記号処理 26－5
（1983. 12. 16）

LISP関数の並列実行モデルとその評価

伊藤貴康　和田慎一
（東北大学工学部通信工学科）

1．まえがき

データフローマシンなどの並列的アーキテクチャの上でLISP処理系を実現しようとする研究が行われつつある。しかし、LISP処理系またはLISP関数に内在している並列性を解析し、LISP向きの並列実行モデルを理論的に解明しようとする研究は殆んど行われていないのが現状である。

LISPおよびその処理系に内在している並列性を整理し、LISP向きの並列実行モデルを考え、その上での処理速度について解析しておくことは理論的観点から重要なだけでなく、LISP向き並列アーキテクチャ、関数型言語向き並列アーキテクチャを考える上で重要である。このような観点から、筆者等は文献[1]および[2]でPure LISPを対象としていくつかの並列実行モデルを提案した。

LISP処理系の実現法としてインタプリタ方式とコンパイラ方式がある。逐次処理のスタックマシンモデル上でインタプリタ実行とコンパイラ実行の処理速度の理論的な比較を文献[1]で行い、Pure LISP関数のコンパイラ実行はインタプリタ実行よりも15倍以上高速であることを示した。

本稿では、まず並列実行モデルとその上でのインタプリタ実行・コンパイラ実行について述べ、各モデルでの並列性導入による高速化について概念的に比較する。次いで、インタプリタ実行とコンパイラ実行の実行ステップ数の比較を行ない、並列実行モデル上での処理速度について比較・検討する。

2．Pure LISPと並列実行モデル

この章ではPure LISPの処理における並列性と並列性導入による効果について説明し、並列性導入のレベルの違いによって3種類のモデルを文献[1]および[2]を基に与える。

2.1 evalquoteの高速化の方針

Pure LISPインタプリタevalquoteは図1のようにPure LISPによって記述されるが、evalquote向きの並列実行モデルについて考えてみる。文献[1]でも論じたようにevalquoteのインタプリタとしての処理内容は次のように分類できる。

①基本関数の処理
②式や関数のタイプの分類
③関数の評価
④関数の適用
⑤引数の評価
⑥条件式の処理
⑦変数の束縛
⑧変数値の探索

したがってevalquoteの高速化は

[方針1]：①～⑧の処理内容を並列実行モ

```
evalquote[fn;x] <=
  apply[fn;x;NIL]

apply[fn;x;a] <=
  [atom[fn]                                  基本関数の処理
      -> [eq[fn;CAR]    -> caar[x];
          eq[fn;CDR]    -> cdar[x];
          eq[fn;CONS]   -> cons[car[x];cadr[x]];
          eq[fn;ATOM]   -> atom[car[x]];
          eq[fn;EQ]     -> eq[car[x];cadr[x]];
          T             -> apply[eval[fn;a];x;a]];
                           関数適用   関数の評価
   eq[car[fn];LAMBDA]
                        -> eval[caddr[fn];pairlis[cadr[fn];x;a]];
                           関数本体の評価      変数束縛
   eq[car[fn];LABEL]
  関数型の分類          -> apply[caddr[fn];x;
                           関数適用  cons[cons[cadr[fn];caddr[fn]];a]]
                                                変数束縛

eval[e;a] <=
  [atom[e]                                  -> cdr[assoc[e;a]];----変数値の探索
   atom[car[e]]
      -> [eq[car[e];QUOTE]                  -> cadr[e];---------- QUOTEの処理
          eq[car[e];COND]                   -> evcon[cdr[e];a];---- CONDの処理
          T                                 -> apply[car[e];evlis[cdr[e];a];a];
   T                                        -> apply[car[e];evlis[cdr[e];a];a]]
  式の分類                                    関数適用          引数評価

assoc[x;a] <=
  [eq[caar[a];x] -> car[a];
   T             -> assoc[x;cdr[a]]]

evcon[c;a] <=
  [eval[caar[c];a] -> eval[cadar[c];a];
   T               -> evcon[cdr[c];a]]

evlis[m;a] <=
  [null[m] -> NIL;
   T       -> cons[eval[car[m];a];evlis[cdr[m];a]]]

pairlis[x;y;a] <=
  [null[x] -> a;
   T       -> cons[cons[car[x];car[y]];pairlis[cdr[x];cdr[y];a]]]
```

[注]ASSOCでは本来equalが使われるべきところを引数α(α-list)の構造からeqに置きかえてある

図1 Pure LISPインタプリタevalquoteの定義

デル上で実行することによる高速化と考えることができる。また、evalquoteは並列実行モデルの(機械)命令プログラムにコンパイルされてから実行されると考えると

[方針2]：evalquoteのコンパイルおよびそのオブジェクトの実行段階での並列性導入による高速化

も採用すべきである。

2.2 関数的処理の並列化

まず、Pure LISPの関数的な性質から、次のような並列性の導入が考えられる。これは[方針2]に基づく並列性の導入と考えてよい。

[IF]：関数適用の式 $f[e_1;\cdots;e_n]$ において引数 $e_1\cdots e_n$ を並列評価してから関数 f を適用する。

[C]：条件式 $[p_1\to e_1;\cdots;p_n\to e_n]$ において

[C-1]：各条件 $p_1\cdots p_n$ を並列的に評価し、$p_1=\cdots=p_{i-1}=$ false かつ $p_i=$ true[注] なる i を調べ、対応する結果 e_i を評価する。ただし i の値が確定した時点で $p_{i+1}\cdots p_n$ に対する評価は打ち切られるものとする。

[C-2]：各条件 $p_1\cdots p_n$ 並びに各結果 $e_1\cdots e_n$ を並列的に評価し、$p_1=\cdots=p_{i-1}=$ false かつ $p_i=$ true[注] なる i を調べ、e_i の値を結果として返す。ただし i の値が確定した時点で $p_{i+1}\cdots p_n, e_1\cdots e_{i-1}, e_{i+1}\cdots e_n$ に対する評価は打ち切られるものとする。

[IF]による高速化

[IF]による並列化の効果が現われるのは複数個の大きな処理時間を要する式が1つの関数の引数として与えられる場合である。図2のように任意の木構造の引数に対するコピーを作る関数copyは引数が深さ n の完全二進木の場合、処理時間は逐次処理ならば $O(2^n)$ であるのに対し、[IF]の採用により $O(n)$ となる。この例からも知られるように再帰的関数の場合には[IF]のような関数的並列性の導入は特に大きな効果を持つ。

```
copy[x] <=
    [atom[x] -> x;
     T -> cons[copy[car[x]];
               copy[cdr[x]]]]
```

図2 copyの定義

[C]による高速化

[C-1]による並列化の効果が現われるのは条件式の条件の数が多いとき、または複数個の大きな処理時間を要する式が条件として並んでいる場合である。また、[C-2]の並列化では条件式 $[p_1\to e_1;\cdots;p_n\to e_n]$ において、各条件 $p_1\cdots p_n$ の評価とは並列に各結果 $e_1\cdots e_n$ の部分も並列的に評価されているので

(条件式の処理時間)
$=\max\left[\max_{1\le k\le n}[(\text{条件}p_k\text{の処理時間})];\ (\text{結果}e_i\text{の処理時間})\right]$

となり、全体の処理時間において max の2つの項のうち、小さなほうが省かれることになる。そのため、2つの項の値がどちらも大きいときには効果が大きい。

今まで述べた関数的並列性はデータフローマシンなどで採用されている並列処理を理想化したものと考えてよい

2.3 リスト構造処理の並列化

evalquoteのPure LISPインタプリタとしての処理内容(2.1の①～⑧)を考えると、項目③,⑤,⑥,⑦および⑧は種々のリスト構造に対する処理となっていることがわかる。これらの処理の高速化のために次の並列性の導入を考える。

[IL]：リスト構造 $(l_1\cdots l_n)$ の各要素 $l_1\cdots l_n$ に対する独立した処理を並列実行する。

この並列性を「リスト構造の各要素を一斉に取り出して並列処理を行なう」,「並列的に処理した結果によるリストの再構成を並列的に行なう」などの高度の並列性を持つマシンの機能としてモデル化する。これは[方針1]に基づくモデルである。マシンモデルに並列的にリスト処理を行なう基本関数を導入し、これらの関数を用いてevalquoteに現われるリスト操作の関数を実現することができ、このようにして高速化を計るのである。たとえば変数値の探索に用いられるassoc[x;a]は図3のように定義を書き換えることによりa-listとして用いられる第2引数aの各要素のcar部とxが等しいかを並列的に調べ、その結果によりaの要素を選び出すという形に並列化できる。この結果、リストaの長さに関係なく一定の時間でassoc[x;a]の処理ができることになる。[IL]に基づく並列化については3.4で詳しく説明する。

2.4 並列実行モデルの設定

[注] 以降では真理値true, falseにより真偽の判定を行う代わりにnon-NIL, NILにより行う。

並列実行の基本的なメカニズムを
- [IF]
- [C-1], [C-2]
- [IL]

によって規定したが、Pure LISP処理系を実現する観点からこれらの有効な組み合わせとして次のようなモデルを設定する。

(逐次) assoc[x;a] <=
[eq[caar[a];x] -> car[a];
T -> assoc[x;cdr[a]]]

(並列) ≠assoc[x;a] <=
≠scan-select[@match[x;@car[a]];a]

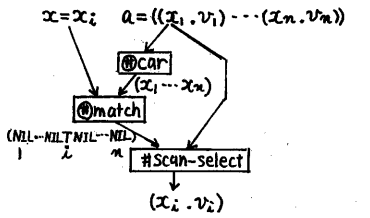


図3 assocの並列処理

(1) Δ-model
- Δ1-model：[IF], [C-1]に基づく実行モデル
- Δ2-model：[IF], [C-2]に基づく実行モデル

(2) #-model：[IF], [C-1], [IL]に基づく実行モデル

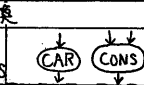
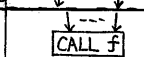
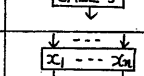
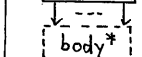
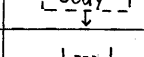
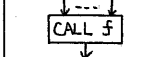
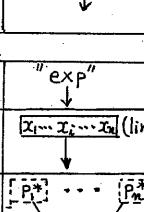
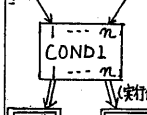
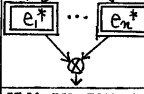
Δ-modelは関数的処理の並列化のみを採用したモデルでデータフローマシンの抽象的モデルとも考えられる。#-modelはさらにリスト構造処理の並列化を採用したモデルである。

3. 並列実行モデルでのコンパイラ実行方式・インタプリタ実行方式と並列実行による高速化

この章では前節で設定した各並列実行モデルでのコンパイラ実行について3.1でその実行方式を説明し、3.2で処理速度を概念的に比較する。次にインタプリタ実行の実行方式と処理速度についての概念的な検討結果について、3.3でΔ-modelの場合、3.4で#-modelの場合を説明する。また、3.2～3.4においてはインタプリタ実行、コンパイラ実行の速度の違いについても検討し、3.5で各モデルでの速度比の相違について検討する。

3.1 コンパイラ実行方式

コンパイラ実行方式ではPure LISP関数が並列実行モデルの命令語プログラムに変換されてから実行される。並列実行モデルでの命令の意味記述をグラフ(表現)を用いて与えることにし、図4にPure LISPの式や関数のグラフへの変換規則を示した。変換規則は(1)関数定義、(2)関数、(3)式に対する規則から成る。定義関数の適用に対しては"CALL f"というノードが生成されるが、そのノードが実行される時に関数fを定義するグラフにおきかえられ、そのグラフが実行されるものとする。またLAMBDA関数に対するグラフではlinkによりLAMBDA変数の引き渡しが表わされる。linkの入力はLAMBDA関数の引数であり、出力は本体(body)内での変数参照に用いられる。

(1)関数定義の変換		
(DE f args body)	f ⇐ (LAMBDA args body)*	関数fの定義内容をグラフへ変換して関数fの定義内容を登録する
(2)関数の変換		
基本関数 ex. CAR, CONS		car, consを適用する
定義関数 f		実行時には関数fの定義内容のグラフがコピーされそれが実行される.
LAMBDA関数 (LAMBDA (x1・・・xn) body)		linkはn個の入力が局所変数x1・・・xnとして扱われることを示す。body内部での局所変数の参照によりlinkからの出力がとられる。
LABEL関数 (LABEL f (LAMBDA args body))		別に(DE f args body)によりfを定義された関数として変換しそれを呼ぶ
(3)式の変換		
定数 (QUOTE exp)		定数expを出力
局所変数 xi		linkからxiに相当する出力をとり出す
条件式 (COND (P1 e1) ・・・ (Pn en)) [Δ1-modelの場合]		COND1がP1*・・・Pn*からの入力を調べP1*=・・・=Pn*=NIL, Pn+1*≠NILなるiを求めeiの入力制御を送る。eiのみが実行され、その値が"⊗"のノードを経て出力される。(P1・・・Pnの並列評価) □の内側では実行制御(⇒)の入力が与えられることにより実行を開始する。
条件式 [Δ2, #-modelの場合]		COND2がΔ1-modelの場合と同様なiを求めeiの値を出力する。(P1・・・Pn, e1・・・enの並列評価)
関数適用の式 (f e1・・・en)		並列的にe1・・・enの値が評価されfに適用される。

[注] 式eに対しe*はeのグラフであるとする

図4 Pure LISPからグラフ表現への変換規則

グラフでは基本関数(後述の#scan-selectを除く)と"CALL f"のノードは全ての入力がそろってから実行を開始し、他のノードはそれぞれに対して必要な入力が与えられれば実行を開始するものとする。ただし、図4の規則では自由変数、関数引数の処理は扱えない。

3.2 コンパイラ実行方式の処理速度の概念的評価

まず、コンパイラ(のオブジェクトの)実行をインタプリタ実行と比較すると、逐次処理の場合、文献[1]で説明したように次の点で高速になる:

(1) 式や関数のタイプの分類は不要である。
(2) 変数値は探索せずに取り出せる。
(3) 関数の評価は不要である。
(4) 引数の評価、変数の束縛などを引数リスト、変数リストなどのリスト処理でなく、より直接的に行なうことができる。

逐次処理のインタプリタを基準として各モデルでの処理速度を概念的に比較した表が図5である。コンパイラオブジェクト実行モードで考えると#-model と Δ_2-model には実質上差異はないと考えてよい。(なお#-model では、モデルに固有の基本関数が考えられるが、それらを除外して考えたとき)

3.3 Δ-model のインタプリタ実行方式と処理速度の概念的比較

Δ_1, Δ_2-model のインタプリタ実行方式は図1のevalquoteをその構造を変えないまま3.1における方法と同じ方法によりコンパイルして生成されたオブジェクトを実行する方式であると考える。

インタプリタ実行ではevalquoteの記述に含まれる並列性が

Δ_1-model ; [IF], [C-1]

Δ_2-model ; [IF], [C-2]

によって並列化される。以下 各並列化の項目ごとにその導入による効果を検討する。

[IF]による高速化

[IF]の導入により、インタプリタとして関数適用の式 $(f\ e_1 \cdots e_n)$ の評価を行なう際、並列化の効果が現われる。図6はevalquoteの関数適用の式 $(f\ e_1 \cdots e_n)$ に対する評価の様子を表わしているが、evlisが再帰的に呼び出されると一番下の木のような状態になる。そこではn個のconsにおいて[IF]によりそれぞれの引数の並列評価が行われるため、eval[e_1;a]…eval[e_n;a]が並列的に行われる。すなわち、インタプリタ上でもコンパイラ実行の場合と対応した[IF]の並列化が実現できることになる。ただし、eval[e_n;a]の呼び出しはn回のevlisの再帰的呼び出しの後に行われる、また、答えのリストを構成するためのn個のconsも下から順に行われなければならないなど、リスト構造を逐次的に処理するための時間(リスト構造処理のオーバーヘッドと呼ぶ)が全体の実行時間に含まれることになる。この意味で前に述べた「インタプリタ上での[IF]の実現」は不完全なものである。

処理内容 \ モデル	逐次処理モデル I	逐次処理モデル C	Δ_1-model I	Δ_1-model C	Δ_2-model I	Δ_2-model C	#-model I	#-model C
基本関数の処理	—	—	—	—	—	—	—	—
式や関数のタイプの分類	—	不要	並列化1	不要	並列化2	不要	並列化2	不要
引数の評価(引数リストの処理)	—	L	[IF]	[IF],L	[IF]	[IF],L	[IF],L	[IF],L
条件式の処理(条件式本体のリストの処理)	—	L	—	[C-1],L	[C-2]	[C-2],L	[C-2],L	[C-2],L
変数の束縛(変数リストの処理)	—	L	—	一定時間 L	—	一定時間 L	一定時間 L	一定時間 L
変数値の探索 関数の評価(a-listの処理)	—	一定時間 L	—	一定時間 L	—	一定時間 L	一定時間 L	一定時間 L

—:逐次処理のインタプリタと比較して概念的な差はない
並列化1:各分類の処理の並列化
並列化2:分類並びに分類に応じた処理の並列化
[IF],[C-1],[C-2]:各並列化
L:リスト構造処理のオーバーヘッドの消滅
一定時間:一定の時間で処理可能
I はインタプリタ実行
C はコンパイラ実行

図5 処理速度の概念的比較

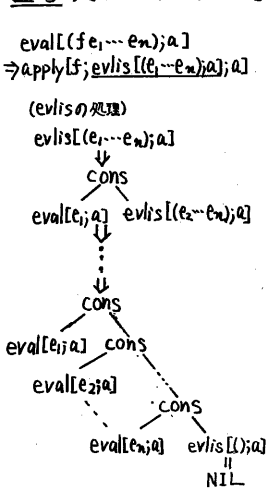


図6 インタプリタの関数適用の式に対する処理

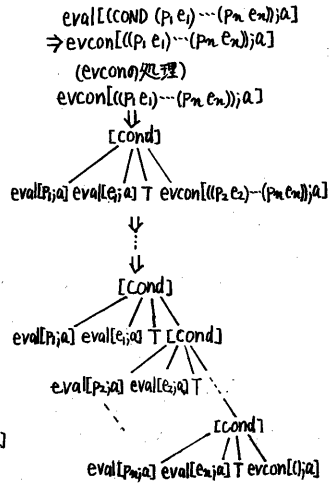


図7 インタプリタの条件式に対する処理

[IF]の導入により、インタプリタ、コンパイラの両方とも引数の並列的評価が行なわれるようになるが、インタプリタの関数定義を詳細に分析すると[IF]の導入はコンパイラよりもインタプリタの高速化により有効であると考えられる。

[C-1]による高速化

[C-1]の並列性の導入によりeval, applyの中の条件式の各条件が並列評価され、式や関数のタイプの分類が高速化される。しかし、[C-1]の導入によりコンパイラ実行で生じる効果と同様な効果がインタプリタの上で生じることはない。すなわち、インタプリタが条件式を評価するとき、

各条件 $P_1 \cdots P_n$ に対する評価 eval[P_i;a] ～～ eval[P_n;a] は並列的に実行されることはなく、逐次的に実行される。このように、[C-1]の導入はインタプリタ、コンパイラに対し、それぞれ異なった形の高速化をもたらすが、多くの場合、条件式の条件の数は少ないため、[C-1]の導入もコンパイラよりインタプリタの高速化に有効であると考えられる。

[C-2]による高速化

まず、[C-2]の並列性は[C-1]を包括するので[C-1]による高速化が行われる。さらに以下のように高速化される。

(a) 式や関数のタイプの分類と同時に各場合に応じた処理が並列的に行われる。

(b) インタプリタ上においてもコンパイラ実行の場合と同様の[C-2]導入の効果が現われる。

(b)の効果の現われ方は[IF]による高速化と類似した形のものである。図7にevalquoteが条件式を評価する様子が示されているが、一番下の木で、各[cond]に対し、その子孫にあたる各部分が並列的に実行されるため、全体として eval[P_i;a] … eval[P_n;a] 並びに eval[e_i;a] … eval[e_n;a] が並列的に実行されることになる。ただし、条件式のリストの構造に応じてevconを再帰的に呼び出すなどリスト構造を逐次的に処理するための時間(リスト構造処理のオーバーヘッド)が全体の実行時間に含まれる。[C-2]の導入により、インタプリタでは(b)の効果に加えて(a)の効果が現われる。したがって[C-2]の導入もコンパイラよりインタプリタの高速化により有効であると考えられる。

3.4 #-modelのインタプリタ実行方式と処理速度

#-model1はΔ_2-model1にリスト構造処理の並列性[IL]を加えたものである。#-modelのインタプリタ実行はリスト構造処理の並列性[IL]を具体的に与えることによって表現することができる。文献[1]で与えた#-evalquoteはその一例である(図8参照)。#evalquoteもΔ_2-model1と同じ方法により([IF],[C-2]の並列性の採用)コンパイルして実行されるものとする。#evalquoteではリスト構造の並列処理のため次のような関数が用いられている。

(1) ㊞一関数：関数名の頭に㊞がついており、引数として与えられるリストの各要素に対してその関数が並列的に適用され、その結果をリストにして返す関数である。図8の㊞car, ㊞cdrなどは㊞一関数の例である。㊞一関数の処理は
(各要素に対する処理時間)＋(オーバーヘッド)
の時間でできるとし、理想的にはオーバーヘッドは0である。

(2) #scan-select[($x_1 \cdots x_n$);($y_1 \cdots y_n$)]：第1引数のリスト($x_1 \cdots x_n$)の各要素 $x_1 \cdots x_n$ を並列的に調べ、$x_1 = \cdots = x_{i-1} = NIL$, $x_i \neq NIL$ となるiを求め、対応する第2引数のリストの要素 y_i を返す。ただし、#scan-selectはそれに対する引数が評価されている段階から実行が開始され、第1引数の各要素ごとに値の確定を調べ、$x_1 \cdots x_i$ の値が確定してiが求まっ

```
#evalquote[fn;x] <= #apply[fn;x;NIL]

#apply[fn;x;a] <=
  [ atom[fn] -> [#eq[fn;CAR]  -> caar[x];
                 #eq[fn;CDR]  -> cdar[x];
                 #eq[fn;CONS] -> #cons[car[x];cadr[x]];
                 #eq[fn;ATOM] -> atom[car[x]];
                 #eq[fn;EQ]   -> #eq[car[x];cadr[x]];
                 T            -> #apply[#eval[fn;a];x;a]];
    #eq[car[fn];LAMBDA]
      -> #eval[caddr[fn];#pairlis[cadr[fn];x;a]];
    #eq[car[fn];LABEL]
      -> #apply[caddr[fn];x;#cons[#cons[cadr[fn];caddr[fn]];a]]]

#eval[e;a] <=
  [atom[e]  -> cdr[#assoc[e;a]];
   atom[car[e]]
     -> [#eq[car[e];QUOTE] -> cadr[e];
         #eq[car[e];COND]  -> #evcon[cdr[e];a];
         T                 -> #apply[car[e];#evlis[cdr[e];a];a]];
   T -> #apply[car[e];#evlis[cdr[e];a];a]]

#assoc[x;a] <= #scan-select[@match[x;@car[a]];a]

#evcon[c;a] <= #scan-select[@eval[@car[c];a];
                            @eval[@car[@cdr[c]];a]]

#evlis[m;a] <= @eval[m;a]

#pairlis[x;y;a] <= #conc2[@cons[x;y];a]


@car[(el ... en)] = (car[el] ... car[en])

@cdr[(el ... en)] = (cdr[el] ... cdr[en])

@cons[(x1 ... xn);(y1 ... yn)]
  = (#cons[x1;y1] ... #cons[xn;yn])

@eval[(el ... en);a]
  = (#eval[el;a] ... #eval[en;a])


@match[x;(a1 ... an)] = (#eq[x;a1] ... #eq[x;an])

#scan-select[(x1 ... xn);(y1 ... yn)] = yi
     (x1 = NIL, ... ,xi-1 = NIL and xi ≠ NIL のとき)

#conc2[(x1 ... xn);(y1 ... ym)] = (x1 ... xn y1 ... ym)
```

図8 #evalquoteの定義

た段階で y_n を第2引数から取り出し、答えとして返す。 #scan-select の処理のためには
($x_1 \cdots x_n$ に基づき i を決定するための時間)
+(オーバーヘッド)
の時間が必要である。理想的にはオーバーヘッドは0であり、また i を決定する処理は $O(\log n)$ で行なうことができ、簡単なものであろから、ほぼ一定の時間で処理できると考えられる。

(3) #conc2[$(x_1 \cdots x_n)$;$(y_1 \cdots y_m)$]: append[$(x_1 \cdots x_n)$;$(y_1 \cdots y_m)$] と同じ意味であるが、第1引数のリストに対する並列処理を行なうことによりリスト長に関係なく一定時間で処理する。

なお、関数名の頭に#の付いた関数は引数が2以上であり、#-model上で並列処理される関数であることを示している。前述の関数を用いることにより #evalquote では図9の関数においてリスト構造に対する処理が並列化されている。このためリスト構造処理のオーバーヘッドがなくなる。また、変数値の探索等のための #assoc は #scan-select, ㊓match の導入により a-list を完全に並列的に扱うため、a-list の長さによらず一定時間で処理できる。特に a-list の深い位置にある自由変数の探索に対して大きな効果が現われる。

関数	並列的に扱われるリスト	対応する処理内容
#assoc[x;a]	a:a-list	変数値の探索・関数の評価
#evcon[c;a]	c:条件式本体のリスト ((p1 e1)…(pn en))	条件式の処理
#pairlis[x;y;a]	x:変数(仮引数)リスト y:実引数リスト	変数束縛
#evlis[m;a]	m:(評価前の)引数のリスト	引数評価

図9 リスト構造処理の並列化の適用される関数

図5の各モデルのインタプリタについての内容は上述の議論に基いたものである。

3.5 インタプリタ実行とコンパイラ実行の速度比のモデル間での相違

3.2 ～ 3.4 の検討結果を用いてインタプリタ実行とコンパイラ実行の速度比のモデルによる違いについて整理する。

まず、3.3で検討したように[IF],[C-1],[C-2]の導入はいずれもインタプリタの高速化により有効であることから速度比は逐次型モデル、Δ_1-model, Δ_2-model の順に小さくなっていくと考えられる。さらに[IL]の並列性を導入した#-model では多くの場合インタプリタのみに[IL]の並列化が適用されるため当然速度比はより小さくなる。

Δ_1, Δ_2-model では逐次型モデルと比較した場合、インタプリタにおいて式や関数のタイプの分類が並列化され高速処理される。さらに、#-model では、リスト構造処理のオーバーヘッドが消滅し、変数値の探索、関数の評価が一定時間で行われるため、3.2で検討したインタプリタとコンパイラの概念上の差がほぼなくなっている。実行ステップ数による比較については4.2を参照されたい。

4. インタプリタ実行とコンパイラ実行の実行ステップ数による速度比較

3章において各モデルでのインタプリタ実行とコンパイラ実行の速度の相違について概念的に比較したが、この章では実行ステップ数による比較・検討を行なう。文献[1]でスタックマシンに対して用いた方法と同様な方法により比較を行う。逐次処理のモデルも含めて各モデルのインタプリタ実行とコンパイラ実行の速度比最悪値を示した。

4.1 Pure LISP関数の実行ステップ数の計算

ステップ数は理想的な仮定の下で計算する。すなわち、並列実行の制御、種々のデータ転送が理想的に行なわれ、それらによるオーバーヘッドはなく、基本関数処理のような Pure LISP と直接対応した処理のみが実行時間に反映されると仮定する。また、局所変数や関数呼び出しのための処理も実行時間に影響を与えないと仮定する。図10の命令実行ステップ値に基いて Pure LISP 関数の実行ステップ数の計算を行なう。Pure LISP の基本関数のステップ値は文献[1]のスタックマシンの場合と同じにした。

関数	値
car	1
cdr	1
cons	1
atom	1
eq	1
cond※	1

関数	値
#scan-select	1
#conc2	1
㊓match	1
他の㊓一関数	㊓のないときの関数の実行ステップ

※:condとは条件式[P1→e1;…;Pn→en]において P1=…=Pi-1=NIL, Pi≠NIL なるiを求めるための時間をさす。

図10 各命令(関数)の実行ステップ値

ここで新たに並列実行モデルとの比較のために逐次処理モデル『seq-model』を設定する。seq-model は各命令の実行ステップ値は並列実行モデルと同じであるが、[IF],

[C-1],[C-2],[IL]のいずれの並列化も採用されないと仮定したモデルである。

次にPure LISP関数や式に対するコンパイラオブジェクトの実行ステップ数を求める方法を説明する。Pure LISP関数fnまたは式eに対するコンパイラオブジェクト実行ステップ数を名前の頭にSをつけてSfnあるいはSeのように表わすものとし、また、モデルに応じて変化する次のような関数を用いることにする。

- 定数 $C \Rightarrow 0$
- 局所変数 $x \Rightarrow 0$
- 条件式 $[p_1 \to e_1; \cdots; p_n \to e_n]$

$$\Rightarrow [p_1 \to \max 2[sp_1+1; se_1];\ p_2 \to \max 2[\max 1[sp_1; sp_2]+1; se_2];\ \vdots\ p_n \to \max 2[\max 1[sp_1; \cdots; sp_n]+1; se_n]]$$

- 関数適用の式 $f[e_1; \cdots; e_n]$

$$\Rightarrow Sfn[v_1; \cdots; v_n] + \max 1[se_1; \cdots; se_n]$$

($v_1 \cdots v_n$ は $e_1 \cdots e_n$ の評価された値)

$$Sfn[x_1; \cdots; x_n] = \begin{cases} fnの命令実行ステップ値 & (fnが基本関数のとき) \\ sbody\ (fn自身またはその定義内容が\ lambda[[x_1; \cdots; x_n]body]のとき) \end{cases}$$

図11 コンパイラ実行ステップ数導出の方法

```
sevalquote[fn;x]  =  sapply[fn;x;NIL]

sapply[fn;x;a] =
  [atom[fn] ->
      [eq[fn;CAR]     -> <6,6,2,2>                              ;
       eq[fn;CDR]     -> <7,6,2,2>                              ;
       eq[fn;CONS]    -> <12,9,5,5>                             ;
       eq[fn;ATOM]    -> <9,6,2,2>                              ;
       eq[fn;EQ]      -> <12,7,3,3>                             ;
       T              -> <9,4,0,0> + seval[fn;a]
                         + sapply[eval[fn;a];x;a]               ];
   eq[car[fn];LAMBDA] -> <9,5,2,2>+spairlis[cadr[fn];x;a]
                         + seval[caddr[fn];
                                 pairlis[cadr[fn];x;a]] ;
   eq[car[fn];LABEL]  -> <20,12,9,9>
                           + sapply[caddr[fn];x;
                                    cons[cons[cadr[fn];
                                         caddr[fn]];a]]

seval[e;a] =
  [atom[e]              -> <3,3,1,1>+sassoc[e;a]                ;
   atom[car[e]] ->
    [eq[car[e];QUOTE]   -> <9,8,3,3>                            ;
     eq[car[e];COND]    -> <10,7,1,1>+sevcon[cdr[e];a]          ;
     T                  -> <11,7,1,1>+sevlis[cdr[e];a]
                           +sapply[car[e];
                                   evlis[cdr[e];a];a]           ];
   T                    -> <6,3,1,1>+sevlis[cdr[e];a]
                           +sapply[car[e];
                                   evlis[cdr[e];a];a]           ]

sassoc[e;a] = <5*n,5*n,n+3,3>  (xが連想リストaのn番目の位置にあるとき)

sevcon[c;a] =  ( c=((P1 e1)⋯(Pn en))でP1⋯Pi-1が不成立, Piが成立のとき)
          i
  <4*i+2+ Σ seval[pk;a]+seval[ei;a],
          k=1
          i
   4*i+2+ Σ seval[pk;a]+seval[ei;a],
          k=1
   2+max[seval[p1;a]+1;...;seval[pi;a]+i;seval[ei;a]+i],
   2+max[seval[p1;a];...;seval[pi;a];seval[ei;a]+1]>

sevlis[m;a] =  ( m=(e1⋯en)のとき)
          n
  <7*n+2+ Σ seval[ei;a],
          i=1
   2+max[6*i+seval[ei;a]],
    1≤i≤n
   2+max[4*i+seval[ei;a]],
    1≤i≤n
     max[seval[ei;a]]>
    1≤i≤n

spairlis[x;y;a] = <12*n+2,6*n+3,4*n+3,4>  (xとyが長さnのリストのとき)
```

図12 インタプリタ実行ステップ数sevalquote

$$max1 = \begin{cases} +(加算) & ; seq\text{-}mode1 \\ max(最大値) & ; \Delta_1, \Delta_2\text{-}mode1 \end{cases}$$

$$max2 = \begin{cases} +(加算) & ; seq, \Delta_1\text{-}mode1 \\ max(最大値) & ; \Delta_2, \#\text{-}mode1 \end{cases}$$

Pure LISPの式は図11のような方法でコンパイラオブジェクトの実行ステップ数を表わす式に変換できる。

定数 (QUOTE e)

C: sobj[(QUOTE e)] = <0,0,0,0>

I: seval[(QUOTE e);a] = <1,3,3,3>

局所変数 x　xがa(リスト)のn番目にあるとき

C: sobj[x] = <0,0,0,0>

I: seval[x;a] = $<5n+3, 5n+3, n+4, 4> \geq <8,8,5,4>$

条件式 (COND $(p_1\ e_1) \cdots (p_n\ e_n)$)

$p_1 \cdots p_{i-1}$ が不成立で p_i が成立するとき

C: sobj[(COND $(p_1\ e_1) \cdots (p_n\ e_n)$)]

$= max2[1+max1[sobj[p_1]; \cdots; sobj[p_i]]; sobj[e_i]]$

I: seval[(COND $(p_1\ e_1) \cdots (p_n\ e_n)$);a]

$$= \begin{cases} 4i+12+\sum_{k=1}^{i} seval[p_k;a]+seval[e_i;a] & \cdots seq \\ 4i+9+\sum_{k=1}^{i} seval[p_k;a]+seval[e_i;a] & \cdots \Delta_1 \\ 3+\max[1+seval[p_1;a]; \cdots; i+seval[p_i;a]; i+seval[e_i;a]] & \cdots \Delta_2 \\ 3+\max[seval[p_1;a]; \cdots; seval[p_i;a]; 1+seval[e_i;a]] & \cdots \# \end{cases}$$

関数適用の式 ($f\ e_1 \cdots e_n$)

基本関数

CAR (CAR e)

C: sobj[(CAR e)] = 1+sobj[e]

I: seval[(CAR e);a] = <26,21,9,3> + seval[e;a]

CDR (CDR e)

C: sobj[(CDR e)] = 1+sobj[e]

I: seval[(CDR e);a] = <27,21,9,3> + seval[e;a]

CONS (CONS $e_1\ e_2$)

C: sobj[(CONS $e_1\ e_2$)] = 3+max1[sobj[e_1];sobj[e_2]]

I: seval[(CONS $e_1\ e_2$);a]

= <27,24,12,6> + max1[seval[e_1;a]; <12,6,4,0> + seval[e_2;a]]

ATOM (ATOM e)

C: sobj[(ATOM e)] = 1+sobj[e]

I: seval[(ATOM e);a] = <29,21,9,3> + seval[e;a]

EQ (EQ $e_1\ e_2$)

C: sobj[(EQ $e_1\ e_2$)] = 1+max1[sobj[e_1];sobj[e_2]]

I: seval[(EQ $e_1\ e_2$);a]

= <27,22,10,4> + max1[seval[e_1;a]; <12,6,4,0> + seval[e_2;a]]

定義された関数 ($f\ e_1 \cdots e_n$) (DE $f(x_1 \cdots x_n)$ body)の形で定義

C: sobj[($f\ e_1 \cdots e_n$)] = $sobj[body] + \max_{1 \leq i \leq n} 1[sobj[e_i]]$

I: seval[($f\ e_1 \cdots e_n$);a] = $<19n+41, 6n+29, 4n+13, 11> + seval[body;a] + \overset{*}{sargs}$

LAMBDA関数 ((LAMBDA $(x_1 \cdots x_n)$ body) $e_1 \cdots e_n$)

C: sobj[((LAMBDA ⋯) ⋯)] = $sobj[body] + \max_{1 \leq i \leq n} 1[sobj[e_i]]$

I: seval[((LAMBDA ⋯) ⋯);a] = $<19n+19, 6n+16, 4n+10, 7> + seval[body;a'] + \overset{*}{sargs}$

LABEL関数 ((LABEL f (LAMBDA $(x_1 \cdots x_n)$ body)) $e_1 \cdots e_n$)

C: sobj[((LABEL ⋯) ⋯)] = $sobj[body] + \max_{1 \leq i \leq n} 1[sobj[e_i]]$

I: seval[((LABEL ⋯) ⋯);a] = $<19n+39, 6n+28, 4n+19, 16> + seval[body;a'] + \overset{*}{sargs}$

$$*: sargs = \begin{cases} \sum_{i=1}^{n} seval[e_i;a] & \cdots seq \\ \max_{1 \leq i \leq n}[seval[e_i;a]+6i] & \cdots \Delta_1 \\ \max_{1 \leq i \leq n}[seval[e_i;a]+4i] & \cdots \Delta_2 \\ \max_{1 \leq i \leq n}[seval[e_i;a]] & \cdots \# \end{cases}$$

図13 インタプリタ実行ステップ数とコンパイラ実行ステップ数の比較

きる。また、3.3, 3.4で述べたように各モデルのインタプリタはevalquote(または#evalquote)のコンパイラオブジェクトの実行であるので、同じ方法をevalquote(または#evalquote)に対して適用することによりインタプリタ実行のステップ数sevalquoteを得ることができる(図12)。ただし、seq, Δ_1, Δ_2, #-modelにおける値がa,b,c,dのとき<a,b,c,d>のように表現している。(この記法を以降でも用いる。)

4.2 インタプリタ実行とコンパイラ実行の速度比較

式eに対するインタプリタ実行ステップ数は図12のsevalquote内のseval[e;a]によって与えられる。また式eに対するコンパイラオブジェクトの実行ステップ数をsobj[e]により表わす。seval, sobjをPure LISPの式の各場合について比較すると図13のようになる。図13において比較されているseval, sobjの式の右辺において引数の対応するseval, sobjを除いた部分(2重下線)を比較し、構造的帰納法を用いることにより、すべてのPure LISPの式について次のようになることが示される。

$$\frac{\text{seval}[e;a]}{\text{sobj}[e]} \geq \langle 13, 8, 4, 2 \rangle$$

最低値は図13の"CONS"の場合に対する比較により与えられる。

関数append, copyに対してステップ数を計算してみると図14のようになる。

インタプリタ実行、コンパイラ実行の速度比は最低値, 図14での具体的な値, 図13の二重下線部の値のいずれにおいてもseq, Δ_1, Δ_2, #のモデルの順に小さくなっていく傾向がある。この傾向は3章において概念的に検討した結果とも合致している。

5. あとがき

データフローマシンなどで採用されている関数的処理の並列化を採用した

Δ-model

さらにリスト構造処理の並列化を導入した

#-model

を用いてPure LISP関数の処理速度を比較した。#-modelではインタプリタ#evalquoteにおいてリスト構造処理の並列化の採用により、高速化される。特に変数値の探索を行なうassocの処理が#scan-select, #matchを用いることにより一定時間で処理できるが、assocの処理の並列化はLISP処理に限らず、他の場合でも適用できると考えられる。

また各モデルでのインタプリタ実行とコンパイラ実行の速度を比較したが、速度比は、逐次, Δ_1, Δ_2, #のモデルの順に小さくなっている。コンパイラ実行の場合、コンパイル操作を別に行なう必要があり、また関数引数などの完全な扱いが困難であることなどの課題が残る。さらにインタプリタのほうがプログラミング環境として好ましいことから、#-modelのような並列処理においてはインタプリタのほうが理論、実際の両面から好ましいとも考えられる。

今後の課題として

(1) コンパイラでの関数引数、自由変数の処理を可能にするモデル
(2) ハードウェア実現上の制約を考慮したモデルとその上での評価
(3) リスト構造処理の並列性をインタプリタ内部で用いるだけでなく、評価の対象とする関数や式についても適用できるようにインタプリタおよびモデルを拡張すること

などがあげられる。

		APPEND	n=0	n=10	n=∞	COPY	n=0	n=10	n=∞
seq-model	I	$265n+162$	162	2812	—	$353 \cdot 2^n - 224$	129	361248	—
	C	$7n+2$	2	72	—	$9 \cdot 2^n - 7$	2	9209	—
	I/C		81	39.0	37.8		64.5	39.2	37.2
Δ_1-model	I	$152n+112$	112	1632	—	$149n+95$	95	1585	—
	C	$6n+2$	2	62	—	$6n+2$	2	62	—
	I/C		56.0	26.3	25.3		47.5	25.6	24.8
Δ_2-model	I	$54n+46$	46	586	—	$50n+38$	138	538	—
	C	$4n+2$	2	42	—	$4n+2$	2	42	—
	I/C		23	13.9	13.5		19.0	12.8	12.5
#-model	I	$31n+24$	24	334	—	$31n+24$	24	334	—
	C	$4n+2$	2	42	—	$4n+2$	2	42	—
	I/C		12.5	8.0	7.8		12.5	8.0	7.8

n : APPEND --- 第1引数のリストの長さ
COPY --- 完全二進木の深さ

図14 具体的なステップ数計算の例

<文献> [1] 伊藤、田村、和田: LISPコンパイラとインタプリタの処理速度の理論的比較, 情処学会記号処理研23-3 (1983), [2] 伊藤、和田: LISP関数の並列実行モデルとその評価 ― Pure LISP関数に対する関数型並列実行モデル ―, 情処学会第27回全大 1G-9 (1983), [3] McCarthy etel.: LISP1.5 Programmer's Manual, MIT Press (1962), [4] J. Allen: Anatomy of LISP, McGraw-Hill (1978)