プログラミング言語　12−2
（１９８７・７・２４）

ベクトル計算機による論理型言語プログラムの高速実行をめざして
― 各種ORベクトル実行方式の実現と性能 ―

金田　泰
日立製作所中央研究所

ベクトル計算機(スーパ・コンピュータ)による論理型言語プログラムの高速実行方式の研究の一部として，ORベクトル実行方式 (OR並列実行方式)の研究をおこなっている．ORベクトル実行方式にはマスク演算方式，インデクス方式，圧縮方式の3種類があるが，これらをそれぞれ詳細化し，Nクウィーン問題をとくプログラムに適用した．すなわち，原始プログラムを論理型言語による中間語にプログラム変換したのちFortranとPascalとにハンド・コンパイルし，日立S-810による実行性能を測定した．その結果，どの方式でも8クウィーン全解探索の時間は20ms弱，大型汎用計算機の8~9倍の性能であり，ベクトル計算機における各実行方式の有望性が確認できた．

Towards High-Speed Execution of Logic Programming Language Programs
― Implementation and performance of or-vector execution methods ―

Yasusi Kanada
Central Research Laboratory, Hitachi Ltd.

As part of research on high-speed execution method of logic programming language programs on vector processors (supercomputers), we are studying "or-vector execution method" (a kind of or-parallel execution method). There are three kinds of or-vector execution methods, i.e., masked operation method, indexing method, and compressing method. We have refined these methods and have applied them to the program solving N queens problem. First, we converted the source program into the intermediate language program in some logic programming language. Second, we compiled it into Fortran and Pascal by hand. Then, we measured the performance of the object program on Hitachi S-810. We found that the execution time of eight queens exaustive search is less than 20 ms in any of the three methods, that is eight or nine times faster than that on large general purpose computers. These results show the effectiveness of these execution methods on vector processors.

1. はじめに

論理型言語プログラムを高速実行する方法として，Cray 2や日立S-810のようなパイプライン型ベクトル計算機(以後単にベクトル計算機とよぶ)をつかう方法が有望だとかんがえられる[1, 2]. しかし，これまでのところこのアプローチは構想段階であり，実用化のためにはおおくの課題を解決しなければならない.

まず，これまでの研究をまとめ，そこで解決すべき課題についてのべる.

金田[3]ではベクトル計算機による探索問題の実行方法であるORベクトル実行方式および並列バックトラック方式をしめした. そして，これをNクウィーン問題のプログラムに適用してFortranでプログラムを記述し，N≧10のとき，汎用計算機むきのプログラムをそれで実行した場合にくらべて約4.5倍の性能をえている.*1 この性能比は，Fortranコンパイラの性能向上によってその後約9倍に達している(表1参照).*2

金田[1]では，ベクトル実行に適したOR並列性やAND並列性とはどのようなものか，また，どのような方式でベクトル計算機で実行すればよいかといった問題について論じた. その中心となるのはORベクトル実行方式および並列バックトラック方式を論理型言語プログラムに適用することである. しかし，(1)とくにリスト処理プログラムに関しては，具体的にその方式を適用できるところまで詳細化されていなかった. また，(2)論理型言語とそのベクトル計算機むき目的プログラムとの意味上のギャップがうめられていなかったため，自動変換(コンパイル)はできなかった.

これまでの研究で，(1)に関してはリスト処理をふくむプログラムの実行方式をほぼ確立した. また，(2)に関してはモード宣言のようなある程度のユーザ・オプションの存在のもとでのORベクトル実行方式のための自動プログラム変換実現のみとおしをえた. これらの技術にもとづいてNクウィーン問題の解をもとめる論理型言語プログラムをマスク演算方式，インデクス方式*3，圧縮方式[1]のそれぞれにしたがってハンド・コンパイルし，S-810において性能を比較したので，それについて報告する.

第2章ではORベクトル実行方式の概要をしめす. 第3章および第4章ではそれをより詳細にしめす. まず第3章では決定性述語の実行方式を，マスク演算方式，インデクス方式，圧縮方式のそれぞれについてのべる. つぎに第4章で非決定性述語の実行方式についてのべる. 第5章では，ハンド・コンパイルしたプログラムのS-810による測定結果をしめし，各方式を比較検討する. 最後に第6章で結論をのべる.

2. ORベクトル実行方式の概要

この章ではベクトル計算機による論理型言語のOR並列実行方式(ORベクトル実行方式とよぶ)の概要について説明する.

金田[1]でのべたように，ORベクトル実行方式では，非決定性述語の実行によって生成される複数の部分解を配列に蓄積し，その配列にベクトル演算を適用する. もとの論理型言語のプログラムを直接このように解釈・実行することは困難であるから，コンパイラがプログラム変換をおこなってベクトル命令をふくむ目的プログラムを生成し，それを実行する. 通常のノイマン型計算機とベクトル計算機とのギャップをうめるこの種の変換をFortranにおいてはベクトル化とよんでいる. したがって，論理型言語の場合にもベクトル化とよぶことにする.*4

ベクトル化は，目的プログラムの生成とはきりはなして原始プログラムから原始プログラムへの変換としておこなうこともできる. コンパイラの処理も，図1のようにベクトル化とコード生成という2段階にわけるのが適当だとかんがえられる. ベクトル化は非決定性プログラムの決定性プログラムへの変換をともなっているから，上田[4, 5]やCodish and Shapiro[6]の変換ににている. また，その適用範囲も上田の方法ににている.

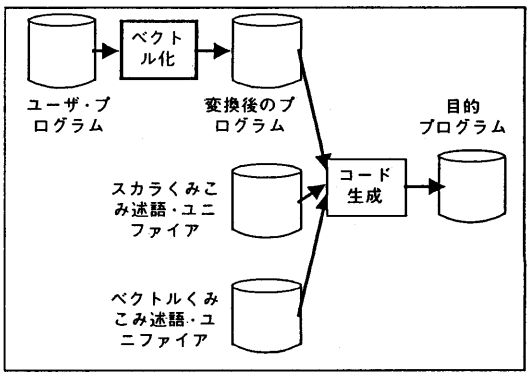

図1 パイプライン型ベクトル計算機における論理型言語のコンパイル方式

以下，ORベクトル実行方式によるプログラムの実行の概要をしめす. 例としてNクウィーン問題のプログラム(中島[7]による)における述語putをつかう(Nクウィーンのプログラム全体を図2にしめす). かんたんのため，4クウィーンの場合をかんがえる.

```
queen(Q) :- put([1, 2, 3, 4, 5, 6, 7, 8], [], Q).

put([], Q, Q).
put(Qs, B, Q) :-
    select(Qs, Q1, R), not_take(B, Q1), put(R, [Q1 | B], Q).

select([A | L], A, L).
select([A | L], X, [A | L1]) :- select(L, X, L1).

not_take(R, Q) :-
    Qa is Q+1, Qs is Q-1, not_take1(R, Qa, Qs).

not_take1([], Qa, Qs).
not_take1([Q | R], Qa, Qs) :-
    Q =\= Qa, Q =\= Qs,
    Qaa is Qa+1, Qss is Qs-1, not_take1(Q, Qaa, Qss).
```

図2 Nクウィーン問題をとく論理型言語プログラム(中島[7]による)

プログラムはつぎのとおりである(DEC-10 Prologの記法にしたがう).

```
put([], Q, Q).                                    ..... (2.1)
put(Qs, B, Q) :-
    select(Qs, Q1, R), not_take(B, Q1),
    put(R, [Q1 | B], Q).                          ..... (2.2)
```

*1 ただし，このプログラムはデータ構造として配列を使用している. したがって，リストを使用するPrologのプログラムとはきちんと対応していない.

*2 金田[3]では汎用機の20倍以上の性能がえられると予測したが，実際は9倍にとどまった.

*3 金田[1]ではリスト・ベクトル方式とよんでいる.

*4 ORベクトル実行の場合には，とくにこの変換をORベクトル化とよぶ. ANDベクトル実行の場合には，ANDベクトル化とよぶ.

?– put([1, 2, 3, 4], [], Q). ……(2.3)

述語putの第1引数はこれから配置すべきクウィーンのリストをあらわす．第2引数はそれらのクウィーンの配置まえのチェス・ボード(いいかえれば，すでに配置されたクウィーンのリスト)をあらわす．第3引数が4クウィーン問題の解である．

つぎに，ORベクトル実行方式における述語putの実行について図3をつかって説明する．論理型言語の実行において，ひとつの論理変数はことなる部分解ごとにことなる値をとる．ORベクトル実行方式では，これらの値をひとつの配列として出力する．金田[1]ではこの配列を部分解配列とよんだが，あいまいさをさけるためにこの報告では論理変数値配列とよぶ．図3の

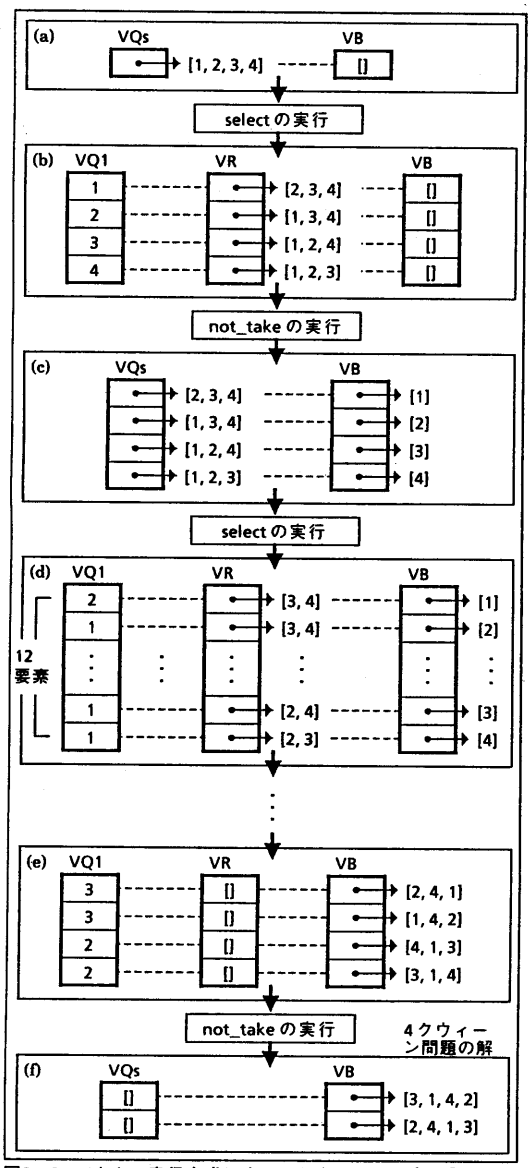

図3 ORベクトル実行方式による4クウィーン・プログラムの実行概要

VQ1, VRなどが論理変数値配列である．ここで，VQ1の第i要素とVRの第i要素とが対応している．

質問(2.3)に関しても，各引数をそれぞれ要素が1個の配列にする．以後，$a_1, a_2, ..., a_n$を要素とする配列を$\#(a_1, a_2, ..., a_n)$とあらわす(空配列は$\#()$とあらわす)．すると，ORベクトル実行方式では，上記のプログラムをつぎのようなインタフェースをもつプログラムに変換したものを実行することになる．

?– v_put(#([1, 2, 3, 4]), #([]), VQ).*1

通常のOR並列実行方式とはちがって，ORベクトル実行方式では非決定性述語における各部分解の生成を逐次的におこなう(第4章参照)．しかし，全解探索においては，非決定性述語から出力される配列の各要素に関する後続の計算を並列(パイプライン的)におこなうことができるので，実行の高速化をはかることができる．述語putにおいては，述語selectが非決定性であるために，出力される配列の要素数が図3(b), (d)のようにふえる．したがって，後続の述語not_take1およびputの再帰よびだしにおけるselectの実行が高速化される．ベクトル計算機ではCPU中での演算が加速されるだけではなく，主記憶とベクトル・レジスタとのスルー・プットがおおきいために主記憶アクセスも加速される．そのため，主記憶のアクセス頻度がたかいリスト処理においても高速化が期待できる．

述語selectの計算において，VQ1, VRだけではなく，ほかの配列についても上記のような要素間の対応をつけておく必要がある．したがって，Q1, Rとともに節(2.2)における以下の計算に使用する論理変数Bに関しても，図3(b), (d), (f)のようにもとの配列の同一要素の値を複数個ふくむ配列VBをつくって，これを使用する．このような操作を環境複写とよぶ．論理変数Qsの値はこれ以後使用しないしQの値は未定なので，これらを環境複写する必要はない．*2

述語not_take1のような決定性述語の実装法について第3章で説明する．また，述語selectのような非決定性述語の実装法すなわち解の増殖のさせかたについて第4章で説明する．

3. 決定性述語のORベクトル実行方式

この章では，決定的であることがわかっている述語のORベクトル実行方式についてのべる．

ORベクトル実行方式では，ことなる部分解に関する計算を並列(パイプライン的)におこなう．並列に計算している部分解のなかには，失敗(fail)するものもある．これ以後，失敗した部分解をふくむ配列要素を無効要素とよび，そうでない配列要素を有効要素とよぶ．逐次実行方式の場合には失敗がおこるとただちにバックトラックするが，ORベクトル実行方式においては有効要素が1個でも存在するかぎり計算を続行する．したがって，有効要素と無効要素とを判別して前者だけを演算対象とする必要がある．金田[1]でのべたように，ORベクトル実行における条件制御方式として3方式がかんがえられる．これらはベクトル計算機における条件制御の3方式[1]とそれぞれ対応している．これらの条件制御方式について図4をつかって説明する．

(1) マスク演算方式

論理変数の値をふくむ配列と同数の論理値を要素とする配列をつかう．この配列をマスク・ベクトルとよぶ．マスク・ベクトルの要素の値がfalseならば対応する配列要素がふくむ論理変数値は無効であり，そうでなければそれは有効である(図では無効要素にはハッチングしてある)．マスク・

ベクトルは複数の論理変数値配列に対して1個あればよい(図4(1)).

(2) インデクス方式(リスト・ベクトル方式)

論理変数の値をふくむ配列のすべての有効要素の添字からなる配列をつかう.*3 この配列をインデクス・ベクトルとよぶ. インデクス・ベクトルがさす(添字をふくむ)要素は有効であり, それがささない要素は無効である. インデクス・ベクトルも複数の論理変数値配列に対して1個あればよい(図4(2)).

(3) 圧縮方式

つねに有効要素だけをふくむように各配列から無効要素をとりのぞき, 圧縮された配列をつかう(図4(3)).

これらの方式の得失については金田[1]でのべているので, ここでは省略する.

以下これらの方式についてNクウィーンにおける述語not_take1を例にとって説明する.

3.1. マスク演算方式

Nクウィーンのプログラムにおける述語not_take1は, 述語selectで選択されたクウィーンが, すでにチェス・ボード上に配置されたクウィーンによってとられるかどうかを検査する. その結果, とられない部分解だけがいきのこる. 述語not_take1の原始プログラムはつぎのとおりである.

```
not_take1([], _, _).                                    ..... (3.1)
not_take1([Q | R], Qa, Qs) :-
    Q =\= Qa, Q =\= Qs,
    Qaa is Qa + 1, Qss is Qs - 1,
    not_take1(R, Qaa, Qss).                             ..... (3.2)
```

節(3.1)は第1引数が空リスト[]のときだけ成功する. 節(3.2)は第1引数が空でないリストのときだけ成功する. したがって, 述語not_take1は第1引数が具体化されていない(uninstantiated)変数でないかぎり決定的(すなわち, 2個の節がともに成功することはない)である.

述語not_take1をマスク演算方式にしたがってベクトル化した結果のプログラムを論理型言語の述語(v_not_take1)としてしめす.

```
1. v_not_take1(_, _, _, MI-MI) :-
2.     v_finished(MI), !.                               ..... (3.3)
3. v_not_take1(B, Qa, Qs, MI-MO) :-
4.     v_null(B, MI, MO1),
5.     v_carcdr(Q, R, B, M1, M2),
6.     v_ne(Q, Qa, M2, M3), v_ne(Q, Qs, M3, M4),
7.     v_add1(Qa, Qaa, M4), v_sub1(Qs, Qss, M4),
8.     v_not_take1(R, Qaa, Qss, M4-MO2),
9.     v_end_or(MO1, MO2, MO).                          ..... (3.4)
```

述語v_not_take1の第1～第3引数はnot_take1の第1～第3引数とそれぞれ対応している. 第4引数はMI-MOというかたちをしているが, ここでMIが入力マスク・ベクトルであり, MOが出力マスク・ベクトルである. 第1～第3引数である配列

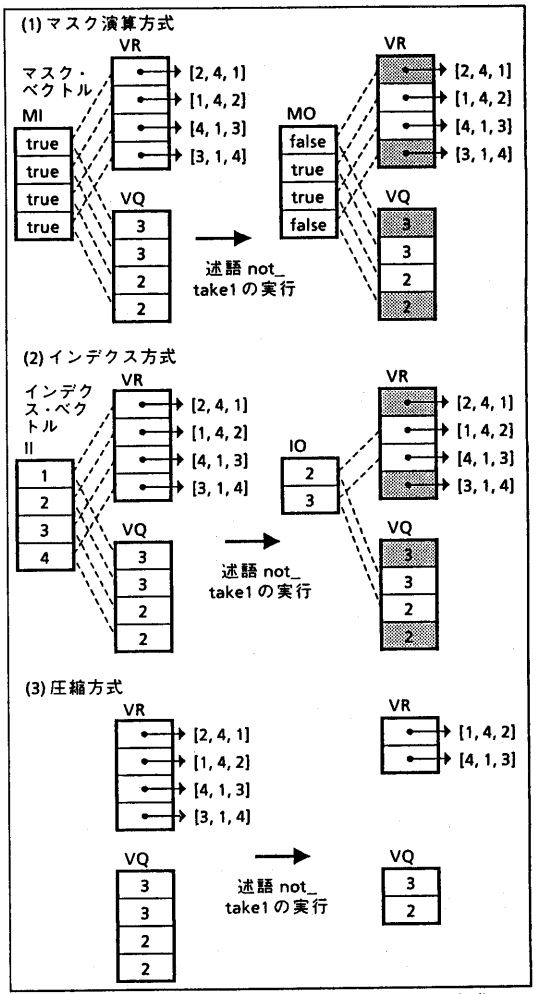

図4 ORベクトル実行方式における条件制御の3方式
(図2(e)~(f)の部分)

の第i要素はMI(i)がfalseならv_not_take1の実行前から無効である. not_take1のベクトル化においては, すべての引数が入力モードでつかわれることを利用している. この点は3.2~3.3節においても同様である.*1

上記のプログラムはつぎの4部分にわけることができる.*2

(1) 停止条件判定部

節(3.3)はベクトル化前のプログラムには対応する部分がない. この節は, 後述する理由により, 停止条件判定部とよぶ.

(2) 第1節対応部

行4はベクトル化前のプログラムの第1節すなわち節(3.1)に対応しているので, 第1節対応部とよぶ.

---

*1 代案として, つぎのようなインタフェースもかんがえられるが, この報告ではあつかわない.
?- v_put(#(#r([1, 2, 3, 4], [], VQ))).
ここで、$\#r(e_1, e_2, ..., e_n)$は, $e_1, e_2, ..., e_n$を要素とするレコードをあらわす.

*3 効率上は添字をつかうより配列の先頭から有効な要素までの変位をつかうほうがよいので実際にはそうするが, この報告ではわかりやすさのために添字そのものをふくむことにする.

*2 環境複写はOR並列化によるオーバヘッドであり, これをモード解析やデータフロー解析などによってへらすことが, ORベクトル実行方式における重要なコンパイル技術となるはずである(なお, 環境複写のために, 本来はわたす必要のない論理変数Bの値を述語selectにわたさなければならないので, ORベクトル実行方式においては, 述語間のインタフェースを巧妙にしないかぎりは述語をコンパイル単位とすることができない).

(3) 第2節対応部

行5~8はもとのプログラムの第2節すなわち節(3.2)に対応しているので，第2節対応部とよぶ．

(4) マスク合成部

行9は第1節対応部と第2節対応部とから出力されるマスク・ベクトルを合成するので，マスク合成部とよぶ．*3

これらの部分のうち(4)がマスク演算方式に特有の部分である．これらの部分について図5を使用して順に説明する．

停止条件判定部(節(3.3))は，v_not_take1の再帰よびだしを停止させるために必要である．v_finished(MI)は入力されたマスク・ベクトルMIの要素がすべてfalseのとき成功し，そうでないとき失敗する(図5(a))．v_finished(MI)が成功すればv_not_take1の実行は終了し，要素の値がすべてfalseのマスク・ベクトルが出力される．失敗すれば節(3.4)が実行される．もし節(3.3)が存在しないとすれば，すべての配列要素が無効になっても再帰よびだしがくりかえされ，v_not_take1の実行は終了しない．

第1節対応部(行4)では，配列Bの各要素が空リスト[]かどうかを判定し，その結果をMO1に反映させる(図5(b))．すなわち，B, MIの要素数をnとすると，$1 \leqq i \leqq n$なるiについて，MI(i)=false，またはMI(i)=trueかつB(i)≠[]ならばMO(i)の値をfalseにする．また，それ以外すなわちMI(i)=trueかつB(i)=[]ならばMO1(i)の値をtrueにする．

第2節対応部では，つぎのように計算する．行5は節(3.2)の頭部の[Q|R]に対応している．配列Bの各有効要素を頭部(car)と尾部(cdr)とに分解し，前者からなる配列をQとし，後者からなる配列をRとする(図5(c))．配列Bがリスト以外の要素をふくむときはその要素B(i)に対応するマスク・ベクトルの要素M2(i)はfalseにする．*4

行6の2個の述語よびだしは，節(3.2)本体のくみこみ述語よびだしQ =\= Qaおよび Q =\= Qsに対応している．すなわち，配列Qと配列Qaの有効要素どうしの比較および配列Qと配列Qsの有効要素どうしの比較をおこなう．各配列の要素の有効性は述語v_neの第3引数であるマスク・ベクトルによってしめされる．いずれも，述語v_neの第4引数であるマスク・ベクトルの要素のうち，一致しない要素に対応するものをtrueにし，一致した要素に対応するものをfalseにする(図5(d)~(e))．

行7の2個の述語よびだしは，節(3.2)のくみこみ述語よびだしQaa is Qa+1およびQss is Qs−1に対応している．すなわち，配列Qaの各有効要素に1をたしたものを配列Qaaの各有効要素とユニファイし，配列Qsの各有効要素から1をひいたものを配列Qssの各要素とユニファイする(図5(f)~(g))．無効要素の値は適当にきめればよい．図ではその値を*でしめしている．*5

行8では述語v_not_take1を再帰よびだししている．その内部処理の説明は省略する．再帰よびだしの結果マスク・ベクトルMO2が出力される(図5(h))．

マスク合成部では，第1節対応部から出力されたマスク・ベクトルMO1と，第2節対応部から出力されたマスク・ベクトルMO2とから，述語v_not_take1が出力すべきマスク・ベクトルMOの値を合成する(図5(i))．すなわち，MO1(i)=falseかつMO2(i)=falseのときだけMO(i)=falseとし，それ以外のときはMO(i)=trueとする．これは，第1節対応部と第2節対応部のいずれでも無効とされた要素だけを無効とする(実行が失敗したとみなす)ということを意味する．

3.2. インデクス方式

述語not_take1をインデクス方式にしたがってベクトル化した結果のプログラムを論理型言語の述語としてしめす．

```
1. x_not_take1(__, __, __, II-#()) :-
2.       x_finished(II), !.                          ..... (3.5)
3. x_not_take1(B, Qa, Qs, II-IO) :-
4.       x_null(B, II, IO-IO1),
```

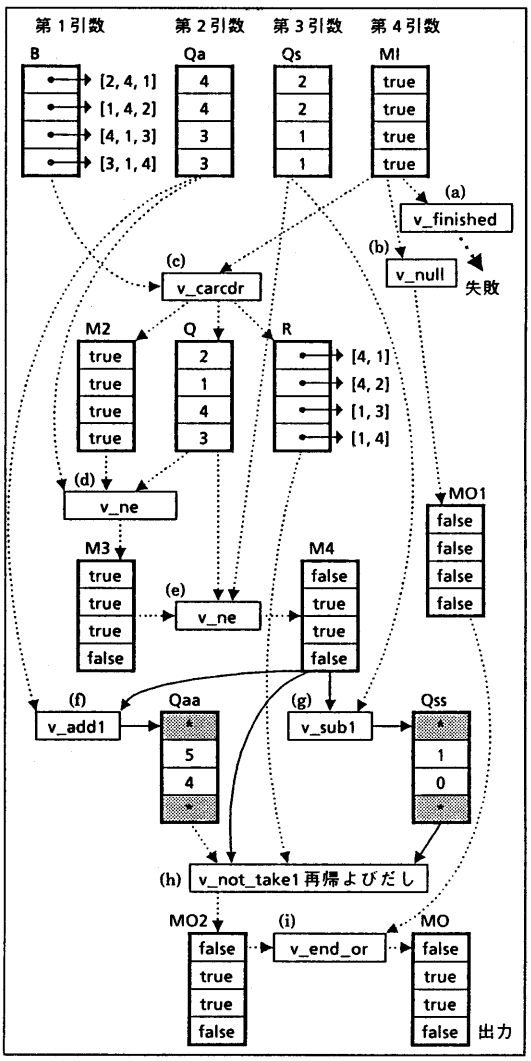

図5 マスク演算方式による述語not_take1の実行
(図4(a)の実行の一部)

*1 述語v_not_take1の各節の本体でよびだされるv_not_take1以外のすべての述語は，ORベクトル実行方式の論理型言語処理系においてはくみこみ述語にするべき汎用の述語である(図1においては「ベクトルくみこみ述語・ユニファイア」とよんでいる)．

*2 ただし，節(3.4)の頭部はいずれにもふくめていない．

*3 マスク合成部の機能はPrologの演算子';'(or)の機能に対応している．

*4 この演算は，S-810のようなベクトル計算機においては，ベクトル比較命令(VCEQ命令またはVCNE命令)とリスト・ベクトルのロード命令(VLX命令など)をくみあわせることによって容易に実現できる．

*5 でたらめの値がはいっているとガベジ・コレクションにさしつかえるなら，さしつかえのない値をいれる．

5. x_carcdr(Q, R, B, II, I2),
6. x_ne(Q, Qa, I2, I3), x_ne(Q, Qs, I3, I4),
7. x_add1(Qa, Qaa, I4), x_sub1(Qs, Qss, I4),
8. x_not_take1(R, Qaa, Qss, I4-IO1). ……(3.6)

述語 x_not_take1 の第 1～第 3 引数は not_take1 の第 1～第 3 引数とそれぞれ対応している．第 4 引数は II-IO というかたちをしているが，ここで II が入力インデクス・ベクトルであり，IO が出力インデクス・ベクトルである．第 1～第 3 引数の第 i 要素は i が II の要素としてふくまれていなければ x_not_take1 の実行前から無効である．*1

上記のプログラムはつぎの 3 部分にわけることができる．*2

(1) 停止条件判定部

節 (3.5) はマスク演算方式における節 (3.3) と同様に，x_not_take1 の再帰よびだしを停止させるためにもうけている．したがって停止条件判定部とよぶ．

(2) 第 1 節対応部

行 4 はベクトル化前のプログラムの第 1 節すなわち節 (3.1) に対応しているので，第 1 節対応部とよぶ．

(3) 第 2 節対応部

行 5~8 はもとのプログラムの第 2 節すなわち節 (3.2) に対応しているので，第 2 節対応部とよぶ．

このプログラムにはインデクス方式に特有の部分はないが，とくに (2) における処理方法に特徴がある．これらの部分について図6を使用して順に説明する．

停止条件判定部 (節 (3.5)) においては，x_finished(II) は入力されたインデクス・ベクトル II が空配列すなわち要素が 0 個の配列であるとき成功し，そうでないとき失敗する (図6(a))．x_finished(II) が成功すれば x_not_take1 の実行は終了し，空のインデクス・ベクトルが出力される．失敗すれば節 (3.6) が実行される．もし節 (3.5) が存在しないとすれば，インデクス・ベクトルが空になっても再帰よびだしがくりかえされ，x_not_take1 の実行は終了しない．

第 1 節対応部 (行 4) では，配列 B の各要素が [] かどうかを判定し，その結果をインデクス・ベクトル IO に反映させる (図6(b))．すなわち，インデクス・ベクトル II の要素数を n とすると，$1 \leqq i \leqq n$ なる i について，B(II(i))=[] ならば II(i) の値を IO の要素とし，B(II(i))≠[] ならばそれを IO の要素とはしない．x_null の実行前には IO は空配列である．配列 IO1 は配列 IO の空領域をあらわす．IO1 は空配列とみなされる．述語 not_take1 は決定的なので，x_not_take1 の再帰よびだしで配列 IO に要素をつめていっても，その要素数は配列 B, Qa, Qs の要素数をこえることはない．したがって，あらかじめそれらと同量の領域を確保しておけば，配列を拡張することなく要素を追加していくことができる．*3

第 2 節対応部では，つぎのように計算する．行 5 は節 (3.2) の頭部の [Q|R] に対応している．インデクス・ベクトル II からさされる配列 B の各要素を頭部と尾部とに分解し，結果を配列 Q と R とにいれる (図6(c))．配列 B がリスト以外の要素 B(II(i)) をふくむときは，II(i) の値はインデクス・ベクトル I2 の要素としない．配列 Q および R の要素数は配列 B のそれと同数で，要素に対応関係があるようにする．*0

行 9 の 2 個の述語よびだしは，節 (3.2) 本体のくみこみ述語よびだし Q =\= Qa および Q =\= Qs に対応している．すなわち，配列 Q と配列 Qa の有効要素どうしの比較および配列 Q と配列 Qs の有効要素どうしの比較をおこなう．各配列の有効性は述語 v_ne の第 3 引数であるインデクス・ベクトルによってしめされる．いずれも，述語 v_ne の第 4 引数であるインデクス・ベクトルには，一致しない要素に対応する第 3 引数のインデクス・ベクトル要素だけをふくめる (図6(d)~(e))．

行 7 の 2 個の述語よびだしは，節 (3.2) 本体のくみこみ述語よびだし Qaa is Qa+1 および Qss is Qs−1 に対応している (図6(f)~(g))．

行 8 では述語 v_not_take1 を再帰よびだししている．その結果は IO1 にインデクス・ベクトルとしてえられる (図7(h))．IO1 は IO の空領域であるから，第 1 節対応部でえられた要素のあとにこれらの要素がつづけて格納されることになる．

3.3. 圧縮方式

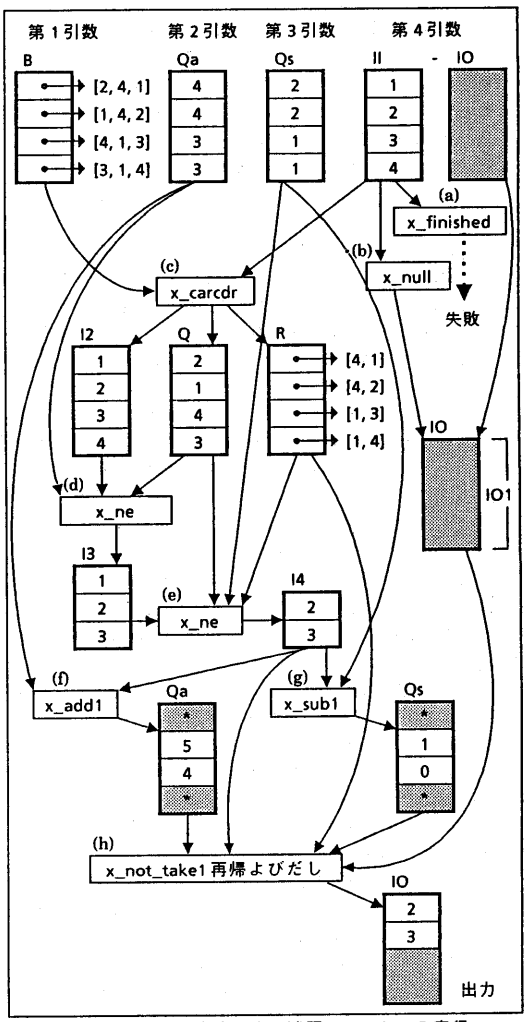

図6 インデクス方式による述語 not_take1 の実行
(図4(b) の実行の一部)

*1 述語 x_not_take1 の各節の本体でよびだされる x_not_take1 以外のすべての述語は，くみこみ述語にするべき述語である．

*2 ただし，節 (3.6) の頭部はいずれにもふくめていない．

*3 B(II(i)) の値をベクトル・レジスタにロードするには，リスト・ベクトルのロード命令をつかえばよい．

圧縮方式では配列要素が無効になるたびにつめあわせをおこなう．しかし，命令レベルで圧縮方式をサポートする命令は現在のベクトル計算機にはない．また，あらたなハードウェアをつくるにしても，通常はつめあわせるべき配列は複数個あるから命令レベルで圧縮方式をサポートするのは困難である．したがって，折中的な方法をとる．すなわち，いったんはマスク・ベクトルやインデクス・ベクトルの値をレジスタ上につくったのち，それをつかって配列の圧縮をおこなう．

述語 not_take1 を圧縮方式にしたがってベクトル化した結果のプログラムを論理型言語の述語としてしめす．

```
1.  c_not_take1(QR, Qa, Qs, B-B, R-R, Q-Q) :-
2.        v_finished(QR), !.                          ..... (3.7)
3.  c_not_take1(QR, Qa, Qs, BI-BO, RI-RO, QI-QO) :-
4.        v_null(QR, __, M1),
5.        c_compress(BI, M1, BO-BOe),
6.        c_compress(RI, M1, RO-ROe),
7.        c_compress(QI, M1, QO-QOe),
8.        v_carcdr(Q, R, QR, __, M2),
9.        v_ne(Q, Qa, M2, M3), v_ne(Q, Qs, M3, M4),
10.       c_compress(R, M4, R1-#()),
11.       c_compress(Qa, M4, Qa1-#()),
12.       c_compress(Qs, M4, Qs1-#()),
13.       c_compress(BI, M4, BI1-#()),
14.       c_compress(RI, M4, RI1-#()),
15.       c_compress(QI, M4, QI1-#()),
16.       v_add1(Qa1, Qaa, __), v_sub1(Qs1, Qss, __),
17.       c_not_take(R1, Qaa, Qss, BI1-BOe,
                     RI1-ROe, QI1-QOe).             ..... (3.8)
```

述語 c_not_take1 の第 1～第 3 引数は not_take1 の第 1～第 3 引数とそれぞれ対応している．第 4 引数は述語 put における論理変数 B の値をふくむべき配列の対であり，BI-BO というかたちをしている．BI が c_not_take1 の実行前の R の値からなる配列であり，BO がその実行後の B の値からなる配列である．また，第 5～第 6 引数は述語 put における論理変数 R および Q の値をふくむべき配列の対であり，それぞれ第 4 引数と同様の構造をしている．*1

上記のプログラムはつぎの 5 部分にわけることができる．*2

(1) 停止条件判定部

節 (3.7) を停止条件判定部とよぶ．

(2) 第 1 節対応部

行 4 はベクトル化前のプログラムの第 1 節すなわち節 (3.1) に対応しているので，第 1 節対応部とよぶ．

(3) 出力配列圧縮部

行 5～7 はそれぞれ配列 B1, R1, Q1 の有効要素を，述語 c_not_take1 が出力する配列 BO, RO, QO につめる．c_not_take1 の実行前にはこれらの配列は空である．配列 BOe, ROe, QOe はこれらの配列の空領域をあらわす．これらの配列は空配列とみなされる．

(4) 第 2 節対応部

行 8～9, 16 はもとのプログラムのプログラムの第 2 節すなわち節 (3.5) に対応しているので，第 2 節対応部とよぶ．

(5) 引数配列圧縮部

行 10～15 は，述語 c_not_take1 の再帰よびだしにさきだって，その引数配列の圧縮をおこなうので，引数配列圧縮部とよぶ．

これらの部分について図7 を使用して順に説明する．

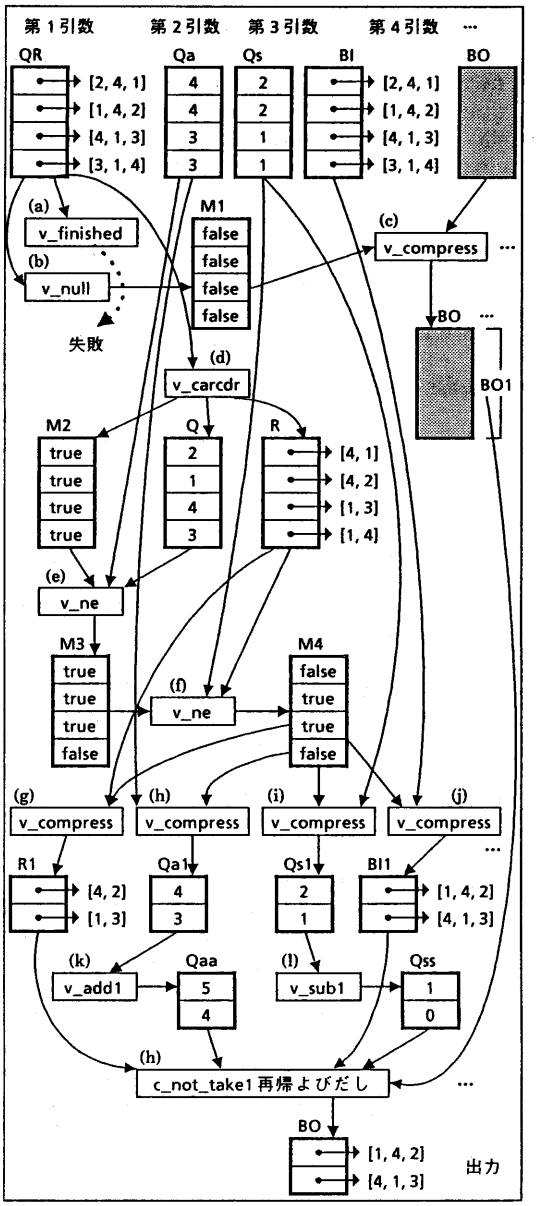

図7 圧縮方式による述語 not_take1 の実行
(図4(c) の実行の一部)

*0 B(j) ($1 \leq j \leq n$) が無効またはリスト以外のときは Q(j) および R(j) の値は適当でよい．

*1 述語 c_not_take1 の各部の本体でよびだされる c_not_take1 以外のすべての述語は，くみこみ述語にするべき述語である．'v_' を冠した述語はマスク演算方式による述語 v_not_take1 にあらわれたのと同一の述語である．

*2 ただし，節 (3.7) の頭部はいずれにもふくめていない．

*3 上記のプログラムでは B が空かどうかを判定しているが，QR, Qa, Qs, R, Q の要素数は常にひとしいので，どの配列で判定してもよい．

*4 自動ベクトル化においては，圧縮のタイミングの選択の最適化が重要な課題となるはずである．

停止条件判定部(節(3.7))は，c_not_take1の再帰よびだしを停止させるためにもうけている．c_finished(B)はc_not_take1に入力された配列が空のとき成功し，そうでないとき失敗する(図7(a)).*3 c_finished(B)が成功すればc_not_take1の実行は終了し，すべての引数について空配列が出力される．失敗すれば節(3.8)が実行される．

第1節対応部(行4)では，配列Bの各要素が空リスト[]かどうかを判定し，その結果をマスク・ベクトルM1に反映させる(図7(b)). すなわち，BIの要素数をnとすると，$1 \leq i \leq n$なるiについて，B(i)≠[]ならばM1(i)の値をfalseにする．また，B(i)=[]ならばM1(i)の値をtrueにする．述語v_null は3.1節にあらわれたv_nullと同一のくみこみ述語である．v_nullのよびだしにおける第2引数には本来その実行開始前のマスク・ベクトルの値をあたえるが，この場合には具体化されていない変数'_'をわたしている．これはすべての要素が具体化されていない配列をわたすのと等価である．

出力配列圧縮部では，第1節対応部でえられたnot_take1の解を配列BO, RO, QOに追加する．これらの配列は最初は空である．インデクス方式におけるIOと同様に，ここでも述語not_take1が決定的であることを利用している．図7(c)においてはマスク・ベクトルIIの要素がすべてfalseなので，要素は出力されず，BO=BO1となる．図では省略しているがRO, Qoについても同様である.*4

第2節対応部では，つぎのように計算する．行8~9はマスク演算方式におけるのとほぼ同様である(図7(d)~(f)). 行10~15では配列R, Qa, Qs, BI, RI, QIを圧縮している(図7(g)~(j)). くみこみ述語の1回のよびだしごとに圧縮をおこなうのはかえってオーバヘッドをふやすので，リスト分解から比較までの演算がおわってから圧縮するようにしている．

行16の2個の述語よびだしは，節(3.2)本体のくみこみ述語よびだしQaa is Qa+1およびQss is Qs−1に対応している．行16では述語c_not_take1を再帰よびだししている．その結果，配列BOe, ROe, QOeに解が蓄積される．これらはそれぞれ配列BO, RO, QOの空領域であるから，第1節対応部でえられた解のあとにこれらの解が格納されることになる．

4. 非決定性述語のORベクトル実行方式

述語が非決定性のとき，あるいは決定性であるかどうかがわからないときには，この章でのべるように実行する．第3章のインデクス方式および圧縮方式においては配列の最大要素数が推定できるために，あらかじめわりあてた配列に要素を追加していくことができた．しかし，非決定性述語の場合には最大要素数がわからないために，拡張可能な配列を使用しないかぎりはこのような方式は適用できない．したがって，下記の方式では解がえられるたびに解からなる配列をリストにつないでいき，最後にそれらの配列をひとつの配列に併合する．この方式を蓄積方式とよぶ．

例としてはNクウィーン問題の解をもとめるプログラムにあらわれる述語selectを使用する．

```
select([A|L], A, L).                                  ……(4.1)
select([A|L], X, [A|L1]) :- select(L, X, L1).         ……(4.2)
```

述語selectは第1引数であるリストから1個の要素をえらんで第2引数とユニファイするとともに，のこりのクウィーンからなるリストを第3引数とユニファイする．したがって，述語よびだしselect([2, 4], Q1, R)においては，つぎのような2個の解がもとめられる．

Q1=2, R=[4].
Q1=4, R=[2].

述語selectを蓄積方式にしたがってベクトル化した結果のプログラムを論理型言語の述語v_selectとしてしめす．

```
1.  v_select(AL, X, Y, MI, BI-BO) :-
2.      v_select1(AL, X1L, Y1L, MI-ML),
3.      v_merge(X1L, X, ML), v_merge(Y1L, Y, ML),
4.      v_repeat(BI, BO, ML).                          ……(4.3)

5.  v_select1(__, [], [], MI-[]).
6.      v_finished(MI), !.                             ……(4.4)
7.  v_select1(AL, [A'|X1L], [L'|Y1L], MI-[M1'|ML]) :-
8.      v_carcdr(A', L', AL, MI, M1'),
9.      v_carcdr(A, L, AL, MI, M1),
10.     v_select1(L, X1L, L1L, M1-ML1),
11.     mapcar(v_cons(A), L1L, Y1L, ML1, ML).          ……(4.5)
```

蓄積方式も述語内部での条件制御のしかたによって，マスク演算蓄積方式，インデクス蓄積方式，圧縮蓄積方式の3種類にわけられるが，下記のプログラムはマスク演算蓄積方式によっている．述語v_selectの第1~第3引数はそれぞれselectの第1~第3引数に対応している．第4引数は入力マスク・ベクトルである．出力時には配列を圧縮して無効要素をとりのぞくので，出力マスク・ベクトルは引数にしていない(配列の複写・併合は必須なので，その際に圧縮をおこなうようにしている)．第5引数はBI-BOというかたちをしているが，BIは各出力配列と要素間の対応をつけるべき配列であり，BOは対応づけの結果えられる配列である.*1

上記のプログラムはつぎの5部分にわけることができる.*2

(1) 停止条件判定部
　節(4.3)を停止条件判定部とよぶ．
(2) 第1節対応部
　行8はベクトル化前のプログラムの第1節すなわち(4.1)に対応しているので，第1節対応部とよぶ．
(3) 第2節対応部
　行9~11はベクトル化前のプログラムの第2節すなわち(4.2)に対応しているので，第2節対応部とよぶ．第2章でものべたように，通常のOR並列実行とはちがって，第1節対応部と第2節対応部は原則としては逐次的に実行される.*1
(4) 配列併合部
　行3はv_selectによって出力される配列を要素とするリストの要素を併合してひとつの配列にするので，配列併合部とよぶ．
(5) 環境複写部
　行4は，配列BIの要素を配列X, Yと対応づけるために複写して，複数の同一要素をふくむ配列をつくるので，環境複写部とよぶ．

これらの部分のうち(4)と(5)とが蓄積方式に特有の部分である．これらの部分について図8を使用して順に説明する．

---

*1 述語v_selectおよびv_select1の各節の本体でよびだされるv_select1以外のすべての述語は，くみこみ述語にするべき述語である．

*2 ただし，行1~2, 5はいずれにもふくめていない．

停止条件判定部のはたらきは決定性述語の場合とおなじである(図8(a)). 第1節対応部すなわち行8では, 配列ALの各有効要素を頭部と尾部とに分解する(図8(b)).

第2節対応部では, つぎのように計算する. まず, 行9で配列ALの各有効要素を頭部と尾部とに分解する. この処理は第1節とまったくおなじであるから, 共通化(共通式削除)することができる. 図8および第5章で実行結果をしめすハンド・コンパイルされたプログラムにおいては共通化をおこなっている.

行10では述語v_selectを再帰よびだししている(図8(c)). 結果は第2~4引数に配列のリストとしてえられる. すなわち, 図にしめしたように, 第2引数X1Lにはえらばれたリスト要素の配列からなるリスト, 第3引数L1Lにはのこりのリスト要素の配列からなるリスト, 第4引数の後半すなわちMLにはマスク・ベクトルのリストがえられる. これらのリストの要素は対応している. X1LおよびL1Lの第2要素は, ML1の第2要素であるマスク・ベクトルの要素がfalseであるから無効である.

行11は節(4.2)の頭部における[A|L]に対応している. すなわち, リストL1Lの要素である配列の各要素に配列Aの各要素をconsしてリストY1Lをえる(図8(d)). ただし, MLによってしめされる無効要素に対してはこの操作をおこなわない. 述語v_select1の実行の最後に配列A'とリストX1Lとをconsして, 出力すべき第2引数の値をえる. また, 配列L'とリストY1Lと

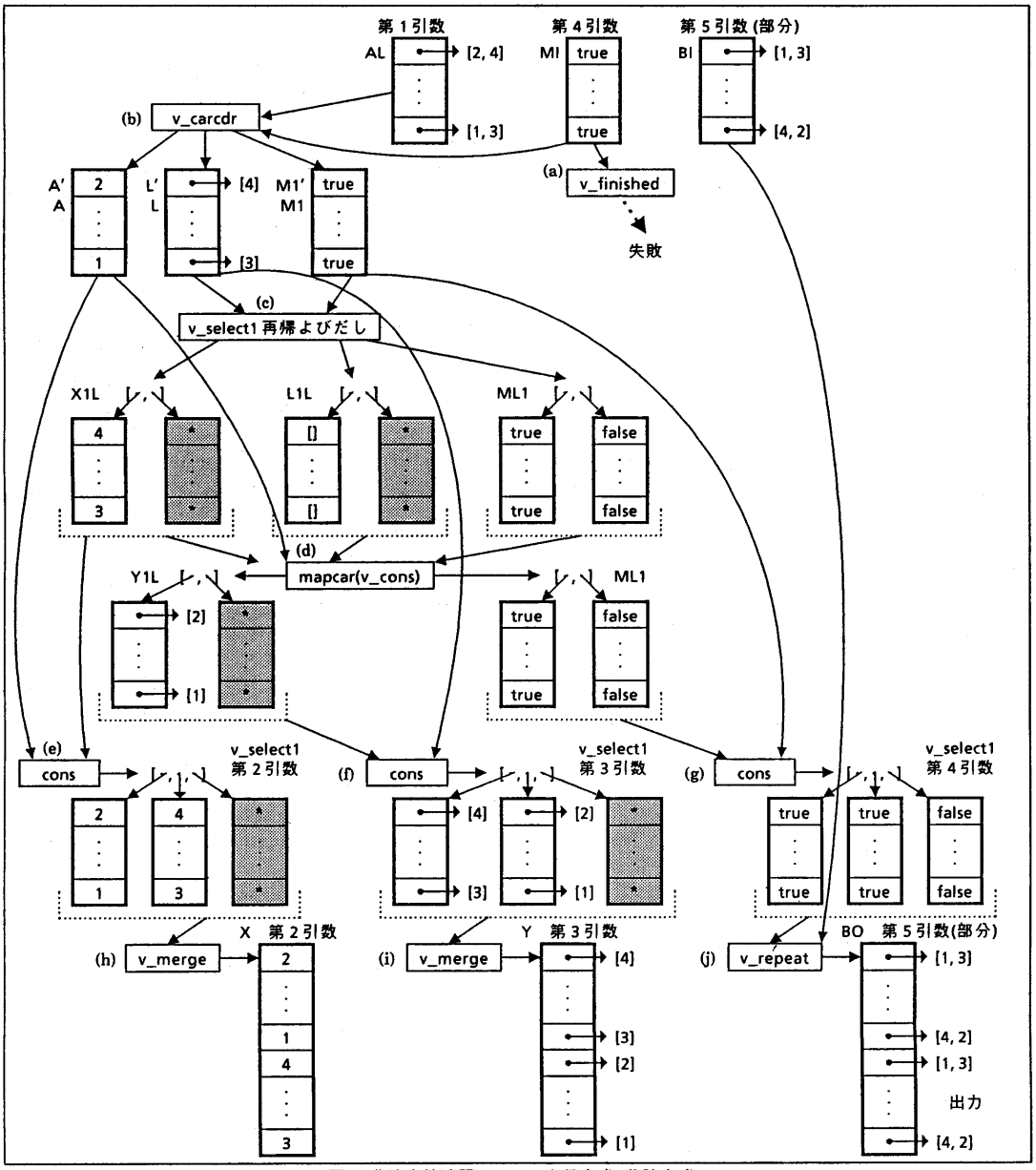

図8 非決定性述語selectの実行方式(蓄積方式)

をconsして第3引数の値をえる(図8(e)~(f)). さらに, 配列M1'とリストMLとをconsして第4引数の後半をえる(図8(g)).

配列併合部(行3)では, リストX1L, Y1Lの要素である配列を併合してそれぞれ1個の配列XおよびYにする(図8(h)~(i)).

環境複写部(行4)では, 配列X, Yと要素どうしの対応がつくように配列BIの要素を配列BOに複写する(図8(j)).

5. 測定結果と検討

図2にしめしたNクウィーン問題をとく論理型言語プログラム(第3~4章で使用した例題はその一部である)を第2~4章の方式にしたがってハンド・コンパイルし, 8クウィーン全解探索のS-810における実行性能を測定した. 測定したプログラムは, マスク演算方式によるもの, インデクス方式によるもの, 圧縮方式によるものの3個である. ベクトル実行するべき部分はFortranで, それ以外の部分はPascalで記述した(Fortranで記述したプログラムの一部を参考図にしめす). なお, ガベジ・コレクションは実装していない.

Fortran部分をベクトル実行した場合とスカラ実行した場合の実行時間を, いずれも表1にしめす. 比較のため, 金田[3]における配列版Nクウィーンの性能をあわせてしめした.*2 また, 実行時間のうちわけを図9にしめす.

とくに重要な点はつぎのとおりである.

(1) マスク演算方式, インデクス方式, 圧縮方式のいずれも, 8クウィーンの実行においては性能上ほとんど差はない. この性能は配列版の性能の約半分である. また, いずれもS-810においてはベクトル実行対スカラ実行の性能比は8~9倍である.*3

(2) いずれの方式でも, 配列要素がnilかどうかの判定とリストの分解, 合成の実行比率がたかい(ベクトル実行で40%程度).

(3) リスト合成は加速率がひくい.

(4) 環境複写・配列併合の実行比率は, マスク演算方式とインデクス方式では数%程度, 圧縮方式では20%程度である.

この結果からつぎのようなことがいえる.

(1) この結果にはFortranで記述したことなどによるオーバヘッドがふくまれているので, 目的プログラムの最適化をはかれば8クウィーンの全解探索が10ms以下で実行できると期待される.*4

(2) リスト版Nクウィーンの性能が配列版Nクウィーンの約半分にとどまっているのは, 上記のオーバヘッドがあること, リスト・ベクトルを多用しているために主記憶とベクトル・レジスタとのスルー・プットが減少していることなどがおもな理由だとかんがえられる.

(3) リスト合成の加速率がひくいのは, ベクトル計算機においては主記憶とベクトル・レジスタとのスルー・プットが汎用機よりはおおきいものの, リスト合成においてはそれ以上のスルー・プットが要求されていることをしめしている.

表1 ベクトル化後の8クウィーンのプログラムの性能

条件制御方式	S-810ベクトル実行時間(ms)	S-810スカラ実行時間(ms)	加速率
マスク演算方式	18	167	9.3
インデクス方式	18	140	7.8
圧縮方式	19	160	8.4
(配列版)	9	79*	8.8*

* 配列版のスカラ実行時間は汎用機むきプログラムの性能であり, ベクトル実行時間と同一のプログラムではない.

(4) マスク演算方式とインデクス方式では, Nクウィーンにおいては環境複写と配列併合はオーバヘッドになっていない. しかし, 圧縮方式ではややオーバヘッドになっている.*5

6. 結論

この報告ではリスト処理をふくむ論理型言語のORベクトル実行方式をしめし, それにもとづいてハンド・コンパイルしたNクウィーン問題の全解探索プログラムの性能の測定結果をしめした. マスク演算方式, インデクス方式, 圧縮方式のいずれにおいてもS-810で20ms弱という速度がえられた. したがって, ORベクトル実行方式は論理型言語の実行方式として有望だということができる.

今後の課題としては, 自動ベクトル化法の確立とコンパイラの開発, より広範囲に適用できるベクトル化方式の開発, 並列バックトラック方式[1,3]の実現などがあげられる.

参考文献

[1] 金田 泰: スーパー・コンピュータによるPrologの高速実行, 第26回プログラミング・シンポジウム報告集, pp.47~56, 1985.

[2] Nilsson, M.: —FLENG Prolog— The Language which turns Supercomputers into Parallel Prolog Machines, *Logic Programming Conference '86*, pp.209~216, 1986.

[3] 金田 泰: Nクウィン問題のベクトル計算機むき解法 —並列バックトラック解法—, 情報処理学会第29回全国大会報告集, 5N-3, pp.1251~1252, 1984.

[4] 上田 和紀: 全解探索プログラムの決定的論理プログラムへの変換, 日本ソフトウェア学会第2回大会報告集, pp.145~148, 1985.

[5] Ueda, K.: Making Exhaustive Search Programs Deterministic, *Third International Conference on Logic Programming, Lecture Notes in Computer Science*, No.225, pp.270~282, Springer-Verlag, 1986.

[6] Codish, M. and Shapiro, E.: Compiling OR-parallelism into AND-parallelism, *Third International Conference on Logic Programming, Lecture Notes in Computer Science*, No.225, pp.283~297, Springer-Verlag, 1986.

[7] 中島 秀之: Prolog, 産業図書, 1983.

---

*1 ただし, 複数のパイプラインをもつベクトル計算機においては, 複数の節が並列に実行されうる.

*2 金田[3]より最近の測定結果である.

*3 ベクトル化したプログラムはスカラ実行には適さないために, スカラ実行むきの最適なプログラムにくらべるとややひくい性能にとどまっているとかんがえられる.

*4 Fortranで記述したためのオーバヘッドの例としては, 参考図にしめしたリストの分解における除算の使用があげられる.

*5 配列版では配列複写・圧縮の実行比率が40%以上であり, オーバヘッドになっている.

*6 たとえば, Nクウィーン問題と同様にOR並列性がたかいプログラムのなかでもBUP(bottom up parser)[8]のように制御が広範囲にちらばるプログラムは, 上記の方式をそのまま適用するとかえって実行がおそくなってしまうとかんがえられる. また, カットやnotなどをふくむプログラムへの適用法もまだわかっていない.

[8] Matsumoto, Y., et. al. : BUP — A Bottom up Parser Embedded in Prolog, *New Generation Computing*, Vol.1, No.2, pp.145~155, 1983.

```
      SUBROUTINE  CARCDV(LEN, CONS, CAR, CDR,
                         MASK, NILP, COUNT, DMY, DMYADR)
      INTEGER  LEN, DMYADR, NILTAG, CCLTAG
      PARAMETER  (NILTAG = 3*65536, CCLTAG = 1*65536)
      INTEGER  CONS(2*LEN), COUNT
      REAL*8  CAR(LEN), CDR(LEN), DMY(*)
      LOGICAL  MASK(2*LEN), NILP(2*LEN)
      DMYADR = DMYADR .AND. 'FFFFFF'Z
      COUNT = 0
      DO 10 I = 1, LEN
        NILP(2*I) = MASK(2*I) .AND. CONS(2*I - 1) .EQ. NILTAG
        MASK(2*I) = MASK(2*I) .AND. CONS(2*I - 1) .EQ. CCLTAG
        IF (MASK(2*I)) THEN
            IX = (CONS(2*I) - DMYADR) / 8
            CAR(I) = DMY(IX + 3)
            CDR(I) = DMY(IX + 2)
            COUNT = COUNT + 1
        END IF
   10   CONTINUE
        END
```

参考図　Nクウィーンのハンド・コンパイル・コードの一部
(マスク演算方式によるリストのnil判定・分解)

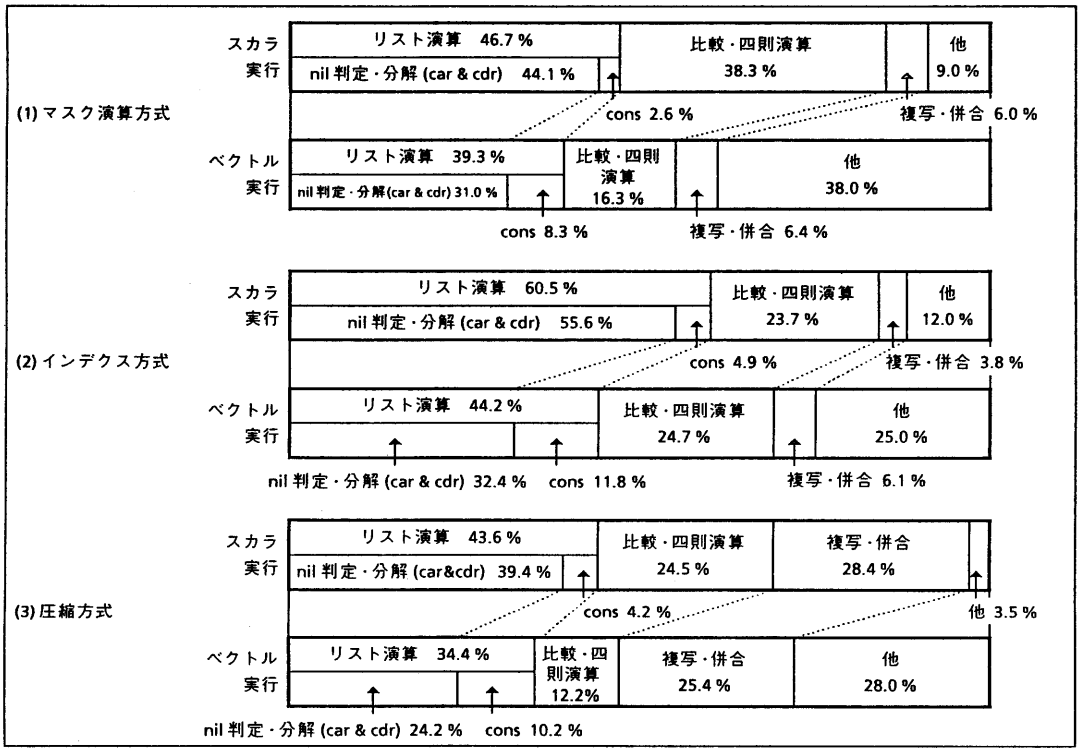

図9　ベクトル化後のプログラムにおける各種演算の実行比率