プログラミング 5－6
（1996. 1. 26）

Schemeプログラムの自動並列化とスティール評価法による実行

川本 真一　　伊藤 貴康

東北大学情報科学研究科

本稿は，Schemeプログラムを構造化並列構文を用いた並列的Schemeプログラムに自動変換し，スティール評価法によって効率よく実行する，制約的自動並列化と呼ぶ簡便な自動並列化法を提案する．Lispの方言であるSchemeで書かれた副作用のないプログラムを考え，制約的自動並列化法の有効性を示す．並列構文としては並列Lisp言語の **pcall** などの構造化並列構文を用いる．Schemeプログラムの構文と実行コストの知識に基づいて，プロセス生成コストなどによる実行時オーバーヘッドが小さく効率良く実行できる並列プログラムを生成し，構造化並列構文を効率良く実行できるスティール評価法によって，得られた並列プログラムを効率よく実行するというのが制約的自動並列化法である．評価実験は，並列計算機DEC7000上に実現されているPaiLisp/MTシステムを拡張して行い，本論文の制約的自動並列化の有効性が実験的にも示されている．

Automatic Parallelization of Scheme Programs and their Evaluations by Steal-based Evaluation Strategy

Shin-ichi Kawamoto　　Takayasu Ito

Department of Computer and Mathematical Sciences,
Graduate School of Information Sciences,
Tohoku University

This paper introduces a simple and efficient method of automatic parallelization of functional programs written in Scheme. The method firstly transforms a sequential functional Scheme program into the corresponding parallel Scheme program with structured parallel constructs like **pcall**, and then the resultant parallel Scheme program will be efficiently evaluated with the Steal-based Evaluation (SBE) strategy. In order to suppress excessive insertions of parallel constructs that incur overheads caused by process creation the transformation from *sequential* into *parallel* employs a method called *Constraint-based Automatic Parallelization* (CAP) that uses simple syntactic information of basic Scheme expressions and their computation-cost estimates. The CAP method based on the SBE strategy is shown to be very efficient for a set of benchmark programs. The experimental evaluations were done with an extended version of PaiLisp/MT on DEC7000 with 6 Alpha processors under OSF/1 OS.

1 はじめに

Lispプログラムの自動並列化の試みとして，Harrison[2]による研究がある．この方法はFORTRANプログラムなどの自動並列化において行われている手続き間の関係やプログラムの繰り返しおよび再帰構造の解析によって並列性を自動抽出する方法であるが，Lispプログラムの再帰性や不規則なデータ構造のため解析手法は複雑である．Lispプログラムの並列化に関しては，並列Lisp言語とその処理系を作成し，プログラマが並列プログラミングを行うのが主流であり，様々な並列Lisp言語[1,3]が設計され，その処理系が作成されて来ている．

並列Lisp言語の研究で得られた構造化並列構文を用いてSchemeプログラムを並列化し，スティール評価法という構造化並列構文を効率良く実行できる簡便な自動並列化の方法が文献5)で提案されている．この考え方を下に，Schemeプログラムの式の評価コストに基づく制約を用いて簡便な自動並列化を行う制約的自動並列化変換を与える．また，並列計算機DEC7000上で実現されたPaiLisp/MTシステム[5]を用いた評価実験の結果からこの並列化法が有効な方法であることを示す．

2 Schemeプログラムの並列化の問題点

Schemeプログラムを並列構文を用いて並列化する場合，並列構文として **future** 構文を用いる方法と **pcall** 構文などの構造化並列構文を用いる方法がある．

式 $(f\ e_1\ e_2)$ において，$(f$ (**future** $e_1)\ e_2)$ のように **future** 構文を挿入すると，e_1 の実行が親プロセス対して並列的に行われる．**future** 構文にはLTC (Lazy Task Creation) と呼ばれる効率の良い評価法が提案され，Mul-T[6]やPaiLisp/MT[5]において実装が行われ，その有効性が確められている．しかし，**future** 構文はそれを挿入する位置や個数によってプログラムの実行効率が変り，最適な所に **future** 構文を自動的に挿入することは困難であることが知られている．

一方，並列Lisp言語PaiLispには，**pcall**, **par**, **par-and**, **par-or**, **plet**, **pletrec** などの様々な構造化並列構文が用意されている．**pcall** 構文は，関数引数の評価を並列に行う構文で，式 $(f\ e_1 \cdots e_n)$ の並列化は (**pcall** $f\ e_1 \cdots e_n$) のように書かれ，その意味は，まず $e_1, \cdots, e_n$ を並列に評価し，次に f を評価し，$e_1, \cdots, e_n$の値に f の値を適用する．**pcall** 構文のSchemeプログラムへの挿入は，関数適用式の関数の前に **pcall** 構文を挿入するだけなので容易である．なお，式 (**par-and** $e_1 \cdots e_n$) は，引数 $e_1, \cdots, e_n$ を並列に評価し，その中の一つ e_kが偽となれば式の全体の値として偽を返し，他のプロセスを全て強制終了させる．式 (**plet** $((x_1\ e_{11}) \cdots (x_n\ e_{1n}))\ e_{21} \cdots e_{2m}$) は $e_{11}, \cdots, e_{1n}$ を並列に評価し，得られた値を変数 x_iに束縛した環境の下で，式 $e_{21}, \cdots, e_{2m}$ を並列に評価し，すべての式の評価を待って式 e_{2m} の値を返す．与えられた逐次Schemeプログラムのに **and** や **let** といった構文が現れれば，それらをそれぞれ **par-and**, **plet** 構文で置き換えることによって容易に並列化できる．従来 **pcall** などの並列構文の評価法として ETC (Eager Task Creation) が用いられてきた[1,7]．ETC は (**pcall** $f\ e_1 \cdots e_n$) を評価する際に，引数 $e_1, \cdots, e_n$ を評価するプロセスを必ず生成する方法で，プロセスの過剰生成によるオーバーヘッドが大きく実行効率が悪いという問題があった．ところが，文献[4]のスティール評価法を用いると，**pcall** 構文の実行が非常に効率良く行えることが分っている (スティール評価法については付録を参照)．逐次Schemeプログラムに関数適用や **and** や **let** などの構文が現れれば，それに **pcall** や **par-and** や **plet** を導入して並列化し，実行はスティール評価法で行えば，効率の良い自動並列化が実現される．この考え方に基づいて，逐次Schemeプログラムの自動並列化を行う簡便な方法を提案するのが本論文の目的である．

例えば，フィボナッチ数を求める逐次Schemeプログラムに無制約的に並列構文を導入したのが図1である．しかし，式 (**pcall** $< n\ 2$), (**pcall** $- n\ 1$), (**pcall** $- n\ 2$) は，変数 n や定数 $1, 2$ を評価するコストがプロセス生成のコストよりも小さいため，並列化を行わない ($< n\ 2$), $(- n\ 1)$, $(- n\ 2)$ より効率が悪い．人間が考えて並列化を行う場合は，このような過剰な並列化は行わず，+演算の引数の評価を並列化するだけである．そこで，式の計算コストが明らかに小さい場合には，並列構文を導入しないという制約の下で並列化を行う制約的並列化の考え方を導入する．

```
(define (fib n)
  (if (pcall < n 2) n
      (pcall + (fib (pcall - n 1))
               (fib (pcall - n 2)))))
```

図 1: 無制約に並列構文を導入した fib プログラム

3 制約的自動並列化変換

まず自動並列化の対象となるSchemeプログラムを定義し，それらに対する制約的自動並列化変換を与える．

3.1 自動並列化の対象とする逐次Schemeプログラム

自動並列化の対象とするSchemeプログラムを図2のような関数定義 d の集りであるとする．ただし，ラムダ束縛はデータとして扱われるSchemeのオブジェクトを表し，ラムダ式は関数適用に用いられる関数を表す．

3.2 Schemeプログラムのコストの分類

本研究の制約的自動並列化は，並列構文を挿入するとオーバーヘッドにより明らかに効率の悪い場合に並列構文の挿入を避け，また並列化が困難な場合にも並列化を行わないという簡便な自動並列化法である．並列構文の挿入で効率が悪くなるのは，並列に実行される式の評価コストがプロセス生成のコストよりも小さい場合である．例えば $(f\ e_1\ e_2)$ を並列化した (**pcall** $f\ e_1\ e_2$) の場合，e_1, e_2 が並列に実行されたとき，式 e_1, e_2 の評価は e_1, e_2を評価するプロセスの生成に置き換えられる．従って，プロセス生成のコストが e_1, e_2の評価コストより大きければ，式 (**pcall** $f\ e_1\ e_2$) は式 $(f\ e_1\ e_2)$ より評価コストが大きい．プロセス生成コストと式の

```
d   ::= (define f (lambda (x1 ··· xn) e))          (関数定義)
      | (define (f x1 ··· xn) e)
e   ::= x                                          (変数)
      | number | #t | #f                           (定数)
      | (quote datum) | 'datum                     (クオート式)
      | (lambda (x1 ··· xn) e)                     (ラムダ束縛)
      | (f e1 ··· en)                              (関数適用)
      | (begin e1 ··· en)                          (逐次実行)
      | (and e1 ··· en) | (or e1 ··· en)           (論理式)
      | (let ((x1 e11) ··· (xn e1n)) e21 ··· e2m)  (let 束縛)
      | (letrec ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
      | (if e1 e2 e3)                              (条件式)
      | (cond (e11 e12) ··· (en1 en2))
      | (case e (l1 e1) ··· (ln en))

f   ::= dによって定義された関数                    (定義関数)
      | (lambda (x1 ··· xn) e)                     (ラムダ式)
      | cons | car | cdr | list                     (基本関数)
      | + | - | * | = | > | < | >= | <=
      | eq? | eqv? | equal? | pair? | not
      | null? | list? | number? | symbol?
```

図 2: Scheme プログラムの構文

評価コストの関係は，マシンのアーキテクチャやシステムの構成法に依存する．演算レベルの細かい並列性を効率良くサポートするデータフローマシンのような場合，(pcall < n 2) のような式が効率的に実行されると考えられるが，本論文で対象としている共有メモリ型マルチプロセッサマシンでは，プロセス生成のコストが式 n や 2 の評価コストより大きいため，(pcall < n 2) のような並列化は行うべきではない．

図 2からわかるように，自動並列化の対象とする逐次 Scheme プログラムの式は，変数，定数，クオート式，ラムダ束縛からなる基本式と，ユーザ定義関数，ラムダ式，基本関数の各関数と，begin, and, let, if などの各構文の組み合わせによって合成される．また，ユーザ定義関数やラムダ式は基本式と基本関数と構文から合成される．最も評価コストが小さいのは基本式で，継いで基本式と 1 つの基本関数から合成される式のコストが小さく，式が複雑になるにつれてその評価コストは大きくなる．例えば，式 (cons 'a 'b) と式 (cons (cons 'a 'b) 'c) では，前者は基本式 2 つと基本関数 1 つ，後者は基本式 3 つと基本関数 2 つなので，後者のほうが評価コストは大きい．即ち，式の評価コストはその式に含まれる基本式や基本関数や構文の個数に依存するので，これらの値からその式の評価コストを推定し，プロセス生成のコストよりも大きいか小さいかを判別する．式 e における基本式，基本関数，構文の出現回数をそれぞれ n_e, n_f, n_s とすれば，式 e の評価コストは $C_e n_e + C_f n_f + C_s n_s$ と表される．従って，プロセス生成のコストよりも評価コストが小さな式を識別する述語 low-cost?を図 3のように定義する．関数 bexps は，式の中に含まれる基本式をすべて数え上げて n_eを求める．bfuns は式の中から関数適用を捜し，その関数部分が基本関数である場合の数を数える．また，式の中にユーザ定義関数があれば，ユーザ定義関数は再帰や繰り返しを含むため評価コストが大きいので，bexps の値を ∞ とすることによって，low-cost?[e] が常に偽を返すようにする．syntaxes は式の中に現れる，begin や let などの構文の数を数える．定数 C_e, C_f, C_s, C_pはそれぞれ基本式の評価コスト，基本関数の適用コスト，合成構文の実行コスト，プロセス生成コストを表す．

```
low-cost?[e] ::= (Ce bexps[e]+Cf bfuns[e]+Cs syntaxes[e] ≤ Cp)

bexps[e] ::=
   if basic-exp?[e] then 1
   else if e = (f e1 ··· en) then
     if f = (lambda (x1 ··· xn) e') then
        bexps[e'] + bexps[e1] +···+ bexps[en]
     else
        bexps[e1] +···+ bexps[en]
   else if e = (begin e1 ··· en) ∨ e = (and e1 ··· en)
         ∨ e = (or e1 ··· en)      then
      bexps[e1] +···+ bexps[en]
   else if e = (let ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
         ∨ e = (letrec ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
      bexps[e11]+···+bexps[e1n]+bexps[e21]+···+bexps[e2m]
   else if e = (if e1 e2 e3) then
      bexps[e1] + max{bexps[e2],bexps[e3]}
   else if e = (cond (e11 e12) ··· (en1 en2)) then
      bexps[e11] +···+ bexps[en1]
      + max{bexps[e12],···,bexps[en2]}
   else if e = (case e' (l1 e1) ··· (ln en))
      bexps[e'] + max{bexps[e1],···,bexps[en]}

bfuns[e] ::=
   if basic-exp?[e] then 0
   else if e = (f e1 ··· en) then
     if basic-fun?[f] then
         1 + bfuns[e1] + ··· + bfuns[en]
     else if f = (lambda (x1 ··· xn) e') then
         bfuns[e'] + bfuns[e1] + ··· + bfuns[en]
     else ∞
   else if e = (begin e1 ··· en) ∨ e = (and e1 ··· en)
         ∨ e = (or e1 ··· en)  then
      bfuns[e1] +···+ bfuns[en]
   else if e = (let ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
        ∨ e = (letrec ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
      bfuns[e11]+···+bfuns[e1n]+bfuns[e21]+···+bfuns[e2m]
   else if e = (if e1 e2 e3) then
      bfuns[e1] + max{bfuns[e2], bfuns[e3]}
   else if e = (cond (e11 e12) ··· (en1 en2)) then
      bfuns[e11] +···+ bfuns[en1]
      + max{bfuns[e12],···,bfuns[en2]}
   else if e = (case e' (l1 e1) ··· (ln en)) then
      bfuns[e'] + max{bfuns[e1],···,bfuns[en]}

syntaxes[e] ::=
   if basic-exp?[e] then 0
   else if e = (f e1 ··· en) then
      syntaxes[e1] +···+ syntaxes[en]
   else if e = (begin e1 ··· en) ∨ e = (and e1 ··· en)
         ∨ e = (or e1 ··· en) then
      1 + syntaxes[e1] +···+ syntaxes[en]
   else if e = (let ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
         ∨ e = (letrec ((x1 e11) ··· (xn e1n)) e21 ··· e2m)
      1 + syntaxes[e11] +···+ syntaxes[e1n]
      + syntaxes[e21] +···+ syntaxes[e2m]
   else if e = (if e1 e2 e3) then
      1 + syntaxes[e1] + max{syntaxes[e2], syntaxes[e3]}
   else if e = (cond (e11 e12) ··· (en1 en2)) then
      1 + syntaxes[e11] +···+ syntaxes[en1]
      + max{syntaxes[e12],···,syntaxes[en2]}
   else if e = (case e' (l1 e1) ··· (ln en))
      1 + syntaxes[e'] + max{syntaxes[e1],···,syntaxes[en]}

basic-exp?[e] ::= var?[e]∨const?[e]∨quote?[e]∨lambda-bind?[e]
```

図 3: 式の評価コストの判定アルゴリズム

3.3 逐次 Scheme プログラムの並列化とその規則

制約的自動並列化変換 $\mathcal{P}$ は，関数適用や構文の引数の評価コストがプロセス生成のコストより小さな式ばかりである場合には並列化せず，そうでない場合には pcall などの構造化並列構文によって並列化する．図 2に示した Scheme プログラムの各要素に対して，low-cost?により明らかに計算コストが小さい式の判定が行えるとの仮定[1]の下で，並列化の考え方を示し，制約的自動並列化変換 $\mathcal{P}$ を与える．

関数定義の並列化

関数定義の並列化はそのボディ部を並列化する．関数定義に対する制約的自動並列化変換 $\mathcal{P}$は次のように与

[1] PaiLisp/MT におけるこの仮定に関しては 3.4 を参照

えられる．

$\mathcal{P}$[(define f (lambda ($x_1 \cdots x_n$) e))] ::=
  (define f (lambda ($x_1 \cdots x_n$) $\mathcal{P}$[e]))
$\mathcal{P}$[(define (f $x_1 \cdots x_n$) e)] ::=
  (define (f $x_1 \cdots x_n$) $\mathcal{P}$[e])

基本式 (変数，定数，クオート式，ラムダ束縛)

基本式は並列化できないと仮定すると，基本式に対する制約的自動並列化変換 $\mathcal{P}$ は次のように与えられる．

$\mathcal{P}$[e] ::= e (basic-exp?[e] = true)

関数適用 (f $e_1 \cdots e_n$) の並列化

1) 引数が 0 又は 1 つの場合，式は (f) や (f e) の形をしているため，引数の並列評価は行わない．f や e の評価コストが大きければそれぞれの評価の並列化は行う．
2) すべての引数の評価コストが小さいか，1 つだけ計算コストが大きい場合，評価コストの小さな式を並列評価すると，並列化に伴なうオーバヘッドにより効率が悪くなるため，引数の並列評価は行わない．評価コストの大きな引数自体の評価は並列化する．
3) 2 つ以上の引数の評価コストが大きな場合，pcall 構文によって引数を並列に評価する．

以上の仮定より関数適用に対する制約的自動並列化変換 $\mathcal{P}$ は次のようになる．

$\mathcal{P}$[(f)] ::= ($\mathcal{P}$[f])
$\mathcal{P}$[(f e)] ::= ($\mathcal{P}$[f] $\mathcal{P}$[e_1])
$\mathcal{P}$[(f $e_1 \cdots e_n$)] ::=
  if (low-cost?[e_1] $\wedge \cdots \wedge$ low-cost?[e_n])
    $\vee$ heavy-only-one-arg?[($e_1 \cdots e_n$)] then
   ($\mathcal{P}$[f] $\mathcal{P}$[e_1] $\cdots$ $\mathcal{P}$[e_n])
  else
   (pcall $\mathcal{P}$[f] $\mathcal{P}$[e_1] $\cdots$ $\mathcal{P}$[e_n])

heavy-only-one-arg?[($e_1 \cdots e_n$)] ::=
 ($\neg$ low-cost?[e_1] $\wedge$ low-cost?[e_2] $\wedge \cdots \wedge$ low-cost?[e_n])
$\vee$($\neg$low-cost?[e_2] $\wedge$ low-cost?[e_1] $\wedge$ low-cost?[e_3] $\wedge \cdots \wedge$ low-cost?[e_n])
 $\cdots$
$\vee$($\neg$low-cost?[e_n] $\wedge$ low-cost?[e_1] $\wedge \cdots \wedge$ low-cost?[e_{n-1}])

low-cost?[e] は実験に用いた PaiLisp/MT では
3 bexps[e]+ 10 bfuns[e]+ 15 syntaxes[e] $\leq$ 60 となる．

[注] 式 (f $e_1 \cdots e_n$) において，e_1 の評価コストが大きく，他の評価コストが小さいとき，式 $e_2, \cdots, e_n$ の評価をひとまとめにし，e_1 の評価と並列に行えば，効率的な実行が達成できる可能性がある．同様な問題は以下の構文でも発生する．

逐次実行 (begin $e_1 \cdots e_n$) の並列化

関数適用と同様に，式 $e_1, \cdots, e_n$ のうちの 2 つ以上が評価コストの大きな式であれば，引数の並列評価を行う．ただし，(begin $e_1 \cdots e_n$) は e_n の値を式の値として返さなければならないので，pcall 構文を用いて次のように変換する．

$\mathcal{P}$[(begin e)] ::= (begin $\mathcal{P}$[e])
$\mathcal{P}$[(begin $e_1 \cdots e_n$)] ::=
  if (low-cost?[e_1] $\wedge \cdots \wedge$ low-cost?[e_n])
    $\vee$ heavy-only-one-arg?[($e_1 \cdots e_n$)] then
   (begin $\mathcal{P}$[e_1] $\cdots$ $\mathcal{P}$[e_n])
  else
   (pcall (lambda ($x_1 \cdots x_n$) x_n) $\mathcal{P}$[e_1] $\cdots$ $\mathcal{P}$[e_n])

論理式 (and $e_1 \cdots e_n$), (or $e_1 \cdots e_n$) の並列化

関数適用と同様に，引数 $e_1, \cdots, e_n$ の 2 つ以上が評価コストの大きな式であるとき，and や or をそれぞれ par-and，par-or で置き換えられる可能性がある．しかし，例えば次の式の評価を考えると，

(and (not (null? e)) (f (car e)) (g (cdr e)))

式 e が空リストであれば，式 (not (null? e)) の値が #f となり式全体の値も偽になるが，and を par-and で単純に置き換えた次の式

(par-and (not (null? e)) (f (car e)) (g (cdr e)))

の場合には，e が空リストのとき (car e)，(cdr e) でエラーを起すので，正しい変換とは言えない．null?の引数である e は，(car e)，(cdr e) とデータ依存関係があるために，実行の順序が問題となる．一般に，データ依存関係があるときの並列性の抽出には，データ依存関係の解析を行う必要があるが，ここでは詳細なデータ依存解析をしない簡便法を考える．式 (and$e_1 \cdots e_n$) において，$e_1, \cdots, e_n$ の 2 つ以上が評価コストの大きな式であり，かつ $e_1, \cdots, e_n$ が以下の論理式条件を満足するとき，par-and 構文によって並列化を行う．and に対する制約的自動並列化変換 $\mathcal{P}$ は次のようになる．or 構文の場合は，以下の規則の and を or で，par-and を par-or でそれぞれ置き換えたものとなる．

$\mathcal{P}$[(and e)] ::= (and $\mathcal{P}$[e_1])
$\mathcal{P}$[(and $e_1 \cdots e_n$)] ::=
  if (low-cost?[e_1] $\wedge \cdots \wedge$ low-cost?[e_n])
    $\vee$ heavy-only-one-arg?[($e_1 \cdots e_n$)] then
   (and $\mathcal{P}$[e_1] $\cdots$ $\mathcal{P}$[e_n])
  else if logical-cond?[($e_1 \cdots e_n$)] then
   (par-and $\mathcal{P}$[e_1] $\cdots$ $\mathcal{P}$[e_n])
  else
   (and $\mathcal{P}$[e_1] $\mathcal{P}$[(and $e_2 \cdots e_n$)])

なお，logical-cond?[e] は次の論理式条件が成立するときに真となる述語であるとする．

[論理式条件] 論理式 e としては，(and (f_1 e_1)$\cdots$(f_n e_n))，(or (f_1 e_1)$\cdots$(f_n e_n)) の場合だけを対象とし，次の (L1)，(L2)，(L3) のいずれかの条件を満足する場合に logical-cond?[e] が真となる．

(L1) $f_1, \cdots, f_n$ が引数 $e_1, \cdots, e_n$ に対する car，cdr による分解操作を含まない場合，および，全ての関数 $f_1, \cdots, f_n$ において f_i の引数がそれぞれ空，(car e_i)，(cdr e_i) のうちのどれかであり，$f_1, \cdots, f_n$が car，cdr による分解操作を含まない場合．

(L2) (f_1 e_1),$\cdots$,(f_n e_n) において，各 f_j の引数データの独立性が，car と cdr の多重適用によっても継承される場合．独立性の検査は，左から右に式を走査していき，独立性の継承を調べる．例えば，(f (car e)) と (g (cddr e)) の場合，(g (cddr e)) は (cdr (cdr e)) の略記であるからこれらの独立性は，(car e) と (cdr e) の独立性から言えることになる．即ち二進木を辿って，従属関係がなければ独立であると判断する．

(L3) (f_1 e_1)，$\cdots$，(f_n e_n) において，$f_1, \cdots, f_n$ が +，−，*，= などの数値演算である場合．

let 束縛 (let ((x_1 e_{11}) $\cdots$ (x_n e_{1n})) $e_{21} \cdots e_{2m}$)，(letrec ((x_1 e_{11}) $\cdots$ (x_n e_{1n})) $e_{21} \cdots e_{2m}$) の並列化

let においては，式 e_{1i} は他の式と独立なので，それらの式の評価を並列に行うことができる．関数適用の場合と同様に，式 $e_{11}, \cdots, e_{1n}$ の 2 つ以上の評価コストが大きいとき，それらを並列に評価し，式 $e_{21}, \cdots, e_{2m}$ の 2 つ以上の評価コストが大きいとき，それらを並列に評価する．$e_{21}, \cdots, e_{2m}$のみを並列化する場合は，ボディ部を (begin $e_{21} \cdots e_{2m}$) に置き換えてそれを並列化する．$e_{11}, \cdots, e_{1n}$のみを並列化する場合は，let 構文を plet 構文に置き換え，ボディ部を begin 構文によって逐次化する．$e_{11}, \cdots, e_{1n}$ と $e_{21}, \cdots, e_{2m}$ を共に並列

化する場合は，letをpletで置き換える．let構文に対する制約的自動並列化変換 $\mathcal{P}$ は次のようになる．

$$
\begin{array}{l}
\mathcal{P}[(\text{let } ((x_{11}\ e_{11})\cdots(x_{1_n}\ e_{1_n}))\ e_{21}\cdots e_{2_m})] ::= \\
\quad \text{if } n=1 \vee (\text{low-cost?}[e_{11}] \wedge\cdots\wedge \text{low-cost?}[e_{1_n}]) \\
\qquad \vee \text{heavy-only-one-arg?}[(e_{11}\cdots e_{1_n})] \text{ then} \\
\quad\ \text{if } m=1 \vee (\text{low-cost?}[e_{21}]\wedge\cdots\wedge \text{low-cost?}[e_{2_m}]) \\
\qquad \vee \text{heavy-only-one-arg?}[(e_{21}\cdots e_{2_m})] \text{ then} \\
\qquad (\text{let } ((x_1\ \mathcal{P}[e_{11}])\cdots(x_n\ \mathcal{P}[e_{1_n}]))\ \mathcal{P}[e_{21}]\cdots\mathcal{P}[e_{2_m}]) \\
\quad \text{else} \\
\qquad (\text{let } ((x_1\ \mathcal{P}[e_{11}])\cdots(x_n\ \mathcal{P}[e_{1_n}]))\ \mathcal{P}[(\text{begin } e_{21}\cdots e_{2_m})]) \\
\text{else} \\
\quad \text{if } m=1 \vee (\text{low-cost?}[e_{21}]\wedge\cdots\wedge\text{low-cost?}[e_{2_m}]) \\
\qquad \vee \text{heavy-only-one-arg?}[(e_{21}\cdots e_{2_m})] \text{ then} \\
\qquad (\text{plet } ((x_1\ \mathcal{P}[e_{11}])\cdots(x_n\ \mathcal{P}[e_{1_n}]))\ (\text{begin } \mathcal{P}[e_{21}]\cdots\mathcal{P}[e_{2_m}])) \\
\quad \text{else} \\
\qquad (\text{plet } ((x_1\ \mathcal{P}[e_{11}])\cdots(x_n\ \mathcal{P}[e_{1_n}]))\ \mathcal{P}[e_{21}]\cdots\mathcal{P}[e_{2_m}])
\end{array}
$$

letrecに対する変換は，以下の規則において letを letrecに pletを pletrecに置き換えたものである．

条件式の並列化

if, cond, caseについては，構文としての並列化は行わず引数式の並列化のみを行う．

if構文 (if e_1 e_2 e_3) の場合，式 e_1, e_2, e_3の評価コストが大きいとき，e_1, e_2, e_3 を並列に実行することが考えられるが、if文の逐次的意味の保存を考えると andのようなデータ依存関係を考慮する必要も生じるので if構文そのものの並列化は行わないものとする．

cond構文 (cond $(e_{11}\ e_{12})\cdots(e_{n1}\ e_{n2})$) の場合，式 $e_{11},\cdots,e_{n1}$ の評価コストが大きいとき、式 $e_{11},\cdots,e_{n1}$ を並列実行するか，式 $e_{11},\cdots,e_{n1}$, $e_{12},\cdots,e_{n2}$ を並列に実行することが考えられるが cond 文の逐次的意味の保存を考えると,and, orにおけるようなデータ依存関係を考慮する必要も生じるので cond 構文そのものの並列化は行わないものとする．

case構文 (case e $(l_1\ e_1)\cdots(l_n\ e_n)$) の場合，$l_i$の各要素と式 e の値との比較のコストは非常に小さいので，並列化は行わないものとする．

以上より，if, cond, caseに対する制約的自動並列化変換は次のようになる．

$$
\begin{array}{l}
\mathcal{P}[(\text{if } e_1\ e_2\ e_3)] ::= (\text{if } \mathcal{P}[e_1]\ \mathcal{P}[e_2]\ \mathcal{P}[e_3]) \\
\mathcal{P}[(\text{cond } (e_{11}\ e_{12})\cdots(e_{n1}\ e_{n2}))] ::= \\
\qquad (\text{cond } (\mathcal{P}[e_{11}]\ \mathcal{P}[e_{12}])\cdots(\mathcal{P}[e_{n1}]\ \mathcal{P}[e_{n2}])) \\
\mathcal{P}[(\text{case } e\ (l_1\ e_1)\cdots(l_n\ e_n))] ::= \\
\qquad (\text{case } \mathcal{P}[e]\ (l_1\ \mathcal{P}[e_1])\cdots(l_n\ \mathcal{P}[e_n]))
\end{array}
$$

3.4 PaiLisp/MTにおける制約的自動並列化変換

上記の制約的自動並列化変換におけるコスト判定述語 low-cost? は DEC7000 上の PaiLisp/MT では次のようになる．

$$\text{low-cost?}[e] ::= 3\ \text{bexps}[e] + 10\ \text{bfuns}[e] + 15\ \text{syntaxes}[e] \le 60$$

PaiLisp/MT は，共有メモリアーキテクチャを仮定した PaiLisp のインタプリタで，fを基本関数とする関数適用 (f e_1 e_2) の PaiLisp/MT による評価は，まず e_1,e_2 の評価が行われ，e_1 の値と e_2 の値に対して，基本関数 f が適用される．PaiLisp/MT においては基本関数の適用コストはほぼ一定である．従って，その評価コストは，“C_f+e_1,e_2の評価コスト” となる．構文の場合も同様で，評価コストは “C_s + 引数の評価コスト” となる．基本式の場合の評価コストは基本式の種類によらずほぼ一定で C_{exp} となる．DEC7000 の PaiLisp/MT による実験から，C_e=3[μsec], C_f=10[μsec], C_s=15[μsec], スティール評価法におけるプロセス生成のコスト C_p は約 60[μsec] と求まっている．例えば，式 (cons 'a (cons 'c 'd)) は，基本式 3 つと基本関数 2 つから構成されるため，その評価コストは 29[μsec] と見積られ，low-cost? は真を返す．基本関数と基本式だけから構成される式を考えれば，基本関数 4 つと基本式 6 つから構成される式まではコストが小さいと判定される．

4 自動並列化の例と評価実験

4.1 ベンチマークプログラムによる評価

フィボナッチ数を計算する fib，引数をたらい回しする tarai，Nクイーン問題を解く queen，クイックソート qsort，論理式の真理値を求める truth，fib を繰り返す fatwalk に対し，1) 制約的自動並列化変換，2) 人手で並列化，の 2 つの方法を適用して並列化し，実験を行った．6 つのすべてのプログラムにおいて制約的自動並列化変換と人手で並列化した場合とで得られる並列プログラムが一致した．例えば，queen の場合得られたプログラムは図 4 となった．並列化は，8 行目に pcall 構文が挿入されているが，この理由は関数 + の引数にそれぞれ ok?, try というユーザ定義関数が含まれており，low-cost? が偽となるためである．17 行目の and 構文は，3 つのうち 2 つの引数の評価コストが 60[μsec] より多少小さいため，low-cost? の値が真となって並列化されない．

```
 1: (define (queen n) (try (1-to n 1) '() '()))

 2: (define (1-to n i)
 3:   (if (= n 0) '()
 4:       (cons i (1-to (- n 1) (+ i 1)))))

 5: (define (try x y z)
 6:   (if (null? x)
 7:       (if (null? y) 1 0)
 8:       (pcall +
 9:        (if (ok? (car x) 1 z)
10:            (try (append (cdr x) y)
11:                 '()
12:                 (cons (car x) z))
13:            0)
14:        (try (cdr x) (cons (car x) y) z))))

15: (define (ok? r d p)
16:   (if (null? p) #t
17:       (and (not (= (car p) (+ r d)))
18:            (not (= (car p) (- r d)))
19:            (ok? row (+ d 1) (cdr p)))))
```

図 4: queen の制約的自動並列化の結果

得られた並列プログラムをスティール評価法を導入して拡張された PaiLisp/MT[5] システムの下で実行した．実験は，DEC 7000 (Alpha 20164 × 6) の下で行った．実験結果を表 1に示す．制約的自動は制約的自動並列化変換を適用，無制約自動は無制約に構造化並列構文を挿入，人手は人間が効率が良いと考えて並列構文を導入したプログラムであり，逐次はベンチマークプログラムそのものである．プロセッサは 6 台使用した．

この結果から，制約的自動は逐次に比べプロセッサ台数 6 台のもとでどれも 3 倍~4 倍効率良く実行されることが分る．制約的自動並列化と人手による並列化で同一のプログラムが得られたため，両者の実行時間はどれも同じとなっている．参考までに行った無制約自動の場合，制約自動や人手の場合の約 2 倍の時間がかかるが，並列構文の入れ過ぎによるプロセス生成のオーバーヘッドが原因であると考えられる．

表 1: ベンチマークプログラムの実験結果 [sec]

プログラム	制約的自動	無制約自動	人手	逐次
(fib 20)	0.20	0.36	0.20	0.66
(tarai 8 4 0)	0.11	0.25	0.11	0.40
(queen 8)	0.48	3.78	0.48	2.17
(qsort ls1000)	0.60	1.45	0.60	2.45
(truth E)	0.005	0.005	0.005	0.016
(fatwalk 20 3)	0.56	1.06	0.56	1.92

4.2 Scheme-in-Scheme による評価

逐次 Scheme のインタプリタを Scheme によって記述した Scheme-in-Scheme の並列化実験を行った．Scheme-in-Scheme は，eval と apply を呼び出しながら入力として与えられた逐次 Scheme プログラムを評価する．実験は，Scheme-in-Scheme に対して，1) 制約的自動並列化，2) 手作業で並列化，の 2 つの方法で並列化し，4.1 で用いた逐次ベンチマークプログラムを入力に与え実行した．人手で並列構文を導入した場合は，逐次 Scheme プログラムの関数適用式の引数を並列に評価するように，pcall 構文を 1 つ導入した．制約的自動並列化は，この部分の並列化に加えて，apply 関数と eval 関数の適用の際に，その引数を並列に評価する．実験結果を表 2に示す．プロセッサは 6 台使用した．

表 2: Scheme-in-Scheme の実験結果 [sec]

プログラム	制約的自動	人手	逐次
(eval (fib 20) e)	11.83	11.22	51.19
(eval (tarai 8 4 0) e)	6.86	6.61	29.19
(eval (queen 8) e)	61.69	58.97	268.93
(eval (qsort ls300) e)	6.27	6.64	23.45
(eval (truth E) e)	0.40	0.38	1.22
(eval (fatwalk 20 3) e)	35.36	33.60	153.85

この結果から，制約的自動は逐次の約 4 倍速く，効率的に実行されることが分る．制約的自動と人手を比較すると，制約的自動が人手より若干遅いが，これは余分に挿入された pcall のオーバーヘッドによるものと考えられる．以上の結果から，簡単なベンチマークプログラムや，やや規模の大きい Scheme-in-Scheme のようなプログラムにおいては，制約的自動並列化変換とスティール評価を用いる本論文の方法は，人手によって効率良く並列化された場合とほぼ同程度の効率が得られる有効な方法であるといえる．

5 おわりに

Scheme プログラムの評価コストの情報を利用した制約条件を用いて構造化並列構文を自動的に挿入し，その結果得られた並列プログラムをスティール評価法によって実行する制約的自動並列化法という簡便な自動並列化の方法を与え，その有効性を実験によって示した．

今後の課題として，if や cond などの構文を効率良く並列化するための規則や論理式におけるデータ依存関係解析の詳細化，副作用がある場合への拡張は今後の課題である．また関数適用の所の [注] に述べたような事柄についても今後の課題である．

参考文献

1) R.Halstead,Jr.: Multilisp: A language for concurrent symbolic computation, ACM Trans. on Programming Languages and Systems, Vol.4, No.7, pp.501-538 (1985).

2) W.L.Harrison III, Z.Ammarguellat.: The design of automatic parallelizers for symbolic and numeric programs, LNCS 441, pp.58-100, Springer (1990).

3) T.Ito, M.Matsui.: A parallel Lisp language PaiLisp and its kernel specification, LNCS 441, pp.58-100, Springer (1990).

4) T.Ito, S.Kawamoto.: Efficient Evaluation Strategies for Structured Concurrency Constructs in Parallel Scheme Systems, PSLS'95 October, Beaune, France, (1995) [to be published in a volume of Springer LNCS]

5) S.Kawamoto, T.Ito.: Multi-threaded PaiLisp with granularity adaptive parallel execution, LNCS 907, pp.94-120, Springer (1995).

6) E.Mohr, D.A.Kranz, R.Halstead,Jr.: Lazy task creation: A technique for increasing the granularity of parallel programs, IEEE Trans. Parallel and Distributed systems, Vol.2, No.3, pp.264-280 (1991).

7) 清野智弘, 伊藤貴康: PaiLisp の並列構文の実現法と評価, 情報処理学会論文誌, Vol.34, No.12, pp.2578-2591 (1993).

付録: スティール評価法

スティール評価法 (SBE: Steal-based Evaluation) は，並列 Lisp 言語 PaiLisp の構造化並列構文 pcall, par, par-and, par-or などを効率良く実行できる評価方式で，LTC (Lazy Task Creation)[6] の基本機能であるスティール (steal) とインライニング (inlining) の適用を拡張したものである．ホームプロセッサと呼ばれる 1 つのプロセッサ上で，式 (pcall f $e_1 \cdots e_n$) を実行したとする．ホームプロセッサは pcall 構文があっても，$e_1, \cdots, e_n$ を実行するプロセスを生成せず，逐次的に実行を進めていく．もし，他のプロセッサが空き状態であれば，ホームプロセッサから引数 $e_1, \cdots, e_n$ のうち未評価の式を取って (この動作をスティールと呼ぶ) 実行する．空状態のプロセッサがなければ，式 (pcall f $e_1 \cdots e_n$) は式 (f $e_1 \cdots e_n$) のように逐次実行されるが，これはインライン化と呼ばれる．スティール評価法は，ETC 法に比べてプロセスの生成を抑制できるので，効率の良い並列実行が可能となる．なお，future 構文のスティール評価法は LTC と同じように実行される．スティール評価法の実験結果を表 3に示す．頭に p の付いたものは pcall によって，頭に f の付いたものは future 構文によって並列化されたプログラムを示す．この結果から，スティール評価法は ETC より効率が良く，また，スティール評価法の下で pcall を用いた並列プログラムが future を用いた場合とほぼ同等の効率を有することが分る．

表 3: スティール評価法の評価結果 [sec]

プログラム	SBE	ETC
(pfib 20)	0.20(50)	0.82(21890)
(ffib 20)	0.19(70)	0.58(10945)
(pqueen 8)	0.49(132)	0.73(11016)
(fqueen 8)	0.48(80)	0.61(5508)
(ptarai 8 4 0)	0.11(200)	0.33(9454)
(ftarai 8 4 0)	0.14(85)	0.20(5059)