# スーパー・コンピュータによるPrologの高速実行

金田　泰　　　日立製作所中央研究所

## 1. はじめに

Prologでかかれたプログラムのもつ並列性をひきだして、マルチ・プロセッサやデータフロー計算機によって高速実行させようというこころみがさかんにおこなわれている（たとえば[1]、[2]）。また、Prologをスカラ的（汎用計算機的）なパイプラインで高速化しようというこころみもみられる（[3]、[4]）。一方、Cray-1に代表されるスーパー・コンピュータによってPrologの並列化可能な計算をパイプライン処理しようというこころみは、これまでみられなかった。しかし、すくなくともちかい将来においては、後者は前者以上に高速化の可能性がたかいとかんがえられる。

この報告ではPrologプログラムをスーパー・コンピュータで実行する方法と、その問題点についてのべる。まず第2章では、スーパー・コンピュータの特質をのべる。第3～4章では、その特質をいかすにはどのような並列実行方式をとればよいかを、AND並列化、OR並列化のそれぞれについてのべる。AND並列化についてはまだ充分な検討をおこなっていないので、第4章でのべるOR並列化がこの報告の中心である。Prologのスーパー・コンピュータによる高速実行のさまたげとなりうる問題点がいくつかあるが、第5章ではそのうち最大の問題とかんがえられるユニフィケーションの高速化の問題についてのべる。そして、そのほかの問題について第6章でのべる。

## 2. スーパー・コンピュータの特質と高速化の可能性

Cray-1で代表されるスーパー・コンピュータは数値計算を高速化するためにつくられた計算機であり、実際、それ以外の目的でつかわれたことは、まれである。（国内をみるかぎりでは、数値計算の一種ではあるが、やや毛色のかわったものとして円周率の計算[5]がおこなわれた程度である）。数値計算においては通常は配列計算が支配的であり、スーパー・コンピュータはその高速化に焦点をおいている。そのため、スーパー・コンピュータは、くりかえし回数（ベクトル長）が数10回以上の均質なくりかえし計算に対しては、それを並列化することによって汎用計算機の20倍以上の高速化をはかることができる。（ここで「並列化」とよんだ操作は正確には「パイプライン化」だが、この報告では、かんたんのため「並列化」とよぶ）。

逆に、並列度があっても2～3程度の計算、あるいは並列度はたかいが均質性がない計算においては、ベクトル演算パイプラインをいかすことができないため無力である（図1、図2）。（記憶装置や1本のバスを共有するマルチ・プロセッサにおいては通常、逆にプロセッサ台数が10台以下で性能があたまうちになるので、

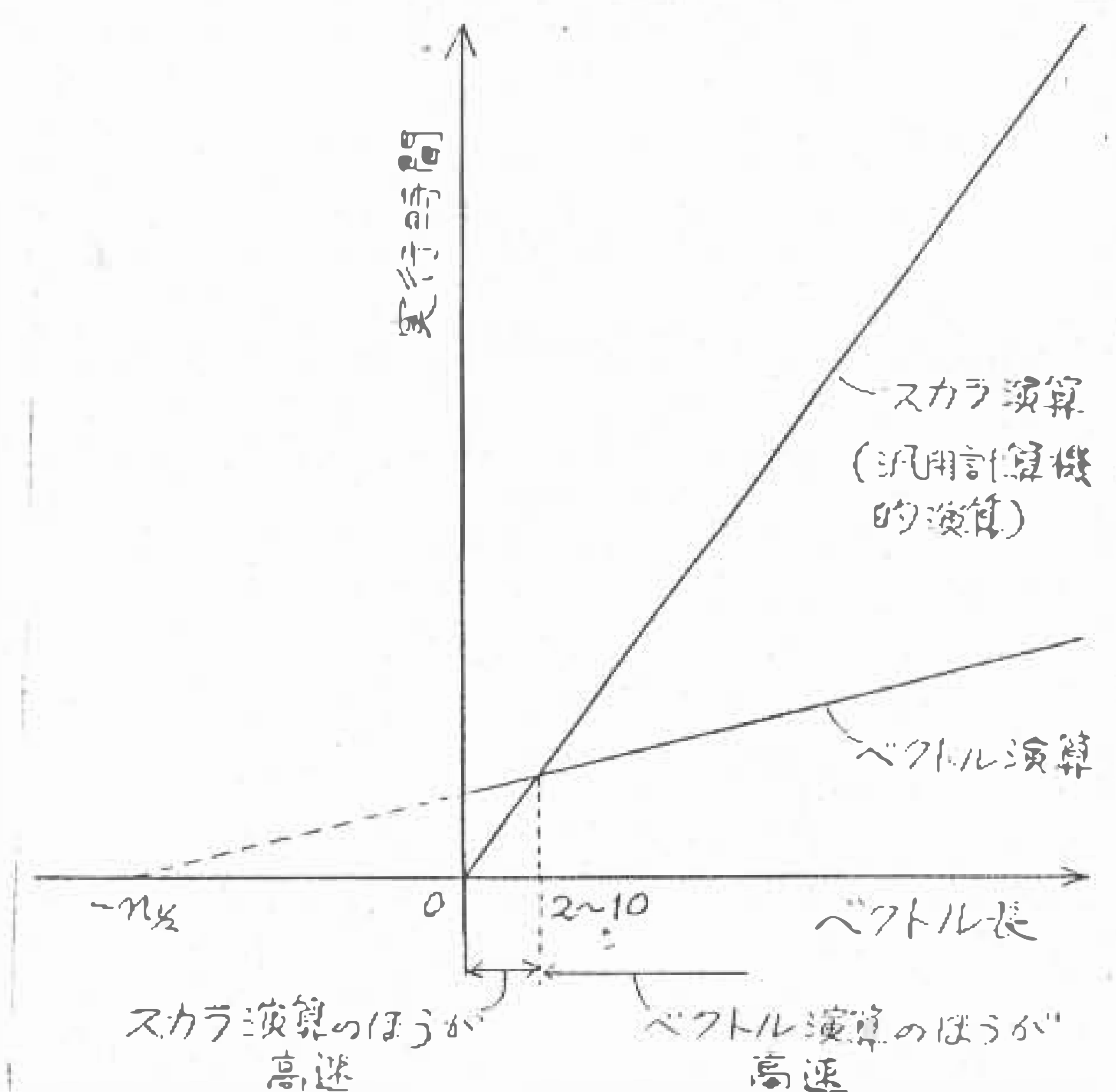



図1 スーパー・コンピュータにおけるベクトル長と実行時間との関係

```
      DO 10 I = 1, N
        A(I) = B(I) * C(I) + D(I)
   10 CONTINUE
```

(a) Iに関して均質な計算の例

```
      DO 10 I = 1, N
        IF (I.EQ.1) THEN
          ...
        ELSE IF (I.EQ.2) THEN
          .
          .
          .
        ELSE IF (I.EQ.N) THEN
          ...
        END IF
   10 CONTINUE
```

(b) Iに関して不均質な計算の例
-- 複雑なループ可変条件

```
      DO 10 I = 1, N
       ...
        IF (C) GOTO 20
       ...
   10 CONTINUE
   20 CONTINUE
```

(c) Iに関して不均質な計算
--ループとびだし

図2 均質な計算と不均質な計算

スーパー・コンピュータはそれとは対照的だといえる）。また、くりかえし回数が数100程度で高速化はあたまうちになる。すなわち汎用計算機との性能比がほぼ一定になる。

このような特質をかんがえると、Prologプログラムから高度かつ均質な並列性をみいだすことが、スーパー・コンピュータによるPrologプログラムの高速化を成功させるかぎになる。このような並列性は、もちろんすべてのプログラムにみいだせるわけではない。すなわち、スーパー・コンピュータですべてのFortranプログラムが高速化されるわけでないのと同様に、高速化がのぞめないPrologプログラムもある。しかし、すくなくともPrologが得意としているある種の探索問題においては、均質な並列性をみいだすことができる。たとえば、N-queen問題（8-queen問題を一般化したもの）はその例である。N-queen問題の並列解法のプログラムは、スーパー・コンピュータS-810において、逐次解法にくらべてつぎのように高速化することができた [6]（ここで、全解探索とはすべての解をもとめることであり、単解探索とはひとつだけ解をもとめることをいう）。

☆全解探索の場合：　Fortranで、N≧10のとき4.5倍（図3）。機械語で20倍以上（推定）。

☆単解探索の場合：　N≧15のときは逐次解法より高速（Fortranで。図4）。

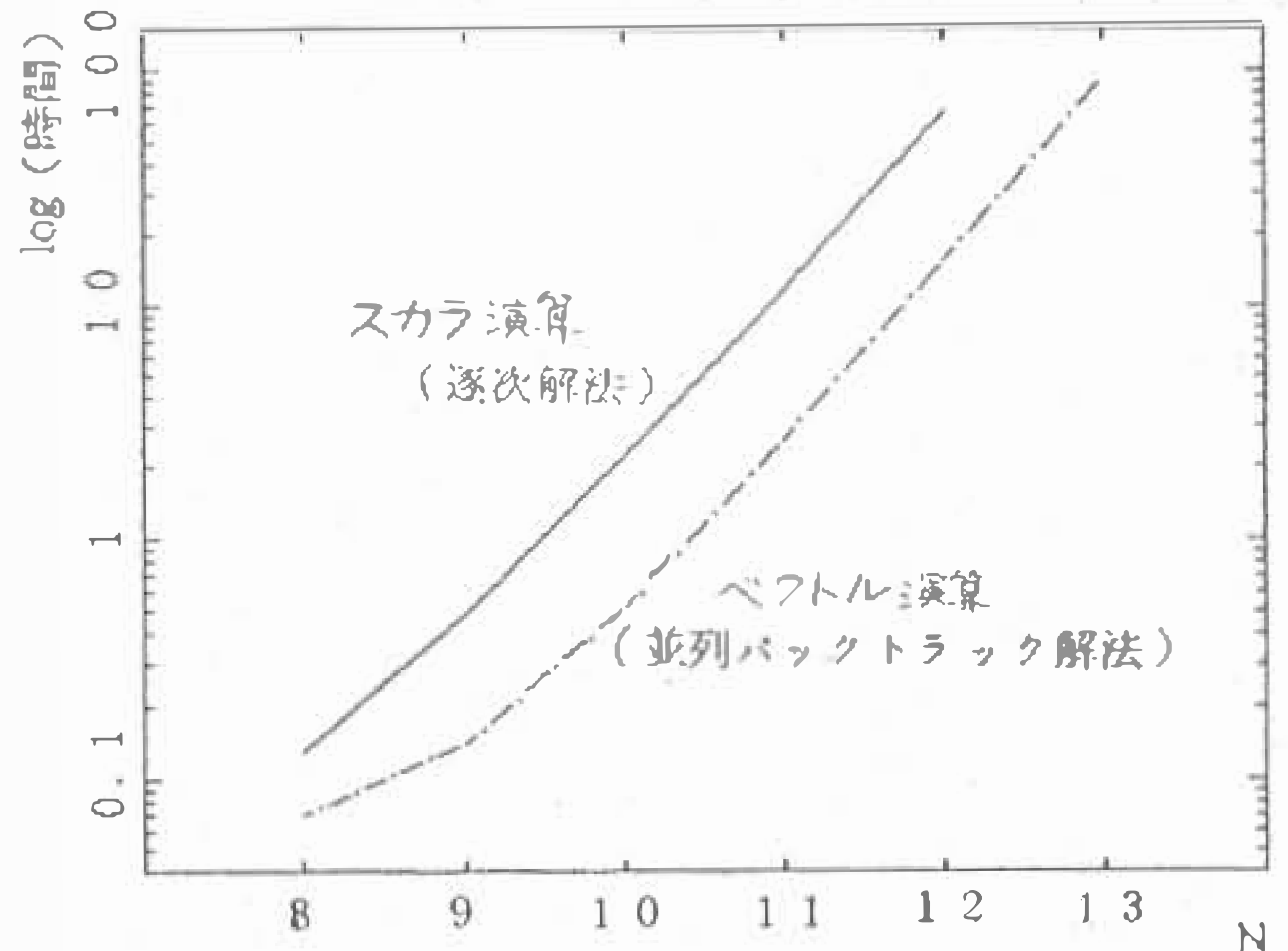


図3　N-queen問題の全解探索（S-810による実測値）

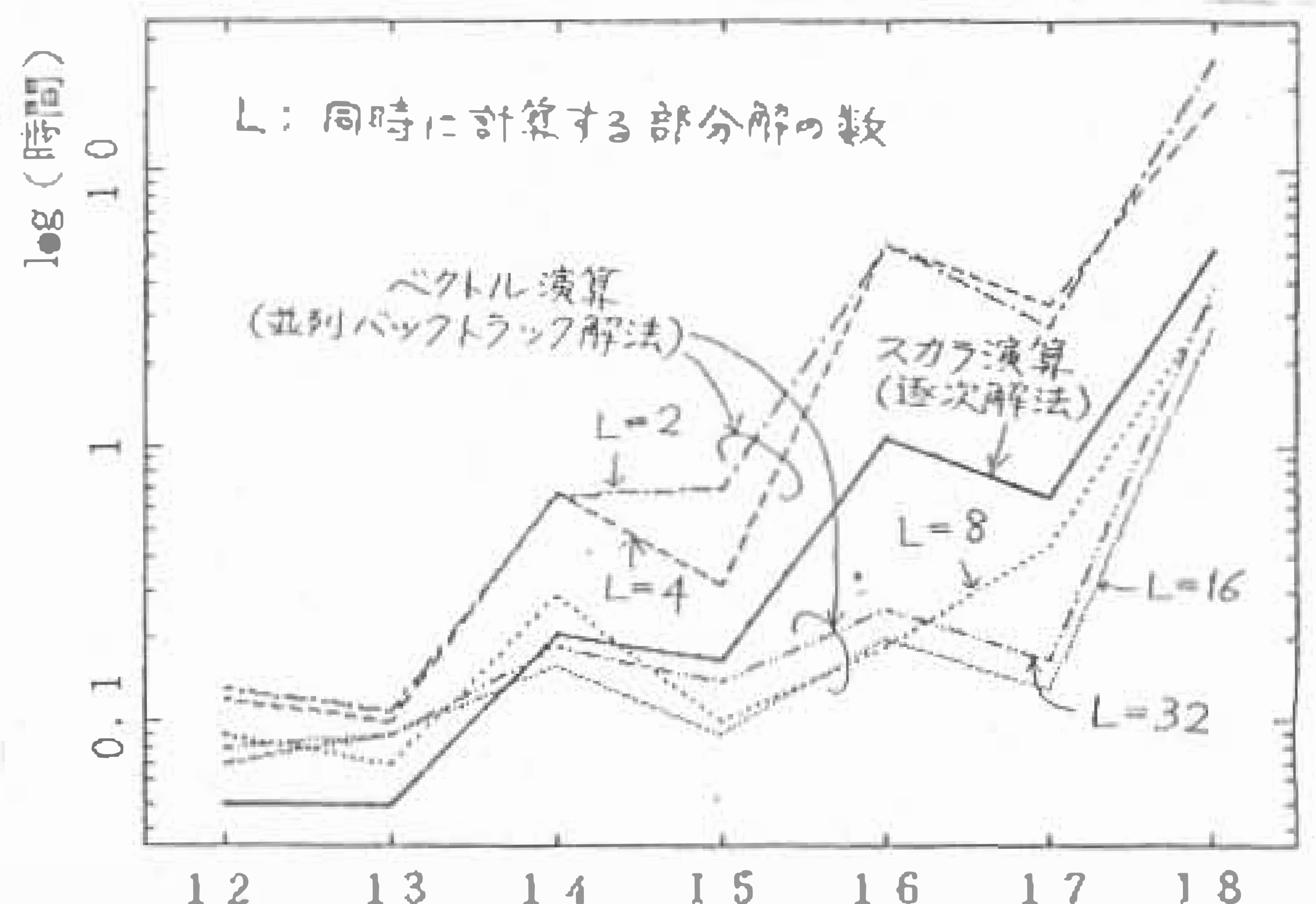


図4　N-queen問題の単解探索（S-810による実測値）

Prologで記述されたプログラムの実行に関しては、Fortranなどの手続き型言語にはない問題が多々あるので、この数字をそのままあてはめるわけにはいかない。しかし、上記の結果をえたFortranのプログラムは第4章でのべる並列バックトラックによるOR並列化をシミュレートしたものであるから、逐次型Prolog、並列型Prologの両処理系において充分な最適化をおこなえば、Prologにおける性能比もこの数字にちかづけることができるとかんがえられる。したがって、スーパー・コンピュータをうまくつかえれば2MLIPS以上の計算速度をえることができるとかんがえられる。なぜなら、汎用計算機の場合で400kLIPS程度の計算速度をえることができるとかんがえられるからである（いずれもコンパイルした場合）。

## 3. AND並列化

第2章でのべたスーパー・コンピュータの特質をいかしてPrologを高速に実行するには、場合によってAND並列化とOR並列化とをつかいわける必要があるとかんがえられる。OR並列化については第4章でのべることにして、この章ではAND並列化についてのべる。

AND並列性とは、ひとつの解をもとめる過程に存在する並列性のことである。AND並列性は、さらにつぎの2つにわけることができる。

（1）局所的AND並列性

通常のスカラ・パイプライン処理においてとりだされるような並列性である。局所的AND並列性には、命令デコードや命令実行の並列性というような低水準のものもあり、「引数間並列性」のような高水準のものもある。（Tickら [3]，[4] は局所的AND並列性をいかす（スカラ）パイプライン・アーキテクチャを研究している）。

（2）大域的AND並列性

アルゴリズム・レベルの大域的な並列性である。ひとつの節を構成する述語のあいだの並列性、再帰よびだしにおける同一の述語のことなるよびだしのあいだの並列性などがある。（局所的AND並列性と大域的AND並列性との境界はあまりはっきりしない。たとえば、ひとつの節を構成する隣接したくみこみ述語のあいだの並列性は、局所的AND並列性にふくめるべきである）。

マルチ・パイプライン型のスーパー・コンピュータにおいては局所的AND並列性もいかすことができるが、ベクトル演算機構を利用するためには、まず大域的AND並列性あるいは（2）でのべるOR並列性をとりだすことが必要である。しかも、均質な並列性をとりだす必要がある。局所的並列性はハードウェアで検出することができるが、大域的並列性はソフトウェアで検出しておかなければ、それをいかすことはむずかしい。Fortranプログラムを実行する場合にも、ベクトル演算機構を利用するために「ベクトル化」とよばれるプログラム構造変換によって並列処理が可能なプログラムに変換している（図5）が、Prologにおいても同様はプログラム構造変換が必要である。この点は、次節でのべる再帰並列化にも共通している。

AND並列化についてはまだ充分な検討をおこなっていないが、「どのような並列化をおこなうべきか」という問題に対して、つぎのようなことはいえるだろう。

（1）項どうしの並列化はスーパー・コンピュータむきではない。

つぎのような節においてQ1, . . . . , Qnを並列（パイプライン）処理することができる。

```
P:-Q1,Q2,...,Qn.
```

しかし、このような並列処理は通常はスーパー・コンピュータには適さないとかんがえられる。その理由は、Q1, . . . . , Qnの各項が均質な処理であることはまれだということである。すなわち、これらの項は通常はことなる述語名をもっているか、引数の構造が大幅にことなっている。ただし、もちろん場合によってはスーパー・コンピュータむきの並列性が存在しうる。

（2）再帰よびだしの並列化はスーパー・コンピュータむきである。

再帰よびだしには逐次性のつよいものもあるが、並列性があるならば、それはスーパー・コンピュータに適した並列性である可能性がたかいとかんがえられる。

```
      DO 10 I = 1, N
        A(I) = B(I) + C(I)
        S = S + D(I) * E(I)
   10 CONTINUE
```

（a）原始プログラム

```
      A(1:N) = B(1:N) + C(1:N)
      S = S + DOTPRODUCT(D(1:N),E(1:N))
```

（b）ベクトル化後のプログラム

図5 ベクトル化の例

その理由は、各よびだしごとの処理の均質性が、通常、たかいことにある。ただし、並列度がひくいなどの理由でスーパー・コンピュータに適さないこともある。

以後は再帰よびだしの並列化についてだけかんがえる。例として、つぎのような述語をかんがえる。

```
not-take1 ([], *q+, *q-) :- .                             ・・・(3.1)
not-take1 ([*q | *r], *q+, *q-) :-
        not (*q==*q+) , not (*q==*q-) ,
        plus (*q+, 1, *q++) , minus (*q-, 1, *q--) ,
        not-take1 (*r, *q++, *q--) .                      ・・・(3.2)
```

これは中島[7]による8-queen問題の求解プログラムの一部（queenが他のqueenによってとられないかどうかを検査する述語）である。not-take1には第1引数としてながさが0～8のリストがあたえられ、notでバックトラックがおこらなければそのながさのぶんだけの回数、再帰的によびだされる。その再帰よびだしのうち最後の1回以外は節（3.2）がよびだされ、よびだしごとにほぼ均質な処理がおこなわれる。

notで生じるバックトラックは均質性を破壊しているが、これらの計算には副作用がない。したがって、バックトラックするまえに、本来はおこらない、plus, minusおよびnot-take1の再帰よびだしをおこなったとしても、（notで失敗したことを記憶しておけば）結果はかわらない。したがって、この再帰よびだしはループ外とびだしのない均質なループに変換することができ、スーパー・コンピュータで並列実行できる。（バックトラックをもとのままシミュレートしようとすると、ループ外とびだしが必要になり、並列化できない。図6にもとのプログラムとバックトラックをおくらせたプログラムをそれぞれFortranでシミュレートしたものをしめす）。

```
      FAIL = .FALSE.
      DO 10 I = X-1, 1, -1
        IF (Q(X).EQ.Q(I) .OR.
     *        Q(X).EQ.Q(X+X-I)) THEN
          FAIL = .TRUE.
          GOTO 20
        END IF
   10 CONTINUE
   20 CONTINUE
```

（a）not-take1をそのままシミュレートするもの

```
      FAIL = .FALSE.
      DO 10 I = X-1, 1, -1
        IF (Q(X).EQ.Q(I) .OR.
     *        Q(X).EQ.Q(X+X-I)) THEN
          FAIL = .TRUE.
        END IF
   10 CONTINUE
```

（b）バックトラックをおくらせたもの

図6 not-take1をシミュレートするFortranプログラム

ただし、リストの表現をかえないかぎりは、リストの尾部（Lisp用語でいえばcdr）をもとめる処理は逐次的におこなわなければならないであろう。そして、その部分が並列化されないかぎりは飛躍的な高速化はのぞめないであろう。また、この問題はべつにしても、N-queen問題ではNが充分におおきくならなければ、並列度がひくいために逐次計算より高速にはならない。"8"-queenでは不充分である（付録1参照）。

原理的には以上のとおりだが、実際には並列化対象にする述語における計算のパタンによって、ことなる方法で並列化をおこなわなければならない。とくに、つぎのような点が重要だとおもわれる。

（1）ループ化可能な再帰よびだしかどうかの判定

まず、対象としている再帰よびだしが、くりかえしにかきかえられるかどうかを判定する必要がある。さらに、その述語の実行の途中でバックトラックがおこらず、したがってループ外とびだしのないループに変換することができるかどうか、あるいは上記の8-queenの例のように、バックトラックをおくらせることによって、とびだしのないループに変換することができるかどうか、などを判定する必要がある。

（2）擬似並列命令の適用

表1にしめすような擬似並列命令をうまくつかうことによって逐次処理部分を極力へらすようにすれば、スーパー・コンピュータのハードウェアをいかすことができる。したがって、本来は並列化できないが擬似並列命令によって「ベクトル化」できる部分をみつけて、それを適用することが必要である。あるいはハードウェア設計にフィードバックできれば、Prologむきの擬似並列命令をくみこんで使用することもかんがえられる（ただし、現在はその必要があるかどうかもわかっていない）。

（3）リスト処理の並列化

not-take1でわかるように、再帰よびだしを高速化するにはリスト処理を並列化可能にすることが必要である。そのためにはリストを配列としてあらわすようなくふうが必要であろう。

しかし、一般にどのようにすればこれらの変換を自動的におこなうことができるかは、いまのところわかっていない。人間がこれらの変換をおこなえば、比較的容易にAND並列化をおこなうことができるとかんがえられるが、それでは、問題むきの記述ができるというPrologの利点を充分にいかすことができない。（ただし、ある種のくりかえしをあらわす述語（"for"のような）を導入することは、スーパー・コンピュータだけでなく人間にも利点があるだろう）。

表1 擬似並列命令の例

```
(1) 累和
    S = ΣA(J)
         J

(2) 内積
    P = ΣA(J)*B(J)
         J

(3) 算術数列生成
    A(1) = S
    A(I) = A(I-1) + C   (I=2,3,....,N)

(4) 1階逐次計算
    A(1) = S
    A(I) = A(I-1) * B(I) + C(I)
                (I=2,3,....,N)
```

## 4. 並列バックトラック方式によるOR並列化

### 4.1. OR並列化

AND並列化とならぶもうひとつの並列化方式はOR並列化である。「OR並列化」とは、ことなる解をもとめる処理を並列実行させることである。したがって、バックトラックのないプログラムはOR並列性がなく、並列化することができない。一方、多数の解をもとめる必要がある場合、あるいは解をもとめるまでに必要なバックトラックが膨大である場合にはOR並列化することができ、スーパー・コンピュータで実行することによって高速化されうる。（ただし、その場合にも処理の均質性が充分でなければむしろ低速になる可能性もある。均質性をそこなうおおきな原因はユニフィケーションにあるとかんがえられるが、それについては第5節でのべることにする）。

つぎのような節をかんがえる。

P:-Q1,Q2,....,Qn.

ここで、おおざっぱにいうと、つぎのようにすればスーパー・コンピュータむきのOR並列化をはかることができる（図7）。Pがよばれるとまず Q1を実行するが、その際、すべての部分解をもとめて配列にする。その配列をQ2にわたし、Q2はその要素について並列に、つぎの処理をおこなう： その配列のうち解となりえないものはすてて、のこりの部分解を成長させた結果を配列にして、Q3にわたす。この配列のおおきさは、Q2にわたされた配列よりちいさいこともあり、おおきいこともある（ちいさくなるのは一部の解が失敗したときであり、おおきくなるのはOR並列性があるときである）。以下同様にしてQnまで計算をすすめ、そこでえられた配列をPの結果とする。すなわち、その配列の各要素が解である。

より具体的な例をつかって説明する。つぎにしめすのは、第3節でとりあげた、中島[7]による8-queenの

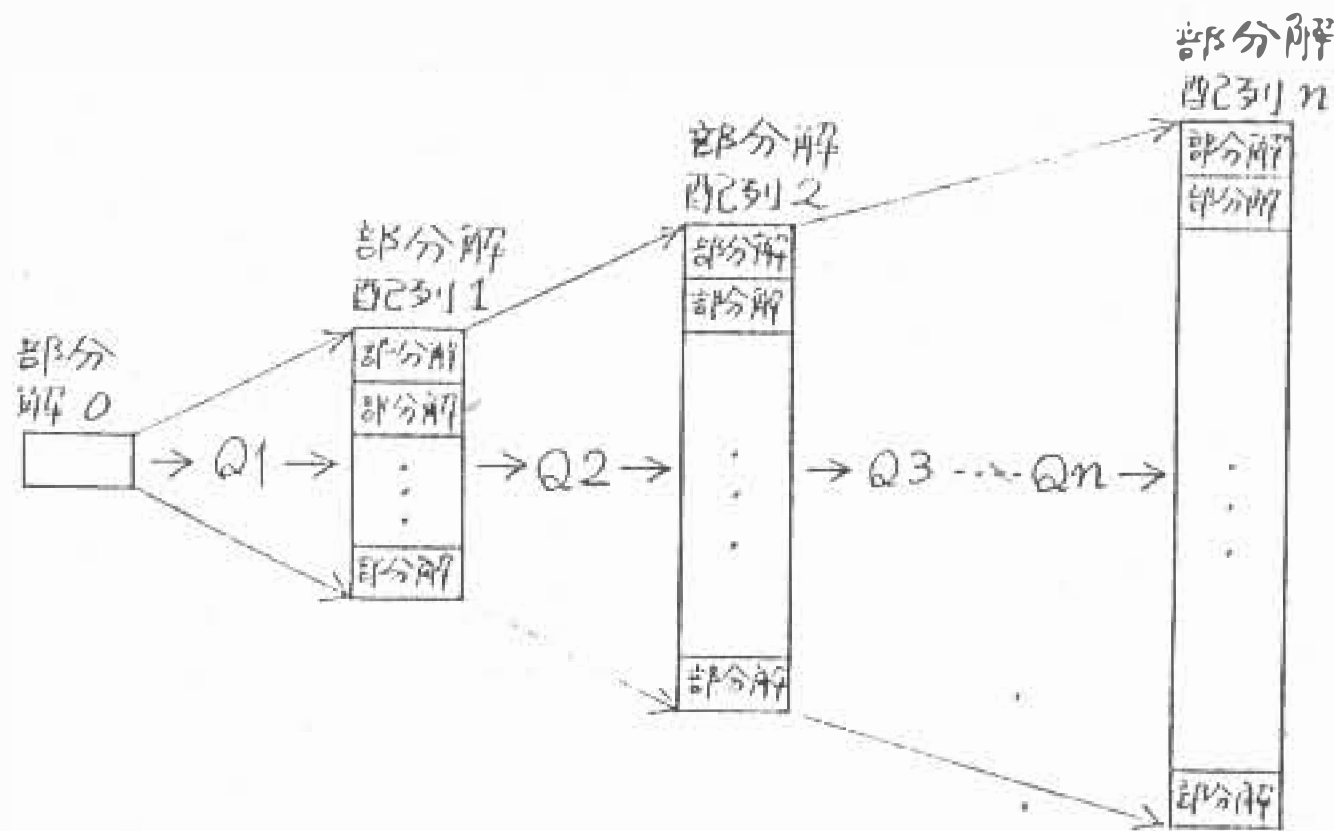



図7 OR並列方式による実行

プログラムの全体である（not-take1は第3章のものとおなじである）。

```
queen (*q) :-
        put ([1,2,3,4,5,6,7,8], [], *q) . ････ (4.1)

put ([], *q, *q) :- .                         ･･･ (4.2)
put (*qs, *b, *q) :-select (*qs, *q1, *r) ,
        not-take (*b, *q1) ,
        put (*r, [*q1 | *b], *q) .            ････ (4.3)

select ([*x | *y], *x, *y) :- .               ････ (4.4)
select ([*x | *y], *z, [*x | *z]) :-
        select (*y, *z, *w) .                 ････ (4.5)

not-take (*r, *q) :-
        plus (*q, 1, *q+) , minus (*q, 1, *q-) ,
        not-take1 (*r, *q+, *q-) .            ･･･ (4.6)

not-take1 ([], *q+, *q-) :- .                 ･･･ (4.7)
not-take1 ([*q | *r], *q+, *q-) :-
        not (*q = *q+) , not (*q = *q-) ,
        plus (*q+, 1, *q++) , minus (*q-, 1, *q--) ,
        not-take1 (*r, *q++, *q--) .          ･･･ (4.8)
```

ここで?-queen (*x) が実行されると、putがよびだされる。（4.2）は失敗するので（4.3）がよびだされるが、ここまでは通常のPrologの実行方式にしたがえばよい。通常のPrologの実行においては、selectはqueenのリスト*qsからひとつのqueenをえらびだして*q1とし、のこりのqueenのリストを*rとする（図8a）。しかしOR並列方式では、えらびだされるべきすべてのqueenからなる配列すなわち*qsにふくまれるすべてのqueenからなる配列を*q1とし、その要素に対応する「のこりのqueenのリスト」からなる配列を*rとする（図8b。したがって、*q1および*rの要素数はひとしい）。

not-takeにはこれらの配列がわたされる。plusおよびminusの計算は、通常のベクトル演算として並列におこなうことができる。その結果*q+および*q-がもとめられるが、これらも*q1，*rと同数の要素をもつ配列である（plusおよびminusではバックトラックがおこらないゆえに同数になる）。同様にしてnot-take1の計算も並列化されるが、この場合にはnotでバックトラックが生じて部分解配列のおおきさがへるので、ひとくふうする必要がある。すなわち、notでふるいおとされる要素は配列からとりのぞくか、または適当な方法で無効にする必要がある。そのためには、Fortranの条件文を実行するときにつかわれるマスク演算方式、圧縮方式またはリスト・ベクトル方式をつかえばよい。（これらの方式名のうち、「マスク演算方式」および「リスト・ベクトル方式」は一般的な名称だが、「圧縮方式」はそうでないということをことわっておく）。

これらの方式について順次説明する。まず「マスク演算方式」（図9a）であるが、この方式では部分解配列と同数の要素をもつ論理型配列を用意して、部分解配列の有効な要素に対応する要素はtrue、無効な要素に対応する要素はfalseとする。このような配列を「マスク・ベクトル」とよぶ。（4.8）の場合、はじめはマスク・ベクトルのすべての要素がtrueだが、notの実行によってfalseがふえていくわけである。つぎに「圧縮方式」（図9b）であるが、この方式では、ひとつの配列計算の結果として部分解配列に無効な要素が生じるたびに、それを除去して配列をつめあわせていく。

*qs = [1, 2, 3, 4, 5, 6, 7, 8]

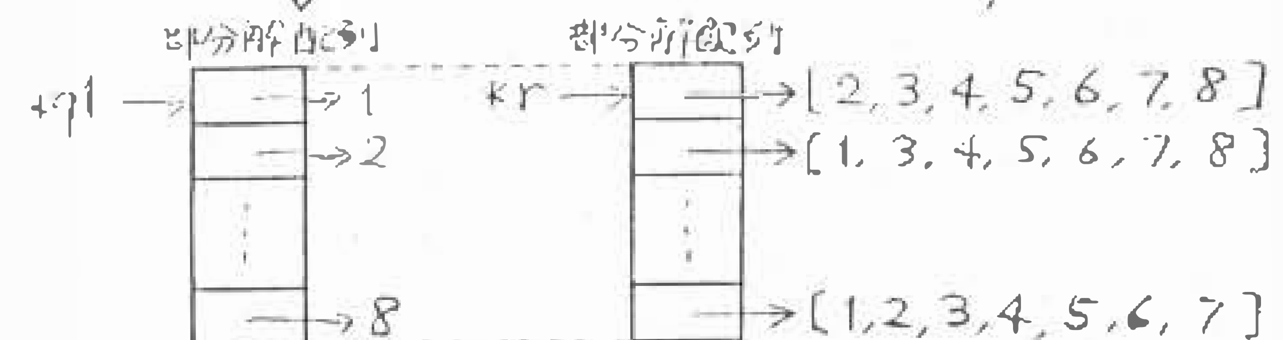
⇓ select

*q1 = 1
*r = [2, 3, 4, 5, 6, 7, 8]

(a) 逐次実行の場合

*qs = [1, 2, 3, 4, 5, 6, 7, 8]

⇓ select




(b) OR並列実行の場合

図8 述語select の実行

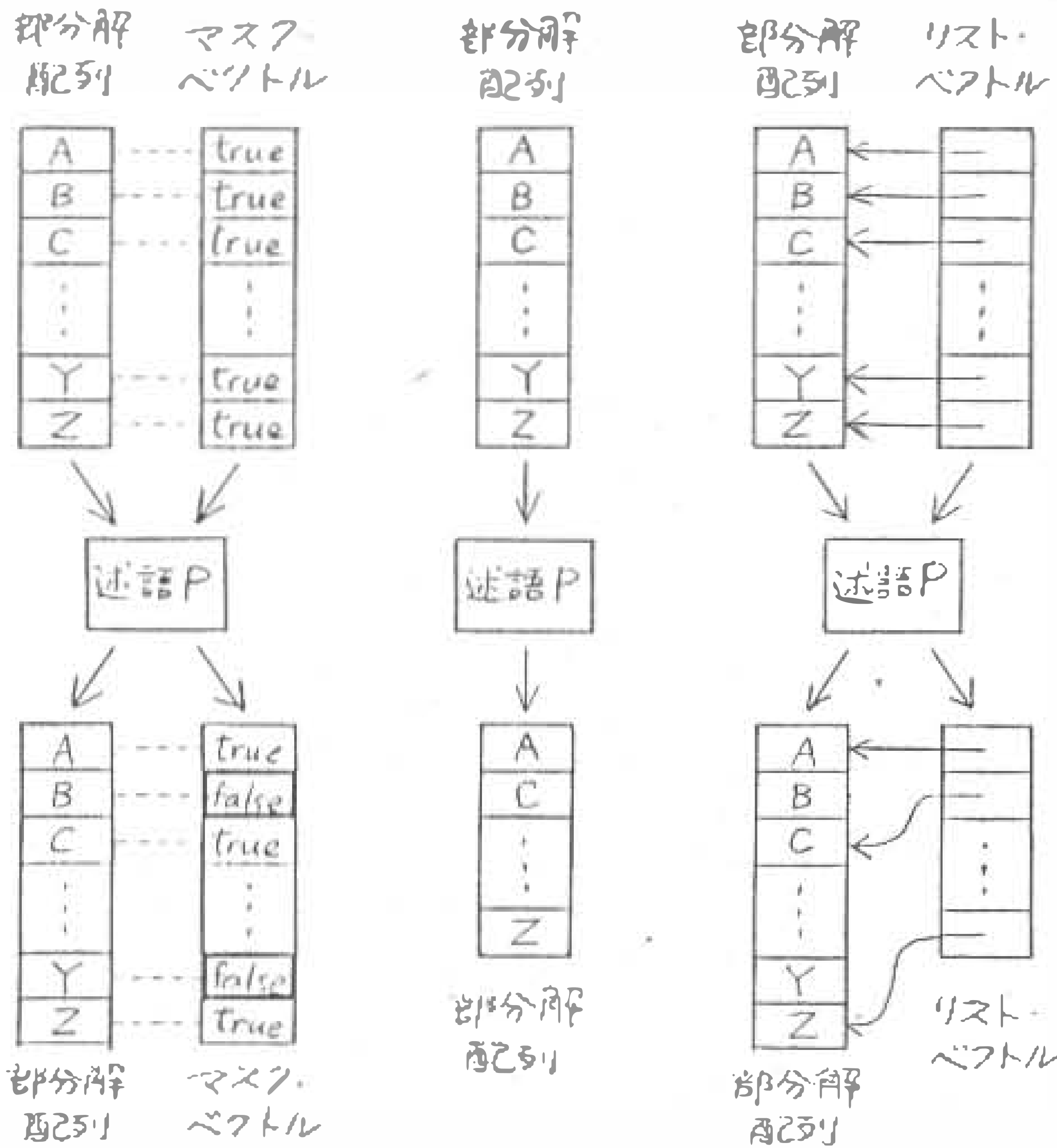



図9 部分解配列縮小の実装法

(4.8)の場合、notを実行するたびに*q、*q+および*q-をつめあわせる。おわりに「リスト・ベクトル方式」(図9c)であるが、この方式では部分解配列の有効な要素のアドレス(というより変位)からなる配列をもちいる。このような配列をリスト・ベクトルという。リスト・ベクトルじたいは圧縮方式と同様につめあわせをおこなうが、部分解配列はつめあわせる必要がない(図9c)。

これらの方式の長短はつぎのとおりである。

(1) マスク演算方式

つめあわせの必要がないので、演算量がすくないとき、あるいはバックトラックがすくない(部分解配列に無効な要素がすくない)ときには効率がよい。しかし、演算量がおおくかつバックトラックがおおいときには無効演算(無効な要素に対する演算)がふえるので非効率である。また、つめあわせをしないため、記憶効率もわるい。

(2) 圧縮方式

無効演算がないというのが利点だが、すべての部分解配列についてつめあわせをするので、そのオーバヘッドがおおきい。同一のマスクのもとで多数のマスク演算をおこなうよりは効率がよいが、Prologにおいてはこのような状況はまれだとかんがえられる。

(3) リスト・ベクトル方式

圧縮方式と同様に無効演算がない。また、複数の部

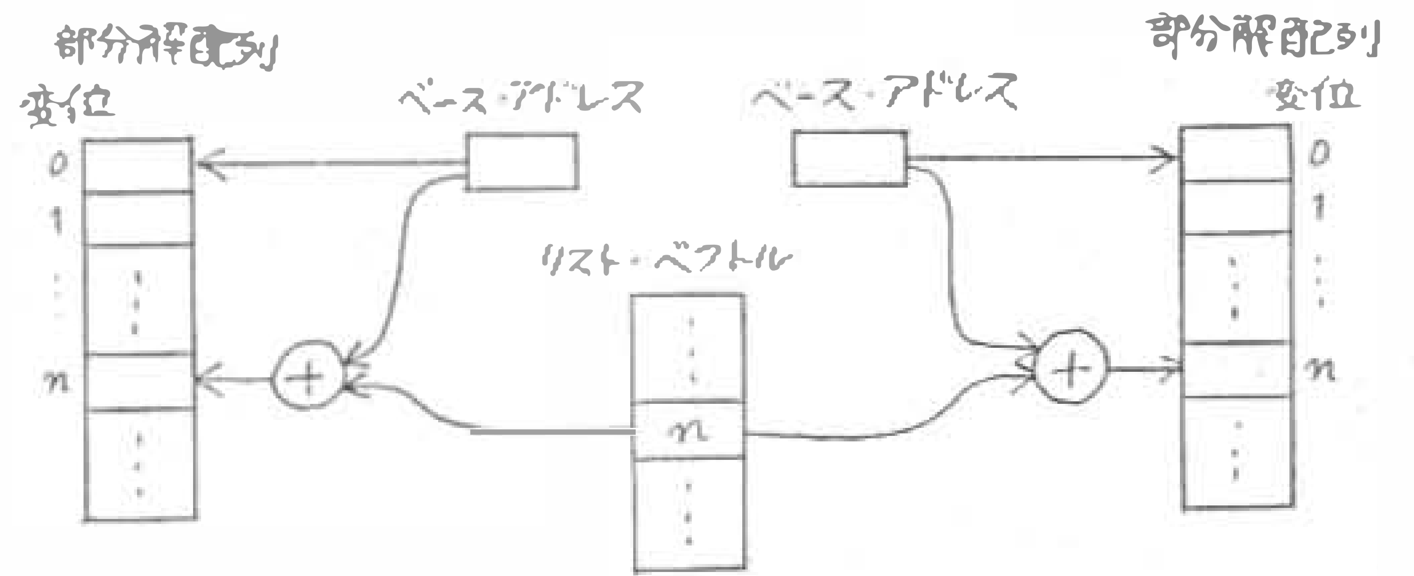



図10 ひとつのリスト・ベクトルによる
複数の部分解配列のアクセス

分解配列をそれぞれことなるベース・アドレスとただひとつのリスト・ベクトルとでアクセスするようにすれば(図10)、つめあわせはリスト・ベクトルに対してだけおこなわれるので、圧縮方式にくらべて順序的であり、マスク演算方式にくらべても通常はまさっているとかんがえられる。ただし、部分解配列が間接アドレスでアクセスされるので、そのアクセスにやや時間がかかる。また、部分解配列をつめあわせないので、マスク演算と同様に記憶効率がわるい。

これらの長短をくらべると、リスト・ベクトル方式が比較的よいようにおもわれる。したがって、リスト・ベクトル方式を核にしたうえで、その欠点をおぎなうためにほかの方式も併用するようにするのがよいとかんがえられる。(どの方式をとるかは定量的なデータにもとづいてきめるべきだが、まだそのようなデータはもとめていない)。

4.2 並列バックトラック方式

ところが、これまでのべてきたOR並列方式では、いわゆる数の爆発をひきおこす。すなわち、OR並列性がたかいプログラムにおいては部分解配列要素の数すなわち部分解の数が膨大になって、メモリからあふれてしまう。また、ただひとつ解をもとめればよい場合にもすべての解をもとめてしまうので、むだである。そこで、つぎのような方式をかんがえる:

(1) 部分解の数が一定数をこえたら部分解配列を分割し、
(2) そのうちのひとつについて計算をすすめる。
(3) その部分解がすべて失敗したときは、のこりの部分解配列のうちのひとつについて、同様の処理をくりかえす。

このようにすることによって、数の爆発をふせぐことができる。また、最初の解がもとめられた時点で計算をうちきれば、ただひとつ解をもとめる場合にも適用されうる。この方法は並列に処理される部分解の集合に対してバックトラック計算をするので、並列バッ

クトラック方式とよんでいる。

図11にしめしたのが、並列バックトラック方式によるPrologプログラムの実行の様子である。図11の(a)，(b)，(c)がそれぞれ上記手順の(1)，(2)，(3)に対応している。

第2章で、スーパー・コンピュータにおいては、くりかえし回数が数100回で高速化があたまうちになるということをのべた。この事実を利用すると、並列バックトラック方式において部分解の数がつねに数100個になるように制御してやれば、スーパー・コンピュータの高速性を最大限にいかしつつ数の爆発もふせぐことができることになる。

しかし、部分解配列の要素数を数100個にするとメモリもすくなくとも数100倍必要になるので、よほど使用メモリがすくないプログラムでなければ、たえられない。また、ただひとつ解をもとめればよい場合には、よほど解のかずがすくなくなければ、よぶんな解をもとめるために低速になってしまう。そこで、部分解配列の要素数は数個～数10個とする。すなわち、ただひとつ解をもとめる場合あるいはメモリの制限がきついときには数個、メモリに余裕があって、かつすべての解をもとめる必要があるときには数10個とすればよいだろう。これによって高速性はややそこなわれるが、ほどほどのメモリで計算することができるようになる。(Cray-1タイプすなわちベクトル・レジスタをもつスーパー・コンピュータのばあいは半性能(ベクトル)長が10～20なので[8]、最大性能の半分以上はだせるはずである。ただし、数個～数10個という値はN-queen問題において適当とかんがえられる値であって、ほかの問題についてはためしていないので、適当かどうか、かならずしもわからない)。

第2章でしめしたN-queen問題のFortranによる求解例は、この方式をFortranでシミュレートしたものである(部分解配列縮小の実現法としては、圧縮方式をもちいている(§4.1参照))。

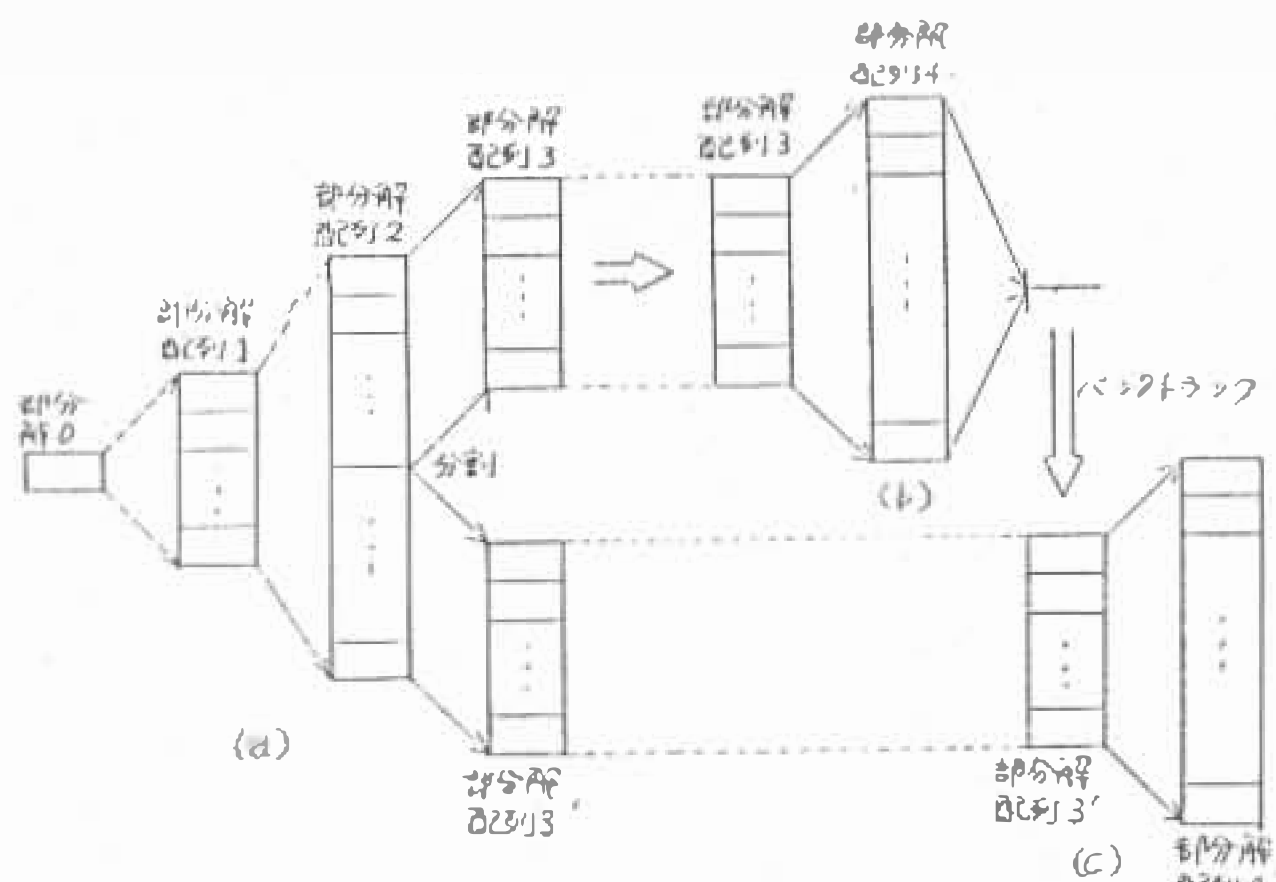



図11 並列バックトラック方式による実行

## 5. ユニフィケーションの高速化

この章ではOR並列/並列バックトラック方式におけるユニフィケーションの高速化についてかんがえる。並列バックトラック方式によれば四則演算などはスーパー・コンピュータがもつ命令によって容易に高速化されうる。しかし、Prologプログラムの実行時間の大半をしめるのはユニフィケーションであり、その時間が短縮されなければ、スーパー・コンピュータによる実行はむしろたかくつく。ところが、Prologであつかわれるデータ構造は柔軟であり、ユニフィケーションのアルゴリズムは多数の条件分岐をふくんでいる。したがって、それをそのままベクトル化するとスーパー・コンピュータには不利である。

そこで、部分解配列が出現頻度のひくい(あるいは複雑な手順を要する)パタンをふくまない場合には、出現頻度のひくいパタンをふくむ場合だけに必要な手順を省略するようにする。すなわちユニフィケーションにあたっては、まず部分解配列の要素に対して並列に、例外的な(出現頻度のひくい)パタンかどうかを検査する。そして、例外的でないものについてはユニフィケーションをおこなう。例外的なパタンがなければ、部分解配列のユニフィケーションはこれでおわりである。もし例外的なパタンがみつかった場合には、そのあと、そのものについて逐次に(あるいは、例外が多数あるときには並列に)ユニフィケーションをおこなう。

具体的にどのようなユニフィケーション・アルゴリズムにすればよいかはハードウェアの特性につよく依存するし、S-810用のアルゴリズムは現在検討中であるから、詳細は省略する。しかし、上記のかんがえかたを説明するために、一部だけの例をあげる。

部分解配列Aの各要素と[]とのユニフィケーションをかんがえる。Aの要素はつぎのうちのいずれかだとする:

定数([]をふくまない)
[]
リスト・セル
変数

これらは(ポインタにつけられた)タグまたはポインタ値によって区別されるものとする。また、束縛変数と非束縛変数は、変数が指示するさきの値で区別される。[]とユニファイされる可能性があるのは[]と変数だけであるが、変数があらわれる確率はひくいと仮定する。すると、まず2個のベクトル比較をおこなって、つぎのようなマスク・ベクトルをつくればよい:

M1: ⇔Aの対応する要素は[]である;
M2: ⇔Aの対応する要素は変数である;

これと同時に、つぎのような整数値（スカラ）をもとめる。

m: =Aの要素のなかで変数であるものの数；

（S-810の場合は、マスク・ベクトルの要素のうちtrueであるものの数をかぞえる命令があるので、これをつかえばよい）。

そのあと、スカラ命令をつかって、m=0であるかどうかをしらべる。m=0なら、ユニフィケーションは完了である。すなわち、結果としてつぎのうちのいずれかがもとめられる：

（1）M1、すなわちAの各要素のユニフィケーションが成功したかどうかをしめすマスク・ベクトル（マスク演算方式の場合）
（2）M1のtrueである要素に対応するAの要素をつめあわせた配列（圧縮方式の場合）
（3）M1のtrueである要素に対応するAの要素をさすポインタからなるリスト・ベクトル（リスト・ベクトル方式の場合）

もしm≠0なら、M2をつかって変数である要素だけからなる配列をつくり、そのそれぞれの要素についてユニフィケーションをおこなう。すなわち、それが単純変数の場合は仮引数に束縛する。また、それが構造変数の場合には、その値とBとのユニフィケーションをおこなう。mが充分おおきければそのユニフィケーションを並列におこなうのがよいが、その場合は上記の手順を再帰的に適用すればよい。（この場合、各ステップでもとめられた結果の合成を、効率をおとさないように、また場合によっては部分解の順序をかえないように、注意ぶかくおこなわなければならない）。このようにすれば、部分解配列のなかに変数があらわれないときには、非常に高速にユニフィケーションをおこなうことができる。また、変数がわずかにあらわれる場合にも、その処理をベクトル演算でおこなうことによるオーバーヘッドをさけることができる。

## 6. 他の問題

ユニフィケーション以外の2つの問題についてのべる。

### 6.1. 部分解配列の縮小

OR並列方式あるいは並列バックトラック方式では、部分解配列の要素数が0になったときにバックトラックするが、0になるまでは要素の数はへる一方である。それが0にちかい状態で計算をつづけるとスーパー・コンピュータの並列性能をいかすことができない。そこで、はやめにバックトラックすることをかんがえる。

すなわち、部分解配列の要素数がある一定数をしたまわったときには、その部分解配列を保存してバックトラックする。あとでふたたびその点まで計算がすすんだときに、あらたにもとめられた部分解ともとのそれとをあわせてひとつの部分解配列にする。そして、それについて計算をすすめる、というようにするのがよいとかんがえられる（これは並列バックトラック方式における部分解配列の分割と逆の操作である）。この方式を「残留バックトラック方式」とよぶ。

しかし、残留バックトラック方式においてはバックトラック時に環境の一部を保存する必要が生じる。時間保存のオーバヘッドがおおきいと、残留バックトラック方式の利点をいかすことができない。だが、適当な環境保存法はまだ確立されていない。

### 6.2. Structure Sharing vs. Structure Copying

論理型言語の実行方式としてstructure sharing、structure copyingというふたつの方式がある。スーパー・コンピュータで並列実行する場合には、structure sharing方式では主記憶のバンク衝突を回避することができないとかんがえられるため、structure copyingは不可欠とかんがえられる。しかし、うまく最適化してコピーの回数をへらさなければ効率の低下をまねくであろう。また、並列実行ができない部分はstructure sharing方式で実行する、すなわち両方式をくみあわせてつかうということもかんがえられる。

しかし、いまのところ両方式をうまくくみあわせることができるかどうか、また、くみあわせたほうがよいかどうかはわかっていない。

## 7. むすび

この報告では、OR並列化の方式を中心としてスーパー・コンピュータによるPrologの並列実行方式についてのべてきた。そして、再帰よびだしのAND並列化、並列バックトラックによるOR並列化、リスト・ベクトル方式の採用、ユニフィケーションの方法、残留バックトラック方式などを提案してきた。

スーパー・コンピュータによるProlog実行に関する研究はまだこれからであり、以上の提案について、今後実際にProlog処理系を作成して有効性をたしかめていきたい。

謝辞

この研究のための時間と発表の機会をあたえてくださった日立製作所中央研究所の高橋栄主任研究員ほかの方々に感謝する。

## 参考文献

[1] 松田秀雄 他：K-Prolog（並列Prolog）の並列処理方式とその評価．情報処理学会第25回全国大会，7B-2，pp.245〜246

[2] 板敷晃弘 他：PROLOGの並列処理システム．情報処理学会第25回全国大会，7B-4，pp.24[illegible]〜25[illegible]

[3] Tick,E. and Warren,D.H.D.: Towards a Pipelined Prolog Processor, 1984 International Symposium on Logic Programming, IEEE Computer Society, pp.29〜40, 1984

[4] Tick,E.: A Multiple Pipelined Prolog Processor, Proceeding of the International Workshop on High Level Computer Architecture, pp.4.7〜4.17, 1984

[5] 田村良明 他：円周率 ―高速計算法と数値の統計性―，第25回プログラミング・シンポジウム報告集 p.1〜15

[6] 金田 泰：Nクウィン問題のベクトル計算機むき解法，情報処理学会第25回全国大会，5N-3，pp.1251〜1252

[7] 中島秀之：Prolog入門，産業図書，1983

[8] Hockney,R.W. and Jesshope,C.R.: 並列計算機，共立出版，1984（原書：Adam Hilger Ltd, 1981）


## 付録1 N-queen問題の並列解法（Fortran版）

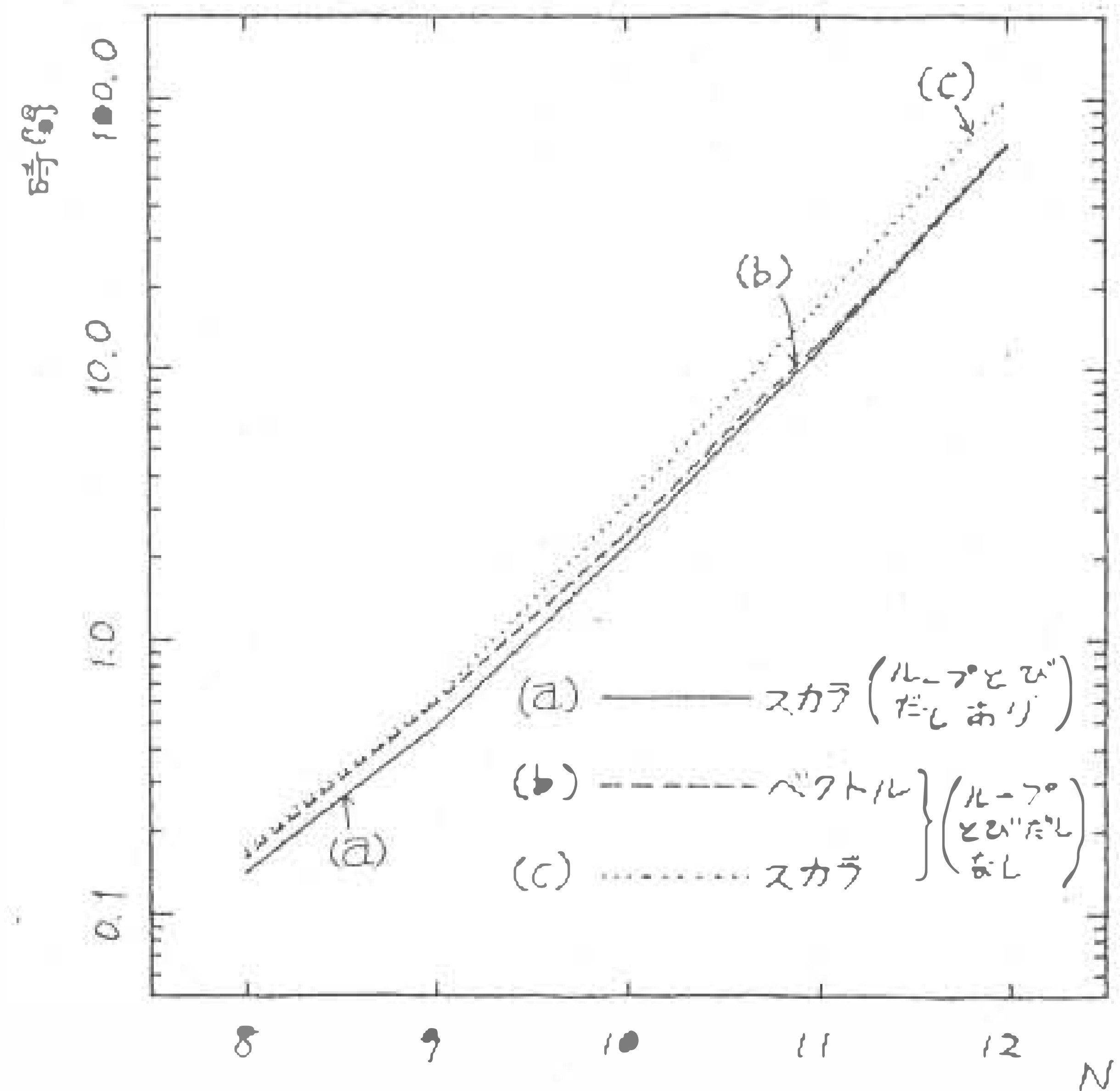
ここでは、金田[6]でしめさなかったデータを中心に、N-queen問題の並列解法などに関するいくつかのデータをしめす。

表S1は並列解法には関係ないが、N-queen問題の解の数をしめした（対称な解もすべてことなるものとしてかぞえている）。

8-queen問題の解を並列バックトラック解法にもとづいて（Fortranで）もとめるのに要する時間は、すでに図3〜4にしめした。ここでは、それをAND並列解法でもとめるのに要する時間をしめす（図S1）。並列化されているのは、第3章（3.1）〜（3.2）および図6にしめしたプログラムに相当する部分である。（すなわち、これから配置しようとしているqueenがすでに配置したqueenによってとられるかどうかをしらべる部分である。ただし、図6のプログラムとはややこ

表S1 Nクウィン問題の解の数

| N | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 解の数 | 0 | 0 | 0 | 2 | 10 | 4 | 40 |
| N | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 解の数 | 92 | 352 | 724 | 2680 | 14200 | 73712 | |



図S1 逐次版およびAND並列版の実行時間

表S2 並列解法におけるベクトル化率

（単位：%）

| N | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|
| AND並列解法 | 55.6 | 59.5 | 63.8 | 66.7 | 69.0 |
| OR並列解法 | 85.6 | 88.5 | 91.6 | 92.3 | — |

となっている）。曲線aは図6aに相当するプログラムをHitac S-810でスカラ実行したものである（S-810のスカラ演算はHitac M-280Hよりわずかにはやい）。曲線bおよびcは、それぞれ図6bに相当するプログラムをS-810でスカラ実行およびベクトル実行したものである。この図からわかることは、ループからのとびだしをなくしたプログラムのスカラ実行とベクトル実行をくらべるとベクトル実行のほうがわずかに高速だが、そのベクトル実行をループからのとびだしがあるプログラムの（スカラ）実行とくらべると、すくなくともN≦12の範囲では後者のほうが高速だということである。

表S2には、AND並列解法およびOR並列解法によるプログラムをS-810で実行する場合のベクトル化率をしめす。