A1-1

抽象度の高いアルゴリズム記述とプログラムの並列実行

Abstract description of algorithms and parallel execution of programs

田中 久美子† 岩崎 英哉† 武市 正人†
Kumiko TANAKA Hideya IWASAKI Masato TAKEICHI

†東京大学工学部

概 要

アルゴリズムを抽象度の高いレベルで記述し, 内在する並列性を抽出して, それを実現する並列プログラムを得る方法を考察する. アルゴリズムは高階関数を用いた関数プログラムで記述し, 標準の評価順序を変更するための少数の特殊関数を用いて並列性を表現する. これによって, 同一のアルゴリズムに対する逐次プログラムと同じ構造の並列プログラムを得ることができる. また, 関数プログラムの参照透明性によってプログラムの正しさを並列性とは独立に証明できる. 本発表では, 数種の並列アルゴリズムについて, 実際に関数プログラムの並列実行系を用いて実行した結果を報告する.

1 関数プログラムの並列実行

1.1 関数プログラムと並列性

関数プログラムは手続き型のプログラムに比べて並列化しやすいと言われるが, 現実には並列実行部分の抽出法は確立されていない. 本研究の焦点は, 計算機アーキテクチャや処理系とは独立に, プログラムに内在する並列アルゴリズムとプログラムの言語構造だけを最大限に利用して, 高い並列性を得るためのプログラム技法を確立することである.

ここでは以下の3つの並列性を追求する.

- プログラムに内在する並列アルゴリズムを生かした並列性
- 逐次実行時とプログラムの構造を変えない並列性
- 遅延評価と共存する並列性

1.2 アルゴリズムと並列性

効率良く高い並列性を得るためには, プログラムのもともとのアルゴリズムが並列的でなければならない. Quinn [2]は並列性を有するアルゴリズムを, 以下の3つに分類している.

- 分割法
  プログラムを同類の小問題に分割し, それぞれを並列に実行する.
- パイプライン法
  プログラムを互いに異なる小問題に分割し, 一つの小問題の出力を別の小問題の入力として並列に実行する.
- 緩和法 [5]
  並列に簡約を進め, 先に得られた一部分の結果を用いて計算結果を得る. 計算結果が得られる順序は非決定的である.

本稿でははじめの2つを扱う.

1.3 関数プログラムによるアルゴリズムの記述

関数プログラムでは高階関数の概念を用いることにより, 数少ない基本関数を用いて多種多様のアルゴリズムを抽象度の高いレベルで表現できる [1]. これらの少数の基本関数を並列化したものと, アルゴリズムに応じた並列化のための高階関数を用いれば, 逐次プログラムの構造を変更せずに並列に実行することができる. また, プログラムの正当性はアルゴリズムの並列性とは独立に示すことができる.

2 関数プログラムへの並列性の導入

2.1 評価系と並列関数 spec

本稿で扱う関数プログラムは遅延評価により頭部正規形 (WHNF) を得るものとする．以下のプログラムは関数型言語 Haskell [6]の表記法を用いる．

これまでに提唱されてきた，関数プログラムの並列化の表現にはさまざまなものがある [4] [7]が，本研究では 2 引数関数 spec を用いる．その意味は，

表示的意味　`spec f x = f x`

操作的意味　`x` と `f x` を並列に実行

である．spec は関数であるため処理系に一つの基本関数として加えるだけで済み，アノテーションなどのように関数とは別の構造を付け加える必要はない．しかも，関数であるために第一級の対象として扱うことができ，表現力も高い．

遅延評価を用いた関数型言語の処理系では，計算機資源を有効に使うために，評価順序を変更する strict という関数が用いられることがある [1]．その意味は，

表示的意味　`strict f x = f x`

操作的意味　`x` の評価を終えてから `f x` を実行

である．strict は spec を使ったプログラムの細かい操作には不可欠である．次の例で x と，y あるいは z を並列実行するには，

```
spec (\u -> if pred
            then (strict (u+) y)
            else (strict (u-) z)) x
```

と表現する．このように spec による並列実行をより生かすために，strict を用いる．

2.2 並列関数の簡約規則

プログラムがプログラマの意図どおりの並列動作をすることを示すのは，逐次動作の証明よりはるかに難しい．したがって，簡約規則を定めることはプログラムの並列動作の正当性を示すためにも重要である．簡約は正規順序（最外簡約）で行なう．以下の簡約規則では f や e は通常の式を表し，α や β は // の右辺に現れるものを指す．

strict

1. e が WHNF のとき
   `strict` $f\ e \Rightarrow f\ e$
2. $e \Rightarrow e'$ のとき
   `strict` $f\ e \Rightarrow f\ e'$

spec

1. e が WHNF のとき
   `spec` $f\ e \Rightarrow f\ e$
2. `spec` $f\ \alpha \Rightarrow f\ \alpha$
3. それ以外のとき
   `spec` $f\ e \Rightarrow f\ \alpha$ `//` α `= proc` e

proc

1. e が WHNF のとき
   `proc` $(e{:}es) \Rightarrow e$`:(susp` es`)`
   `proc` $e \Rightarrow e$　　（上記以外の時）
2. $e \Rightarrow e'$ のとき
   `proc` $e \Rightarrow$ `proc` e'

susp

`susp` $e \Rightarrow \alpha$ `//` α `= proc` e

//

1. e_1 または e_2 が WHNF のとき
   e_1 `//` α `= proc` $e_2 \Rightarrow e_1$
2. $e_1 \Rightarrow e_1'$ のとき
   e_1 `//` α `= proc` e_2
   $\Rightarrow e_1'$ `//` α `=` e_2
3. $e_2 \Rightarrow e_2'$ のとき
   e_1 `//` α `= proc` e_2
   $\Rightarrow e_1$ `//` α `= proc` e_2'

「プロセス」を簡約の過程に取り入れる．proc, susp, // はプログラム中には直接あらわれないが，簡約の途中段階で生じるものである．// によって並列に動作する簡約プロセスを表し，e_1 // e_2 では e_1 と e_2 のそれぞれを簡約する．proc はプロセスの動作している状態を表し，susp はプロセスの一時停止状態を表す．一時停止しているプロセスは，評価の要求を受けると proc となり，再び並列に簡約が行なわれる．また各構成子ごとに proc や susp の簡約規則を定める必要があるが，ここでは本稿で必要なリスト場合のみ示した．

2 つの式 e_1, e_2 が「等しい」($e_1 \cong e_2$) とは簡約規則による変換により，同じ WHNF に帰着できることと定義する．すると次のような法則の正当性を示すことができる．

法則 1

`spec . spec` $\cong$ `spec`

左辺　`spec (spec f) x`

spec の第 3 則より

$\Rightarrow$ `(spec f` α`) //` α `= proc x`

spec の第 2 則より

⇒ (f α) // α = proc x
右辺　spec f x
spec の第3則より
⇒ (f α) // α = proc x
法則2
strict f (spec (const x1) x2)
≅ spec (const (strict f x1)) x2
(但し, const x y = x)
左辺　strict f (spec (const x1) x2)
spec の第3則と strict の第2則より
⇒ strict f (const x1 α)
// α = proc x2
// の第2則より
⇒ (strict f x1) // α = proc x2
右辺　spec (const (strict f x1)) x2
spec の第3則より
⇒ (const (strict f x1) α)
// α = proc x2
// の第2則より
⇒ (strict f x1) // α = proc x2

3 関数プログラムの並列化

頻繁に使う少数の基本関数の並列版, およびアルゴリズムに即した並列化のための高階関数を用意し, プログラムの構造を変えずに並列実行する.

3.1 基本関数の並列化

foldl, foldr, map, filter の4つの基本関数は関数プログラムでは頻繁に使うものであり, これらを用いれば多種多様のプログラムを記述することができる [4]. したがってこれらを並列化した foldl_spec,map_spec,filter_spec を導入する. foldr は spec を挿入する意味のある場所がないので, これを並列化したものは用意しない.

foldl_spec の定義は以下である.

```
foldl_spec f a [] = a
foldl_spec f a (x:xs)
 = spec (foldl_spec f) (f a x) xs
```

遅延評価のために foldl はメモリを一時的にリストの長さに比例する分だけ必要とするが, 上の位置に spec を入れると, f a x の計算を並列に進めてメモリを節約することになる [1].

map_spec の定義は

```
map_spec f [] = []
map_spec f (x:xs)
   = (spec (:) (f x)) (map_spec f xs)
```

である. map-spec はリストの各要素の対する関数 f の適用を並列に行なうので, その並列性の本質は 1.2 節の分割法アルゴリズムに相当する.

filter_spec は map_spec と foldr を使って定義することができる [3].

```
filter_spec p xs = concat (map_spec f xs)
  where f x | p x        = [x]
            | otherwise  = [ ]
concat xs = foldr (++) [] xs
```

foldr の定義は

```
foldr f a [] = a
foldr f a (x:xs)
   = (1)(2)f x) (3)(4)(5)foldr f) a) xs)
```

である. spec や strict を挿入する可能性のある場所は, 上に番号をふって示した. 場所1と場所2は関数 f の性質に依存するので, foldr の定義を変えるのではなく, f として spec や strict をつけた関数を与えるべきである. 場所2については foldr は第3引数に対し正格であるため, 場所4については foldr の計算中は a が不変であるため, また, 場所5については foldr f が既に正規形であるために, いづれの場所も spec・strict を入れるのは適当ではない. したがって, foldr を並列化した基本関数は用意しないことにした.

3.2 分割法アルゴリズムの並列化

分割法の基本的なプログラムは map, filter を用いたものであるので, とくに分割法アルゴリズムのための並列化の関数は用意する必要はない. たとえば, 以下に示すのは第1引数のリストを第2引数のリストから探すプログラムである.

```
match_list xs ys
 = map (match xs) (tails ys)
match [] ys = True
match xs [] = False
match (x:xs) (y:ys)
 | x==y       = match xs ys
 | otherwise  = False
tails [] = []
tails xs = xs:(tails (tl xs))
```

match は計算量が多い関数である．これをリストの各要素に並列に適用するためにはプログラムの map を map_spec に変えるだけでよい．

3.3 パイプライン法アルゴリズムの並列化

関数プログラムではパイプラインは，ストリーム（リスト）を入力・出力とする関数の合成を，引数に対して適用することが，パイプラインに相当する．最も基本的なパイプラインは，

```
(f . g) xs
```

である．遅延評価は要求駆動であるから，たとえば

```
spec f (g xs)
```

のように，並列に関数を実行するだけでは並列化の効果は得られない．したがって，各関数の間に適当な大きさ（k とする）のバッファを設ける [4]．ここでは簡単のため，xs を無限列とする．

```
pipe k ls
 = spec (const (next ls (drop k ls)))
        (list_spec (take k ls))
next (x:xs) ys
 = spec (const (x:(next xs (tl ys))))
        (hd ys)
list_spec xs
 = spec (const xs) (ls xs)
   where ls []     = []
         ls (x:xs) = spec (ls xs) x
```

k を適当な定数とすれば，バッファ付きの関数合成 $\odot_k$ は，

```
f ⊙k g
  = f . (pipe k) . g
```

と定義される．これを用いて上のプログラムを並列化するには通常の関数合成を $\odot_k$ に変え，

```
(f ⊙k g) xs
```

とすればよい．

無限関数列をストリームに適用するプログラムは，再帰定義を用いることが多いので，関数合成がプログラム中に必ずしも陽には現れない．たとえば，素数列を生成するプログラムは

```
primes = map hd (iterate nr_sieve [2..])
iterate f xs = xs:(iterate f (f xs))
nr_sieve [] = []
nr_sieve (x:xs)
 = filter (pred x) xs
        where pred x y = (~=0).mod y x)
```

と記述できる [1]．ここで iterate は再帰定義により関数をストリームに繰り返し適用するプログラムを表現するための高階関数である．これを並列化した iterate_spec によってパイプラインのプログラムを並列実行する．

```
iterate_spec f xs
 = xs:(iterate_spec f (f (pipe k xs)))
primes_spec
 = map hd (iterate_spec nr_sieve [2..])
```

素数のプログラムの map hd に相当する部分は，問題に応じて適当なものを用いればよい．

4 まとめと展望

関数型言語を並列実行するために関数 spec を導入し，簡約規則を構築した．また，並列性を有する代表的なアルゴリズムのうち分割法，パイプライン法を用いて，抽象度の高いレベルで記述したプログラムをその構造を変えずに並列実行する手法を示した．そのためにプログラムで頻繁に使われている基本関数を並列化し，またアルゴリズムに応じて並列化のための高階関数を構成した．

今後は，緩和法に関して，分割法やパイプライン法と同様の，プログラムの構造を変えない並列化の手法を確立することが課題である．さらにこれらのアルゴリズムを組み合わせたより高いレベルのプログラムの効率的な並列化を研究していきたい．

参考文献

[1] R.Bird, P.Wadler. *Introduction to Functional Programming*, Prentice Hall, 1988.

[2] M.J.Quinn. *Designing Efficient Algorithms for Parallel Computer*, McGraw-Hill International Editions, 1987.

[3] R.Bird, R.Backhouse, G.Malcolm. *International Summer School on Constructive Algorithmics Part 1*, Lecture Notes at the Summer School, 1989.

[4] P.Roe. "Some Ideas On Parallel Functional Programming," *Functional Programming-Proceedings of the 1989 Glasgow Workshop*, Springer, 1989, pp.338-351.

[5] F.W.Burton. "Encapsulating non-determinacy in an abstract data type with determinate semantics," *Journal of Funcional Programming 1*, pp.3-20, 1991.

[6] P. Hudak et al. *Report on the Programming Language Haskell*, 1991.

[7] B.K.Szymanski et al. *Parallel Functional Languages*, ACM Press, 1991.

[8] 武市正人．「関数プログラムの並列実行に関する研究」，平成2年度科研費報告書，1991.