記　号　処　理 77－3
（1995．1．13）

Flow Graph 解析を用いた並列関数型言語処理系の実装

金子裕之　中西正和
慶應義塾大学

関数型言語は並列性を内在しているためプログラムからすべての並列性を引き出す事は並列化のオーバーヘッドの増大につながり、効率の良い並列実行を困難とする。本稿では P.Hertel の手法である Flow Graph 解析を拡張し関数プログラムの並列性の抽出を試みる。P.Hertel らは逐次型言語の解析に利用されていた Flow Graph 解析を関数型言語に適用して高度な最適化を行なった。Flow Graph の使用により様々な解析 (strictness analysis, boxes analysis 等) を統一的に扱う事が可能となる。

Implementing Parallel Functional Language using Flow Graph Analysis

Hiroyuki KANEKO and Masakazu NAKANISHI
Department of Computer Science
Graduate School of Science and Technology
Keio University
3-14-1, Hiyosi, Kouhoku, Yokohama 223, Japan

To detect all parallelism in the functional program increases the overhead for the functional languages have parallelism and prevent the efficient parallel execution. In this paper We extend flow graph analysis and try to maximize useful parallelism in the functional program. P.Hertel applied the flow graph analysis used to analysis the imperative language to the functional language. To use flow graph analysis enables to treat the variety of analysis(strictness analysis , boxes analysis and so on).

1 はじめに

一般に言語に並列性を導入する場合、陽の並列性 (explicit parallelism) と暗黙の並列性 (implicit parallelism) が存在する。このうち暗黙の並列性はプログラムの構文には手を加えず、コンパイラがプログラム中の並列性を抽出する方式である。しかし関数型言語においては並列性を内在しているためプログラムからすべての並列性を引き出す事は並列性の爆発につながる。そこで、どのように並列性を引き出すかという事は重要な研究課題である。関数プログラムの解析の方法に P.Hertel らが提案、実装している Flow Graph 解析 [6][2] がある。Flow Graph 解析は関数プログラムを Flow Graph に変換して様々な解析 (strictness analysis, boxes analysis 等) を行なう。解析の結果は逐次実行の高速化のために利用されていたが本論文においてはこの解析を拡張し Flow Graph から並列性の抽出を試みる。

2 Flow Graph 解析

フロー解析は手続き型言語の解析において一般的に用いられる手法である。フロー解析とはデータ、変数の使用、コンパイル単位の外とのデータの流れを追跡したものである。Pieter Hartel らはフロー解析を関数型言語に適用して strictness analysis 等の解析と組み合わせて高度な最適化を行なった。Flow Graph はコンパイルされた関数プログラムを表しているグラフであり node とデータの経路を表す edge からなる。edge はそれぞれ整数で番号つけがされている。node は図 1 の 4 つのデータのタプルで表される。

input(s)	その node に入ってくる edge を表す
output(s)	その node から出ていく edge を表す
type	11 種類のラベル (図 3) を表す
qualifier	その node 特有のデータを表す

図 1: Flow Graph の node

BIND	関数適用を表す
LAMBDA	関数抽象を表す
SOURCE	基本的なデータを表す
SINK	破壊されるデータを表す
IMPORT	他の関数から輸入される情報を表す
EXPORT	他の関数に輸出する情報を表す
USE	共有されるデータを規定する
SWITCH	条件判断文を表す
MERGE	条件判断文を表す
CALL	
RESULT	

図 2: Flow Graph のラベルの種類

関数プログラムを Flow Graph に書き換え、strictness analysis やその他の解析の情報を蓄積して統一的に扱う事が可能になる。

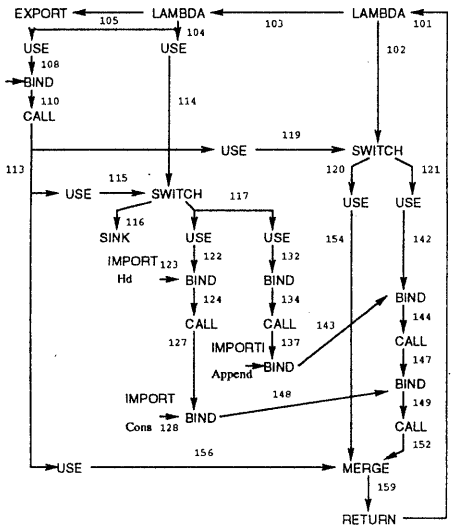

図 3: 関数 Append の Flow Graph

3 Strictness analysis

遅延評価型関数型言語では、引数が必ずしも評価されるとは限らない。もしコンパイル時に必ず評価される引数 (正格引数) がわかれば、先行評価や並列実行について有効であると言える。

- 逐次実行
  遅延評価の実現にはオーバーヘッドを伴う。正格引数がわかれば先行評価が可能になり効率良く評価する事が出来る。

- 並列実行
  正格引数は確実に評価されるので並列に計算を実行しても無駄になることはない。非正格引数は計算結果が使われない可能性があるので出来る限り正格引数を並列化する方が効率良く並列実行出来る。

Strictness analysis は Flow Graph に対して後方の RETURN node から行なう。strict かどうかという事は SUP(Strictness UP) によって表す。

- SUP=0
  情報がない。つまり strict かも知れないし、non-strict かも知れない。

- SUP=1
  strict である。

後方から edge をたどりながら node の SUP の値を次々と伝播していく。定義されている解析関数にしたがって解析は進行する。

4 並列性の抽出

関数型言語は副作用が存在しないために、プログラムから並列性を引き出す事は容易である。しかし、プログラムから全ての並列性を引き出すと並列化時のオーバーヘッドが増大して効率良い結果が得られない。そこでプログラムを解析し、並列実行する箇所を規定する必要がある。

5 並列化の方針

各関数、および各データオブジェクトについて属性として GRA を設定する。GRA は以下のようにして決定される。

- データオブジェクト
  一律で 0

- 関数
  LAMBDA で 0 からカウントを開始し、edge を RETURN までたどる。edge が分岐している場合は分岐してたどる。Graph 上で node を 1 移動するたびに GRA を 1 増加する。全てのルートの GRA の平均をその関数の GRA と定める。

GRA が大きい関数ほど処理単位が大きい関数だと考えられる。そこで GRA の値を参照して並列性の抽出を行なうと、粒度の大きい並列処理を行なうことが出来ると考えられる。そこで、並列化の方針として

- 正格な引数についてのみ並列化の対象とする。

- GRA の大きい引数から優先的に並列化を行なう。

解析の例

以下に解析の例を示す。

- 例
  f (Cons 2 3) (h x y) (i z) という関数適用があり、各関数の GRA が Cons = 2、h = 10、i = 8 とすると、コンパイラが GRA を比較して (h x y) と (i z) のタスクのみ並列に実行するコードを生成する事が可能となる。

6 実装

疎結合並列計算機 AP1000(64CPU) 上に遅延関数型言語処理系 Ginger[5] のサブセットを実装し、Flow Graph 解析を組み込む。

参考文献

[1] Benjamin Goldberg. Multiprocessor execusion of functional programs. *International Journal of Parallel Programming, Vol.17,No.5*, pp. 425–473, 1988.

[2] Pieter Hartel Hugh Glaser. A pragmatic approach to the analysis and compilation of lazy functional languages. *Proceedings of the 2nd Conference on Parallel and Distributed Processing*, 1990.

[3] David R.Lester Hugh Kingdon and Geoffrey L.Burn. The HDG-machine:a highly distributed graph-reducer for a trransputer network. *The Computer Journal,Vol.34,No 4*, pp. 290–302, 1991.

[4] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *The Journal of Functional Programming 2(2)*, pp. 127–202, 1992.

[5] Mike Joy. Ginger - a simple functional language. *Technical report RR235, University of Warwick*, 1992.

[6] John M.Wild Pieter H.Hartel, Hugh Glaser. Compilation of functional languages using flow graph analysis. *Software-Practice and Experience, Vol.24(2)*, pp. 127–173, 1994.

[7] 田中哲朗. 祖結合並列計算機用の関数型言語処理系. 日本ソフトウェア科学会第 10 回大会論文集, pp. 329–332, 1992.