp1 ではなく Vec で V を初期化する．また，Res を参照する時点ではすべての更新が終っている必要があるため，wait を用いてこれを保証している．

しかし，このようなループの分割が常に可能なわけではない．まず，next で修飾されている変数が，コピー除去により破壊代入されていて，各繰り返しで同じ配列を参照している必要がある．また，破壊代入を矛盾なく行なうため，異なる繰り返し間で，同じ要素の参照，代入があってはならない．これは，並列化コンパイラにおける繰り返しを跨ぐ依存関係の解析と全く同じ状況である．したがって，GCD テストなど並列化コンパイラで良く研究されている手法をそのまま利用することができる．

6　まとめ

Committed-Choice 型言語 Fleng へのループ記述構文の導入と，最適化の手法について述べた．今後の課題として，本稿で述べたような最適化を行なうプログラムを実装することがあげられる．

参考文献

[1] 荒木拓也，田中英彦．Committed-choice 型言語 fleng のインライン展開による粒度制御手法．情報処理学会第 53 回全国大会，Vol. 1, No. 3E-8, pp. 347–348, September 1996.