Committed-Choice 型言語 Fleng における配列処理の最適化

荒木拓也[†,☆] 坂井修一[†] 田中英彦[†]

Committed-Choice 型言語 Fleng は，データフロー同期の機構を用いてすべてのゴール（手続きに相当する）を並列に実行することで，容易に大量の並列性を抽出することが可能な言語である．しかし，データフロー同期を実現するために，変数は単一代入であり書き換えることができない．このため，元の配列の一部を更新した配列を得るためには，新しい配列を確保して，変更部分以外を元の配列からコピーする必要がある．また，Fleng は配列を扱う際にデータ並列性を直接抽出する機構を持たない．このため，本来データ並列的に実行できるプログラムでも並列性が抽出できない場合がある．本研究では，配列の更新におけるコピー操作を除去し，その後データ並列性を抽出することで，配列処理の最適化を行う手法を提案，実装，評価した．配列のコピー除去は，プログラムのデータフロー，配列の利用を大域的に解析し，それに基づきプログラムの部分的な逐次化を行うことで実現する．また，データ並列性の抽出は，コピー除去を行ったプログラムに対して，並列化コンパイラで行われているような，ループ運搬依存の解析を行うことで実現する．我々は本手法を Fleng コンパイラに実装し，並列計算機 PIE64 上で評価を行った．これにより，コピー除去により数倍〜数百倍，データ並列性の抽出により数倍程度のきわめて高い速度向上が達成できることが示された．

Optimization of Vector Processing of a Committed-Choice Language Fleng

Takuya Araki,[†,☆] Shuichi Sakai[†] and Hidehiko Tanaka[†]

A committed-choice language Fleng can extract much parallelism easily by executing all goals in parallel using dataflow synchronization. However, in order to realize dataflow synchronization, Fleng utilizes single assignment variables whose values cannot be changed. Thus, to get a new array whose one element is changed from the original array, it is necessary to allocate a new array and copy all elements except the changed one from the original array. In addition, Fleng does not exploit data-parallelism explicitly while processing an array. This usually makes it impossible to extract certain amount of parallelism which can be extracted by simple data-parallel languages. In this study, we proposed, implemented, and evaluated a vector processing optimization method with which vector copy can be eliminated and data-parallelism can be extracted. Copy elimination is realized by partial sequentialization of a program with global dataflow and array utilization analysis. Data-parallelism extraction is applied to a program after copy elimination, and it utilizes loop-carried dependency analysis like parallelizing compilers. We implemented this method on a Fleng compiler and evaluated on a parallel computer PIE64. The evaluation shows that quite high speedup (up to a few hundred times speedup with copy elimination and several times speedup with data-parallelism extraction) can be attained with this method.

1. はじめに

Committed-Choice 型言語 Fleng[11]は，データフロー同期の機構を用いてすべてのゴール（手続きに相当する）を並列に実行することで，容易に大量の並列性を抽出することが可能な言語である．

しかし，データフロー同期を実現するために，変数は単一代入であり書き換えることができない．このため，元の配列の一部を更新した配列を得るためには，新しい配列を確保して，変更部分以外を元の配列からコピーする必要がある．

また，Fleng はデータ並列性を直接抽出する機構を持たない．このため，本来データ並列的に実行できるプログラムでも並列性が抽出できない場合がある．

配列更新時のコピーの問題は，関数型言語などの他の単一代入変数を用いる言語でも存在し，研究が行われている．しかし，逐次型言語のみを対象とした手法であったり，実行時にコストがかかったりするなどの

† 東京大学工学系研究科
School of Engineering, The University of Tokyo
☆ 現在，NEC C&C メディア研究所
Presently with NEC C&C Media Research Laboratories

問題があった．また，単一代入変数を用いる言語で，プログラマが明示しないデータ並列性を抽出するものはほとんど知られていない．

本研究では，コンパイル時に配列の更新におけるコピー操作を除去し，その後データ並列性を抽出することで，配列処理の最適化を行う手法を提案，実装，評価した．

配列のコピー除去は，プログラムのデータフロー，配列の利用を大域的に解析し，それに基づきプログラムの部分的な逐次化を行うことで実現する．また，データ並列性の抽出は，コピー除去を行ったプログラムに対して，並列化コンパイラで行われているような，ループ運搬依存の解析を行うことで実現する．

後者の手法の重要な点は，単一代入変数を扱う言語において，並列化コンパイラで行われている技術を適用可能にする手法を提示した点にある．これは配列のコピー除去をどのように行うかに依存しており，本研究のコピー除去法はこれを念頭において設計されている．

本研究で提案する手法は他の単一代入変数を用いる言語においても効率的な適用が可能であると考えられる．

本論文の構成は以下のとおりである．まず，2 章で対象言語である Fleng について説明する．次に 3 章で配列のコピー除去法について述べ，4 章でデータ並列性の抽出法について述べる．5 章で実装と評価について述べ，6 章で関連研究について述べた後，最後に 7 章で結論を述べる．

本論文で述べる手法は，すべて Fleng コンパイラの一部として実現済みである．本論文では基本的に処理の手順を示すことで手法を説明するが，細かすぎる点は例を用いることで説明する．

2. Committed-Choice 型言語 Fleng

Fleng は並列論理型言語のうちの，Committed-Choice 型言語と呼ばれる種類の言語である．Committed-Choice 型言語には，ほかに GHC[16]，KL1[7]，Concurrent Prolog[15]などがある．Fleng は，GHC からガードゴールを取り除いたものに，ほぼ相当する．Fleng のガードの機構はヘッドユニフィケーションによってのみ実現されている．これにより，Fleng はより単純な機構によって同期を実現している．

Fleng のシンタックスは Prolog のものとよく似ているが，バックトラックを行わないという点で，セマンティクスは大きく異なる．

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，高並列なプログラムの実行が可能である．この点が Fleng の大きな特徴になっている．

以下に例をあげながら，Fleng の操作的な意味について説明する．

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

これは，`R = (A + 1) * 2` を実行する述語 foo の定義である．述語とは通常のプログラミング言語の手続きに相当し，1 つまたは複数の定義節で定義される．この場合は 1 つの定義節で定義されている．定義節は “`:-`” で区切られており，“`:-`” の左側をヘッド，右側をボディという．

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．この書き換えの操作をリダクションという．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（サスペンド）し，`add` の実行が終了し B の値が決定すると，実行を再開（アクティベイト）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．変数は値が決まっていない（未定義）か決まっているかの 2 つの状態だけを持つ．未定義の変数の値を決めることを束縛するなどという．

分岐は次のように表す．

```
foo(true,R):- R = 1.
foo(false,R):- R = 0.
```

この述語は 2 つの定義節から構成されており，第 1 引数が `true` ならば `R = 1`，`false` ならば `R = 0` を実行する．ここで，`true` や `false` のように小文字で始まるものはシンボルを表す．変数は大文字で始まる（内容に興味がない変数は，無名の変数として “`_`” で表すことができる）．この例の場合，さきほどと同様に，第 1 引数が決まるまでサスペンドし，値が決まったところでアクティベイトされ，その値によって分岐を行う．

また，Fleng には，記述を容易にするためのマクロとして，If-Then-Else マクロと is マクロが用意されている．If-Then-Else マクロは，`(Cond -> Then; Else)` あるいは `(Cond -> Then)` の形で記述する．たとえば，先ほどの分岐の例は次のようになる．

```
foo(A,R):- (A == true -> R = 1; R = 0).
```

is マクロは，算術演算の記述を容易にするもので，た

とえば，add(A,B,C) を C is A + B のような形で記述することができる．これらのマクロはコンパイル時に展開される．

配列は即値としては {1,2,3} のような形で表現する．配列の操作には以下の組み込み述語を用いる．

- vector(Size,V)
  大きさが Size の配列を生成し，V に返す．生成時の配列の要素は未定義変数である．
- element(P,V,E)
  配列 V の P 番目の要素を E に返す．
- set_element(P,V,NewE,NewV)
  配列 V の P 番目の要素を NewE に更新した，新しい配列 NewV を返す．V と NewV には別の配列であるため，単純な実装では，新たに配列を確保し，更新する要素以外の部分をコピーする必要がある．

3. 配列のコピー除去

3.1 基本的な考え方

set_element を用いて配列の要素を更新する際には，新しい配列を確保して，更新する要素以外の部分をコピーする必要がある．これは，更新しようとしている配列を参照しているゴールが他に存在するかもしれないからである．もし，ほかに参照しているゴールが存在しているときに，配列の要素を破壊代入すると，そのゴールが配列を参照したときに変更された値を参照してしまい，プログラムの意味を変えてしまう．

しかし，もしその配列を参照しているゴールが，更新しようとしているゴールだけならば，安全に破壊代入による更新ができる．これがコピー除去の基本的な考え方である．

配列を参照しているゴールが更新時に他にないことを保証するには，配列の参照，更新をコンパイル時に逐次化してしまい，更新時には参照が終わっていることを保証すればよい．ただし，元のプログラム上では並行に動作するものとして記述されていたものを逐次化することになるため，デッドロックを引き起こす危険性がある．

また，逐次化を行うことにより，並列度が低下する危険性もある．しかし，配列のコピーによるオーバヘッドはきわめて大きく，並列度の低下による速度低下よりも性能に大きく影響する．このため，本論文では並列度の低下については考慮せず，可能な限り破壊代入による更新を行う．

3.2 逐次化の方法

ここで，配列参照の逐次化と破壊代入を表現するため，以下の述語を追加する．

- get_element(P,V,E,NewV)
  配列 V の P 番目の要素を E に返す．参照が終了した時点で，NewV を V に束縛する☆．
- assign_element(P,V,NewE,NewV)
  配列 V の P 番目の要素を NewE に破壊代入する．破壊代入が終了した時点で，NewV を V に束縛する．

これらの述語はコンパイラ内の中間表現として用いられる．つまり，ユーザは直接これらの述語を用いることはない（ユーザがこれらの述語を使う可能性については，後で議論する）．

これらを用いると，たとえば

```
V = {1,2,3,4,5},
element(3,V,E),
set_element(3,V,10,NV)
```

というプログラムがあった場合，element による参照と set_element による更新を逐次化してやることで，安全に

```
V = {1,2,3,4,5},
get_element(3,V,E,V1),
assign_element(3,V1,10,NV)
```

のように，破壊代入操作に変換することができる．同じ配列に対する element 操作が複数ある場合は，同様に，それぞれの element を get_element に変更し，逐次化を行えばよい．

しかし，元のプログラムの意味上，並行に動作するものを逐次化するということは，デッドロックの危険性を持つ．これを，**図 1** のプログラムを例として説明する．

この例で，上の段のプログラムを下の段のプログラムのように逐次化することはできない．これは，変数 NV によってゴール (3) がゴール (2) に依存しており，かつ変数 P1 によってゴール (1) がゴール (3) に依存

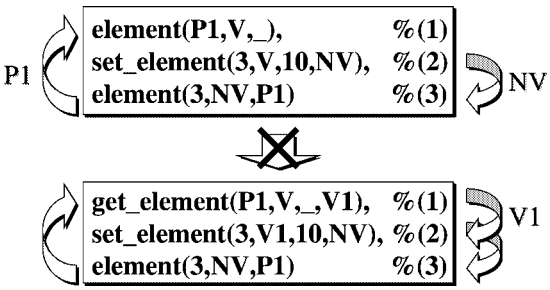

図 1 デッドロックする例（1）
Fig. 1 Example of deadlock (1).

☆ 参照の終わった要素は破壊代入される可能性があるため，E は配列の中を指すポインタにはならないように実装する必要がある．すなわち，E は即値か，配列外のデータを指すポインタになる．

しているからである．依存関係により，ゴール (2) →ゴール (3) →ゴール (1) という順でしか実行できないが，ここにゴール (1) →ゴール (2) という逐次化を行うと，サイクリックな依存を作ってしまうため，デッドロックする．

デッドロックが起きないようにするためには，プログラムのデータフロー解析を行い，逐次化を行おうとしている element，set_element の間に，逐次化しようとしている方向と逆方向の依存関係がないことを保証する必要がある．

3.3 データフロー解析

ここで，依存関係を以下のように定義する：

- 変数 A の値が束縛されない限り，変数 B の値が束縛されないとき，変数 B は変数 A に依存する．
- 変数 A が束縛されない限りゴール foo が実行されないとき，ゴール foo は変数 A に依存する．
- ゴール foo が実行されない限り変数 A が束縛されないとき，変数 A はゴール foo に依存する．
- ゴール foo が実行されない限りゴール bar が実行されない場合，ゴール bar はゴール foo に依存する．

定義から明らかなように，これらの依存関係は推移律を満たす．すなわち，たとえば変数 B が変数 A に依存し，ゴール foo が変数 B に依存する場合，ゴール foo は変数 A に依存する．

また，あるゴールを実行するのに必要な変数（この変数が束縛されない限りゴールが実行されない変数）を入力と呼び，あるゴールを実行によって出力される変数（このゴールの実行がない限り束縛されることはない変数）を出力と呼ぶことにする．すなわち，入力変数からゴールへ，ゴールから出力変数へ，という依存関係が存在する．

たとえば，element の入力はインデックスを表す第 1 引数と配列そのものである第 2 引数であり，出力は要素を表す第 3 引数である．また，set_element の入力は同様にインデックスを表す第 1 引数と配列を表す第 2 引数であり，出力は新しい配列を表す第 4 引数である．

2 つのゴール foo と bar をこの順で逐次化しようとした場合，デッドロックが起きないようにするには，bar から foo への依存関係がないことを保証する必要がある．これには，foo の出力変数から bar の入力変数への依存が存在しないことを確認すればよい．

先ほどの例では，ゴール (1) の入力 P1 は，(ゴール (3) の存在により）ゴール (2) の出力 NV に依存する．このため，ゴール (1) →ゴール (2) という逐次化ができないということが分かる．

ここで，P1 が NV に依存しているという情報は，我々が提案したデータフロー解析手法[20]を用いることで得ることができる．この手法はプログラムのデータフローを大域的に解析することにより，変数間の依存関係を与えるものである．この手法は，存在する可能性のある依存関係をすべて検出するので，安全に解析に用いることができる．

ただし，単純にこの手法を適用したのでは，再帰（相互再帰を含む）する述語の依存情報を正確に求めることは難しい．これは，再帰する述語の依存情報を求めるためには，自分自身の依存情報を必要とするためである．

この問題を解決するため，我々は不動点アルゴリズム[18]（あるいは反復アルゴリズム[1]）を採用した．この手法では，再帰する述語の依存情報を求める際，その述語自身の依存情報として，最初は空の情報を用いる．そして，得られた依存情報を基に計算を再度行い，計算結果が収束するまでこれを繰り返す（依存情報は有限であり，単調に増加するため，必ず停止する）．

3.4 大域的な逐次化

前節の説明は，同一定義節内に配列の生成，参照，更新がある場合について行ったが，実際のプログラムでは，それぞれが別の定義節に存在するのが普通である．本節では，大域的な逐次化を行う方法を示す．

大域的な逐次化を行う際の基本的な考え方は，

- set_element からスタートして，同一配列を参照する述語を逐次化していく．配列の生成までたどりつけば，破壊代入に変更可能．

である．

これを実現するには，

- 内部で配列の参照や生成を行う述語をどう扱うか．
- 配列操作の入力，出力，それらの間のデータ依存解析をどう行うか．

という問題がある．以下の節では，これらについて説明した後，大域的な逐次化がどのように行われるかについて説明する．

3.4.1 配列操作のタイプ

配列操作が別々の定義節に存在する場合，それぞれの操作を内部から呼び出す述語を，その操作を行う述語として扱う．たとえば，

```
foo(P,V,E):- bar(P,V,E).
bar(P,V,E):- element(P,V,E).
```

という述語があった場合，まず bar は element を呼び出しているので，element と同様，配列を参照する述語だとして扱われる．次に foo は配列を参照する述

表 1 配列操作のタイプ
Table 1 Type of array operation.

alloc	配列の生成のみである.
seq(N)	すでに逐次化されていて，次に第 N 引数を調べる必要がある.
ref	配列の参照のみである.
allocref	配列の参照と生成を含む.
seqref(N)	配列の参照を含むが，次に第 N 引数を調べる必要がある.
invalid	無効なタイプ.

語 bar を呼び出しているので，同様に配列を参照する述語だとして扱われる．

これは参照の例だが，実際には，配列の生成や，= などによって追跡する変数が変わる場合も扱う必要がある．

ここで，配列操作のタイプを**表 1** に示すように分類する．

たとえば，

```
foo(A):- element(1,A,_).
```

の場合，述語 foo の第 1 引数の配列の参照タイプは refである，などとして使う．ここで，配列を更新する述語をタイプとして扱っていない．これは，アルゴリズムでは，更新述語から生成述語までたどっていくため，たどってきた元が更新述語であるということが分かるからである．たどっていく途中に別の更新述語が現れた場合，本質的に配列のコピーが必要であるため，タイプとして invalid を用いる．全体の構成は，後で詳しく述べる．

alloc では vector による生成，即値による生成（A = {1,2,3} など）のほか，set_element も第 4 引数の配列を生成するものとして扱う．

seq(N) は以下のような場合である：

```
foo(A,B):- get_element(1,A,_,B).
```

というプログラムで，B をたどってきた場合，すでに参照は逐次化されており，次に変数 A を調べる必要がある．このような場合，foo の第 2 引数のタイプは seq(1) になる．この例では，get_element を用いたが，典型的には=によってこのような場合が起こる．たとえば，

```
foo(A,B):- B = A.
```

の foo の第 2 引数のタイプも seq(1) である．

allocref，seqref(N) は，alloc，seq(N) のほかに ref タイプの述語が同じ配列を参照する場合である．たとえば

```
foo(A):- A = {1,2,3}, element(1,A,_).
```

において，foo の第 1 引数の参照タイプは allocref になる．

配列参照のタイプを調べるには，次のようにする．

(1) その述語の定義節内で，その配列を引数に持つ述語を集める．

(2) その述語がユーザ定義の述語の場合はこの手続きを再帰的に呼び出すことによって，その述語のタイプを求める〔再帰（相互再帰を含む）する述語の場合は，データフロー解析を行う場合と同様，不動点アルゴリズムを用いる〕．

(3) 以下のルールに従いタイプの情報を集める．その述語から呼び出される述語のタイプが，

- ref のみであれば，結果は ref である．
- alloc が 1 つだけなら，結果は alloc である．
- alloc が複数，あるいは alloc と seq(N)/seqref(N) があれば，invalid である．
- alloc/allocref と ref があれば allocref である．
- seq(N)/seqref(N) が複数あれば，invalid である．
- seq(N)/seqref(N) と refがあればそれを seqref(N) として以下に進む．
- seq(N)，seqref(N) があれば，N で表された変数を引数に持つゴールに対しても，タイプ情報を求める．
  N で表された変数がヘッドに存在する場合（ヘッドに存在する変数の場所が M 番目だとする），N で表された変数を引数に持つゴールのタイプが，
  - alloc，allocref，seq(N)，seqref(N) であれば invalid とする．alloc/allocref が invalid なのは，呼び出し側で配列が確保されていることを仮定しているためである．今回の実装では，全体を通して，ヘッドに対象の変数が現れた場合，呼び出し側で配列の確保が行われているものとして扱う．これは実装を容易にするためである．
  - ref であれば，seqref(M) になる．
  - N で表された変数を参照するゴールが存在しなければ，元のタイプに応じて seq(M) または seqref(M) になる．

  N で表された変数がヘッドに存在しない場合は，基本的に N で表された変数を引数に持つゴールのタイプが，全体のタイプとなる．ただし，元のタイプが seqref(N) の場合は逐次化されていない配列の参照があるため，N で表された変数を参照するゴール

のタイプが seq(M) でも seqref(M) となる．

(4) 定義節が複数ある場合は，各定義節でこれを行い，結果を上記と同様のルールに従い集める．ただし，seqref(N)，seq(N) のルールは異なり，

- seq(N)/seqref(N) と ref があれば seqref(N) になる．
- seq(N)/seqref(N) と seq(M)/seqref(M) があれば，M と N が一致しないと invalid である．一致すれば seq(N) または seqref(N) である．
- seq(N)/seqref(N) と alloc があれば，それぞれ seq(N)/seqref(N) になる．
- seq(N)/seqref(N) と allocref があれば，seqref(N) になる．

上で述べたルールの中で，同一節中で seq(N)，seqref(N) を扱うルールは複雑なため，例をあげて説明する．

```
foo(X,Y):- bar(X), get_element(1,X,E,Y).
```

のようなプログラムにおいて，foo の第 2 引数のタイプを考える．Y は get_element の第 4 引数である．get_element の第 4 引数のタイプは seq(2) である．ここで，get_element の第 2 引数 X は bar の第 1 引数であり，またヘッドにも現れている．したがって，bar の第 1 引数のタイプが ref ならば，foo の第 2 引数のタイプは seqref(1) になり，それ以外ならば invalid となる．

また，

```
foo(W,Y):- bar(W,X), get_element(1,X,E,Y).
```

の場合は，get_element の第 2 引数 X は bar の第 2 引数であるが，ヘッドには現れていない．この場合，bar の第 2 引数のタイプが foo の第 2 引数のタイプとなる．

また，alloc が複数ある場合は invalid となるが，これはありえないと考えられるかもしれない．しかし，配列の生成が A = {1,2,3} のような形の場合，配列の生成ではなく，要素の取り出しを行っている場合がある．これは，Fleng における = がユニファイを表すためである．たとえば，

```
A = {1,Q,3}, A = {P,2,R}
```

というプログラム片があった場合，実行後は P = 1, Q = 2, R = 3 になる．このような配列の参照は扱うことができないので，両者を alloc として扱い，同じ配列に対して複数の alloc があった場合，invalid として処理をあきらめることにしている．

3.4.2 依存解析

先に述べたとおり，安全に逐次化を行うためには配列操作間の依存関係を解析する必要がある．

ある述語の配列操作における入力/出力変数を求めるには以下のようにする．

(1) その述語の定義節内で，その配列を引数に持つ述語を集める．

(2) その述語がユーザ定義の述語の場合はこの手続きを再帰的に呼び出すことによって，その述語で入力/出力されている変数を求める〔再帰（相互再帰を含む）する述語の場合は，不動点アルゴリズムを用いる〕．

(3) データフロー解析の結果を用いて変数間の依存関係を求める．この結果から，入力変数が依存する可能性のある変数の集合と，出力変数に依存する可能性のある変数の集合を求める．

(4) これらの変数の集合のうち，ヘッドに現れた変数が，その述語のその定義節での入力/出力変数となる．

(5) 定義節が複数ある場合は，各定義節でこれを行い，入出力変数について集合和をとる．

たとえば，

```
foo(I,V,O):-
  add(I,1,P), element(P,V,E), add(E,1,O).
```

において，foo の第 2 引数に現れた配列 V について考える．ここで，V を引数に持つ述語は element だけである．element の入力変数は P，出力変数は E である．

この定義節中で P が依存している変数は I，E に依存している変数は O であることが，データフロー解析の結果から知ることができる．I，O はともにヘッドに現れているので，foo の第 2 引数の配列参照における入力変数は第 1 引数の I，出力変数は第 3 引数の O である．

3.4.3 参照の逐次化

次に，同一節中の参照の逐次化を以下のように行う．

(1) 対象となる配列を引数に持つ述語がユーザ定義の述語の場合は，この手続きを再帰的に呼び出すことで，その述語での配列参照を逐次化する．未知の述語や，set_element の入力になっていた場合は，逐次化をあきらめる（後者は，本質的にコピーを必要としているので，コピー除去不能である）．また，element は get_element に変更する．

(2) それらの述語のタイプを求める．invalid の場合は逐次化をあきらめる．

(3) これらの参照において，入力，出力となる変数を求める．

(4) データフロー解析の結果から，変数間の依存関

係が分かる．その結果と，配列参照の入力/出力変数から，それぞれの参照間の依存関係を求める．その依存関係の順に述語をソートし，依存関係の順序を保存するように，引数を付け直す．これにより，参照の逐次化を行う．ここで，タイプ情報を元に，配列の生成があった場合は，それを起点に逐次化する．seq(N) か seqref(N) のタイプを持つものがあった場合，N で表された変数を参照する述語に対しても，同様に逐次化を行う．

また，逐次化の際，参照後の配列を渡すためにヘッドの引数を 1 つ増やす必要がある．

たとえば，

```
foo(V,E):-
  bar(V,P1), add(P1,1,P2),
  element(P2,V,E).
bar(V,P1):- element(1,V,P1).
```

というプログラムにおいて，foo の第 1 引数の配列の参照を逐次化することを考える．この場合，まず bar の第 1 引数の配列参照を逐次化する．

bar は中で element を 1 つ呼び出しているだけであるから，

```
bar(V,P1,V1):- get_element(1,V,P1,V1).
```

のように逐次化できる．また，bar の第 1 引数の参照のタイプは ref であり，入力変数はなし，出力変数は第 2 引数の P1 である．

次に，foo の中で逐次化を行う．bar の出力変数である P1 から element の入力変数である P2 に対して依存関係が存在するため，bar から element の方向への逐次化を行う．

element は get_element に逐次化できて，bar は逐次化すると第 3 引数に参照後の配列を返す bar になるので，それらの変数を付け直すと，以下のようになる．

```
foo(V,E,V2):-
  bar(V,P1,V1), add(P1,1,P2),
  get_element(P2,V1,E,V2).
bar(V,P1,V1):- get_element(1,V,P1,V1).
```

実は，このやり方では，逐次化する順序が限定される．たとえば

```
foo(V,...):- bar(V,...), baz(V,...).
bar(V,...):-
   element(...,V,...),  % (1)
   element(...,V,...).  % (2)
baz(V,...):-
   element(...,V,...),  % (3)
   element(...,V,...).  % (4)
```

のような定義において，foo の第 1 引数の V の参照を逐次化する場合を考える．そのためには bar 内，baz 内の配列参照を逐次化し，さらに bar，baz 全体で逐次化する．したがって，配列参照 (1)，(2)，(3)，(4) の逐次化される順番は，

- ((1) → (2)) → ((3) → (4))
- ((2) → (1)) → ((3) → (4))
- ((1) → (2)) → ((4) → (3))
- ((2) → (1)) → ((4) → (3))

以外には作れない．すなわち，プログラムのデータフローから，たとえば (1) → (3) → (2) → (4) という順番でしか実行できない場合は，この方法では逐次化できないことになる．上の依存解析では，依存関係が両方向にあるというように解析される．このような場合は逐次化をあきらめる．

3.4.4 コピー除去処理

最後に全体の処理は，次のようになる．

(1) set_element を見つけ，それを assign_element に書き換える．

(2) その定義節内でその配列を引数に持つ述語のタイプを求める．

(3) タイプが参照を含むもの（ref, allocref, seqref）は，その内部で呼ばれる参照を前述の手続きを用いて逐次化する．

(4) 参照の逐次化の場合と同様に,（複数の）参照→更新の順に逐次化を行う．参照の逐次化の場合と同様に，参照/更新における入力/出力変数を求め，データフロー解析順に逐次化する．更新から参照への依存がある場合は逐次化をあきらめ，配列のコピーを挿入する．また，タイプ情報から，配列の生成を行う述語があると分かった場合，それを先頭にして逐次化を行う．seq(N) か seqref(N) のタイプを持つものがあった場合，N で表された変数を参照する述語に対しても，同様に逐次化を行う．

(5) ヘッドにその配列が現れている場合は，呼び出し側で配列が生成されるとして，参照→更新の順に逐次化する．そのあと，その述語を呼び出している部分にさかのぼって，元の述語を更新操作だとしてこの手続きを繰り返す．ここで，ヘッド部にその配列が現れているにもかかわらず，ボディ部の述語で配列が生成されている場合は，解析が困難になるためあきらめ，コピーを挿入する．

例をあげて説明する．

```
foo(E,NV):-
  V = {1,2,3,4,5},
  element(1,V,E), bar(V,NV).
bar(V,NV):-
  element(2,V,P),
  set_element(P,V,10,NV).
```

というプログラムを考える．

まず，bar 内の set_element を assign_element に書き換える．次に set_element と同じ配列を参照する element を逐次化する．element は get_element に変換される．set_element の出力から element の入力への依存は存在しないので，bar は以下のようになる．

```
bar(V,NV):-
  get_element(2,V,P,V1),
  assign_element(P,V1,10,NV).
```

次に，bar を更新述語であるとして foo 内の処理を行う．element は get_element に変換される．bar の出力変数は NV である．NV から element の入力への依存関係は存在しない．したがって，配列の生成 V = {1,2,3,4,5}，element，bar をこの順で逐次化することができて，最終的に以下のようになる．

```
foo(E,NV):-
  V = {1,2,3,4,5},
  get_element(1,V,E,V2), bar(V2,NV).
bar(V,NV):-
  get_element(2,V,P,V1),
  assign_element(P,V1,10,NV).
```

この処理法では，コピー除去をあきらめる際に，set_element を残すのではなく，プログラムの呼び出し木のできるだけ上部で配列のコピーを挿入している．これは，配列のコピーが繰返しの中で行われると実行時間に与える影響が大きいため，なるべくコピー操作を繰返しの外で行いたいためである．これにより，分割コンパイルなどで配列の生成と使用が別ファイルになってしまうような場合でも，コピーを 1 回だけ行って，あとは破壊代入で処理するといったようなことが可能になる．

また，プログラム全体のデータフロー解析は各更新の処理ごとに毎回行わなければならないことに注意する必要がある．これは，プログラムの逐次化によってデータフローが変わってしまうためである．たとえば，**図 2** のようなデータフローを持つプログラムを考える．この図では更新が 2 つあるが，プログラムに元からあった依存だけを考慮に入れると，両者の更新に対して右側の破線で表したような逐次化を行ってしまう．

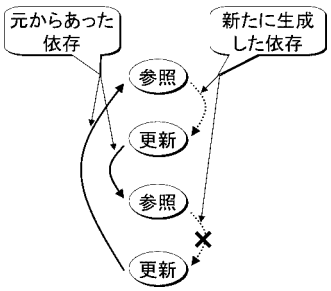

図 2 デッドロックする例 (2)
Fig. 2 Example of deadlock (2).

この場合，図のようにサイクリックな依存を作ってしまうのでデッドロックを引き起こす．これを避けるためには，1 つの更新の処理ごとに依存関係を再計算する必要がある．

プログラム全体のデータフローを毎回計算するのはコストが高いため，データフロー解析は必要な部分だけ行うことにする．すなわち，ある定義節内の依存関係が必要な場合，その定義節から呼ばれる述語のデータフロー解析のみを行う．解析した結果は保持しておくが，プログラムの逐次化が行われ依存関係が変化した述語（あるいは依存が変化した述語を呼び出す述語）は再計算が必要であるため，結果を破棄する．

3.5 多次元配列の扱い

これまでの議論では，基本的に 1 次元配列のみを扱ってきた．Fleng での多次元配列は，配列を要素に持つ配列として扱われているので，そのまま扱うのはそれほど簡単ではない．これは，一時的に現れる配列に対しても，解析を行う必要があるためである．

そこで，多次元配列のアクセスをプログラム上で明示的に示すことで，中間的な配列を使わないことにした．たとえば，多次元配列版の element は，以下のように定義する．

- melement([P_1,P_2,...,P_N],MV,E)
  N 次元配列 MV の第 1 次元目が P_1，第 2 次元目が P_2,..., 第 N 次元目が P_N の要素を E に返す．

配列の生成，更新，破壊代入なども，同様に定義する．

4. データ並列性の抽出

Fleng で配列の各要素に対して何らかの操作を行うプログラムを記述したとする．この操作が十分に大きな操作なら，それぞれの操作をゴールとして独立に実行することで並列処理が可能である（**図 3**）．

しかし，この操作が非常に小さい（たとえば，各要素を 2 倍するというような操作）場合，並列に実行することは，効率が悪い．このため，我々が文献 21)

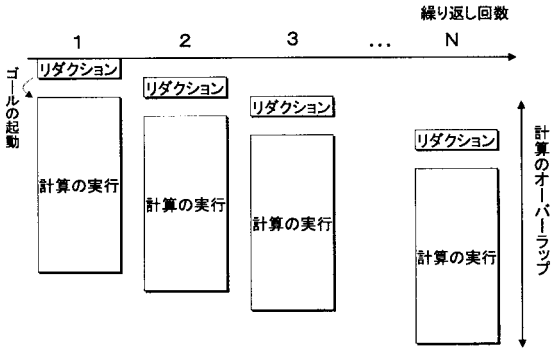

図 3 計算が十分に大きい場合
Fig. 3 When computation is large enough.

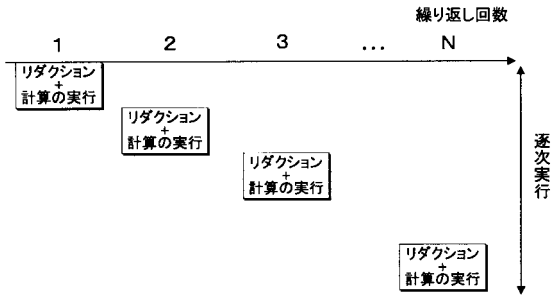

図 4 粒度制御された場合
Fig. 4 After granularity optimization.

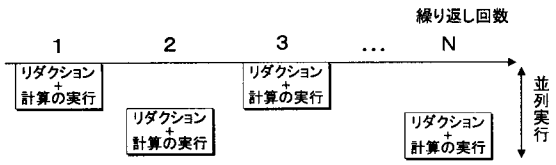

図 5 データ並列性を抽出した場合
Fig. 5 After extraction of data-parallelism.

で述べたような粒度制御を行うことにより，逐次化を行う（粒度制御を行わないとしても，無駄な計算を並列に実行してしまうだけなので，実行時間は短縮しない）．この様子を**図 4** に示す．

しかし，たとえば配列の各要素を 2 倍するというような操作は，各要素ごとに独立に実行できるはずである．この様子を**図 5** に示す．この図では，2 要素ずつ独立に計算を行っている．

実際，Fortran のような言語の並列化コンパイラでは，このようなデータ並列性を抽出することで，並列処理を行っている．

本章では，Fleng でデータ並列性を抽出することで，並列度を向上する手法について述べる．

4.1 基本的な考え方

配列を扱うプログラムでは，単純なゴール起動により並列度が抽出できなくても，データ並列性を抽出することにより，並列実行できる場合がある．以下に例をあげて説明する．

```
main(NV):-
  V = {1,2,3,4,5,6,7,8,9,10},
  vector(10,NV),
  double(V,1,10,NV).

double(V,Crnt,Max,NV):-
  (Crnt =< Max ->
    element(Crnt,V,E),
    NewE is E * 2,
    element(Crnt,NV,NewE),
    Nxt is Crnt + 1,
    double(V,Nxt,Max,NV)
  ).
```

このプログラムは，配列 V の要素を 2 倍した新たな配列 NV を生成するプログラムである．このようなプログラムの場合，繰返しの中で実行する部分は，配列の要素を取り出して 2 倍するだけなので，非常に小さい．したがって，粒度最適化を行うことで，逐次化による高速化が行われる．

しかし，このプログラムの場合，配列の各要素を 2 倍するという操作は，要素ごとに並列に実行可能なはずである．たとえば上のプログラムは以下のように変更することで，並列度を向上することができる（述語 double の定義は同じ）．

```
main(NV):-
  V = {1,2,3,4,5,6,7,8,9,10},
  vector(10,NV),
  double(V,1,5,NV)
  double(V,6,10,NV).
```

この場合，double を 5 要素ずつ 2 つに分けることで，並列度が向上している．

4.2 データ並列性の抽出法

前節で述べた変換はつねに可能なわけではない．たとえば，配列の要素の総和を計算する次のようなプログラムでは，このような並列化はできない☆．

```
main(NV):-
  V = {1,2,3,4,5,6,7,8,9,10},
  sum(V,1,10,0,Sum).

sum(V,Crnt,Max,CrntSum,Sum):-
```

☆ 実際にはリダクション演算として並列化可能であるが，本論文ではリダクション演算としての並列化は扱わない．

```
(Crnt =< Max ->
  element(Crnt,V,E),
  NxtSum is CrntSum + E,
  Nxt is Crnt + 1,
  sum(V,Nxt,Max,NxtSum,Sum)
;
  Sum = CrntSum
).
```

これは，sum の第 4 引数が，繰返しごとに異なっているため，繰返し間にわたる依存関係が存在するためである．すなわち，繰返し間にわたる依存関係が存在しなければ，データ並列性の抽出が可能になる．

したがって，データ並列性が抽出可能である条件は以下のようになる：

- 引数が
  - 誘導変数
  - ループ不変変数

  のみであること．
- 再帰の脱出条件は，誘導変数とループ不変変数との大小比較のみであること．

ここで，誘導変数とは繰返し 1 回で定数分だけ増減する変数であり，ループカウンタであると考えればよい．上の double，sum では第 2 引数（Crnt）が誘導変数である．

4.3 配列のコピー除去を行った場合

前節では配列の更新は扱わなかった．しかし，通常のプログラムでは，配列の更新は頻繁に行われる．本節では，配列の更新を含むプログラムについて述べる．

例として，先ほどのプログラムを配列の更新を用いて記述し，コピー除去を行って set_element を assign_element にしたプログラムを以下に示す．

```
double(V,Crnt,Max,NV):-
  (Crnt =< Max ->
    get_element(Crnt,V,E,V1),
    NewE is E * 2,
    assign_element(Crnt,V1,NewE,NxtV),
    Nxt is Crnt + 1,
    double(NxtV,Nxt,Max,NV)
  ;
    NV = V
  ).
```

この場合も先ほどと同様に，プログラムの字面からは，第 1 引数の V が更新されているように見える．しかし，V に対する更新操作を行っているように見える get_element，assign_element は，実は，**同じ配列を返している**という点に注目する必要がある．すなわち繰返しを通して，double の第 1 引数に渡されている配列は同じ配列を指している．

したがって，この場合もデータ並列性の抽出が可能である．呼び出し側は次のようになる．

```
main(NV):-
  V = {1,2,3,4,5,6,7,8,9,10},
  double(V,1,5,NV1),
  double(V,6,10,NV2),
  (wait(NV1),wait(NV2)-> NV = V).
```

まず，第 1 引数には，同じ配列を渡す．また，NV を参照する際に，更新操作がすべて終わっている必要があるため，wait で終了を確認してから NV を束縛する必要がある．

ただし，すべての場合でこのような変換が可能なわけではない．この場合に可能だったのは，get_element によって参照する要素と，assign_element によって更新する要素が，各繰返しで独立だったからである．たとえば，配列の 1 つ前の要素を 2 倍して現在の場所を更新するようなプログラムでは，先ほどのような変換は不可能である．

すなわち，配列の参照（get_element），更新（assign_element）が，異なる繰返しで同じインデックスの配列要素をアクセスするかどうかが判定の条件になる．

この問題は，逐次型言語の並列化コンパイラにおける，繰返しをまたぐ配列の依存関係の解析問題と等価である．この問題は Fortran のような言語の並列化コンパイラにおいて非常によく研究されており，有名なものとしては GCD テストなどがある[18]．

同様のテストを get_element，assign_element の引数に対して行い，重ならないことを保証すれば，先ほどのようなループの分割が可能である．

以上をまとめると，データ並列性を抽出できる場合の条件は以下のようになる：

- 引数が
  - 誘導変数
  - ループ不変変数
  - assign_element，get_element によって更新される配列

  のみであること．
- assign_element，get_element によって更新される配列が引数にある場合は，配列の参照と更新の間で繰返しをまたぐ依存がないこと．
- 再帰の脱出条件は，誘導変数とループ不変変数との大小比較のみであること．

繰返しをまたぐ依存の解析法は，本実装では GCD

表 2 コピー除去数と並列化したループ数
Table 2 Number of copy elimination and parallelized loops.

プログラム	コピー数		ループ数	
	オリジナル	コピー除去後	並列化可能ループ	並列化したループ
histogram	1	0	1	1
bubble	2	0	0	0
vecqsort	2	**1**	0	0
vecqsort2	2	0	0	0
vecmul	1	0	2	2
gauss	3	0	1	1
gaussjordan	2	0	2	2
fft	2	0	1	**0**

テストを用いた．

このように，容易に並列化コンパイラの技術を応用できる点が，本手法の重要な点である．これは，コピー除去を明示的な配列の更新に変換することによって実現したことに由来する．コピー除去を，たとえば後述する Shallow Binding のような手法で行った場合，本手法のようなデータ並列性の抽出は困難になる．

5. 実装と評価

5.1 実　　装

実装は，Fleng プログラムを入力し，配列処理の最適化を行った Fleng プログラムを出力するプリプロセッサの形で行った．プログラムは Fleng 自身で記述されており，約 6600 行のプログラムである．

コンパイラドライバは，まず Fleng に用意されているマクロを展開するプログラムを起動し，その後本論文で述べた配列処理の最適化を行う．そして，出力されたプログラムを文献 21) で述べた粒度最適化を行うプログラムで処理してからコンパイルを行う．

5.2 評 価 条 件

評価対象のプログラムは以下のものを用いた．

histogram ヒストグラム作成のプログラムである．1 から 100 までの整数が要素である長さ 1000 のリストのヒストグラムを作成した．

bubble 配列を用いたバブルソートである．入力は 100 要素の配列を用いた．

vecqsort 配列を用いたクイックソートである．入力は 200 要素の配列を用いた．

vecqsort2 配列を用いたクイックソートである．`get_element` を用い，手動で逐次化を行っている．入力は 200 要素の配列を用いた．

vecmul 1 次元配列を定数倍するプログラムである．定数倍した結果で `set_element` により元の配列を更新する．配列長は 1024 とした．

gauss ガウスの消去法による，連立一次方程式の求解である．32×32 の 2 次元配列を処理する☆．

gaussjordan ガウスジョルダンの消去法による，連立一次方程式の求解である．32×32 の 2 次元配列を処理する．

fft 高速フーリエ変換を行うプログラムである．データ長は 128 とした．

実行時間は我々の研究室で設計，開発した並列計算機，PIE64 [2] 上で評価した．PIE64 は Fleng の高速実行を目的に設計された並列計算機であり，64 台の要素プロセッサ数を持つ分散共有メモリマシンである．プロセッサはマルチコンテクスト処理をサポートする．また，相互結合網は自動負荷分散の機能を持つ．

コンパイル時間はワークステーション（Sun Enterprise 3000）上の Fleng 処理系で CPU を 1 台だけ用い，測定した．

5.3 コピー除去数と並列化したループ数

表 2 にオリジナル，コピー除去後の配列コピー数と，並列化可能なループ数と実際に並列化したループ数を示す．いずれも，ソースコード上に現れた数であり，実行回数ではない．

まず，コピー数を見てみると，`vecqsort` はコピーが 1 つ除去できていない．これは，データ依存解析の精度が足りず，存在しない依存関係を存在するものとして扱ってしまったことが原因である．ここで，`element` を手動で `get_element` に変換し，逐次化を行った `vecqsort2` では，コピー除去ができている．ここで注意して欲しいのは，`vecqsort2` でも `set_element` を用いているということである．`vecqsort2` では，配列の生成まで逐次化されていることを処理系が確認したうえで，安全にコピーを除去している．その他のプログラムに対してはすべてのコピーが除去できている．

次に並列化を行ったループ数を見てみる．`histogram`

☆ PIE64 には浮動小数点演算器が備わっていないため，固定小数点で計算を行った．`gaussjordan`，`fft` も同様である．

表 3 配列処理最適化による速度向上
Table 3 Speedup with optimization of vector processing.

プログラム	実行時間（速度向上率）				
	オリジナル	コピー除去後		データ並列性抽出後	
histogram	5035 ms	142 ms	（35 倍）	76 ms	（1.9 倍）
bubble	21825 ms	416 ms	（52 倍）		—
vecqsort	7256 ms	5278 ms	（1.4 倍）		—
vecqsort2	7393 ms	480 ms	（15 倍）		—
vecmul	45549 ms	92 ms	（500 倍）	27 ms	（3.5 倍）
gauss	26079 ms	3341 ms	（7.8 倍）	1605 ms	（2.1 倍）
gaussjordan	38525 ms	4597 ms	（8.4 倍）	2134 ms	（2.2 倍）
fft	5325 ms	159 ms	（34 倍）		—

における並列化可能なループとは，配列の初期化の部分である．`vecmul` で並列化可能なループが 2 つあるのも，1 つは配列の初期化部である．また，`fft` では並列化可能なはずのループが並列化できていない．これは，並列化可能かどうかの判定に誘導変数が変化する範囲を知らなければならないためで，GCD テストでは並列化可能であるという判定はできないためである．

5.4 実行時間

次に**表 3** に実行時間の評価を示す．実行時間の評価は，PIE64 上で行った．それぞれのプログラムは文献 21) で述べた粒度最適化を行ってある（ゴール融合，強制インライン展開ともに処理を行った）．実行時間は，3 回実行した値の平均値を用いた（ガーベジコレクションの時間を含む）．プロセッサ台数は 64 台で評価した．

また，データ並列性の抽出を行う際は，どれだけの数のループに分割するかが性能に大きな影響を与える．今回は，分割した各ループで少なくとも 4 回繰返しが行われるようにした（この値はオプションで変更できる）．これは，繰返しの回数が少ないと実行粒度が細かくなりすぎ，オーバヘッドが大きくなるからである．今回の評価では，全体の繰返し回数が 100 回の場合，たとえプロセッサが 64 個あっても 25 個のループにしか分割されない．

まず，コピー除去の効果についてであるが，それぞれのプログラムで数倍〜数百倍と大幅な速度向上を達成している．ただし，速度比は配列をコピーする大きさ，回数に依存するため，入力サイズに依存する．`vecqsort` では 1.4 倍程度とあまり効果が見られないが，これは削除できなかったコピーを再帰の外に移動できなかったためである．

次に，データ並列処理の効果であるが，それぞれのプログラムで数倍程度の速度向上を達成している．しかし，予想されたほどの速度向上ではないのは，入力サイズが小さかったためと，通信にかかる時間が速度向上を制限してしまったためではないかと考えられる．

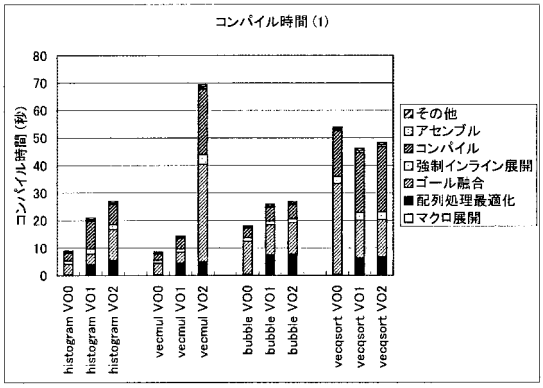

図 6 コンパイル時間（1）
Fig. 6 Compilation time (1).

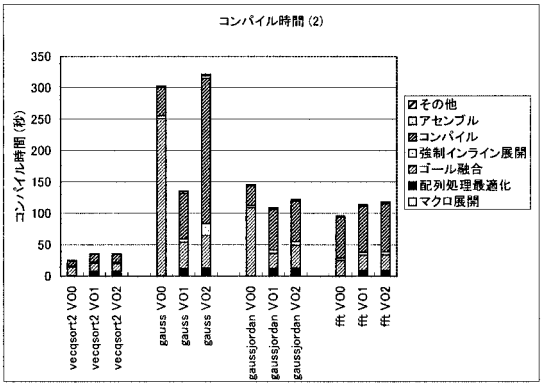

図 7 コンパイル時間（2）
Fig. 7 Compilation time (2).

5.5 コンパイル時間

最後にコンパイル時間についての評価を示す．**図 6** に `histogram`，`vecmul`，`bubble`，`vecqsort` のコンパイル時間を，**図 7** に `vecqsort2`，`gauss`，`gaussjordan`，`fft` のコンパイル時間をそれぞれ示す．

このグラフでは，`V00` が配列処理の最適化なしを，

VO1がコピー除去のみを行った場合を，VO2がコピー除去とデータ並列性の抽出を行った場合を示す．

コンパイル時間はSun Enterprise 3000上のFleng処理系でCPUを1台だけ用い，測定した．

このグラフから，配列処理最適化部の処理時間のコンパイル時間全体に占める割合はそれほど大きくないことが分かる．VO2の方がVO1よりも全体のコンパイル時間が大きくなる傾向があるが，これはデータ並列性を抽出する部分で述語の数が増えるためである．

全体のコンパイル時間に占める割合は，ゴール融合部とコンパイル部が大きい．また，ゴール融合部は入力プログラムの性質によって実行時間が大きく変化する．vecqsortやgauss，gaussjordanでは，配列処理の最適化を行わない方がコンパイル時間が長くなっているが，これはset_elementの定義がassign_elementの定義よりも複雑であるため（両者ともFlengレベルで記述されている部分がある），この部分でゴール融合部が複雑な処理を行い，コンパイル時間が長くなっているものと思われる．

5.6 考　　察

コピー除去に関しては，ほとんどのプログラムですべてのコピーを除去できている．しかしその一方，vecqsortではコピーを除去することができなかった．これは，データフロー解析をさらに精密に行うよう改良することで解決することができる．またそのほかにも，vecqsort2のようにユーザが明示的にget_elementを用い，逐次化を行うことでも対応することができると考えられる．コンパイラは除去できなかった配列のコピーがあると，そのことをユーザに知らせる．ユーザはこのメッセージを見て，手動で配列参照を逐次化することで，問題を解決することができる．

ここで注意が必要な点は，assign_elementを用いた危険な破壊代入を行う必要はないということである．set_elementを用いたままでも，get_elementによって逐次化されていることを処理系が確認し，安全にassign_elementに書き換えることができる．ユーザに破壊代入操作を行わせると，非常に発見しにくい同期バグを引き起こす可能性が高い．

また，別の解決法として，後述するShallow Bindingなどの実行時に行う手法と併用することが考えられる．しかし，この場合Shallow Bindingを用いてコピー除去を行った配列に対しては，データ並列性の抽出は行えなくなる．

6. 関 連 研 究

6.1 コピー除去

コピー除去に関する研究は，単一代入変数を用いる言語でさかんに行われている．

関数型言語においては文献5)，10)，13)などの研究があるが，逐次型の関数型言語に関する研究が中心である．逐次型言語の場合は実行順序は決定しているため，その実行順序において配列の更新時に参照が残っていないことを保証することで，コピー除去を行うことが可能である（ただし，遅延評価を行う関数型言語では，実行順序の解析はそれほど容易ではない）．しかし，Committed-Choice型言語や，データフロー関数型言語のような並列言語では実行順序が実行時まで決定しないため，これらの手法を適用することはできない．

文献13)では，逐次型の関数型言語において，評価順序を変更し，コピー除去が可能な場合を増やす方法について議論されている．評価順序を変更する部分で並列言語への応用も考えられるが，この文献では逐次型の言語についてしか扱われていない．また，この文献では，コピー除去ができなかった場合，コピー操作を再帰の外に追い出すことで，コピー数を減らす方法についても述べられている．この手法は，本研究の手法と類似している．

並列関数型言語に関する研究としては，データフロー関数型言語Vにおける研究[23)]がある．この研究では，本研究と同様に逐次化によってコピー除去できる場合を増やしているが，デッドロックに関する議論は行われていない．また，データフロー関数型言語SISAL[6)]でもコピー除去が行われているが，文献13)によると，関数間にわたる配列に関してはリファレンスカウントを用いている．

また，Idは，コピー除去自体をあきらめ，プログラマに破壊代入の管理をゆだねているが，これはプログラマに対する負担が大きい．

並列論理型言語においても多くの研究がなされており，なかでもKL1においてMRB（Multiple Reference Bit）を用いる手法[8)]が著名である．これは，1ビットのリファレンスカウンタを用い，配列の単一参照性を調べるというものである．単一参照とは，そのデータを参照しているゴールが他に存在しないということを意味する．この手法はリファレンスカウンタを扱うオーバヘッドが存在するほか，できるだけ単一参照性を保つため，注意深くプログラミングする必要があるという問題点がある．

KL1 には，set_vector_element という要素の参照と更新を同時に行う述語があり，これを用いて配列を扱う．単一参照性を保つためにはたとえ要素の参照だけを行いたい場合でも，同じ値で更新するようにして set_vector_element を用いるというプログラムの書き方をする必要がある．MRB を利用する手法では，このようなプログラミングを行わなければ，コピー除去を行うことができない．

静的に単一参照性を検出するという研究には，文献 14)，24) などがあるが，同様なプログラミングを要求される．ただし，このようなプログラミングを行いさえすれば，本研究で提案した手法よりも容易にコピー除去を行うことが可能になる．

また，Shallow Binding を用いる方法[3]もよく知られている．この手法では，更新を行う際に，「その場所の元の値」を表すリストを配列につなげておき，書き換え前の配列はそのリストを指すようにする．そして，配列の要素を破壊代入で更新する（**図 8**）．このようにすれば更新は定数時間で行うことができる．

ここでは配列の古い値の方をリストに保存しているが，更新する値の方をリストにつなぎ，配列の値は破壊代入しないという方法もありうる．しかし，プログラムでは古い配列を参照するよりは，新しい配列を参照することの方が多いので，一般には古い方の値をリストに保存する方法がとられる．

Shallow Binding を用いる方法は解析が不要なため実現が容易であるが，本質的にはリストであり，配列の持つすべての要素を定数時間でアクセスできるという性質は失われる．たとえば，更新が大量に行われる場合は，更新前の配列を参照するのに，リストをたどる手間として，更新回数と同じだけのオーダーの時間ががかかる．

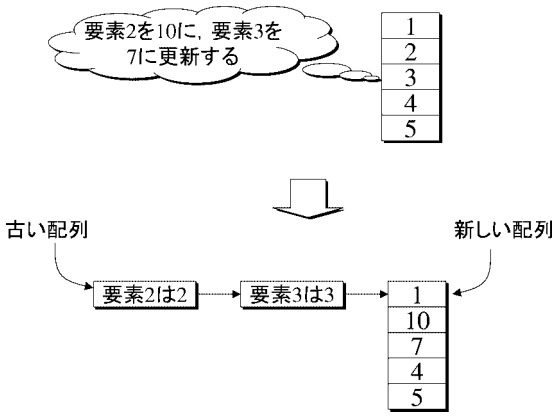

図 8 Shallow Binding
Fig. 8 Shallow Binding.

6.2 データ並列性の抽出

単一代入変数を持つ言語におけるデータ並列性の抽出に関しては，本研究で行ったようなアプローチの研究はあまりなされていない．

データフロー関数型言語 SISAL[6]は，プログラマが明示的に for-all 構文を用いることで，データ並列処理を可能にしている．しかし，プログラマはデータ依存を意識し，各繰返しを独立に実行しても安全であるということを確認してプログラムを書く必要がある．

文献 12) では，関数型言語において Partition と Combine という操作を導入し，コピー除去を行った後，更新操作を並列に行うという手法を提案している．この方法でも，プログラマが並列実行の可能性について意識しなければならない．

論理型言語に関しては，文献 9) において，リストデータ構造を用いたデータ並列処理について述べられている．リストを用いるという点で独創的だが，リストをたどる処理は逐次にしか行えないため，本論文で述べたような，1 回の繰返しでの処理が小さい場合は，データ並列性を抽出できない．

また，文献 4) では，Prolog のデータ並列実装について提案されている．ここではデータ並列実行として，Recursion Parallelism と呼ばれる，再帰的なデータ構造についてデータ並列的に実行する手法と，Bounded Quantification と呼ばれる，述語をある値の範囲ですべて呼び出すという手法を提案している．

前者は，再帰的なデータ構造（典型的にはリスト）上で並列に実行する手法で，先の文献 9) の考え方に近い．この手法の実行形態を**図 9** に示す．

再帰的なデータ構造に対して再帰的な繰返し処理を行う際，ヘッドユニフィケーションを前もって数回分行っておき，そのあとでそれぞれの再帰のボディー部を並列に実行するというモデルである．このような並列実行が可能なのはボディ部のゴールにおける共有変数への束縛が分岐によらない場合であるとしている．これは，プログラムの解析により判定する．

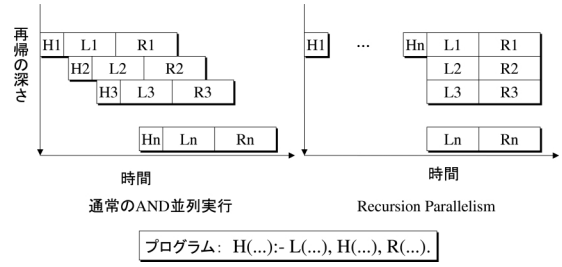

図 9 Recursion Parallelism
Fig. 9 Recursion Parallelism.

Prolog のセマンティクスを保存した並列化手法としては興味深いが，文献 9) における研究と同様，1 回の繰返しでの処理が小さい場合は，データ並列性を抽出できない．

後者の Bounded Quantification は手続き型言語で言うところの for-all に近い．たとえば

`all(1 ≤ I ≤ 100, p(I))`

で，`p(1),P(2),...,p(100)` を呼び出すことを表す．並列実行のためというよりは，記述力を改善するために導入された．SISAL などと同様に，プログラマが並列実行性について意識しなければならないほか，この論文では配列更新時のコピーの問題を解決していない．

Committed-Choice 型言語においては，文献 17), 19) で，GHC（KL1）によるデータ並列計算について議論されている．この手法は文献 24) の手法によりコピー除去を行い，文献 12) と同様の手法で，並列に更新を行うものである．したがって，プログラマは並列性を制御できるが，並列性を意識しなければ並列に動作させることができないという問題がある．

また，文献 22) では，KLIC に分散配置配列と呼ばれる配列を導入し，プログラマが明示的に分散配置した配列に対し，手続き型言語で並列処理を行う手法について述べられている．この手法ではより高い効率を得ることができるが，やはりプログラマが並列性について意識する必要がある．

7. 結　　論

本研究では，Committed-Choice 型言語 Fleng において，配列処理の最適化を行う手法の提案，実装，評価を行った．本研究における配列処理の最適化は，配列更新時のコピー除去と，データ並列性の抽出からなる．評価の結果，コピー除去については数倍〜数百倍，データ並列性の抽出については数倍程度のきわめて高い効率向上が得られることを示した．

本手法は，他の単一代入変数を用いる並列言語にも適用可能であると考えられる．

今後の課題としては，

- できるだけ多くの場合でコピー除去を行うため，より精度の高いデータフロー解析を行うこと
- より多くの場合でデータ並列性を抽出するため，より高度な繰返しをまたぐ配列の依存解析を行うこと

などがあげられる．

参 考 文 献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers – Principle, Techniques, and Tools*, Addison-Wesley (1986). 原田賢一（訳）：コンパイラ原理・技法・ツール I, II, サイエンス社 (1990).
2) Araki, T., Hidaka, Y., Nakada, H., Koike, H. and Tanaka, H.: System Integration of the Parallel Inference Engine PIE64, *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, pp.64–76 (1994).
3) Baker, H.G.: Shallow Binding Makes Functional Arrays Fast, *ACM Sigplan Notices*, Vol.26, No.8, pp.145–147 (1991).
4) Bevemyr, J.: Data-parallel Implementaion of Prolog, PhD Thesis, Uppsala University, Computing Science Department (1996).
5) Bloss, A.: Efficient Implementation of Functional Aggregates, *Functional Programming Languages and Computer Architecture*, pp.26–38, ACM (1989).
6) Cann, D.: Retire Fortran? A debate rekindled, *Comm. ACM*, Vol.35, No.8, pp.81–89 (1992).
7) Chikayama, T.: Operating System PIMOS and Kernel Language KL1, *Proc. International Conference on Fifth Generation Computer Systems 1992*, pp.73–88 (1992).
8) Chikayama, T. and Kimura, Y.: Multiple Reference Management in Flat GHC, *4th International Conference on Logic Programming*, pp.276–293 (1987).
9) Hermenegildo, M.V. and Carro, M.: Relating Data-Parallelism and (And-) Parallelism in Logic Programs, *Euro-Par '95*, LNCS, Vol.966, pp.27–41, Springer-Verlag (1995).
10) Hudak, P. and Bloss, A.: The Aggregate Update Problem in Functional Programming Systems, *POPL '85*, pp.300–314, ACM (1985).
11) Nilsson, M. and Tanaka, H.: Fleng Prolog – the language which turns supercomputers into Prolog machines, *Logic Programming '86*, LNCS, Vol.264, pp.170–179, Springer-Verlag (1989).
12) Sastry, A.V.S. and Clinger, W.: Parallel Destructive Updating in Strict Functional Languages, *Lisp and Functional Programming*, pp.263–272, ACM (1994).
13) Sastry, A.V.S., Clinger, W. and Ariola, Z.: Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates, *Functional Programming Languages and Computer Architecture*, pp.266–275, ACM (1993).
14) Sastry, A.V.S., Sundararajan, R. and Tick, E.: A Compile-Time Memory-Reuse Scheme for Concurrent Logic Programs, Technical Report

CIS-TR-91-24a, Department of Computer and Information Science, University of OREGON (1992).
15) Shapiro, E. (Ed.): *Concurrent Prolog*, The MIT Press (1987).
16) Ueda, K.: Guarded Horn Clauses, Technical Report, TR-103, ICOT (1985).
17) Ueda, K.: Moded Flat GHC for Data-Parallel Programming, *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, pp.27–35 (1994).
18) Zima, H. and Chapman, B.: *Supercompilers for Parallel and Vector Computers*, ACM Press (1991). 村岡洋一（訳）：スーパコンパイラ，オーム社 (1995).
19) 坂本幸司，土山了士，上田和紀：KLIC 並列処理系における配列演算の最適化，並列処理シンポジウム JSPP '99, p.208 (1999). ポスターセッション.
20) 荒木拓也，田中英彦：Committed-Choice 型言語 Fleng における静的粒度最適化，情報処理学会論文誌，Vol.38, No.9, pp.1771–1780 (1997).
21) 荒木拓也，田中英彦：Committed-Choice 型言語 Fleng における粒度制御法の評価，情報処理学会論文誌：プログラミング，Vol.40, No.SIG1 (PRO2), pp.23–31 (1999).
22) 藤瀬哲朗，近山 隆，上田和紀，稲村 雄，関田大吾：KLIC へのデータ並列処理機能の導入について，第 58 回情報処理学会全国大会論文集，Vol.1, No.4N-2, pp.393–394 (1999).
23) 日下部茂，岡崎芳希，谷口倫一郎，雨宮真人：データフロー関数型言語の並列化コンパイラにおける配列の静的コピー除去，並列処理シンポジウム JSPP '95, pp.161–168 (1995).
24) 上田和紀：並行論理プログラムの参照数解析，情報処理学会プログラミング研究会（SWoPP '98）(1998).

(平成 11 年 9 月 13 日受付)
(平成 12 年 1 月 6 日採録)

荒木 拓也（正会員）

1971 年生．1994 年東京大学工学部電気工学科卒業．1999 年同大学院情報工学専攻博士課程修了．工学博士．同年，NEC 入社．現在 C&C メディア研究所勤務．プログラミング言語の実装，特に並列論理型言語，並列化コンパイラ，命令レベル並列性の抽出等に興味を持つ．

坂井 修一（正会員）

1958 年生．1981 年東京大学理学部情報科学科卒業．1986 年同大学院情報工学専門課程修了．工学博士．同年，電子技術総合研究所入所．1991 年 4 月より 1 年間米国 MIT 招聘研究員．1993 年 3 月より 1996 年 2 月まで RWC 超並列アーキテクチャ研究室室長．1996 年 10 月より 1998 年 3 月まで筑波大学助教授（電子・情報工学系）．1998 年 4 月より東京大学助教授（工学系研究科）．計算機システム一般，特にアーキテクチャ，並列処理，スケジューリング問題，マルチメディア等の研究に従事．情報処理学会論文賞（1990 年度），日本 IBM 科学賞（1991 年），市村学術賞（1995 年），ICCD Outstanding Paper Award（1995 年）等受賞．IEEE，ACM，電子情報通信学会各会員．

田中 英彦（正会員）

1943 年生．1965 年東京大学工学部電子工学科卒業．1970 年同大学院博士課程修了．工学博士．同年東京大学工学部講師．1971 年助教授，1978〜1979 年ニューヨーク市立大学客員教授，1987 年教授現在に至る．計算機アーキテクチャ，並列処理，人工知能，自然言語処理，分散処理，CAD 等に興味を持っている．「非ノイマンコンピュータ」，「情報通信システム」著，「計算機アーキテクチャ」，「VLSI コンピュータ I，II」，「ソフトウエア指向アーキテクチャ」共著，New Generation Computing 編集長．電子情報通信学会，人工知能学会，ソフトウェア科学会，IEEE，ACM 各会員．