記号処理プログラムのベクトル処理法と
その論理型言語プログラムへの適用

金山 裕

①

記号処理プログラムのベクトル処理法とその論理型言語プログラムへの適用

金田　泰

要 旨

Cray-1 を祖とするベクトル計算機は，従来，ほとんど数値計算専用につかわれていた．しかし著者は，HITAC S-810 などの第 2 世代ベクトル計算機がもっているおおくの機能は記号処理いいかえれば非数値処理に使用できることを確信した．それにもとづいて研究・開発された各種の記号処理のベクトル計算機における実行のための基礎技術について，この論文ではのべる．すなわち，第 1 に N クウィーン問題をはじめとするさまざまな解探索問題あるいは AI プログラムのベクトル処理を可能にする並列バックトラック技法についてのべる．第 2 に，記号処理に特徴的なリスト，木，グラフなどの可変長データ構造を処理する複雑なくりかえし制御構造を，くりかえし構造の交換やくりかえし構造の 1 重化などにもとづいてベクトル処理可能にするベクトル化法についてのべるとともに，リストのベクトル処理に必要な基本演算のベクトル処理法についてのべる．第 3 に，グラフや共有要素をもつリストや木などの共有部分があるデータの処理や，複数データのハッシュ表への登録のように同一データへの同時かきこみをおこなう可能性がある処理をベクトル処理するための方法についてのべる．

また，これらの基礎技術の応用として開発されたベクトル化法とそれによる実測結果とをしめす．すなわちまず，Prolog で代表される逐次論理型言語によって記述された解探索プログラムのためのベクトル化法についてのべる．このベクトル化法の適用によって，S-810 による N クウィーン問題のプログラムの実行において逐次実行の 9 倍程度の高速化を実現した．また，逐次論理型言語および GHC で代表される並列論理型言語による素数生成問題のプログラムのためのベクトル化法についてのべる．このベクトル化法の適用により，逐次実行の 3 ～ 4 倍の高速化を実現した．また，入出力データのモードが確定した解探索などの逐次論理型言語プログラムから高速なベクトル処理をおこなうプログラムを生成することができる自動ベクトル・コンパイラのプロトタイプを開発したが，その構造と機能とについてのべる．これらの論理型言語プログラムのベクトル化法は，たかい加速率をえようとすれば限定された範囲のプログラムしか変換できないが，今後の開発によってたかい加速率とひろい応用範囲とを両立させられるようになることが期待される．

さらに，これらのベクトル処理法において使用されているマルチ・ベクトルというデータ構造についてのべる．リストなどのポインタを使用したデータ構造をマルチ・ベクトルに変換することによって，広範な記号処理をベクトル処理可能にすることができる．

これらの研究成果により，ベクトル計算機の一部の記号処理への適用可能性を実証し，ベクトル計算機の応用範囲の拡大を実現することができたとかんがえる．また，これら

の技術は，パイプライン型ベクトル計算機にとどまらず，今後飛躍的に発展することが予想される Connection Machine CM-2 のような SIMD 型並列計算機に応用することができるとかんがえられる．

謝 辞

この研究に協力していただいた以下の方々に感謝する．

東京大学の 和田 英一 教授には，指導教官として，この論文を洗練するのをてつだっていただいた．

現日立製作所中央研究所第 7 部の 菅谷 正弘 氏には，著者とおなじ第 8 部に所属していたとき共同研究者としてこの研究に参加していただき，とくに論理型言語プロトタイプ処理系の開発に従事していただいた．小島 啓二 氏には，筆者とおなじ中央研究所第 8 部に所属し，この研究にも関連がふかい HITAC M-680H IDP などの研究に従事している立場から，さまざまな示唆をいただいた．また，*N* クウィーン問題解探索の M-680H IDP 用のプログラム開発にあたって著者と共同作業していただいた．現慶応大学環境情報学部の 安村 通晃 氏には，日立製作所中央研究所在勤時にこの研究に関してさまざまな示唆をいただくとともに，筆者がカーネギー・メロン大学に客員研究員として在勤時に，中央研究所における非数値ベクトル処理研究のテーマ・リーダを著者からひきついで研究をあらたな方向に発展させていただいた．Martin Nilsson 氏には，東京大学博士課程在学時にこの研究と関連のふかい FLENG Prolog の研究に従事し，それをつうじてさまざまな示唆をあたえていただいた．

また，現中央研究所副所長の 高橋 栄 氏には，この研究の開始時に中央研究所におけるユニット・リーダとしてこの研究を支援していただくとともに，その後ソフトウェア工場から支援していただいた．現日立製作所システム開発研究所第 3 部長の 吉住 誠一 氏には，中央研究所における研究テーマとしてこの研究が発足して以来，筆者が所属する第 8 部 692 ユニットのユニット・リーダとして，強力にバックアップしていただいた．さらに，現日立製作所常務の 武田 康嗣 氏，現中央研究所所長の 堀越 彌 氏，現中央研究所企画室の 千葉 常世 氏ほかの中央研究所の方々には，この研究を管理者の立場から支援していただいた．

最後に，妻 麗 はこの論文をまとめるにあたってさまざまな面でバックアップしてくれた．

目 次

第1章
序論

この章では，この論文における研究の背景と動機，研究の目的，論文の構成について順にのべる．

1.1 研究の背景と動機

この研究を開始した背景と動機はつぎの6点にまとめられる．

(1) 記号処理の高速化への要求のたかまり．
(2) ベクトル計算機の機能の“汎用機”化．
(3) ベクトル計算機をふくむ“汎用”計算機の専用計算機に対する優位性．
(4) 並列計算機より早期に並列記号処理を実用化できるという見通し．
(5) ベクトル計算機による記号処理技術のSIMD型並列計算機への応用の可能性．
(6) 記号処理プログラムのプログラム変換による並列化の可能性．

これらの点について，それぞれ以下でくわしくのべる．

1.1.1 記号処理の高速化への要求のたかまり

数値計算においては，最初の商用パイプライン型ベクトル計算機であるCray-1の登場以来，多少のプログラミング環境のわるさには目をつぶってでも高速性がもとめられてきた．ベクトル計算機がもつ桁違いの高性能とそれをもとめる数値シミュレーションの要求のために，数値計算用ベクトル計算機は着実に普及してきた．一方，記号処理においては，従来，数値計算ほど高速処理の要求がつよくなかったとかんがえられるが，記号処理においても高速性をもとめる応用がひらけてきた．膨大な計算時間が必要とされ，かつリアルタイム性が要求される機械翻訳はその代表といえるだろう．また，Prologのような論理型言語やLispの普及により，数値計算ほどではないにせよ，これらの記号処理用言語が高速性をもとめられる応用にしだいにつかわれるようになり，その高速実行がもとめられるようになってきた．

ベクトル計算機は現在のところ，ワークステーションにくらべればもちろんのこと，汎用大型計算機にくらべてもプログラミング環境上の不便がおおい．なぜなら，ベクト

ル計算機のプログラミングにはFortranなどの汎用のプログラミング言語をつかうことができるとはいっても，そのプログラミングのためには，しばしば機械指向の特殊なプログラミング技法がもとめられる．また，ベクトル計算機のコンパイル・コードは原始プログラムからはかけはなれたものになるためデバグも困難である．さらに，わりこみ処理がきめこまかくない，記憶が仮想化されていないなどの理由のためにTSSでの使用はできないかまたは不向きである．

このようなベクトル計算機の欠点にもかかわらず，上記のような理由によって，今後，ワークステーションや汎用機より画期的に高速なら記号処理においてもベクトル計算機の需要が生じるとかんがえられる．また，当初は実記憶でしかつかえなかったベクトル計算機にもその後仮想記憶が導入されるなど，上記の欠点の一部はすでに克服されつつあるという点もベクトル計算機の記号処理への応用にとって有利である．

1.1.2 ベクトル計算機とその機能の汎用機化

パイプライン型ベクトル計算機はもともと数値計算専用機として開発された．数値計算においては配列計算すなわちベクトル演算の高速化が非常に重要であり，それを演算器の高度なパイプライン化，ベクトル・レジスタの導入などによって実現したのがCray-1 やCDC Cyber 205を祖とするパイプライン型ベクトル計算機である．また，その成功に触発されて，HITAC M-180 IAPなど，汎用計算機の付加機構としての内蔵型配列処理機構(Integrated Array Processors)も開発された．これらの計算機は数値計算によくあらわれる配列計算に特化された演算装置をもっていた．

しかしパイプライン型ベクトル計算機は，数値計算むきというその基本的な特性をたもちながらも，現在ではスーパ汎用機とよんでもよいほどの多様なベクトル演算が可能な計算機に発展した．ベクトル計算機の発展過程をいわゆるスーパコンピュータと汎用機内蔵型配列処理機構とにわけて表1.1にまとめる．第2世代以降のベクトル計算機の演算命令のなかには，記号処理の基本演算もふくまれている．たとえば，HITAC S-800シリーズ，富士通VPシリーズ，日電SXシリーズなどのベクトル計算機はいわゆるリスト・ベクトル(間接指標ベクトル)のロード命令，ストア命令をもっている．これらの命令を使用することによって，ある種の記号処理をおこなうことが可能だとかんがえられる(第3章参照)．

これらの「汎用機」的機能は，Fortranプログラムにおいてベクトル演算が適用できる範囲を拡大するためにもうけられた．Fortranプログラムにおいては四則演算それもとくに配列要素に対する四則演算がもっとも重要だが，単純なDOループにおけるそのベクトル化の技術が確立されると，つぎには条件文のもとにある演算や，内積・一次巡回演算(first-order iteration)などの巡回性(recurrence)がある変数や配列の計算，不規則

に変化する添字をもつ配列すなわちリスト・ベクトルのベクトル化などが高速化の目標とされた．それは，ベクトル化率を90％以上の水準までたかめなければパイプライン型ベクトル計算機のもつ高速性を満足にひきだすことができないためである．これらの演算は数値計算においては最重要とはいえない演算だが，記号処理においてはもっとも重要である．すなわち，数値計算専用機が数値計算のより一層の高速化をもとめた結果として記号処理にも適用できる汎用性を獲得するにいたったということができる．

表1.1 ベクトル計算機の分類とその汎用機化のあゆみ

大分類	小分類	例	機能[0]			
			算・論[1]	条件[2]	関係[3]	ソート
スーパコンピュータ	第0世代スーパコンピュータ	STAR-100 (CDC) ASC (TI)	○	△	×	×
	第1世代スーパコンピュータ	Cray-1 (Cray, 1976) Cyber 205 (CDC, 1980)	○	△	×	×
	第2世代スーパコンピュータ	S-810 (日立 [Odaka 83]) VP-200 (富士通 [Hirakuri 83]) SX-2 (日電 [Furumasa 84])	○	○	×	×
	…					
汎用計算機付加機構	第1世代 内蔵配列演算機構 (IAP)	M-180 IAP (日立, 1978 [Horikoshi 83]) ACOS 1000 IAP (日電 [Osaka 82])	△	△	×	×
	…					
	現世代 内蔵配列演算機構	M-680H IAP (日立, 1986) 3090 Vector Facility (IBM, 1986)	○	○	×	×
	内蔵データベース処理機構 (IDP)	M-680H IDP (日立, 1986)	○[4]	○[4]	○	○

[0] 記法について：○機能あり，△限定的な機能あり，×機能なし．
[1] 算術・論理演算．
[2] 条件制御つき演算(マスクつき演算)．
[3] 関係演算(関係データベースに関する集合演算)．
[4] IAPとあわせて使用することにより，算術・論理演算，マスク演算も実行できる．

ベクトル計算機が記号処理に適用できるようになったのは，もうひとつの「汎用機化」のためでもある．もうひとつの「汎用機化」とは，主記憶スループットの増大である．すなわち，大量の配列データを高速に主記憶からロードし主記憶にストアするため，現在のパイプライン型ベクトル計算機は他にほとんど例をみないほど主記憶と処理装置間

のスループットがたかい．このようなたかい主記憶スループットは数値計算の応用においてもとめられたために実現されたものだが，記号処理は数値計算以上にその恩恵をうけることができる．なぜなら，数値計算の応用においては全計算時間のなかで当然のことながら四則演算がしめる割合がたかいが，記号処理においては主記憶間でのデータ転送がしめる割合がたかいからである．

ただし，ここでひとこと注意しておかなければならないが，汎用機化したとはいっても，くりかえし計算が存在することがベクトル・パイプラインを有効に利用するために必須であることにはかわりがない．この点は数値計算においても記号処理においてもかわらない．いいかえれば，ベクトル計算機で実行できる処理はベクトル処理にかぎられるということである．

ところで，ベクトル計算機の応用範囲拡大は，単に数値計算用ベクトル計算機の汎用化だけではなく，一方では記号処理専用のベクトル計算機の開発という形でもすすんだ．すなわち，関係データベース処理用のベクトル計算機 HITAC M-680H IDP (Integrated Database Processor) [Kojima 87, Kojima 90] などが開発された．M-680H IDP はデータベース処理のために機能が最適化された汎用計算機内蔵型のベクトル計算機である．すなわち，M-680H IDP においては関係の和，差，積などの関係演算やマージ・ソートなどをベクトル処理することができる．データベース処理用に最適化されてはいるが，他の各種の記号処理にも使用することができる機能をそなえている．たとえば，ベクトルの要素のなかである条件をみたすものを高速にフィルタする，汎用的な条件処理機能をもっている．

しかし，数値計算用ベクトル計算機の記号処理機能は，もともと数値計算を補足するために付加された機能であり，また記号処理分野ではまだ数値計算分野ほどは高速処理がもとめられていない．そのため，数値計算用ベクトル計算機は記号処理分野ではまだその能力は十分には注目されているとはいえない．また，記号処理用ベクトル計算機の応用もまだデータベース処理やソート・マージなどの一部の記号処理にかぎられている．すなわち，これらの汎用的な機能が十分にはいかされていないのが現状である．

1.1.3 "汎用" 計算機の専用計算機に対する優位性

MIT や ICOT (新世代コンピュータ技術開発機構) をはじめ，おおくの研究機関で記号処理専用のスカラ計算機の研究がさかんにおこなわれ，製品化されたものも Symbolics, Lambda，富士通の Alpha などの Lisp マシン，三菱の PSI などの論理型言語マシンなどがある．しかし，これらの高級言語マシンすなわち専用機にくらべて，RISC (Reduced Instruction Set Computer) にせよ CISC (Complex Instruction Set Computer) にせよ，またメインフレームにせよワークステーションにせよ，汎用機は専用機にくらべて現在のところ優位にたっているといってよいだろう．すなわち，汎用の機能をもった計算機あるいはそれに付加機構をつけるかまたは小規模の改造をくわえて記号処理の機能をもたせた M-680H IDP のような計算機は，専用機に適している一部の応用をのぞいてはむしろコスト・パフォーマンスがたかいといえるだろう．そのおもな理由は，汎用機のほうが生産台数がおおく，設計にも工数がかけられるからである．また，既存のソフトウェアが使用できるという点でも有利である．上記の記号処理専用機のなかには，商用機として一時的に世界を席捲し，たかい売上を記録したものもあるが，IBM System 360 / 370, DEC VAX, SUN Workstations などの特定のアーキテクチャが何世代にもわたって市場を支配している汎用機のばあいとはちがって，2 世代にわたってその記録を持続できたものは皆無である．

おなじ理由によって，まだ応用範囲のせまい記号処理用計算機よりは，すでにはばひろい応用がひらけている (また，すでに周辺のソフトウェアが作成ずみの) 数値計算用計算機のほうが優位にたっているということがいえる．

以上のような現実をみれば，記号処理専用の高速計算機を設計してそのソフトウェアをつくるよりは，既存の数値計算用高速計算機をそのまま，あるいは改良し，そのうえに記号処理用のソフトウェアを開発するほうが魅力的なアプローチだとかんがえられる．このことは，逐次計算機だけではなくベクトル計算機や並列計算機においてもおなじだとかんがえられる．すなわち，数値計算用ベクトル計算機といまだ存在しない記号処理専用ベクトル計算機，あるいは数値計算用並列計算機と記号処理専用並列計算機を比較するかぎりは，いずれのばあいも前者のほうが優位にたつものと予想することができる．

1.1.4 並列計算機とくらべての実用化時期のはやさ

記号処理の高速化をかんがえるばあい，並列計算機の利用が重要な選択枝であることはうたがいがない．しかし，並列計算機はハードウェア，ソフトウェアともに未解決の課題がおおい．数値計算分野では，すでに出荷され，実用に供されている並列計算機もすくなくない．だが，とくに記号処理分野では，研究はさかんにおこなわれているが，商用の並列計算機が安定して供給され，かつ実用に供されるようになるにはまだ時間がかかるとかんがえられる．

一方，数値計算用のベクトル計算機は，この研究を開始した 1984 年の時点ですでに実用段階にあり，ソフトウェアはまだ改良の余地がおおきいものの，ハードウェアは成熟している．そして，上記のように記号処理用の機能ももっている．したがって，ベクトル計算機のためのソフトウェア研究はいますぐ実機をつかって実験することができ，成功すればただちに実用化につなげることができるというおおきな利点がある．

1.1.5 SIMD 型並列計算機への応用可能性

ベクトル計算機は，Flynn の分類にしたがえば一種の SIMD (Single Instruction stream Multiple Data stream) 型計算機であり，マルチ・プロセッサによる SIMD 型並列計算機との共通点もおおい．したがって，そこで開発したソフトウェア技術のおおくは，将来，SIMD 型並列計算機が実用に供されるようになったときに応用できるとかんがえられる．現在研究され製品化されている並列計算機の大半は Flynn の分類にしたがえば MIMD (Multiple Instruction stream Multiple Data stream) 型であるが，研究レベルではイリノイ大学の Illiac IV，ICL の DAP (Distributed Array Processor)，IBM の GF11 などの SIMD 並列計算機が開発されており，さらには MIT / Thinking Machines 社の Connection Machine CM-1 / CM-2 [Hillis 85, Hillis 90] のように商業的にも成功したということができる SIMD 型並列計算機もあり，その将来は有望だとかんがえられる．

SIMD 計算機が有望だとかんがえられる理由はつぎのとおりである．MIMD 型並列計算機においては，プロセッサ間の同期・通信が基本的にソフトウェアによっておこなわれれる．そのため MIMD 型並列計算機においては同期・通信のオーバヘッドがおおきく，したがって，データベース処理やエキスパート・システムなどにおける記号処理におおいとかんがえられる，粒度のこまかい並列処理には適さない．これに対して，SIMD 型並列計算機においてはプロセッサ間の同期・通信が基本的にハードウェアでおこなわれる．したがって，同期・通信を高速に，かつ再現性がたかく (比較的デバグしやすく) 信頼性がたかい方法でおこなうことができるという利点がある．

1.1.6 プログラム変換による並列化の可能性

“日本製スーパコンピュータ” を中心とするパイプライン型ベクトル計算機ハードウェアのこの研究への影響については 1.1.2 節においてすでにのべたが，これらのためのコンパイラ技術もまたこの研究の強力な背景となっている．これらのベクトル計算機に対してはほとんど唯一の言語処理系として Fortran コンパイラがサポートされているが，これらのコンパイラは単純なコンパイルをおこなうだけではなく，逐次処理のかたちで記述されたプログラムをベクトル処理可能なかたちにベクトル化する，すなわちプログラム変換する [Yasumura 87]．すなわち，標準の Fortran に並列処理記述用の特殊な構文を追加した Illiac IV の Fortran などとはちがって，これらのコンパイラにおいては，プログラマは標準 Fortran によって，したがって逐次処理のかたちでベクトル処理のためのプログラムを記述することができる．この「プログラム変換による並列化」を，この研究においても基本的な戦略としている．

Prolog などの論理型言語や Lisp などで記述された記号処理プログラムにも，Fortran によって記述された数値計算プログラムと同様に，パイプライン化あるいは並列化可能

なくりかえし計算がふくまれているばあいがあるとかんがえられる．それらの計算は，Fortran におけるように比較的容易にベクトル計算機で計算できるかたちにプログラム変換できるわけではない．その理由としてはたとえば Fortran における DO ループのかわりにより複雑なくりかえし構造または再帰よびだしで表現されていること，Fortran にはないポインタをつかったデータ構造をつかっていること，記号処理プログラムは数値計算プログラムにくらべて不均質性がたかいことなどがあげられる[注1]．しかし，そのようなばあいでも Fortran プログラムのベクトル化をさらに発展させたプログラム変換をほどこすことによって，ベクトル計算機で計算できるようにすることができるばあいがあるとかんがえられる．そして，プログラム変換の適用範囲を，あらたな技術開発によって Fortran プログラムのベクトル化を応用できる範囲からさらに拡大できれば，ベクトル計算機による記号処理を実用化することができるとかんがえられる．このプログラム変換においては，とくに，不均質な構造を均質な構造に変換することが重要だとかんがえられる．

これらの処理は，変換をおこなうまえは不均質であるため，従来はパイプライン型ベクトル計算機や SIMD 型並列計算機による SIMD 型並列処理に不適であり，したがって SIMD 型並列処理の適用範囲もせまいとかんがえられてきた．現在の並列処理の研究の大半が MIMD 型並列処理にそそがれている理由もここにあるとかんがえられる．しかし，不均質な処理をそのまま並列処理しようとする MIMD 型のアプローチは，プログラミング，デバッグ，通信オーバヘッドなどのさまざまな面で困難にぶつかっている．上記のような不均質な構造をプログラム変換によって均質にすることができれば SIMD 型並列処理に適するようになり，したがってパイプライン型ベクトル計算機や SIMD 型並列計算機の応用範囲を拡大することができ，並列処理の限界を打破することができるとかんがえられる．

[注1] この「不均質性」に関しては 1.3 節でよりくわしくのべる．

1.2 研究の目的

この研究の目的は，ベクトル計算機の応用範囲拡大および記号処理・論理型言語実行のベクトル化による高速化という2点にまとめることができる．

1.2.1 ベクトル計算機の応用範囲拡大

この研究により，汎用機化したベクトル計算機を，数値計算だけでなく，解探索やリスト処理をはじめとする各種の記号処理の応用にも適用することを可能にし，計算の飛躍的な高速化をはかるソフトウェア技術を開発する．この開発により，ベクトル計算機がもつマスク演算機能をはじめとする各種の条件処理機能やリスト・ベクトル処理機能が，リスト処理をはじめとする各種の記号処理において使用することができ，かつそれによって高性能がえられることを実証することを目的とする．そしてさらには，パイプライン型ベクトル計算機，内蔵型ベクトル計算機だけではなくSIMD型並列計算機までふくめたベクトル計算機の応用範囲を拡大することを目的とする．いいかえれば，“汎用機化”したベクトル計算機が実際に記号処理にも応用できるという命題，極端にいえば「ベクトル計算機は記号処理も実行できる汎用計算機になる」という命題を実証することを目的とする．そして，この研究からのアーキテクチャへのフィードバックと応用範囲の拡大による計算需要の拡大とをつうじて，これらの計算機アーキテクチャを発展させることをめざす．当面は大規模な応用への適用はかんがえず，基礎技術の確立をめざす．

1.2.2 記号処理・論理型言語実行のベクトル計算機による高速化

1.2.1節でのべたことのうらがえしになるが，この研究は，高速処理への要求がたかまってきた記号処理，とくに論理型言語によって記述された記号処理プログラムの実行の高速化をベクトル計算機によって実現することを目的とする．とくに，高速化のためのユーザの負担を最小限にするために，自動ベクトル化をおこなう処理系を開発することを最終的な目的とする．すなわち，ベクトル計算機のアーキテクチャを意識せずにかかれたプログラムをベクトル計算機むきのプログラムに自動的にプログラム変換する処理系を開発する．

このような処理系においては，ベクトル計算機のためのFortranの処理系においてそうであるように，どのようなプログラムでもひとしく高速処理可能なプログラムに変換するというわけにはいかないとかんがえられる．すなわち，まずベクトル処理に適したある種のくりかえし処理があることが不可欠な条件であり[注2]，また，プログラミング・

[注2] ただし，Fortranにおけるのよりは柔軟なくりかえし構造をあつかえるようにすることが必須だとかんがえられる．

スタイルに制約が課せられたり，多少の特殊なユーザ・オプションが必要とされたりすることはさけられないとかんがえられる．しかし，これらの条件や制約は，それがうまく設定できたばあいには，汎用言語をつかわずにベクトル処理プログラムを記述するばあいに必要になる条件や制約にくらべれば，むしろとるにたりないものだということができる．

また，この研究は論理型言語プログラムのベクトル化の研究において開発された技術を，Lispやエキスパート・シェルなどをはじめとする他の言語による記号処理プログラムへも適用し，自動ベクトル化をおこなう処理系を開発するための基礎をきずくことをめざす．

1.3 論文の構成

この論文では，まず第2章で解探索のベクトル処理法について説明する．この論文における研究の原点は，解探索のベクトル処理法であるORベクトル計算法と並列バックトラック計算法を開発したことである．Nクウィーン問題などの解探索は，従来はバックトラックをつかって逐次計算されたが，これらのいずれかの方法によってベクトル処理することが可能になった．また，解探索のベクトル処理法はベクトル計算機の記号処理への応用のための基礎技術としてもっとも重要なもののひとつだとかんがえられる．解探索のベクトル処理法により，バックトラックによって形成される一種の不均質なくりかえし処理を，ベクトル計算機に適した均質なくりかえし処理に変換できるようになった．ここでバックトラックによって形成されるくりかえし構造は，くりかえし開始前にくりかえし回数がわかるなどの点で単純な構造をもったFortranのDOループにくらべればはるかに不均質性がたかい．なぜなら，どこでバックトラックが発生するかがコンパイル時には決定できないからである．

第3章と第4章とにおいては，解探索のベクトル処理に関する研究成果をもとにしておこなってきた論理型言語プログラムのベクトル化方法についてのべる．論理型言語プログラムであつかわれるもっとも重要なデータ構造はリストであり，そのベクトル処理の実現のために解探索のベクトル処理以外に，リスト処理基本演算のベクトル処理方法の開発，制御構造変換法の開発が必要だった．これらのうち，リスト処理基本演算のベクトル処理方法については第3章でのべる．ここでリスト処理基本演算とは，ベクトル化に不可欠なリストの分解 (car, cdr)・合成 (cons) などの基本演算のベクトル処理方法のことである．制御構造変換法については第3章においてのべたあと，第4章において一部の方法についてさらにくわしくのべる．制御構造変換法はプログラムの制御構造に1重化，交換などの変換をおこなうことによってベクトル化可能にする方法であり，第1章でのべた解探索の逐次処理法とベクトル処理法とをつなぐ方法でもある．ここでしめす制御構造変換法により，ポインタをつかってつくられた可変長のリストや木などの不均質性があるデータ構造の処理を，ベクトル計算機に適した均質な処理に変換することができるようになった[注3]．第3章では制御構造変換にもとづいてリスト処理をベクトル化する方法についてのべる．第4章では制御構造変換の一種であるくりかえし構造交換のための2つの方法の比較評価をおこなう．

解探索のベクトル処理および論理型言語プログラムのベクトル化においては現在のところ使用されていないが，この研究で開発されたもうひとつの重要な基礎技術として，共有部分があるデータのベクトル処理法である上書きラベル・フィルタ法がある．これ

[注3] これらのデータ構造は可変長であり，構造も一定でないばあいがあるという点で「不均質」であるといえる．

は，ハッシングのベクトル処理のこころみのなかから開発された方法である．ハッシングのベクトル処理を研究した理由は，解探索のベクトル処理において必要なばあいがある構造データの複写を効率よくおこなうために，ハッシングが必要とかんがえられたからである．構造データの効率的な複写方法はいまだ完成していない．しかしながら上書きラベル・フィルタ法の応用は，ハッシングや解探索にかぎらない．これはグラフ・DAG (Directed Acyclic Graphs) あるいはある種のリストや木などのように複数の要素からささされた共有要素をもつさまざまな不均質な構造データのベクトル処理を可能にする方法であり，ベクトル記号処理の応用範囲を拡大するうえで重要だとかんがえられる[注4]．第5章では，この上書きラベル・フィルタ法についてのべる．

第6章および第7章では，制御構造変換法などの基礎技術の応用としての論理型言語プログラムのベクトル化法をしめす．第6章では，この研究の当初の最大の目的であった解探索のベクトル処理プログラムの自動生成をめざした，OR並列化を基本とする逐次論理型言語のベクトル化法をしめす．第6章でのべるベクトル化法はまだ完成されたものとはいえず，限定された範囲にしか適用されない．しかし，その限定された範囲においては形式化されている．すなわち，自動ベクトル化が可能になっている．第7章ではAND並列化を基本とする論理型言語のベクトル化法をしめす．この方法はいまのところ形式化されていないため，適用範囲も明確化されていないが，第7章では素数生成というストリーム処理の逐次論理型言語プログラムと並列論理型言語プログラムの両方のベクトル化をおこない，評価している．第7章では，AND並列化をおこなうばあいのリスト処理基本演算のベクトル化法についてもふれる．

この研究の最終的な目的は自動ベクトル化をおこなう処理系の開発であるが，第8章では，第7章の方法にもとづいて試作した自動ベクトル化をおこなう論理型言語の処理系の構造と中間語の仕様についてのべる．この処理系はベクトル化できるプログラムの範囲が非常にかぎられているため実用にはならないが，論理型言語自動ベクトル化のための一歩をふみだしたものということができる．

第9章では，論理型言語プログラムのベクトル化の研究のなかからみいだされた，ベクトル記号処理において重要な役割をはたすデータ構造であるマルチ・ベクトルの応用，機能，操作法についてのべる．

最後に，第10章で上記のすべての研究成果をまとめる．

なお，この論文の第1章は金田 [Kanada 85] を参照してあらたに記述した．第2章は金田ら [Kanada 88b] を下敷きにし，鳥居ら[Torii 88a] および鳥居ら [Torii 88b] を参照して記述した．第3章は金田ら [Kanada 89b] を下敷きにして記述した．第4章はまったく

[注4] このばあい，各要素データは，共有されていたりされていなかったりするという点において「不均質」であるといえる．

あらたに記述した．第5章は金田ら [Kanada 90a]，金田 [Kanada 91a] を参照して記述した[注]．第5章の内容をもとにした論文を投稿中 [Kanada 91b] である．第6章は金田ら [Kanada 89a] を下敷きにし，金田ら [Kanada 88a]，金田ら [Kanada 89c] および金田 [Kanada 87] を参照して記述した．また，第6章をもとにして金田 [Kanada 91c] を記述している．第7章は金田ら [Kanada 90b] をもとにして，大幅にかきかえた．第8章はあらたに記述した．第9章および第10章もまったくあらたに記述した．第9章に関しては，その内容をもとにして金田ら [Kanada 91b] を執筆した．ただし，既存の論文をもとにして記述した部分でも，ほとんどの図はあらたに記述した．また，あらたに記述した部分もふくめて，各部分を日立製作所中央研究所の研究報告に記載している．

[注] 第5章に記述されたプログラムは金田ら [Kanada 90a] および金田 [Kanada 91a] に記述されているものとほぼおなじである．

第2章
解探索のベクトル処理

要旨

この章では，探索問題に適用することができる，ベクトル計算機むきのあたらしい計算法「並列バックトラック計算法」をしめす．この方法にしたがってFortranでプログラムを記述すれば，数値計算専用とかんがえられていたS-810のようなスーパコンピュータや，M-680H IAP/IDP (内蔵型アレイ・プロセッサ / 内蔵型データベース・プロセッサ) のようなベクトル計算機構を付加した汎用計算機で広範囲の探索問題を高速に実行することができる．またこの方法では，並列度を適切に制御することによって，必要な記憶量を逐次計算法とひとしいオーダにおさえることができる．

この計算法を N クウィーン問題に適用し，つぎのような実行性能をえた．逐次処理にくらべて，エイト・クウィーンの全解探索においてはS-810を使用して約9倍，M-680H IAPおよびIDPを使用して約2倍だった．また，S-810においては $N \geq 14$ のとき単解探索でも逐次処理より高速に実行することができた．

これによって並列バックトラック計算法の有効性がたしかめられるとともに，ベクトル計算機S-810および M-680H IAP/IDP の記号処理への適用可能性がしめされた．また，並列バックトラック計算法は，Prologのような論理型言語のベクトル計算機による高速実行の可能性を示唆している．

2.1 はじめに

パイプライン型ベクトル計算機 (以後は単にベクトル計算機とよぶ) は，ベクトル (配列) の各要素に同一の演算をパイプライン的に実行する演算機構をもうけることによって，ベクトル計算を高速に実行できるようにした計算機である．

ベクトル計算機のもっとも重要な族として，Cray-1 を祖とするパイプライン型スーパコンピュータ (以後は単にスーパコンピュータとよぶ) がある．スーパコンピュータは，数値計算専用機として発展してきた．その結果，Fortran プログラムを部分的にスーパコンピュータむきにかきかえることにより，おおくの数値計算プログラムを超大型汎用計算機にくらべて 10 倍以上高速に実行することができる．

第 1 世代のスーパコンピュータはベクトルとスカラに関する単純な四則演算以外の命令をほとんどもっていなかった．ところが，HITAC S-810 [Odaka 83]，FACOM VP-200 [Hirakuri 83]，日電 SX-2 [Furumasa 84] などの第 2 世代のスーパコンピュータにおいては，大幅な機能の拡張がおこなわれた．すなわち，マスク演算命令，リスト・ベクトル命令，ベクトル圧縮・伸長命令など，条件文のもとの数値計算や複雑な配列添字の計算などに使用される命令がレパートリにくわえられた [Hirakuri 83]．そのため，通常の Fortran プログラムからスーパコンピュータむきのプログラムへの自動変換すなわちベクトル化をおこなうベクトル・コンパイラとあいまって，応用範囲の拡大とベクトル化率の向上による高速化が達成された．

一方，ベクトル計算機のもう 1 つの族として，汎用計算機の付加機構としての内蔵型アレイ・プロセッサ (Integrated Array Processor) がある．すなわち，HITAC M-180 IAP [Horikoshi 83] から M-680H IAP にいたる一連の計算機である．日電 ACOS 1000 IAP [Osaka 82]，IBM 3090 VF (Vector Facility) [Buchnolz 86] などもこの族のなかにくわえることができる．これらの内蔵型アレイ・プロセッサもまた，数値計算専用機として発展してきたが現在では大幅に機能が拡張されているという点は，スーパコンピュータのばあいと同様である．

さらに，汎用計算機の付加機構としては，関係データベースやソーティングの高速処理を目的とした M-680H IDP (内蔵型データベース・プロセッサ Integrated Database Processor) [Torii 87, Kojima 87] が開発されている．上記のベクトル計算機とはちがって IDP は記号処理を目的としているが，関係データベース処理に適したかぎられたデータ形式 (デュアル・ベクトル) だけをあつかうことができる．

このように，いまのところベクトル計算機は，関係データベース，ソーティング [Store 78, Brock 81, Roensch 87, Ishiura 88]，論理シミュレーション [Nagashima 86, Ishiura 86] などをのぞけば，リスト処理で代表される本格的な記号処理には使用されるにいたっていない．しかしそのおもな理由は，ベクトル計算機むきの記号計算のプログ

ラミング方法およびベクトル化 (プログラム変換) 方法が開発されていなかったためであり，ベクトル計算機の機能不足のためではない．なぜなら，現在のベクトル計算機は，すでに本格的な記号的ベクトル処理に必要なベクトル命令をほぼひととおりそなえているからである．この研究の目的は，第 1 に，ある種の記号処理においてはベクトル計算機を適用することによって高速化がはかれることをしめすことである．そして第 2 に，高速化の対象としてえらんだ，探索問題という，記号処理のなかでも 1 つのもっとも重要な分野の問題の，ベクトル計算機で実行するのに適した計算方法を開発することである．

この章では，まず 2.2 節で従来からある探索問題の逐次的な計算方法であるバックトラック計算法についてのべるとともに，この方法によるプログラムをベクトル計算機で実行しようとするばあいの問題点をしめす．2.3 節ではバックトラック計算法によるプログラムに最小限の変更をくわえてベクトル処理する AND ベクトル計算法をしめすが，この方法では飛躍的な性能向上はえられない．そこで，2.4 節では上記の問題点を解決したベクトル計算機むきの計算法である OR ベクトル計算法についてのべるが，まず 2.4.1 節では OR ベクトル計算法のもっとも純粋なかたちである完全 OR ベクトル計算法についてのべるとともに，その問題点をしめす．つぎに 2.4.2 節では，その問題点をも解決した計算法である並列バックトラック計算法についてのべる．2.5 節では，並列バックトラック計算法にしたがって記述したプログラムの S-810 および M-680H IAP および IDP を使用したばあいの実行時間を，(逐次) バックトラック計算法によるプログラムの実行時間と比較するとともに，考察をくわえる．2.6 節では，N クウィーン問題においては必要がなかった並列バックトラック計算法におけるベクトル長増大のための方法についてのべる．

2.2 逐次バックトラック計算法

この節ではまず，N クウィーン問題を例として，探索問題をとく際に従来からつかわれてきた逐次計算法であるバックトラック計算法についてかんたんにのべる．並列バックトラック計算法と明確に区別するため，以後は「逐次バックトラック計算法」とよぶ．そしてつぎに，逐次バックトラック計算法にしたがって記述されたプログラムをそのままベクトル計算機で実行しようとするときに生じる問題点をしめす．

N クウィーン問題は，パズルの一種であるエイト・クウィーン問題を一般化したものである．すなわち，$N \times N$ のおおきさの「チェス・ボード」に，N 個のクウィーンを，どの2つもおなじ行，列，および対角線方向にないように配置する問題である．図2.1にエイト・クウィーン問題の92個の解のうちの1つをしめす．また，表2.1に N クウィーン問題の解の数をしめす．エイト・クウィーン問題は代表的な探索問題であり，LispやPrologなどのベンチマーク・プログラムとして，はばひろく使用されている[Okuno 84].

エイト・クウィーン問題の解法に関してはおおくの研究がおこなわれている．その解法はたとえばBitner and Reingold [Bitner 75]，Dijkstra [Dijkstra 72]，Floyd [Floyd 84] にのべられている．Bitner and ReingoldとDijkstraのプログラムは逐次バックトラック計算法にもとづいている．また，Floydのプログラムは，非決定性プログラミング法 [Floyd 84] にもとづいていて，Prologによるエイト・クウィーンのプログラム [Okuno 84] のもとになっているといえる．Floydのプログラムもその実現手段としては逐次バックトラック計算法がつかわれる．

逐次バックトラック計算法にもとづくエイト・クウィーン問題の全解探索 (すべての解をもとめること) の解法をしめす．解は8個の整数からなるリスト $(x_1, x_2, \ldots, x_8)$ $(0 \leq x_i \leq 7)$ で表現される．各 x_i が第 i 列第 x_i 行におかれたクウィーンをあらわす．たとえば図2.1にしめした解は $(3, 1, 7, 2, 5, 7, 0, 4)$ とあらわされる．

図2.2がそのアルゴリズムである．図2.2のプログラムは $x_1, x_2, \ldots, x_8$ をこの順に決定していく．x_i を決定する際には，x_i の値をかりにさだめて，安全性チェックをおこなう．すなわち，$x_1, x_2, \ldots, x_{i-1}$ があらわすすでに盤面におかれたクウィーンが，x_i があらわすクウィーンと同一の行にないかどうか，また対角位置にないかどうかをチェックする (すなわち，$x_i \neq x_j$, $x_i \neq x_j \pm (i-j)$ $(0 \leq j \leq i)$ がなりたつかどうかをしらべる)．もしその条件がみたされないときは，x_i の値をかえて，つぎの候補をさがす．x_i のすべての候補をつくしたときは，x_{i-1} の値をかえて，つぎの候補をさがす．図2.2のプログラムではこのようにしてエイト・クウィーンの全解探索をおこなう．なお，付録1に図2.2のアルゴリズムをFortranでコーディングした例をしめす．

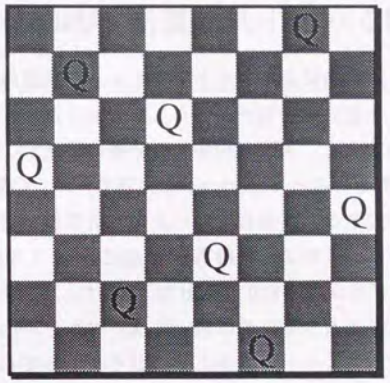


図2.1　エイト・クウィーン問題の解の一例

表2.1　N クウィーン問題の解の数 ($N \leq 13$)

N	1	2	3	4	5	6	7	8	9	10	11	12	13
解の数	0	0	0	2	10	4	40	92	352	724	2680	44200	73712

```
B := emptyChessBoard;                    —Bを空のチェス・ボードとする．
QUEEN(B, 1);
    — 手続き QUEEN をよび，エイト・クウィーン問題の解をもとめて印刷する．

procedure QUEEN(B: chessBoard; x: row) is
    if x>8 then
        PRINT(B);                        —すべてのクウィーンがおかれたので印刷する．
    else
        for y in 1..8 loop
            if not TAKEN(B, x, y) then   —B上の点(x,y)にクウィーンをおいても
                                         —ほかのクウィーンにとられないなら，
                PUT_QUEEN(B, x, y);      —点(x,y)にクウィーンをおく．
                QUEEN(B, x+1);           —のこりのクウィーンをおき，解を印刷する．
                REMOVE_QUEEN(B, x, y);   —点(x,y)のクウィーンをとりのぞく．
            end if;
        end loop;
    end if;
end QUEEN;
```

図2.2　エイト・クウィーン問題の逐次バックトラック計算法

2.3 AND ベクトル計算法 (AND 並列計算法)

逐次バックトラック計算法によるエイト・クウィーン問題のプログラムでもっとも計算時間がかかるのは，上記の安全性チェック，すなわち図 2.2 のプログラム上では関数 *TAKEN* の実行である (ただし，関数 *TAKEN* の定義は図 2.2 にしめしていない)．関数 *TAKEN* は，多少の注意をはらって Fortran で記述すれば，自動ベクトル・コンパイラでベクトル化することができる．すなわち，ベクトル計算機で実行できるようにプログラム変換することができる．このような計算法を AND ベクトル計算法とよぶことにする[注1]．図 2.3 にこのプログラムの S-810 Model 20 における実測結果をしめす．この図からわかるように，すくなくとも S-810 のばあいには，ベクトル化後のプログラムの実行速度はベクトル化前のそれにくらべて，かえっておそくなってしまう．それは，ベクトル長がみじかい (すでに盤面に配置されたクウィーン数以下) うえ，逐次実行にくらべてむだな計算がふえるからである．

以下，このプログラムとその実行結果についてよりくわしく解析する．図 2.2 のプログラムにおける関数 *TAKEN* においては，これからおくべきクウィーン x_i と，すでにおかれたクウィーンのそれぞれ x_j とが条件 $x_i \neq x_j$, $x_i \neq x_j \pm (i-j)$ をみたすかどうかをチェックする．したがって，すでにおかれたクウィーンの数だけのくりかえしが最内側ループとなる．しかも，条件をみたさないことがループの途中でわかれば，以後のくりかえしは実行する必要がないので，**goto** 文でループ外に脱出することになる．したがって，エイト・クウィーンのばあいで平均ループ長はわずか 4 程度になる．

このプログラムはループ外への脱出をなくせばベクトル化することができる．S-810 の Fortran コンパイラではこの変換は自動的におこなわれる．しかし，ループ外脱出をなくすことによってむだな計算がふえるうえ，もともとループ長がみじかいためにベクトル長がみじかい．したがって，図 2.3 にしめしたように，ループ外脱出があるプログラムと脱出をなくしてベクトル化したプログラムとをくらべると，ベクトル化によってかえって実行時間は増加してしまう．したがって，AND ベクトル計算法は *N* クウィーン問題のプログラムをベクトル化する方法としてはつかえない．

[注1] AND ベクトル計算法というなまえの由来は Prolog によって記述したエイト・クウィーン問題のプログラムにおいて AND 関係 (第 3 章参照) となる計算を並列化してベクトル処理していることによる。このなまえについては次章でさらにのべる．

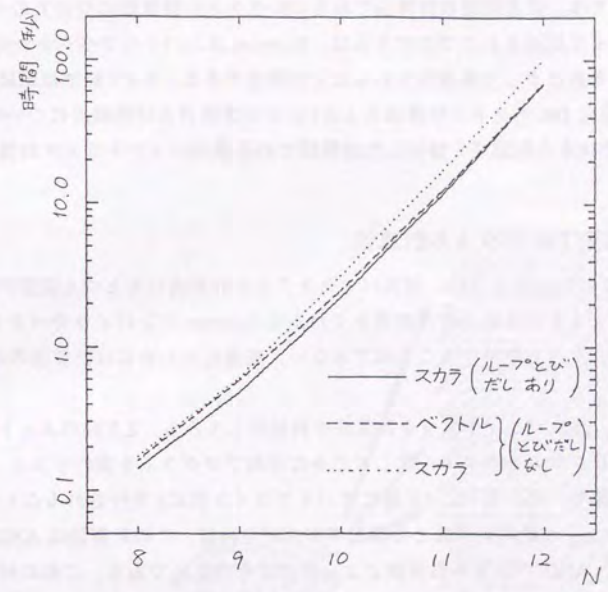


図 2.3　AND ベクトル計算法による *N* クウィーンの相対実行時間
(S-810 旧 **Fortran** コンパイラ)

2.4 OR ベクトル計算法 (OR 並列計算法)

この節では，探索問題の計算法である OR ベクトル計算法についてのべる．この方法にしたがって記述されたプログラムは，Fortran コンパイラでベクトル化することができ，かつそれによって高速化されることが期待できる．2.4.1 節では単純な OR ベクトル計算法 (完全 OR ベクトル計算法とよぶ) とその性能および問題点についてのべる．また，2.4.2 節ではその問題点を解決した計算法である並列バックトラック計算法についてのべる．

2.4.1 完全 OR ベクトル計算法

2.3 節でわかったように，逐次バックトラック計算法にもとづく探索問題のプログラムをそのままたは最小限の変更をくわえて Fortran コンパイラでベクトル化しても，十分な高速実行を期待することはできない．高速化のためには計算法のみなおしが必要である．

そこで，逐次バックトラック計算法を再検討してみる．2.3 節のエイト・クウィーン問題のプログラムをベクトル化してできた目的プログラムを実行すると，1 つの解をもとめる計算の一部が並列に (正確にはパイプライン的に) 実行されることになる．したがって，Prolog の並列処理法との類比でかんがえれば，この計算法は AND 並列処理法に相当する．AND ベクトル計算法とよんだのはそのためである．これに対して，ことなる解をもとめる計算を並列に実行する方法をかんがえることができる．この計算法は OR 並列処理法に相当する．したがって，この計算法を OR ベクトル計算法とよぶ．

OR ベクトル計算法を，エイト・クウィーン問題の全解探索を例として，逐次バックトラック計算法と比較しながら説明する．すでにのべたように，逐次バックトラック計算法では，探索の進行にともなって選択枝が生じるたびに，そのうちの 1 つをえらんで実行する．そして，その選択枝が失敗するとあともどり (backtrack) して，ほかの選択枝をあらためて実行する．これに対して OR ベクトル計算法では，すべての解候補の集合の各要素に対して並列に計算をすすめる．エイト・クウィーンのばあいには，解候補はクウィーンがのせられたチェス・ボードである．解候補および解候補の集合のデータ表現として配列を使用することによって，ベクトル計算機による並列処理 (パイプライン処理) を可能にする．OR ベクトル計算法によるエイト・クウィーンの計算過程の概要を図 2.4 にしめす．選択枝が生じるたびに，その数だけ解候補を複写してそれぞれにことなる選択の結果を反映し，それらのあたらしい解候補の全体からなる集合をつくる．そして，つぎの計算ステップはその集合全体に対して並行しておこなう．

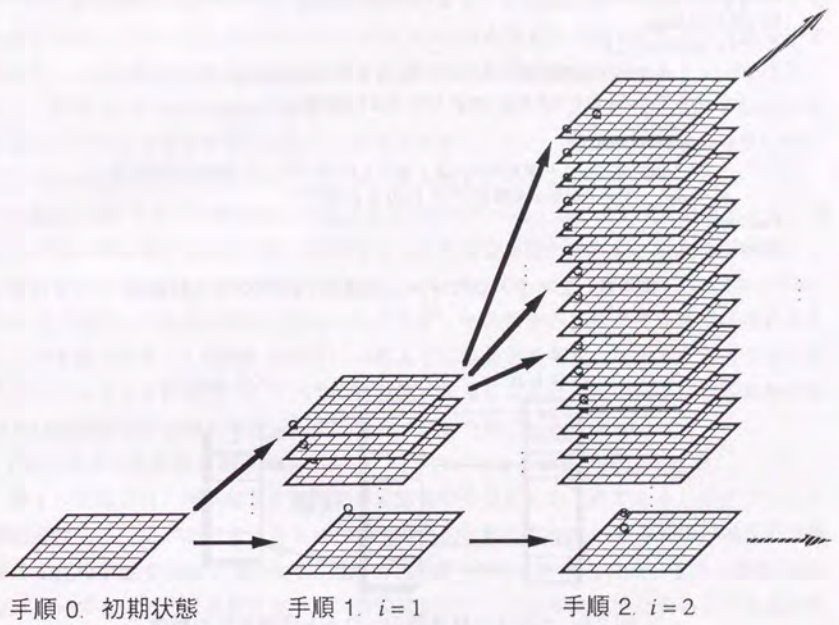


図 2.4　OR ベクトル計算法による計算過程の概要

図2.5にアルゴリズムの概要をしめす．図2.5のプログラムでは，まず変数Cに空のチェス・ボードからなる集合を代入する．そして，第1列から順に8個のクウィーンをおいていく．このプログラムにおいては，最初の代入文と関数*nextCandidate*とが生成検査法 (Generate-and-test Method) における生成 (ただし部分的な解候補の生成) のはたらきをし，関数*nextRow*が検査のはたらきをしている．

```
C := {emptyChessBoard};          —C を1個の空のチェス・ボードからなる集合(ベクトル)とする．
for i in 1..8 loop
   C1 := nextRow(C, i);
      —C の各要素の第 i 列におくことができるクウィーンの番号 x とチェス・ボード b
      — との対 ⟨x,b⟩ からなる集合(ベクトル)を C1 とする．

   C := nextCandidate(C1);
      —C1 の各要素 ⟨x,b⟩ におけるチェス・ボード b にクウィーンをおいたあらたな
      —チェス・ボードからなる集合(ベクトル)を C とする．
end loop;
PRINT(C);
```

図2.5　エイト・クウィーン問題のORベクトル計算法

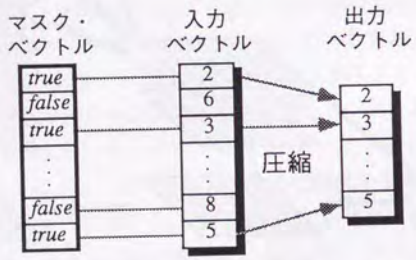


図2.6　ベクトル計算機のベクトル圧縮命令の動作

関数*nextRow*(C, i)はつぎのような機能の関数である．*nextRow*の入力は，第1～$i-1$列にクウィーンがおかれたチェス・ボードの集合Cと，つぎにクウィーンをおくべき列の番号iとである．また，その出力すなわち関数値は，Cの各要素の第i列におくことができるクウィーンの番号x $(0 \le x \le 7)$とチェス・ボードb $(\in C)$との対$\langle x, b\rangle$からなる集合である．すなわち，安全性チェックをおこない，それをみたす解候補を生成するのに必要なデータ$\langle x, b\rangle$を出力する．なお，同一のチェス・ボードbが複数回使用される可能性があるが，この段階ではbの複写はおこなわず，同一のデータをくりかえし使用する．また，関数*nextCandidate*(rb)は上記の対$\langle x, b\rangle$を入力し，チェス・ボードbにクウィーンxをおいたあらたなチェス・ボードからなる集合を値とする関数である．チェス・ボードbは複数の対に使用されている可能性があるので，クウィーンxをおくまえに複写する必要がある．

図2.5のプログラムで実行比率がたかい部分は関数*nextRow*(C, i)と関数*nextCandidate*(rb)とである．このうち関数*nextRow*(C, i)は，第2世代のベクトル計算機においてはベクトル圧縮命令をつかうことによって高速に実行することができる．ベクトル圧縮命令とはベクトルの要素のうち条件をみたすものだけを選択し，圧縮されたベクトルを出力する命令である(図2.6参照)．また，M-680H IDPには，$\langle x, b\rangle$のような対を要素とするベクトル(デュアル・ベクトル)を条件にしたがって圧縮する命令が用意されている[Kojima 87, Kojima 90]ので，やはり上記の機能を実現することができる．また，関数*nextCandidate*(rb)の機能はベクトル計算機がもつベクトルのロード命令およびストア命令を使用することによって実現できる．なお，付録1に図2.5のアルゴリズムをFortranでコーディングしたプログラム例をしめす．

つぎに，ORベクトル計算法の性能予測についてのべる．ORベクトル計算法では，選択点ごとに解候補を選択枝の数だけ複写しなければならないという，逐次バックトラック計算法にはないオーバヘッドがある．したがって，ORベクトル計算法のプログラムをスカラ実行したばあいには，逐次バックトラック計算法のプログラムより低速になることが予想される．しかし，そのオーバヘッドにもかかわらず，ベクトル実行すれば逐次バックトラック計算法にくらべて実行時間は高速になりうる．それは，各解候補の計算をパイプラインにのせて短ピッチで計算できるためである．

ORベクトル計算法は高速ではあるが，つぎのような2つの問題点がある．

第1の問題点は，解候補数に比例する記憶量が必要だという点である．Nクウィーン問題のばあい，Nがおおきくなるにつれて指数的に解の数が増加するので，現在の計算機の主記憶容量ではNが15程度以上のとき計算不能になってしまう．また，計算可能なばあいでも，汎用計算機においては解候補がキャッシュや実記憶からあふれるために計算速度が低下する．

第2の問題点は，ORベクトル計算法は単解探索すなわちただ1つ解をもとめるばあいに不向きだという点である．すなわち，すべての解を並列にもとめるため，単解探索のばあいにはむだな計算がおおくなる．

2.4.2 並列バックトラック計算法

2.4.1節でのべたORベクトル計算法の2つの問題点を解決するために考案したのが，この節でのべる並列バックトラック計算法である．

一般にベクトル計算機においては，ベクトル長が十分にながければ逐次計算機より1桁程度高速に計算することができるが，ベクトル長が2～10以下ではベクトル計算の準備などのオーバヘッドのために逐次計算機よりかえって低速になる(クロス・ポイント

は機種によってことなる)．ベクトル長がこれよりながくなるとしだいに性能が向上し，ベクトル長が64 ～ 1000程度でほぼピークに達する．そして，それ以上のベクトル長では，ベクトルの1要素あたりの実行時間は一定か，またはキャッシュのヒット率低下のためにかえってながくなる．

したがって，ORベクトル計算法においてベクトル長が十分にながくなったときは，つぎのようにするのがよいとかんがえられる．まず，ベクトルを2個以上に分割して，そのうちの1つについて計算を続行する．そして，その計算がおわったあとであともどりして，のこりのベクトルに関する計算をおこなう．この計算法を並列バックトラック計算法とよぶ．並列バックトラック計算法によるエイト・クウィーンの計算過程の概要を図2.7にしめす．

並列バックトラック計算法によるNクウィーンの全解探索のアルゴリズムを図2.8にしめす．図2.8のプログラムでは，解候補の数$|C^i|$が定数*ulim*をこえたとき，解候補の集合(ベクトル)を2個に分割する．解候補の集合の分割の方法としてはより巧妙な方法もかんがえられる．各種の分割法の比較はおこなっていないが，図2.8の分割法は単純なわりには比較的よいとかんがえられる．たとえば一度に3個以上に分割すると，平均ベクトル長が必要以上にみじかくなるので，2分割のほうがこのましいとかんがえられる．なお，図2.8のアルゴリズムをPascalとFortranとでコーディングした例を付録1にしめす．

並列バックトラック計算法においては，2.4.1節でのべたORベクトル計算法の2つの問題点はつぎのように解決されている．

第1の問題点すなわち解の候補数に比例する記憶が必要だという点については，つぎのようにいえる．並列バックトラック計算法で必要な記憶量は，集合の分割が頻繁におこなわれるばあいには，Nに関して指数的にふえる解の候補数に依存しない．ただし，記憶量は*ulim*にほぼ比例するので，全解探索のばあい，十分な高速化のために*ulim*を数100程度とすると，逐次バックトラック計算法のばあいにくらべて2桁程度おおい記憶量が必要である．

第2の問題点すなわち単解探索に不向きだという点については，つぎのようにいえる．単解探索のばあいは，最初の解がもとめられた時点(印刷がおわった直後)で計算をうちきるようにすれば，すべてではないにせよ，かなりのむだな計算をへらすことができる．たとえば，Nクウィーンの逐次実行においては，1つの解をもとめるだけでもかなりの量のバックトラックをくりかえすことが必要だから，単解探索においても逐次バックトラック計算法より高速に実行できる可能性がある．

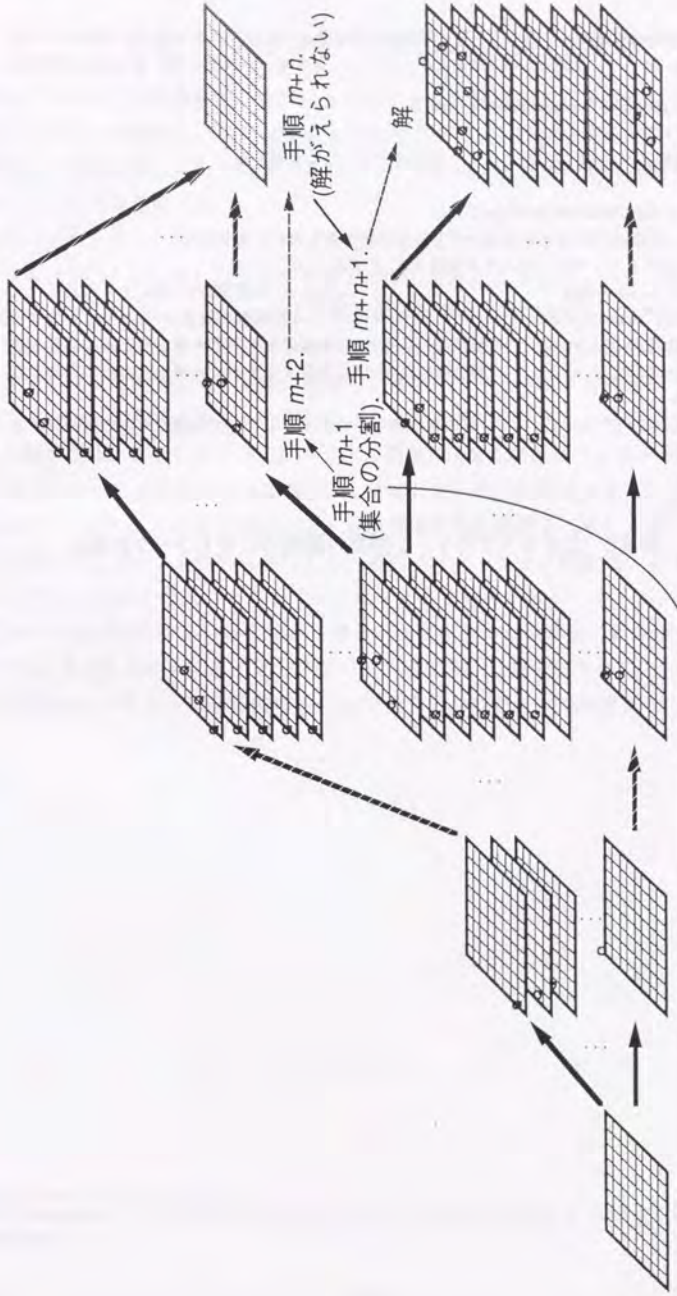


図2.7 並列バックトラック計算法による計算過程の概要

```
B := {emptyChessBoard};                    —Bを1個の空のチェス・ボードからなる集合とする.
QUEEN(B, 1);    — 手続き QUEEN をよび，エイト・クウィーン問題の解をもとめて印刷する.

procedure QUEEN(C : chessBoard; x: row) is
  if x>8 then
    PRINT(C); —すべてのクウィーンがおかれたので印刷する.
  else
    C' := nextCandidate(nextRow(C, i));
      —Cがふくむチェス・ボードbにクウィーンをおいたあらたな
      —チェス・ボードからなる集合をC' とする.
    if |C'| >ulim then                     — もしC' の要素数が ulim よりおおきければ
    split C' into C1, C2;                  — 解候補の集合C' を2つに分解する.
      QUEEN(C1, x+ 1);   —解候補の集合C1 に由来するすべての解をもとめて印刷する.
      QUEEN(C2, x+ 1);   —解候補の集合C2 に由来するすべての解をもとめて印刷する.
    else
      QUEEN(C', x+ 1);   —解候補の集合C に由来するすべての解をもとめて印刷する.
    end if;
  end if;
end QUEEN;
```

図2.8　エイト・クウィーン問題の並列バックトラック計算法

2.5 Nクウィーン解探索の実行結果

逐次バックトラック計算法と並列バックトラック計算法によるNクウィーン問題のプログラムを計3本記述し，その全体および部分ごとの実行時間を測定した．ORベクトル計算法の実行時間も測定したが，並列バックトラック計算法の実行時間とほぼひとしいので，ここでは省略する．

並列バックトラックのプログラムは非IDP用およびIDP用の2本であり，これらのプログラムは図2.8にしめしたように再帰よびだしをふくんでいる．そのため，再帰部分はPascalで記述した．また，ベクトル計算部分はFortranで記述して自動ベクトル化した．IDPはFortranからは使用できないため，IDP用のプログラムにおけるIDP使用部分はアセンブラで記述した．比較に使用した逐次バックトラック計算法のプログラムはすべてFortranで記述した．IDP用をのぞくこれらのプログラムについては，付録1でさらに説明する．IDPにおけるORベクトル計算法(および並列バックトラック計算法)の実現法については鳥居ら[Torii 88a, Torii 88b]にかんたんに記述されている．

図2.9にS-810によるNクウィーンの全解探索の実行時間をしめし，図2.10にそのときの加速率をしめす．S-810においては，並列バックトラック計算法のほうが測定したすべてのNにおいて逐次バックトラック計算法より高速である．エイト・クウィーンのばあいには実行時間は8.7 msであり，逐次バックトラック計算法の約9倍の速度である[注2]．また，S-820によってエイト・クウィーンのばあいを測定したところ，スカラ処理時間は53.4 ms，ベクトル処理時間は3.4 msであり，したがって加速率は15.7に達した．

[注2] 金田 [Kanada 84] では4.5倍の性能にとどまっていたが，その後のFortranコンパイラの性能向上によって9倍に達した．

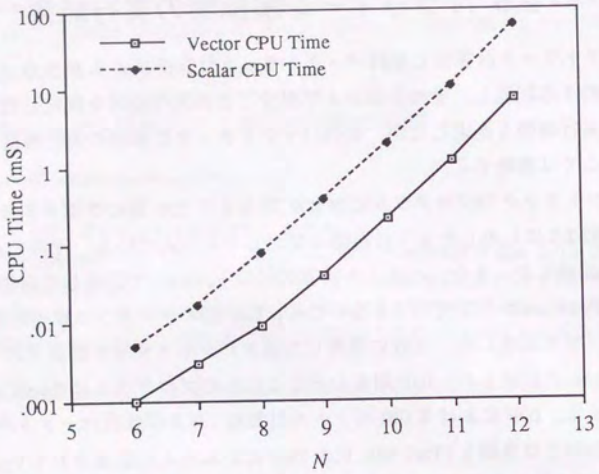


図 2.9　N クウィーン全解探索実行時間 (S-810)

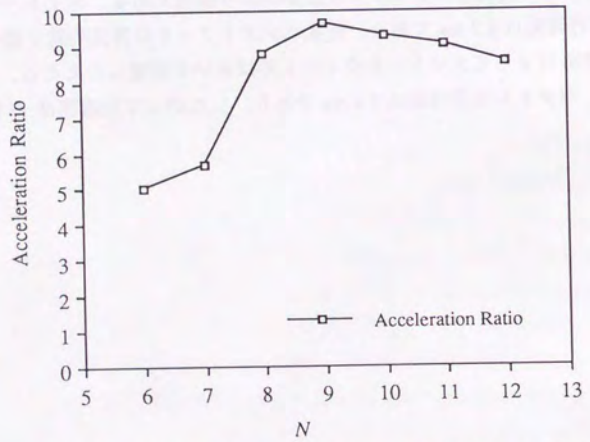


図 2.10　N クウィーン全解探索における加速率 (S-810)

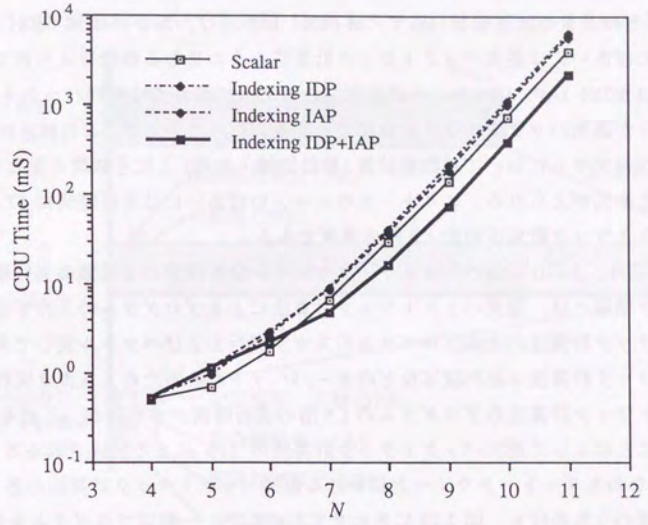


図 2.11　N クウィーン全解探索実行時間 (M-680H　IAP/IDP)

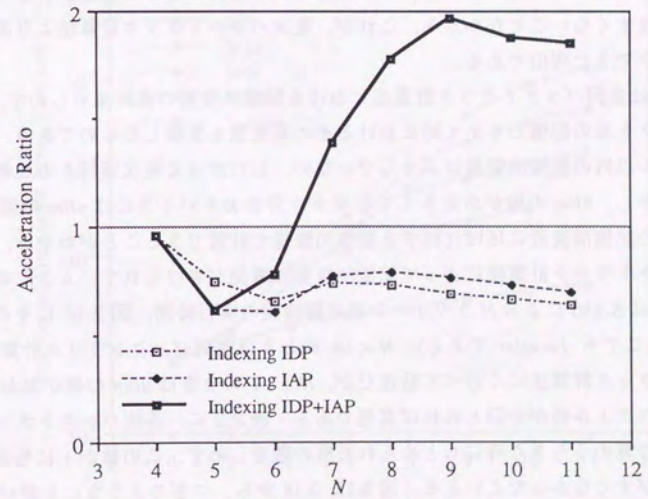


図 2.12　N クウィーン全解探索における加速率 (M-680H　IAP/IDP)

図2.11にM-680H IAP，IDPによるNクウィーンの全解探索の実行時間をしめし，図2.12にそのときの加速率をしめす．M-680H IAPだけ，あるいはM-680H IDPだけを使用したばあいには逐次バックトラック計算法をうわまわる性能はえられていない．しかし，M-680H IAP，IDPをともに使用したばあいにおいては並列バックトラック計算法のほうが逐次バックトラック計算法より高速になっている[注3]．これは，Nクウィーン問題のプログラムにおいては数値計算(整数加減・比較)と記号処理とをともにおこなうためだとかんがえられる．エイト・クウィーンのばあいには実行時間は17 msであり，逐次バックトラック計算法の約1.8倍の速度である．

図2.13には，S-810におけるエイト・クウィーン全解探索の3種類の実行時間をしめす．その3種類とは，逐次バックトラック計算法によるプログラムのスカラ実行，並列バックトラック計算法によるプログラムのスカラ実行およびベクトル実行である．並列バックトラック計算法は配列複写などのオーバヘッドがあるため，スカラ実行すると逐次バックトラック計算法のプログラムの1.5倍の実行時間がかかるが，これをベクトル実行することによって逐次バックトラック計算法の 1/9 にまで加速されることがわかる．

$N=8$すなわちエイト・クウィーンにおける並列バックトラック計算法の各プログラムの実行時間のうちわけを，図2.13にあわせてしめした．一般にプログラムを並列処理するばあいには，しばしばデータを複写する必要が生じるが，並列バックトラック計算法のプログラムにおいても，逐次バックトラック計算法では必要がなかった配列の複写・圧縮が必要になっている．しかし，その実行時間が全体にしめる割合がNクウィーンのばあいにはすくないことがわかる．これが，逐次バックトラック計算法より高速に実行することができた理由である．

図2.14は並列バックトラック計算法における記憶消費量の実測値をしめす．この図の縦軸はベクトルの記憶わりあて時におけるその要素数を累積したものである．l (= *ulim*)はベクトル以外の記憶消費量はふくんでいない．したがって逐次実行との比較もしていない．しかし，*ulim*の値がおおきくてもNが十分おおきいときには*ulim*の値がちいさいばあいの記憶消費量にほぼ比例する記憶消費量で計算できることがわかる．すなわち，並列バックトラック計算法によって記憶消費量の爆発がさけられていることがわかる．

図2.15にS-810によるNクウィーンの単解探索の実行時間，図2.16にその加速率をしめす(ここでも $l=ulim$ である)．$N<14$ のときは並列バックトラック計算法は逐次バックトラック計算法にくらべて低速だが，$N\geq 14$ のときは*ulim*の値がおおきく，したがってベクトル長が十分とれれば高速である．表2.2に，並列バックトラック計算法において計算のうちきり時にもとめられた解の数をしめす．この数が1にちかいほどむだな計算がすくなかったといえる．図2.15, 2.16から，つぎのようなことがいえる．N

[注3] なお，$N\leq 5$のばあいは誤差がおおきいため，データをしめしていない．

クウィーン($N\geq 14$)においては1つのクウィーンをもとめるまでの計算に十分なOR並列性があり，単解探索においても並列バックトラック計算法で十分な並列度がえられる．

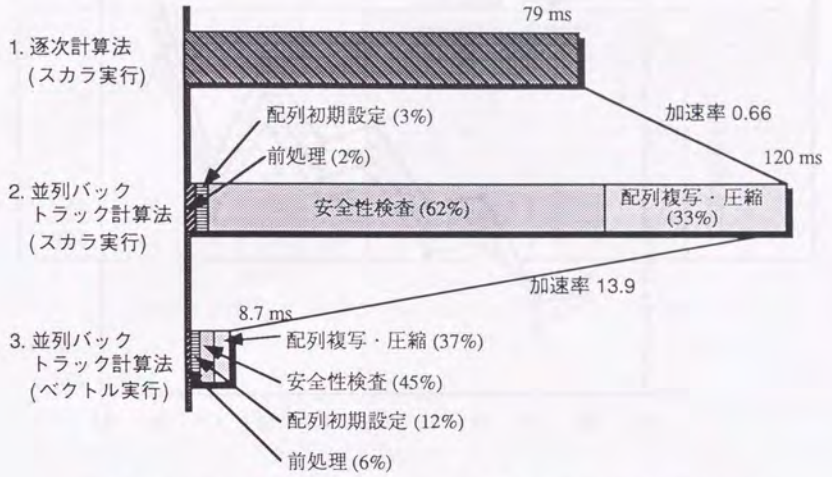


図2.13　各種実行法によるエイト・クウィーン全解探索の実行時間

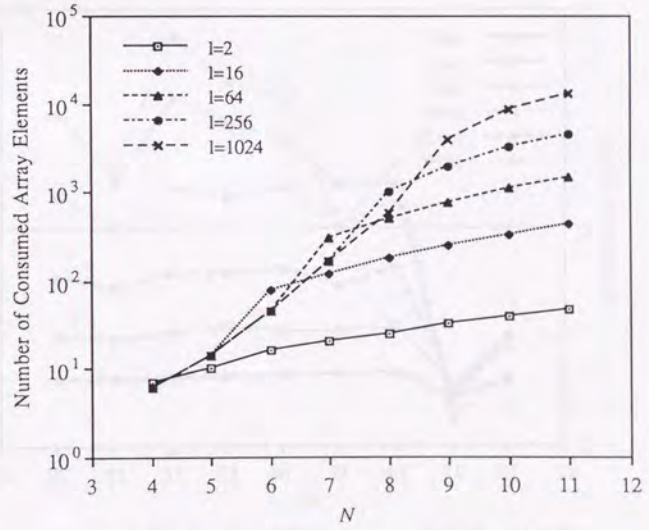


図2.14　Nクウィーン全解探索における記憶消費量 (S-810)

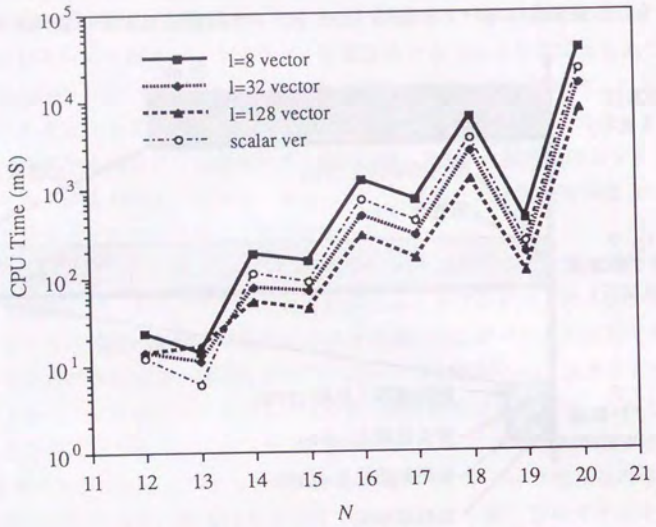


図2.15　N クウィーン単解探索の実行時間 (S-810)

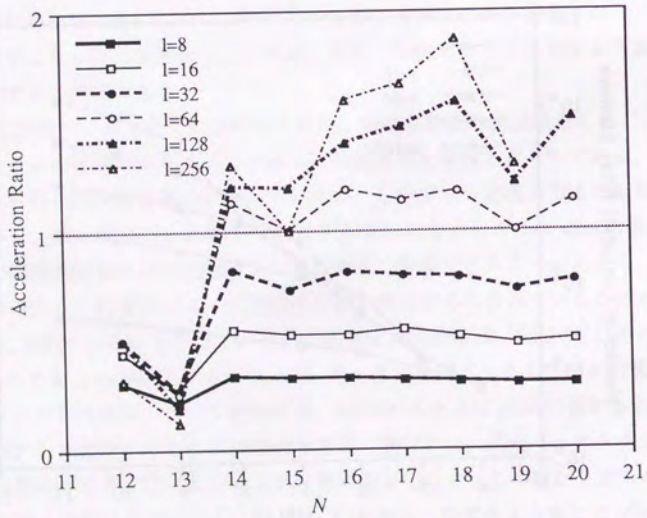


図2.16　N クウィーン単解探索における加速率 (S-810)

表2.2　N クウィーン単解探索で計算うちきりまでにもとめられた解の数 (- は未計算)

分割 ベクトル長	N=8	N=9	N=10	N=11	N=12	N=13	N=14	N=15	N=16	N=17	N=18
2	1	1	1	1	2	1	1	1	1	1	1
4	1	1	1	1	1	1	2	1	1	1	1
8	1	2	2	2	1	1	2	1	1	1	1
16	3	3	1	1	1	1	1	1	1	1	1
32	6	5	6	1	2	1	1	1	1	1	1
64	20	32	19	1	3	1	1	-	-	-	1
128	23	64	17	15	9	4	1	-	-	-	

2.6 並列バックトラック計算法におけるベクトル長増大法

Nクウィーン問題の並列バックトラック計算法によるプログラムにおいては，単純な並列バックトラック計算法によってつねにほぼ適当なベクトル長を確保することができたため，2.5節にしめしたように満足すべき加速率をえることができた．しかし，問題によってはベクトル長が爆発的に増大したあと減少し，みじかいベクトル長のもとで計算がつづけられるばあいがありうる．このようなばあいには，ベクトル計算機の性能をひきだすことができない可能性があるので，つぎのように，はやめにバックトラックする方法によってベクトル長を増大させることがのぞましいとかんがえられる [Kanada 85].

解候補ベクトルの要素数がある一定数をしたまわったときには，そのときのプログラム上の点と解候補ベクトルとを保存してバックトラックする．あとでふたたびプログラム上の同一点まで計算がすすんだときに，あらたにもとめられた解候補ベクトルと保存してあった解候補ベクトルとを併合してひとつの解候補ベクトルとし，それについて計算をすすめる．これは，並列バックトラック方式における解候補ベクトルの分割と逆の操作である．この方式を金田 [Kanada 85] は**残留バックトラック方式**とよんでいる．残留バックトラック方式を実現するためには，どのようにして解候補ベクトルとその環境を保存するか，またどのような条件がなりたったときに解候補ベクトルを併合するかをきめるのが重要であり，そこに研究の余地がある．

2.7 まとめ

この節では，探索問題に適用することができる，ベクトル計算機むきの計算方法である並列バックトラック計算法をしめすとともに，それをNクウィーン問題に適用して，最高 15.7 というたかい加速率がえられることと，しかも並列バックトラック計算法においては記憶消費量の爆発なしにそれが達成されることをたしかめた．また，同時に S-810 や M-680H IAP/IDP のようなベクトル計算機の記号処理への適用可能性がしめされたといえる．

しかしながら，並列バックトラック計算法のプログラムは，ユーザが直接記述するにはあまりにも複雑であり，それを容易にすることが重要な課題だとかんがえられる．すなわち，より単純なプログラムから並列バックトラック計算法のプログラムを自動生成することがのぞましい．ところが，逐次バックトラック計算法にしたがって記述されたプログラムから並列バックトラック計算法のプログラムへのプログラム変換をおこなうのは非常に困難である．バックトラック制御が陽に記述されていない Prolog のような論理型言語のプログラムからの変換は，逐次バックトラック計算法による手続き型言語のプログラムからの変換よりは容易に実現できるとかんがえられる．したがって，それを最初の目標とするのが適当であろう．

また，逆に論理型言語の OR 並列処理方式 [Conery 81] の一種として並列バックトラック計算法をみると，論理型言語のベクトル計算機による高速実行の可能性をしめしているとともに，並列度の制御という点でも示唆をあたえているとかんがえられる．この章でしめしたプログラムはデータを表現するのに配列を使用しているが，論理型言語の実行方式とするためには，リスト処理などのベクトル化が必要である．しかし 2.1 節でものべたように，第2世代のスーパコンピュータや内蔵型アレイ・プロセッサはリスト処理に必要な命令もそなえている．したがって，プログラム変換によって十分なベクトル長がとれさえすれば，高速化されることはまちがいないとかんがえられる．

2.8 付録1：測定用プログラムとその説明

測定に使用した各プログラムの主要部分を図2.17～2.19にしめす．図2.17が逐次バックトラック計算法のプログラム，図2.18がORベクトル計算法のプログラム，図2.19が並列バックトラック計算法のプログラムである．これらのうち逐次バックトラック計算法とORベクトル計算法のプログラムはすべての部分がFortranで記述されている．並列バックトラック計算法のプログラムは計算部分をFortranで記述しバックトラック制御部分をPascalで記述しているが，これは再帰よびだしを使用したかったからである．

```
*****************************************************
* N-QUEEN EXPANDED VER.2 (FOR A SCALAR PROCESSOR)   *
*            BY KANADA,Y.  1984-2-20                *
*****************************************************
      IMPLICIT INTEGER (A-Z)
      PARAMETER (NN=99)
      INTEGER X,Y(0:NN),CNT
      REAL*4  CPTIME
      LOGICAL FAIL
*
      READ*,N
      N1=N-1
      CALL CLOCK(CPTIME, 3)
      CNT=0
      X=0
      Y(0)=0
*     LOOP
    1   CONTINUE
*       FAIL = .NOT. SAFE(X,Y,TAKE,N1)
        FAIL = .FALSE.
        DO 10 XX=0, X-1
          IF (Y(X).EQ.Y(XX) .OR.
     *        X-Y(X).EQ.XX-Y(XX) .OR.
     *        X+Y(X).EQ.XX+Y(XX)) THEN
            FAIL = .TRUE.
            GOTO 15
          END IF
   10   CONTINUE
   15   CONTINUE
*
        IF (.NOT. FAIL) THEN
          IF (X.EQ.N1) THEN
*           CALL SUCCES(Y,CNT,N1)
            CNT = CNT + 1
            FAIL = .TRUE.
*           STOP
          ELSE
            X = X + 1
            Y(X) = 0
          END IF
        END IF
        IF (FAIL) THEN
*         CALL BACKTR(X,Y,N1)
          Y(X) = Y(X) + 1
*         LOOP
   21       IF (Y(X).LE.N1 .OR. X.EQ.0) GOTO 26
            X = X - 1
            Y(X) = Y(X) + 1
            GOTO 21
*         END LOOP
   26     CONTINUE
*
          IF (Y(X).GT.N1) THEN
*                          : FAILED
            WRITE(*, '('' # of solutions ='',I4)') CNT
            CALL CLOCK(CPTIME, 4)
            WRITE(*, '('' CPU time = '',F8.2,'' (ms)'')')
     *        CPTIME*1000
            STOP
          END IF
        END IF
        GOTO 1
*     END LOOP
      END
*****************************************************
      SUBROUTINE SUCCES(Y,CNT,NN)
      IMPLICIT INTEGER (A-Z)
      INTEGER NN
      INTEGER Y(0:NN),CNT
*
*     WRITE(*,'X,I5,'':'',(X,40I3)') CNT, Y
      END
```

図2.17　エイト・クウィーン問題逐次バックトラック計算法のFortranコーディング

測定に使用した並列バックトラック計算法によるプログラムの構造について説明する．とくに，どのような条件制御命令を使用したかをしめす．(これらの命令の代表的な動作を図2.20にしめす)．並列バックトラック計算法にもとづく2つのプログラムのFortran部分(図2.8の*nextRow*, *nextCandidate*に対応)は，それぞれ4つの部分から構成されている．


```
PROGRAM NQUEEN(INPUT, FT06F001);

CONST ulim = 1023;  llim = 1023;
      NMAX = 12; NMAX1 = 11;
      MAXSOL = 40000; (* << REGION_SIZE DIV (4 * NMAX) *)
      LLIMN = 13000; (* >= NMAX * LLIM *)

TYPE  BOARD  = ARRAY [0..NMAX1] OF INTEGER;
      BOARDS = ARRAY [0..MAXSOL] OF BOARD;
      BOARDWK= ARRAY [0..LLIM, 0..NMAX1] OF INTEGER;
      SAFETY = ARRAY [0..LLIMN, 0..NMAX1] OF BOOLEAN;

VAR   ANSCNT : INTEGER;
      SAFE   : SAFETY;
      Y      : BOARDS;
      YWORK  : BOARDWK;
      N, N1  : INTEGER;
      FT06F001 : TEXT;
      CPUTIME1, CPUTIME2 : REAL;
      Z      : INTEGER;

...

PROCEDURE PRINT(VAR Y : BOARDS;  FIRST, NSOL : INTEGER);
  VAR J, I : INTEGER;
BEGIN
  FOR J := FIRST TO FIRST + NSOL - 1 DO BEGIN
    ANSCNT := ANSCNT + 1;
    WRITE(FT06F001, ANSCNT:5,' : ');
    FOR I := 0 TO N-1 DO BEGIN
      WRITE(FT06F001, Y[J,I]:3);
    END(*LOOP*);
    WRITELN(FT06F001);
  END(*LOOP*);
END(*PRINT*);


PROCEDURE QUEEN(NEXTPOS, FIRST, NSOL, X0 : INTEGER);
  LABEL 99;
  VAR CNT, X, X1, J, I, N1, NREST : INTEGER;
BEGIN
  FOR X := X0 TO N-1 DO BEGIN
    QUEEN1(YWORK, Y, Y[NEXTPOS], SAFE, X, X0, FIRST, NSOL,
           MAXSOL-NEXTPOS, N, NMAX);
    IF NSOL > ULIM THEN BEGIN
      N1 := NSOL DIV LLIM;
      FOR I := 0 TO N1 - 1 DO BEGIN
        QUEEN(NEXTPOS+NSOL, NEXTPOS+I*LLIM, LLIM, X+1);
      END(*LOOP*);
      NREST := NSOL - N1 * LLIM;
      IF NREST <> 0 THEN BEGIN
        QUEEN(NEXTPOS+NSOL, NEXTPOS+NSOL-NREST, NREST, X+1);
      END(*IF*);
      GOTO 99;
    END ELSE BEGIN
      FIRST := NEXTPOS;
    END(*IF*);
  END(*LOOP*);
  ANSCNT := ANSCNT + NSOL;
99:
END(*QUEEN*);


BEGIN
  FINT; { Fortran initialization }
  READ(N); N1 := N - 1;
  REWRITE(FT06F001);
  WRITELN(FT06F001, 'SOLUTIONS OF ', N:1, ' QUEENS PROBLEM');
  WRITELN(FT06F001, '  LLIM = ', LLIM:1, '  ULIM = ', ULIM:1);

  {dummy}
    Z := 0;
    QUEEN1(YWORK, Y, Y[0], SAFE, Z, Z, Z, Z,
           2147483647, N, NMAX);

  ANSCNT := 0;
  CLOCK(CPUTIME1, 3);
  QUEEN(0, 0, 1, 0);
  CLOCK(CPUTIME2, 4);
  WRITELN(FT06F001, 'NUMBER OF SOLUTIONS = ', ANSCNT:1);
  WRITELN(FT06F001, 'CPU TIME =',
          (CPUTIME2 - CPUTIME1) * 1000 :8:2, 'ms');
  FEND; { Fortran finalization }
END.
```

```
      SUBROUTINE QUEEN1(Y, Y0, YTMP, SAFE, X, X0,
     *                  FIRST, NSOL,  MAXSOL, N, NMAX)
      IMPLICIT INTEGER (A-Z)
      INTEGER Y(0:NSOL-1, 0:NMAX-1), Y0(0:NMAX-1, 0:MAXSOL)
      INTEGER YTMP(0:NMAX-1, 0:MAXSOL)
      LOGICAL SAFE(0:NSOL-1, 0:NMAX-1)
      REAL*8  TODMCR, TIME1, TIME2, TIME30, TIME40
*
      IF (X.EQ.X0) THEN
        DO 10 X1 = 0, X-1
          DO 10 J = 0, NSOL-1
   10       Y(J,X1) = Y0(X1,FIRST+J)
      ELSE
        DO 20 X1 = 0, X-1
          DO 20 J = 0, NSOL-1
   20       Y(J,X1) = YTMP(X1,J)
      END IF
      DO 50 I = 0, N-1
        DO 30 J = 0, NSOL-1
          Y(J,X) = I
          SAFE(J,I) = .TRUE.
   30   CONTINUE
        DO 40 X1 = 0, X-1
* VOPTION NOFVAL
          DO 40 J = 0, NSOL-1
            T1A = X1 + Y(J,X1)
            TA  = X  + Y(J,X)
            T1S = X1 - Y(J,X1)
            TS  = X  - Y(J,X)
            IF (Y(J,X1) .EQ. Y(J,X) .OR. T1A .EQ. TA .OR.
     *          T1S .EQ. TS)
     *        SAFE(J,I) = .FALSE.
   40   CONTINUE
   50 CONTINUE
        DO 65 X1 = 0, X-1
          CNT = 0
          DO 60 I = 0, N-1
            DO 60 J = 0, NSOL-1
              IF (SAFE(J,I)) THEN
                YTMP(X1,CNT) = Y(J,X1)
                CNT = CNT + 1
              END IF
   60     CONTINUE
   65   CONTINUE
        CNT = 0
        DO 70 I = 0, N-1
          DO 70 J = 0, NSOL-1
          IF (SAFE(J,I)) THEN
            YTMP(X,CNT) = I
            CNT = CNT + 1
          END IF
   70   CONTINUE
        IF (CNT .GE. MAXSOL-NSOL) THEN
          PRINT*,'TOO MANY SOLUTIONS -- LARGEN *MAXSOL*'
          STOP
        END IF
        NSOL = CNT
        END
```

図2.19　エイト・クウィーン全解探索並列バックトラック計算法の Pascal / Fortran コーディング例 (S-810 用)

```
*****************************************************
* N-QUEEN FOR A VECTOR PROCESSOR (VER.3A)           *
*     BY KANADA,YASUSI   1984-2-10                  *
*****************************************************
      PROGRAM NQUEEN
      IMPLICIT INTEGER (A-Z)
      PARAMETER (N=11)
      PARAMETER (MAXSOL=89999)
      INTEGER   Y(0:MAXSOL,0:N-1), YTMP(0:MAXSOL,0:N-1)
      LOGICAL   SAFE(0:MAXSOL)
*
      WRITE(*,'(X,''SOLUTIONS OF'',I3,'' QUEEN PROBLEM'')') N
      NSOL = 1
      DO 80 X = 0, N-1
        DO 10 X1 = 0, X-1
          DO 10 J = 0, NSOL-1
 10         Y(J,X1) = YTMP(J,X1)
        CNT = -1
        DO 70 I = 0, N-1
          DO 30 J = 0, NSOL-1
            Y(J,X) = I
            SAFE(J) = .TRUE.
 30       CONTINUE
          DO 40 X1 = 0, X-1
            DO 40 J = 0, NSOL-1
              IF (   Y(J,X1).EQ.  Y(J,X) .OR.
     *            X1+Y(J,X1).EQ.X+Y(J,X) .OR.
     *            X1-Y(J,X1).EQ.X-Y(J,X)) SAFE(J) = .FALSE.
 40       CONTINUE
          DO 60 J = 0, NSOL-1
            IF (SAFE(J)) THEN
              CNT = CNT + 1
              DO 50 X1 = 0, X
 50             YTMP(CNT,X1) = Y(J,X1)
            END IF
 60       CONTINUE
 70     CONTINUE
        NSOL = CNT + 1
        IF (NSOL.GT.MAXSOL) GOTO 99
        PRINT*,'#SOL =', NSOL
 80   CONTINUE
      CALL PRINT(YTMP,NSOL,N,MAXSOL)
      STOP
 99   PRINT*,'TOO MANY SOLUTIONS -- LARGEN "MAXSOL"'
      END
*****************************************************
      SUBROUTINE PRINT(Y,NSOL,N,MAXSOL)
      IMPLICIT INTEGER (A-Z)
      INTEGER Y(0:MAXSOL,0:N-1)
*
      CNT = 1
      DO 10 J = 0, NSOL-1
        WRITE(*,'X,I5,'':'',(X,40I3)') CNT, (Y(J,I),I = 0,N-1)
        CNT = CNT + 1
 10   CONTINUE
      END
```

図2.18　エイト・クウィーン問題 OR ベクトル計算法の Fortran コーディング例

非 IDP 用のプログラムはつぎのような4部分から構成されている．S-810 においては各部分はいずれもベクトル化可能である．各部の説明の最後に図 2.19 における対応部分をしめす．

(1) 前処理部

Pascal 部分からわたされた解候補からなる配列 (図 2.8 の *C* に対応) から，内部処理用の配列への複写をおこなう[注4]．第2，第3の部分は，すでにおかれたクウィーンの個数だけの回数くりかえし実行される (図 2.19 では DO 10 ～ 20)．

(2) 配列初期設定部

この部分では，(3) で使用する解の有効性をしめす論理型配列 (マスク・ベクトル) の初期化をおこなう (その値をすべて .true. とする．図 2.19 では DO 30)．

(3) 安全性検査部

第3の部分は，図 2.8 の *nextRow* に対応する部分であり，解候補の安全性をチェックする．この部分ではマスク演算命令を使用する (図 2.20 a 参照．図 2.19 では DO 40)．

(4) 配列複写・圧縮部

そして第4の部分は，図 2.8 の *nextCandidate* に対応する．内部処理用の配列につくられたチェックに合格した解候補 (図 2.8 の $\langle x, b \rangle$ に対応) を，上記の論理型配列の値にしたがって Pascal 部分にわたす配列に圧縮しながら複写する．この部分は S-810 の圧

[注4] この部分はプログラムのかきかたしだいではなくすことが可能である．

圧縮命令を使用することによってベクトル処理できる (図 2.20 c 参照) (図 2.19 では DO 65)．

IDP 用のプログラムはつぎのような4部分から構成されている．M-680H においては，IDP を使用する部分以外はいずれも IAP によるベクトル処理が可能である．

(1) 前処理部

Pascal 部分からわたされた解候補からなる配列 (*C*) から，内部処理用のデュアル・ベクトル[注5]などの配列への複写をおこなう．

第2，第3の部分は，すでにおかれたクウィーンの個数だけの回数くりかえし実行される．

(2) 配列初期設定部

この部分は，(3) にそなえて上記のデュアル・ベクトルの一部の初期化をおこなう．

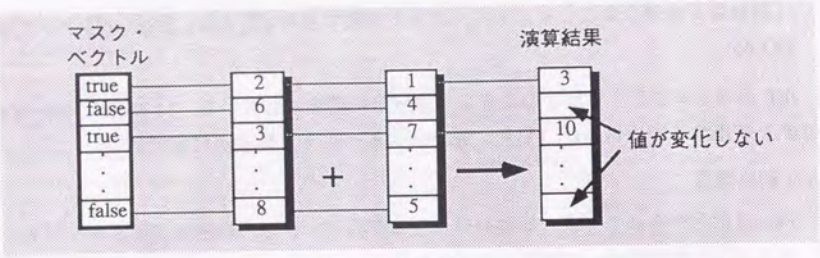
(3) 安全性検査部

図 2.8 の *nextRow* にほぼ対応する部分である．解候補の安全性をチェックするとともに，解候補をふくむデュアル・ベクトルの圧縮をおこなう．この部分では IAP のリスト・ベクトル命令 (図 2.20 b 参照) と IDP のサーチ命令 [Kojima 87] を使用すればベクトル処理できる．

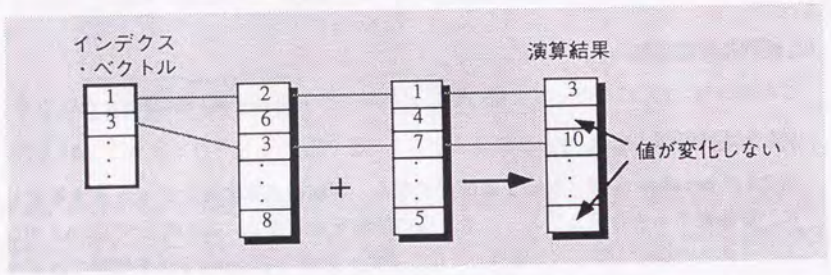
(4) 配列複写・圧縮部

図 2.8 の *nextCandidate* にほぼ対応する部分である．内部処理用の配列につくられたチェックに合格した解候補を，Pascal 部分にわたす配列に圧縮しながら複写する．配列複写・圧縮部ではリスト・ベクトル命令を使用する．この部分の実行前には解候補の配列は圧縮されていないが，それをアクセスするのに，すでに圧縮されているデュアル・ベクトルを使用するので，非 IDP 用のプログラムの (4) にくらべるとはるかに高速に実行できる．

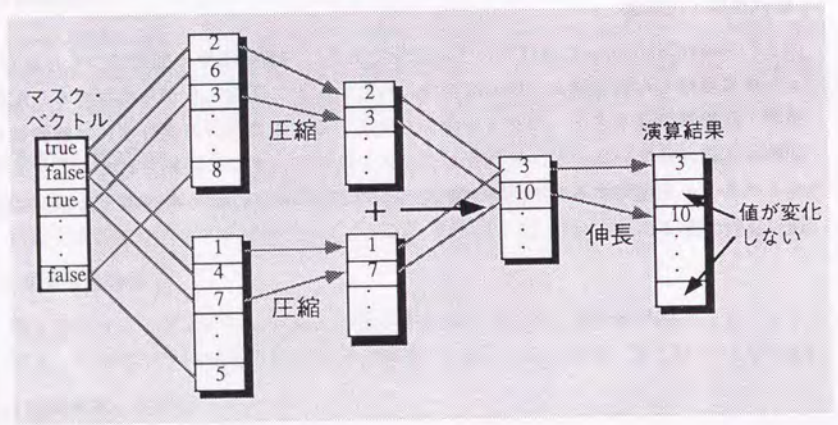
[注5] デュアル・ベクトルは IDP 用のデータ形式である [Torii 87b, Kojima 87]．



(a) マスク演算方式



(b) インデクス方式



(c) 圧縮方式

図 2.20　ベクトル計算機における条件制御方式

2.9 付録 2：N クウィーン解探索の測定データ

表 2.3 に，図 2.9 ～ 2.10 でしめした逐次バックトラック計算法と並列バックトラック計算法による N クウィーン問題の全解探索の S-810 における実行時間および加速率を数値でしめす．表 2.4 に，図 2.11 ～ 2.12 でしめした M-680H IAP/IDP においておこなった同様の比較結果をしめす．表 2.5 には，図 2.14 でしめした N クウィーン問題の全解探索における記憶消費量の測定結果を数値としてしめす．

表 2.3　N クウィーン全解探索実行時間 (S-810)

N	並列 BT CPU 時間 (ms)	逐次計算法 CPU 時間 (ms)	加速率
6	1	5	5.00
7	3	17	5.67
8	**9**	**79**	**8.78**
9	40	386	9.65
10	209	1,944	9.30
11	1,176	10,631	9.04
12	7,315	62,627	8.56

表 2.4　N クウィーン全解探索実行時間 (M-680H IAP/IDP)

N	スカラ処理	CPU 時間 (ms)			加速率		
		IDP使用	IAP使用	IDP+IAP	IDP使用	IAP使用	IDP+IAP
4	0.5	0.52	0.52	0.52	0.96	0.96	0.96
5	0.7	0.94	1.15	1.15	0.74	0.61	0.61
6	1.7	2.60	2.81	2.19	0.65	0.60	0.78
7	6.5	8.85	8.44	4.69	0.73	0.77	1.39
8	29.4	40.52	37.35	16.56	0.73	0.79	1.78
9	143.4	208.44	188.09	72.81	0.69	0.76	1.97
10	732.1	1110.31	1011.77	389.48	0.66	0.72	1.88
11	4037.3	6351.35	5838.65	2183.85	0.64	0.69	1.85

また表 2.6 には，図 2.15 ～ 2.16 でしめした N クウィーン問題の単解探索の S-810 における実行時間および加速率を数値としてしめす．

表 2.5　N クウィーン全解探索における記憶消費量 (S-810, 単位: 配列要素数)

N	l = 2	l = 16	l = 64	l = 256	l = 512	l = 1024
4	7	6	6	6	6	6
5	10	14	14	14	14	14
6	16	79	46	46	46	46
7	20	121	295	164	164	164
8	25	185	509	1009	1062	568
9	33	256	753	1899	3070	3941
10	40	331	1127	3151	5083	8388
11	47	438	1466	4471	8230	13223

表 2.6　N クウィーン単解探索の実行時間 (S-810, 単位 ms)

N	l = 8		l = 16		l = 32	
	ベクトル	スカラ	ベクトル	スカラ	ベクトル	スカラ
6	23	64	16	63	14	74
7	14	43	11	45	11	56
8	182	455	111	415	75	403
9	153	381	96	350	69	340
10	1224	3050	760	2745	492	2577
11	731	1847	428	1637	294	1558
12	6354	16023	3868	14288	2541	13450
13	435	1128	270	999	180	979
14	37567	96840	23032	86155	14915	80723

N	l = 64		l = 128		l = 256	
	ベクトル	スカラ	ベクトル	スカラ	ベクトル	スカラ
6	16	100	14	120	22	221
7	12	79	16	136	24	264
8	54	408	51	466	47	546
9	50	354	42	394	50	592
10	338	2526	286	2619	251	2632
11	206	1542	159	1589	140	1676
12	1684	13010	1250	12819	1059	13036
13	131	1002	108	1073	101	1280
14	10148	78140	7617	76717	-	-

最後に，AND ベクトル計算法に関する測定結果をしめす．表 2.7 に図 2.17 のプログラムの S-810 (旧 Fortran コンパイラ) におけるベクトル化率をしめす．

表 2.7　AND ベクトル計算法のプログラムのベクトル化率

N	8	9	10	11	12
ベクトル化率	55.6	59.5	63.8	66.7	69.0

S-810/20 旧 Fortran コンパイラ (S-810 初出荷当時) を使用して測定．

第3章
制御構造変換にもとづく ベクトル化—1リスト処理

要旨

この章では，ベクトル計算機を使用してリスト処理を高速に実行するためのプログラム変換にもとづいた基本戦略を提案する．この戦略は，リストを使用した解探索のベクトル処理やその他のベクトル記号処理のためのプログラムを，ユーザにとって自然なかたちに記述されたプログラムから変換・生成することを可能にするものである．この戦略は，Fortran プログラムのベクトル化技法の拡張とかんがえることができる「くりかえし構造の交換」と「くりかえし構造の1重化」というプログラム変換技法にもとづいている．変換結果のプログラムにおいては，複数のリストを要素とするベクトルを使用する．それらに関する操作をベクトル計算機のリスト・ベクトル処理機能や数列生成機能などをもちいて実現する．

上記の戦略は，この章においてはまだ形式化されていないため自動的におこなうことはできないが，これをエイト・クウィーン問題の Prolog プログラムに手動で適用し，ベクトル計算機 S-810 においてスカラ処理の約9倍の実行速度をえた．

3.1 はじめに

リスト (linked list) は記号処理においてもっとも重要なデータ構造であり，Lisp や Prolog などの記号処理用言語において欠かせない．したがって，記号処理応用プログラムの高速化および記号処理用言語の高速処理のためには，リスト処理の高速化がもっとも重要な課題である．

リスト処理高速化の方法としては，最適化コンパイラによって逐次計算機用の高速なプログラムを生成するという方法もある．しかし，根本的な高速化のためには，並行処理むきハードウェアの使用が必須である．並行処理むきハードウェアを使用する高速化方法としては，つぎのようなものがかんがえられる．

(1) 実行時の負荷分散にもとづく方法

複数の処理装置をもつ並列計算機において，並行処理可能な部分処理を実行時にことなるプロセッサに負荷分散し，実行する．各処理装置がまったくことなる処理をおこなうので，MIMD 型並列計算機に適した方法である [Sakai 86, Kanada 87].

(2) コンパイル時のプログラム変換にもとづく方法

コンパイル時に並行処理可能な部分をみつけ，ベクトル計算機用または並列計算機用のプログラムを生成し実行する．MIMD 型並列計算機においてもこの方法は利点があるだろうが，逐次処理むきのプログラムをそのままでは実行できない HITAC S-820 や Cray-2 のようなベクトル計算機や，SIMD 型並列計算機に適した方法である．実行時の負荷分散オーバヘッドがない点が (1) より有利だとかんがえられる．

(3) プログラミング時のアルゴリズム改良にもとづく方法

プログラマが並行処理むきのプログラムを記述する．どのようなアーキテクチャの計算機にも有効な方法だが，かなりアーキテクチャに依存したプログラミングが要求される．DO 文のかわりに並列くりかえし文をつかうというようなかんたんな改良で目的が達せられるばあいもある．しかし，おおくのばあい，単純なリコーディングなどでは十分な性能はえられず，アルゴリズムの再検討が要求される [Shapiro 84]．たとえば第 2 章でしめした解探索のベクトル処理方法は，逐次処理のプログラムに対してアルゴリズム改良をおこなった結果えられたものとかんがえられるが，両者のプログラムを比較すればわかるように，その対応は単純ではなく，逐次処理のプログラムからベクトル処理のプログラムをみちびくのは容易ではない．

これらの方法はいずれも研究途上にあり，どの応用においてどの方法が成功するかは現在のところわからない．この論文であつかうのは (2) の方法である．この方法は実行時

の負荷分散オーバヘッドがないため (1) より高速に処理できる可能性があり，また (3) におけるほどユーザに負担をかけることがないという利点がある．したがって，有望な方法だということができる．

これまで，ベクトル計算機はほとんど数値計算専用機とかんがえられてきた．また，現在でも Fortran がベクトル計算機で使用できるほとんど唯一の言語である．したがって，ベクトル計算機のためのコンパイル技術あるいはベクトル化技術は，数値計算プログラムにおいて，また Fortran において発展してきた．

ベクトル計算機は，ベクトル処理を高速におこなうことができる専用ハードウェアと，それに関する命令 (ベクトル命令) とをもっている．ベクトル計算機の Fortran コンパイラにおいては，DO ループ内の演算の実行順序を変更して同種の演算をまとめることにより，それを 1 個のベクトル命令で実行するようにする．これによって，それらの演算をベクトル処理専用ハードウェアで実行することを可能にしている．この演算順序の変更をベクトル化とよんでいる．

しかし，Fortran において発展してきたベクトル化技術をそのまま記号処理プログラムおよび記号処理用言語に適用するだけでは，リスト処理プログラムのベクトル化は実現できない．その理由をのべる．

Fortran コンパイラにおいては，プログラム中のループを検出し，そのうちの可能なものをベクトル化する．Fortran プログラムにあらわれるデータ構造は配列であり，ベクトル化に適するかどうかの判定は配列添字の比較をベースにしておこなわれる．一方，記号処理用言語で記述されたリスト処理プログラムにおいては，第 1 に制御構造としては，ループが存在せず，再帰よびだしがつかわれることがおおい．また，第 2 にデータ構造としては，配列はあらわれず，再帰よびだしごとに独立のリストが処理されることがおおい．したがって，配列添字の比較をベースにしたベクトル化法では，まったく対処できない．

しかも，リスト処理のプログラムは，もとのままでは本質的にベクトル処理に適さない構造のプログラムであることがおおい．ベクトル処理に適さないのは，つぎのような理由による．各要素の処理のあいだにデータ依存性があるとベクトル処理はできないが，リスト処理プログラムの最内側ループは，通常，リストの各要素を順に処理していくため，データ依存性がある．また，最内側ループがみじかすぎるためにたとえベクトル化できても加速されないばあいもある[注1]．したがって，あらたなベクトル化法を開発しなければ，ベクトル計算機によるリスト処理を実現することはできない．

3.2 節では，準備として，Fortran コンパイラでつかわれている 3 つの基礎的なベクト

[注1] 3.4 節でしめすエイト・クウィーン問題のプログラムにおける手続き `not_take1` はその例のひとつである．

ル化技術について説明する．3.3 節では，プログラム変換にもとづくベクトル・リスト処理の戦略についてのべる．3.4 節では，変換後のプログラムでつかわれるリスト処理基本演算のベクトル処理方法についてのべる．3.5 節では，エイト・クウィーン問題のプログラムにおける変換の適用例についてのべる．3.6 節でそのプログラムに関する実測結果をしめす．関連研究については次章でのべる．

3.2 ベクトル化の基礎

この論文でのべるリスト処理プログラムのベクトル化においても，Fortran におけるベクトル化技術が基礎となる．したがって，この節ではベクトル化の基礎についてかんたんに説明する．

ベクトル化は一種のプログラム変換とみなすことができる [Yasumura 87]．ベクトル化はつぎのようなプログラム変換のくみあわせである．

(1) ループ分散 (loop distribution)

ベクトル命令は，ロード，加算，ストアなどの 1 個の基本操作だけを，あたえられたベクトルの各要素におこなう微小なループとみなすことができる．DO ループをこのような微小なループに分割するプログラム変換を，ループ分散という [Kuck 81]．

(2) ループ交換 (loop interchange)

ベクトル化に適さないループをベクトル処理可能にするためかまたはよりたかい実行性能をえるためにおこなわれ，またループ 1 重化はおもに実行性能の向上のために Fortran のプログラムのなかには，ループ分散だけではループ内の一部の操作をベクトル命令に変換できないプログラムもある．このようなばあいには，外側ループとのいれかえ (くりかえし方向の変更) をおこなって最内側ループの全体をベクトル化可能にするというプログラム変換がおこなわれることがある．この変換はループ交換とよばれる [Allen 84, Wolfe 86]．ループ交換は，後述するループ 1 重化と同様に，ベクトル長をのばすことを目的としておこなわれることもある．例を図 3.1 (1) にしめす[注2]．

(3) ループ 1 重化 (loop unrolling)

ベクトル計算機においては，最内側ループのくりかえし回数すなわちベクトル長をのばすことが性能向上につながる．したがって，多重ループを一重ループにする変換がおこなわれる．この変換はループ 1 重化とよばれる [Tsuda 85]．例を図 3.1 (2) にしめす．

これらの変換のうちループ分散はプログラムをベクトル処理可能にするために必須の変換であり，ループ交換およびループ 1 重化は上記のように，ばあいによっておこなわれる．

[注2] かんたんな例にするため，変換前にもベクトル化可能なプログラムをしめしている．

```
   DO 20 I=1,N
     DO 10 J=1,M
       A(I,J)=A(I,J-1)+B(I,J)
10   CONTINUE
20 CONTINUE
```

→

```
   DO 10 J=1,M
     DO 20 I=1,N
       A(I,J)=A(I,J-1)+B(I,J)
20   CONTINUE
10 CONTINUE
```

(1) ループ交換

```
   DO 20 I=1,N
     DO 10 J=1,N
       A(J,I)=B(J,I)+C(J,I)
10   CONTINUE
20 CONTINUE
```

→

```
   DO 10 IJ=0,N*N-1
     I=IJ/N+1
     J=MOD(IJ,N)+1
     A(J,I)=B(J,I)+C(J,I)
10 CONTINUE
```

(2) ループ 1 重化

図 3.1　Fortran におけるループ交換とループ 1 重化

3.3 ベクトル・リスト処理の戦略

この節では，まず，我々が提案するリストのベクトル処理戦略におけるプログラムの処理手順をしめす．つづいて，そこでつかわれる個別のプログラム変換技法について説明する．最後に，この戦略における可変長リストのあつかいについて，とくに説明する．

3.3.1 プログラムの処理手順

このベクトル処理戦略では，およそつぎのような手順でリスト処理をおこなうことをめざす．

(A) プログラミング

プログラマが，記号処理用言語などで自然な (すなわち機械に依存せず過度に並列処理を意識しないですむ) リスト処理プログラムを記述する．この点が，3.1 節でのべたプログラミング時のアルゴリズムの改良にもとづく方法とはおおきくことなる．

(B) プログラム変換

上記のプログラムは，通常そのままではベクトル処理できないので，プログラム変換によってベクトル計算機で処理可能なプログラムに変換する．変換はコンパイラまたはプリプロセッサによっておこなうのがのぞましい．我々が開発した S-810 のための Prolog コンパイラ・プロトタイプにおいては一部のプログラムを自動的に変換することができるが，現在は自動的にプログラム変換できる範囲は非常にかぎられている．したがって，たいていのばあいはプログラマが変換をおこなう必要がある．しかし，ばあいによってはユーザ・オプションというようなかたちでユーザのたすけをかりるにしても，基本的にすべてを自動的に変換することが目標となる．

(C) 実行

変換後のプログラムをベクトル計算機で実行する．ただし，変換後のプログラムが原始プログラムとしてあたえられるばあいは，実行のまえにコンパイルする必要がある．

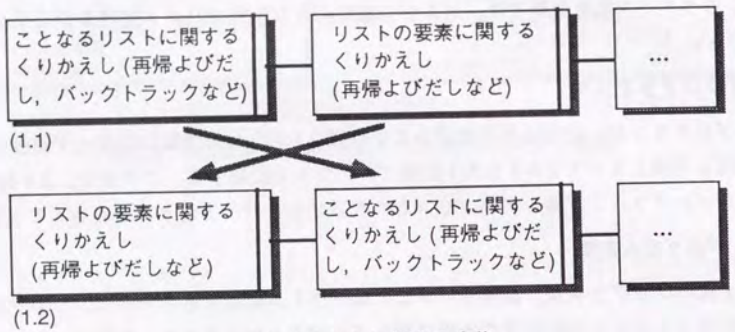
(B) におけるプログラム変換は，3.2 節で説明した Fortran のベクトル化よりこみいっているが，その拡張とかんがえることができる．このプログラム変換はつぎのような 2 種類の変換のくみあわせである．

(1) くりかえし構造の交換
(2) くりかえし構造の 1 重化

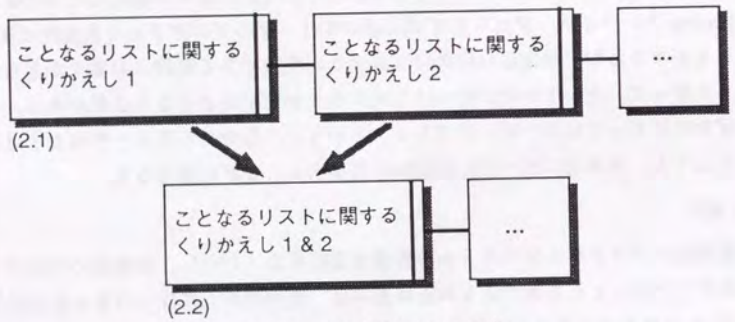
これらについて，つづく 2 つの節で説明する．これらの節では，かんたんのため固定長のリストをあつかう．可変長リストのあつかいについて 3.4 節でのべる．

3.3.2 くりかえし構造の交換

くりかえし構造の交換とは，多重のくりかえし構造における内側のくりかえしと外側のくりかえしとをいれかえることによってベクトル処理できないプログラムをベクトル処理可能にするプログラム変換のことである．図 3.2 (1) に例をしめす．くりかえし構造の変換によって，図 3.2 (1.1) に PAD (Problem Analysis Diagram) をつかってしめしたようなプログラムが図 3.2 (1.2) にしめしたようなプログラムに変換される．

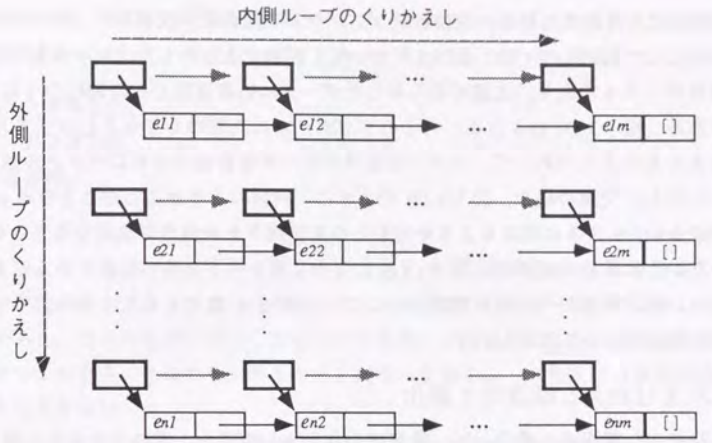


(1) くりかえし構造の交換

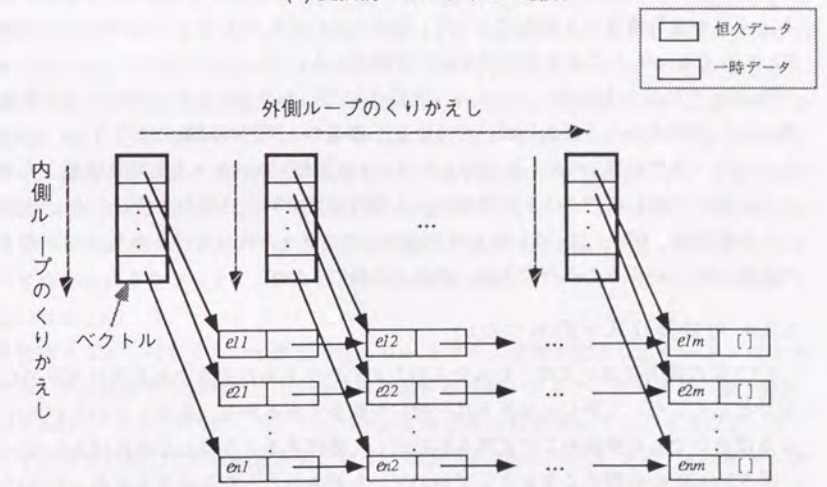


(2) くりかえし構造の 1 重化

図 3.2　リスト処理におけるくりかえし構造の交換と 1 重化



(1) 変換前のプログラムの動作



(2) 変換後のプログラムの動作

図 3.3　くりかえし構造の変換によるベクトルの生成とくりかえし方向の変化

対象となるくりかえし構造は，ループ，末尾再帰よびだし (tail recursion)，あるいは，自動バックトラックをひきおこす一種のくりかえし構造である．Prolog プログラムに関していえば，再帰よびだしによって内側のくりかえしが形成され，再帰よびだしやバックトラックによって外側のくりかえしが形成されている．

図 3.3 はくりかえし構造の交換前のプログラムの動作と，交換後のプログラムの動作とを対比してしめしている．図 3.3 において，細線であらわしたデータは操作対象である複数のリストであり，太線であらわしたデータは処理過程で一時的につくられるデータである．図 3.3 (1) のように，もとのプログラムの内側のくりかえしはリストの要素に関するくりかえしであって，リスト要素間のデータ依存性のためにベクトル処理できない．しかし，交換の結果，図 3.3 (2) のように，内側のくりかえしがことなる (データ依存性がない) リストに関するくりかえしとなり，ベクトル処理可能になる．

くりかえし構造の変換は，図 3.1 (1) にしめしたループ交換の拡張とかんがえることができる．基本演算のベクトル処理方法については 3.4 節でくわしくのべ，くりかえし構造の交換の例は 3.5 節でしめす．

3.3.3 くりかえし構造の 1 重化

くりかえし構造の 1 重化とは，多重のくりかえし構造を 1 重のくりかえし構造に変換するプログラム変換のことである．図 3.2 (2) に 2 重のくりかえし構造の 1 重化を図示する．くりかえし構造の 1 重化によって，図 3.2 (2.1) にしめしたようなプログラムが図 3.2 (2.2) にしめしたようなプログラムに変換される．

Prolog プログラムに関していえば，再帰よびだしやバックトラックによって多重のくりかえしが形成されるばあいに，くりかえし構造の 1 重化の対象になりうる．この変換によってスカラ処理 (ベクトル長が 1 のベクトル処理) をベクトル処理に変換したり，ベクトル長をのばしてベクトル計算機による実行速度を向上させたりすることができる．

この変換は，図 3.1 (2) にしめした Fortran のベクトル化における多重ループの 1 重化の拡張とかんがえることができる．例は 3.5 節でしめす．

3.3.4 可変長リストのあつかい

3.3.2 節の説明においては，かんたんのためにベクトルの要素である各リストのながさをひとしくした．しかし，リストは一般に可変長であるから，各リストのながさがことなるばあいでも変換後のプログラムがただしく動作するようにしなければならない．

図 3.2 (1.2) の外側のくりかえしにおいて，ながさのことなるリストをあつかうばあい，すべてのベクトル要素について一定回数 n 回の処理をおこなうとすれば，つぎのような不都合が生じる．第 1 に，ながさが n をこえるリストについては処理されない要素がのこる．第 2 に，ながさが n 未満のリストについては空リスト (*nil* または [ ]) を分解しようとして，あやまった処理がおこなわれる．

このような不都合をさけ，ちょうど必要なだけの回数の処理をおこなうようにするためには，ベクトル計算機に用意されている 3 種類の条件制御機構のうちのいずれか，またはこれらをくみあわせてつかえばよい．それらは，マスク演算処理機構，リスト・ベクトル処理機構，圧縮・伸長処理機構である [Kamiya 83]．いずれの条件制御機構をおもに使用するかによって，可変長リストをあつかう方法もつぎの 3 種類にわけられる．

(1) マスク演算方式，
(2) インデクス方式，
(3) 圧縮方式．

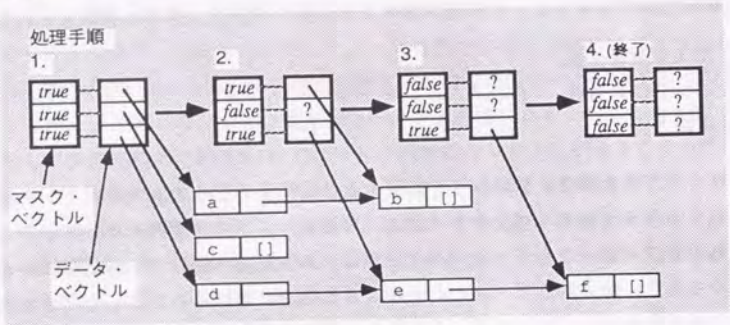
これらの方法を図 3.4 をつかって説明する．操作すべきリストへのポインタを要素とするベクトルをデータ・ベクトルとよぶ．可変長リストの処理のばあいにかぎらないが，いずれの方法においてもデータ・ベクトルはリスト処理の過程で一時的につかわれるベクトルであり，恒久的なデータとしてはスカラ処理におけるのとまったくおなじ形式のリストがつかわれる．すなわち，くりかえし構造の変換によって恒久データの形式は変換されることがない．

図 3.4 (1) はマスク演算方式によるリストのアクセス方法をしめす．この方式では各データ・ベクトルの要素が有効かどうかをあらわす論理値をふくむベクトルを使用する．このベクトルをマスク・ベクトルとよぶ．図 3.4 (1) では 3 個のリストを並列に処理するばあいのマスク・ベクトルとデータ・ベクトルの変化をしめしている[注]．この図においてはデータ・ベクトルは 1 個しかあらわれないが，一般には複数個のデータ・ベクトルが使用され，いずれも同一のマスク・ベクトルの支配をうける．マスク・ベクトルの各要素が，データ・ベクトルの対応する要素 (同一添字要素) が有効かどうか，すなわちその要素に対して処理をおこなうべきかどうかをあらわしている．これに対して，処理のひとつの時点におけるマスク・ベクトルは，基本的にはただ 1 個である．したがって処理中につくられる全データ・ベクトルのベクトル長は唯一のマスク・ベクトルのベクトル長にひとしい．

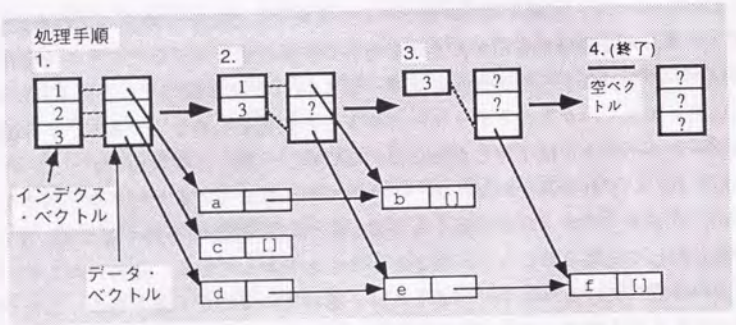
最初はマスク・ベクトルの全要素の値を *true* とする．処理がすすむとデータ・ベクトルの第 i 要素に関する処理がリストの末尾に達するので，マスク・ベクトルの第 i 要素の値を *false* とする．マスク・ベクトルの対応する要素の値が *true* であるようなデータ・ベクトル要素に関してだけ処理をおこなう．処理がすすむにつれてマスク・ベクトル中には *false* を値とする要素が増加する．マスク・ベクトルの全要素の値が *false* になったとき，処理を終了する．このように処理すべき要素がなくなったときに可変長データのベクトル処理を終了する可変長ループのベクトル処理方法を**残存要素検出法**とよぶ．残存要素検出法およびそれ以外の可変長ループ・ベクトル処理方法については次章でくわ

[注] 後述する論理型言語のばあいには，破壊的代入ができないために処理の各ステップでつかわれるベクトルはそのたびごとに再生成される．しかし，これらのベクトルはそもそも一時データであるから，最適化処理系においては記憶わりあてをする必要は通常はなく，ベクトル・レジスタ上だけに存在する．そのばあいには，ベクトル計算機のデータフロー的な性質上，ベクトルの再生成はオーバヘッドにはならない．

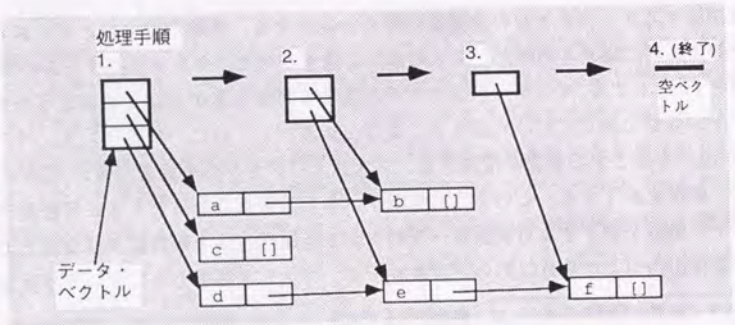
しくのべる．



(1) マスク演算方式



(2) インデクス方式



(3) 圧縮方式

図 3.4　3 種類の条件制御方式によるリストの処理方法

図 3.4 (2) はインデクス方式によるリストのアクセス方法をしめす．この方式では，各データ・ベクトルを直接にアクセスするのではなく，データ・ベクトルの有効要素のインデクス (または変位) をふくむベクトルをつうじてアクセスする．このベクトルをインデクス・ベクトルとよぶ．図 3.4 (2) では，図 3.4 (1) とひとしいリスト処理を例として，インデクス・ベクトルとデータ・ベクトルとの変化をしめしている．マスク演算方式のばあいと同様に，この図においては 1 個のデータ・ベクトルだけが使用されているが，一般には複数個のデータ・ベクトルが，基本的には唯一のインデクス・ベクトルによって支配される．

インデクス方式においては，インデクス・ベクトルから指示される要素だけが処理対象となる．すべてのデータ・ベクトルのベクトル長はひとしく，処理の途中でデータ・ベクトルのベクトル長は変化しないが，インデクス・ベクトルからは処理が終了した要素が除去されていく．したがって，最初は両者のベクトル長はひとしいが，しだいにインデクス・ベクトルのベクトル長は短縮する．そのベクトル長が 0 になったとき，処理を終了する．この方法は，残存要素検出法をインデクス方式に適用したものだということができる．

図 3.4 (3) は圧縮方式によるリストのアクセス方法をしめす．この方式では，データ・ベクトルはつねに有効要素だけをふくむ．そのために，処理が終了した要素が生じるたびに，その要素はデータ・ベクトルから除去されていく．マスク・ベクトルやインデクス・ベクトルのような制御のためのベクトルは基本的に使用されない．そして，すべてのデータ・ベクトルのベクトル長はつねにひとしく，処理がすすむにつれて同期して変化する．このばあいはベクトル長が 0 になったときに処理を終了する．これも残存要素検出法の応用である．

金田 [Kanada 85] でものべているように，これらの方式にはそれぞれつぎのような利点・欠点がある．

(3) 圧縮方式

圧縮方式はマスク・ベクトルやインデクス・ベクトルのような制御のためのベクトルを使用しないため，原理的には他の 2 方式にくらべて単純だということができる．しかし，つぎのような欠点がある．図 3.4 (3) においては各ステップでただ 1 つのデータ・ベクトルがつかわれているが，一般には複数のデータ・ベクトルがつかわれる．このようなばあい，圧縮方式による実行においては，全データ・ベクトルをいっせいに圧縮しなければならない．なぜなら，そうしないと各ベクトル間での要素の対応がくずれて，対応する要素をみつけることができなくなるからである．同時にあつかうデータの量がおおいばあいには，このベクトル圧縮のオーバヘッドは無視できない．

(2) インデクス方式

インデクス方式においては圧縮方式のようにデータ・ベクトルの数だけのベクトル圧縮をおこなう必要はなく，1個のインデクス・ベクトルだけを圧縮すればよい．しかし，データ・ベクトルの要素を間接アドレスでアクセスするため，それを処理のためにベクトル・レジスタにロードするのに時間がかかるという欠点がある．また，データ・ベクトルがむだな要素をふくんだまま使用されるため，圧縮方式にくらべて記憶効率がわるいということができる．

(1) マスク演算方式

マスク演算方式においてはベクトルを圧縮する必要はいっさいない．しかし，マスク演算方式においては，処理のために基本的にマスク・ベクトルのベクトル長に比例する時間がかかるため，マスク・ベクトルの要素に *false* がおおくなるとむだな処理がおおくなるという欠点がある．また，インデクス方式と同様にデータ・ベクトルがむだな要素をふくんだまま使用されるため，圧縮方式にくらべて記憶効率がわるいということができる．

このように各方式は一長一短であり，いずれの方式がよいかはばあいによる．

ところで，上記のプログラム変換戦略はプログラム変換の方針をあたえているだけであり，具体的な変換方法はくりかえし構造がループ，再帰よびだし，バックトラックのいずれによって形成されているか，などによってことなり，それぞれ具体的な変換方法をみいだす必要がある．プログラム中に2重以上のくりかえし構造があることがこれらのプログラム変換を適用するための必要条件だが，それは十分条件ではないから，それをみたすすべてのプログラムがベクトル化可能な構造に変換可能なわけではない．たとえば，部分的に共有された複数のリストの一部をかきかえる処理などは，プログラム変換できない．図3.5にこの理由でベクトル化できない処理の例をしめす[注3]．しかし，おおくのリスト処理のプログラムは多重のくりかえし構造をもっていて，外側のくりかえしごとにことなるリストを処理するので，この戦略の適用範囲はかなりひろいものとかんがえられる．またこの戦略は，可変長リストだけではなく他の可変長データ構造にも適用することができる．

[注3] このような処理のベクトル化法に関しては第5章でのべる．

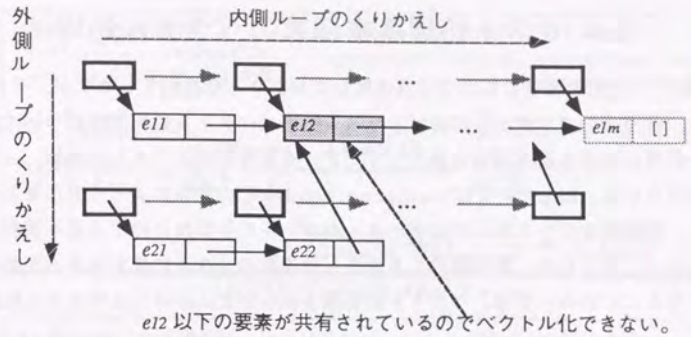


図3.5　2重のくりかえしがあるがベクトル化できない処理の例

3.4 リスト処理基本演算のベクトル処理法

3.5 節で，上記の戦略にもとづく具体的なプログラム変換例をしめすが，それにさきだって，変換後のプログラムにおけるリスト処理のベクトル処理方法について説明する．リスト処理における基本的な演算として，データ型の判定，リストの分解，リストの合成があげられる．Lispでいえば null, car, cons などの関数によっておこなわれる演算である．変換後のプログラムではデータ・ベクトルの各要素に対する基本演算をまとめておこなう．すなわち，最内側のくりかえしは複数のリストに関する基本演算のくりかえしになる．このループは，ベクトル計算機 S-810 などにおいてはベクトル処理可能である．

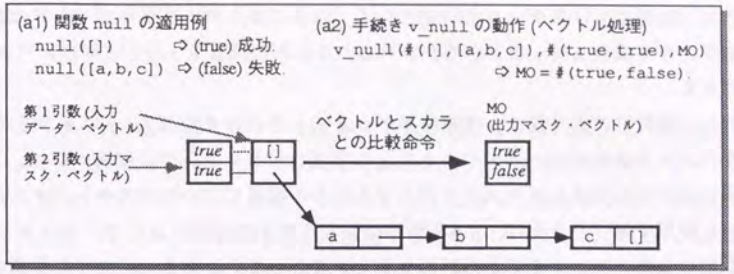
これらの基本演算とそのベクトル処理法について順に説明する．すべての条件制御方式についてのべると煩雑になるので，マスク演算方式についてだけのべる．ほかの条件制御方式も同様に実現可能である．なお，制御構造の変換によらずリストをベクトルにデータ構造変換することによってベクトル処理するばあいのリスト処理基本演算に関しては第 7 章でかんたんにのべる．

3.4.1 データ型判定

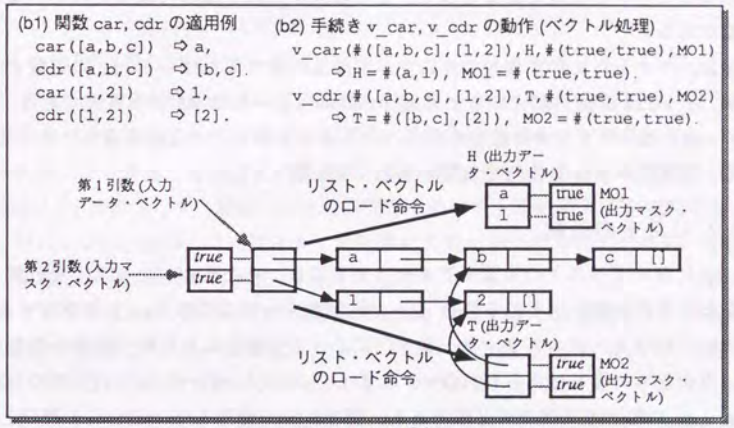
Lisp, Prolog などの記号処理用言語においては，各データは型 (タグ) をともなっている．そして，各データの処理にさきだって，型を判定する必要がある．たとえば，リスト処理においては，通常，空リスト判定 (空リストかどうかの判定) をおこないながらリストをたどる必要がある．空リスト判定をおこなう Lisp 関数 null を例にとり，図 3.6 (a) を使用して，データ型判定の機能とそのベクトル処理の方法を説明する．

関数 null は 1 個の引数をもち，引数が空リストなら *true*，そうでなければ *false* を結果とする．関数 null の適用の例を図 3.6 (a1) にしめす．

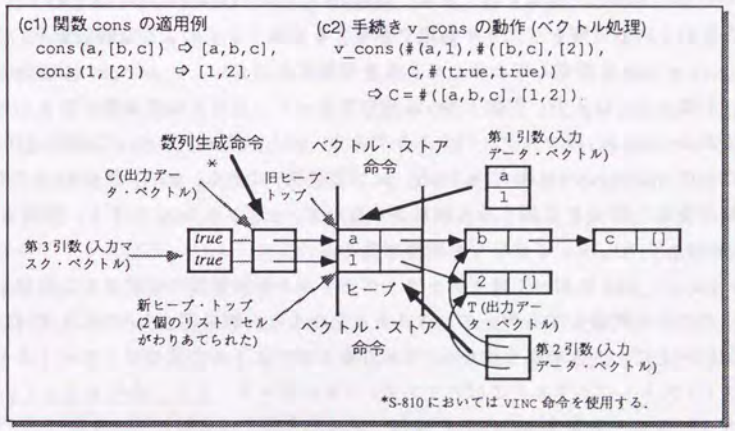
複数のデータに関する空リスト判定をベクトル処理で実現するには，つぎのような機能をもつ手続き v_null を用意しておき，これを使用すればよい．v_null(X,MI,MO) において，引数 X，MI は入力，引数 MO は出力であって，これらは要素数がひとしいベクトルである．以下，データ・ベクトル V の第 i 要素を V[i] であらわす．MI，MO はマスク・ベクトルである．MI[i] が *true* ならば MO[i] に null(X[i]) ($1 \le i \le N$，N は要素数) の値を代入する．MI[i] が *false* ならば X[i] に関する比較はおこなわず，MO[i] も *false* とする．



(a) 空リストかどうかのテスト



(b) リストの分解



(c) リストの合成

図 3.6　リスト処理基本演算とそのベクトル処理方法

リスト処理におけるデータ型判定として，ほかには，データがリストかそれ以外の型かを判定する関数 atom などがつかわれる．これらの演算も null と同様にベクトル処理できる．

データ型判定の実行後に，判定結果が *true* のときだけまたは *false* のときだけ実行可能なベクトル演算をデータ・ベクトル X の要素に対して実行するばあいには，データ型判定において出力されるマスク・ベクトルをその演算においてマスク・ベクトルとして入力し使用する．たとえば，3.4.2 節でのべるリストの分解において，データ・ベクトルの要素のなかにリストとそれ以外のものとがまざっているときは，まず各要素がリストであるかどうかの判定をおこなって，その結果が *true* になった要素だけ分解をおこなう必要がある．

なお，ベクトル X の要素がつねにすべて演算対象であれば v_null の引数 MI は不要だが，v_null が条件制御のもとで実行されるばあいには MI が不可欠である．

v_null は，ベクトル計算機のもつベクトルとスカラとの比較命令をつかうことによって，容易にベクトル命令で実現できる．例を図 3.6 (a2) にしめす．

3.4.2 リストの分解

Lisp においてリストの要素をアクセスするには，あたえられたリストを分解する関数，すなわちその頭部をもとめる関数 car と尾部をもとめる関数 cdr とを使用する．Prolog においてはリストはユニフィケーションによって分解されるため，分解の機能は手続きというかたちで陽にはあらわれない．しかし，ユニフィケーションの実装のために関数 car, cdr に相当する機能が必要である．図 3.6 (b)を使用して，これらの機能とそのベクトル処理の方法とを説明する．まず，関数 car, cdr の適用例を図 3.6 (b1) にしめす．

複数のリスト分解をベクトル処理で実現するには，つぎのような機能をもつ手続き v_car, v_cdr を用意しておき，これらを使用すればよい．v_car(X,Y,MI,MO) において，引数 X, MI は入力，引数 Y, MO は出力であって，これらは要素数がひとしいベクトルである．MI, MO はマスク・ベクトルである．MI[i] が *true* であってかつ X[i] が分解可能ならば Y[i] = car(X[i]) ($1 \le i \le N$, N は要素数) である．MI[i] が *false* または X[i] が分解不能ならば X[i] に関する分解はおこなわず，MO[i] も *false* とする．関数よびだし v_cdr(X,Y,MI,MO) においても同様である．

v_car, v_cdr においては，データ・ベクトルから各要素の頭部または尾部をもとめて，それらを要素とするデータ・ベクトルをつくる．例を図 3.6 (b2) にしめす．より具体的にいえば，つぎのようになる．S-810 などのベクトル計算機は，ベクトル・インデクス (リスト・ベクトルとよばれている) つきのロード・ストア命令をもっている．インデクスつきロード命令の動作を C 言語であらわすと，つぎのようになる．


```
for (i = 1; i <= n; i++) {
    vector1[i] = *(vector2[i] + vectorBase);
}
```

ベクトル化された Fortran プログラムにおいては，通常は vectorBase に相当する命令フィールドに配列の原点 (先頭アドレス)，vector2[i] に相当する命令フィールドにその添字を指定して使用する．しかし，vector2[i] に相当する命令フィールドにリスト・セルの先頭アドレス，vectorBase に相当する命令フィールドにリスト・セルの先頭から頭部または尾部への変位を指定すれば，各リスト・セルの頭部または尾部を要素とするベクトルをもとめることができる．

3.4.3 リストの合成

Lisp においてリストを合成するには，関数 cons をつかう．Prolog においてはリストはユニフィケーションによって合成されるため，手続きというかたちで陽にはあらわれない．しかし，ユニフィケーションの実装のために関数 cons に相当する機能が必要である．図 3.6 (c) を使用して，関数 cons の機能とそのベクトル処理方法を説明する．

関数よびだし cons(X,Y) は，リスト Y の先頭に要素 X をつけくわえたリストを結果とする．関数 cons の適用の例を図 3.6 (c1) にしめす．

複数のリスト合成をベクトル処理で実現するには，つぎのような機能をもつ手続き v_cons をつかえばよい．v_cons(X,Y,C,MI) において引数 X, Y, MI は入力，引数 C は出力であって，これらは要素数がひとしいベクトルである．MI はマスク・ベクトルである．MI[i] が *true* ならば C[i] = cons(X[i],Y[i]) ($1 \le i \le N$, N は要素数) である．MI[i] が *false* ならば合成をおこなわない (C[i] は変化しない)．図 3.6 (c2) に例をしめす．

つぎのようにすれば，v_cons をベクトル命令で実行できる．まず，頭部とするべきデータを要素とするデータ・ベクトル H と，尾部とするべきデータを要素とするデータ・ベクトル T とを入力する．H と T とは要素数がひとしい．そして，H と T の対応する要素の値を頭部および尾部の値とするリスト・セルをつくり，それらをさすポインタを要素とするデータ・ベクトル C をつくる[注5]．そして C を結果とすればよい．リスト・セル長を Lc とし，リスト・セルをヒープに連続してわりあてるとすれば，ベクトル C の各要素のアドレス部は，リスト合成の実行前のヒープ・トップ・ポインタの値に 0, Lc, $2 * Lc$, ... を加算してできた値をふくむ．このような値のベクトルは，ベクトル計算機の数列生成命令 (たとえば S-810, S-820 においては VINC 命令) を使用すれば高速に生成できる．むだのない記憶わりあてのためには[注6]等差でない数列生成またはマスクつ

[注5] v_cons は 1 要素あたり 3 個のポインタを生成することに注意．

[注6] すなわち，「マスクが *false* である要素には記憶をわりあてないためには」．

き等差数列生成が高速実行できることがのぞましいが，現状では高速性を優先して等差数列を使用するのがよいであろう[注7]．

なお，MI[i] の要素のなかに *true* でないものがあるときには，*true* でない要素に対応するデータ・ベクトル要素に対してはリスト・セルをわりあてないようにするのが場所効率はよい．しかし，そうするとベクトル C のとなりあう要素のアドレスの差分が一定ではなくなるため，ベクトル計算機によってはベクトル化できなくなったり，ベクトル化できてもベクトル C の生成に要する時間が増加したりする．したがって，高速性を優先するばあいには，MI[i] が *true* でない要素に対してもリスト・セルをわりあてるほうがよい．

最後に，リストの合成を記号処理における記憶管理という側面からみる．リストの合成は動的記憶わりあてをともなう．したがって，この節ではベクトル処理によって複数のリスト要素の動的記憶わりあてを実行する方法をあたえたことになる．記憶管理におけるもうひとつの重要な操作は記憶の解放である．リスト処理においては記憶の解放のためにガーベジ・コレクションを使用するのが普通である．Appel ら [Appel 89] はベクトル処理によるガーベジ・コレクションの方法をあたえており，これとこの節の記憶わりあて法とをくみあわせることによってベクトル処理による記憶管理法として完結する．

[注7] S-820 をはじめとするいくつかのベクトル計算機においては，等差でない数列生成は等差数列生成よりはるかに低速であるためである．

3.5 エイト・クウィーンの Prolog プログラムへの適用

この節では，3.2 節でのべたプログラム変換の戦略を，Prolog で記述されたエイト・クウィーン問題のプログラムに適用した例についてのべる．3.5.1 節ではエイト・クウィーンのプログラムの一部である手続き not_take1 を例としてくりかえし構造の交換についてのべる．3.5.2 節では，エイト・クウィーンのプログラムの一部である手続き select を例としてくりかえし構造の 1 重化についてのべる．Prolog のプログラムをベクトル化するためには，他のベクトル化技法と併用し，しかるべき手順をふむ必要があるが，その手順については第 6 章であらためて説明することにして，この節では上記の手続きのプログラム変換において使用されているくりかえし構造の交換と 1 重化だけに焦点をあてて説明する．なお，この節ではエイト・クウィーンのプログラムを例題とするが，Prolog で記述されたリストを分解する手続き append におけるくりかえし構造の交換と 1 重化に関しては図 6.8 にしめしているので，参照されたい．また次章では，リスト処理への適用例ではないが，より単純な配列の線形検索のプログラムを例題としてくりかえし構造の交換を手続き型言語上でおこなっている．

3.5.1 くりかえし構造の交換

エイト・クウィーン問題をとく Prolog プログラムは，たとえば中島 [Nakashima 83] でしめされている[注8]．つぎにしめす not_take1 はその一部であり，指定された 1 個のクウィーンがチェス・ボードの対角方向にすでにおかれたクウィーンと衝突するかどうかをしらべる手続きである．not_take1 のすべての引数は入力である．

(1) 原始プログラム

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

第 1 引数はチェス・ボードをあらわすリストであり，第 2 ～ 3 引数はしらべるべき位置 (行番号) をあらわす整数である．

not_take1 はその末尾が再帰よびだしになっていて，各くりかえしにおいて，チェス・ボードをあらわすリストの要素を 1 個ずつしらべていく．エイト・クウィーンのばあいには，このリストの平均長は 4 以下とみじかい．したがって，この再帰よびだしをそのままループに変換したとしてもベクトル計算機の性能をいかすことはできない [Kanada

[注8] 中島のプログラムの全体は第 6 章でしめす．

88b]．しかし，not_take1 は，その外部で発生する深いバックトラックによって，その全体がくりかえし実行される．これらの実行のあいだにはデータ依存関係がないので，このくりかえしを最内側のくりかえしとするようにプログラム変換すれば，ベクトル処理が可能になる．

変換後の手続き v_not_take1 をしめす．

(2) ベクトル化後のプログラム

```
v_not_take1(_, _, _, MI, MI) :-
    v_finished(MI), !.
v_not_take1(B, Qa, Qs, MI, MO) :-
    v_null(B, MI, MO1),
    v_car(B, Q, MI, M1), v_cdr(B, R, M1, M2),
    'v_=\='(Q, Qa, M2, M3),
    'v_=\='(Q, Qs, M3, M4),
    'vs_+'(Qa, 1, Qaa, M4),
    'vs_-'(Qs, 1, Qss, M4),
    v_not_take1(R, Qaa, Qss, M4, MO2),
    v_end_or(MO1, MO2, MO).
```

v_not_take1 の第 1 ～ 3 引数は not_take1 の第 1 ～ 3 引数に対応している．not_take1 はバックトラックによってくりかえしよびだされるが，各回における not_take1 の第 1 引数の値を要素とするデータ・ベクトルを v_not_take1 の第 1 引数とする．第 2 ～ 3 引数も同様である．エイト・クウィーンのプログラムにおける not_take1 以前に実行される部分は，このインタフェースをみたすようにプログラム変換しなければならない．

変換後のプログラムにおける条件制御の方式として 3 種類があるが，いずれのばあいもプログラムの基本構造はかわらないので，ここではマスク演算方式によるプログラムだけをしめす．v_not_take1 の第 4 ～ 5 引数は入力および出力のマスク・ベクトルである．引数であるすべてのベクトルの要素数はひとしい．

(1) から (2) への変換の手順については第 6 章でのべるので，ここでは省略する．(1), (2) の各部分の対応を表 3.1 にしめす．手続き v_null, v_car, v_cdr の機能は 3.3 節でのべたとおりである．他の手続きの機能については表 3.1 を参照されたい．

すでにのべたように not_take1 の末尾再帰よびだしはリストの要素に関するくりかえしである．v_not_take1 の再帰よびだしは末尾再帰ではないが，データ・ベクトル B の要素であるリストの要素に関するくりかえしであり，not_take1 の末尾再帰よびだし

と対応している．しかし，v_not_take1 においては，それがよびだす手続き v_null, v_car, v_cdr などの内部にループがあり，これらは not_take1 におけるバックトラックで形成されるくりかえしに対応している．したがって，not_take1 から v_not_take1 への変換においてくりかえし構造を交換しているということができる．

(2) のプログラムにつぎのような入力をあたえたときの実行過程を図 3.7 にしめす．ただし，ここで #(e_1, e_2) は e_1, e_2 を要素とするベクトルをあらわす．

第 1 引数: #([2,4,1], [4,1,3]).
第 2 引数: #(4,3).
第 3 引数: #(2,1).
第 4 引数: #(true,true).

表 3.1　手続き **not_take1** における原始プログラムとベクトル化プログラムとの対応

原始プログラム	ベクトル化後のプログラム	意味
-*	v_finished(MI)	手続きの実行を終了するかどうか (再帰を停止させるかどうか) を判定する．
[]	v_null(B,MI,MO1)	ベクトル B の要素が空リストかどうかを判定する．
[Q|R]	v_car(B,Q,MI,M1), v_cdr(B,Q,MI,M2)	ベクトルの要素であるリストを頭部と尾部とに分解する．
Q =\= Qa, Q =\= Qs	'v_=\='(Q,Qa,M2,M3), 'v_=\='(Q,Qs,M3,M4)	整数 Q と Qa, Q と Qs とがひとしくないかどうかをしらべる．
Qaa is Qa + 1	'vs_+'(Qs,1,Qaa,M4)	ベクトルの各要素とスカラ・データとの整数加算をする．
Qss is Qs - 1	'vs_-'(Qs,1,Qss,M4)	ベクトルの各要素とスカラ・データとの整数減算をする．
not_take1(R,Qaa,Qss)	v_not_take1(R,Qaa, Qss,M4,MO2)	再帰よびだしをする．
-*	v_end_or(MO1,MO2,MO)	原始プログラム第 1 節に対応する部分と第 2 節に対応する部分から出力されるマスクを合成する．

* 原始プログラムに対応する部分がない．

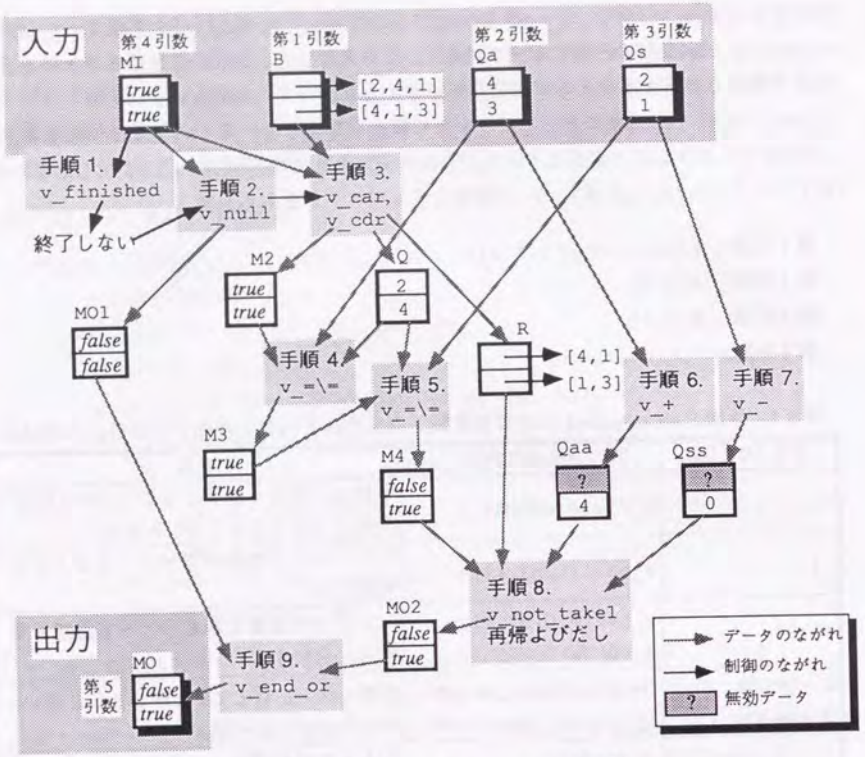


図 3.7　手続き v_not_take1 の実行例

上記の入力をあたえて v_not_take1 をよびだすことは，つぎのような 2 つの手続きよびだしを実行することとほぼ等価である．

```
?- not_take1([2,4,1], 4, 2).        ..... (3.5.1)
?- not_take1([4,1,3], 3, 1).        ..... (3.5.2)
```

(3.5.1) を実行すると失敗し (3.5.2) を実行すると成功する．それに対応して，図 3.7 の実行の結果は #(false,true) となる．実行過程については金田 [Kanada 87] でよりくわしくのべている．

3.5.2 くりかえし構造の 1 重化

エイト・クウィーン問題のプログラムにおける手続き select は，リストを入力して，

そのリストから 1 要素をえらびだした結果と，そのリストからその要素をのぞいたリストとを出力する手続きである．入力されるリストのながさを n とすれば，解の数も n 個だけある．手続き select またはそれと等価な手続き (しばしば delete という名称をあたえられる) は Prolog プログラムにおいてしばしばつかわれるが，エイト・クウィーン問題のプログラムにおいても，クウィーンのリストから 1 個のクウィーンをえらぶのにつかわれる．

select のプログラムをしめす．

(1) 原始プログラム

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

第 1 引数が入力のリスト，第 2 引数がえらばれた要素，第 3 引数がのこりの要素のリストである．

select にはその末尾に再帰よびだしがあって，その各くりかえしにおいて第 1 節から 1 個ずつ解を出力し，select のよびだしもとに復帰する．ただし，select の実行後にバックトラックが生じなければ第 2 節は起動されず，したがって再帰よびだしも起動されない．

すべての解が出力されるまでバックトラックがくりかえされることを前提とすれば，つぎのようなプログラム変換が可能になる．すなわち，1 個の解がもとめられるごとに解を出力するかわりに，ベクトルを用意してそれに解を蓄積していく．すべての解がもとめられたところで，そのベクトルを出力する．この変換後のプログラムの機能は，Prolog の手続き bagof の機能ににている．このようにことなる解を要素とするベクトルをつかうインタフェースは，3.5.1 節の手続き v_not_take1 のそれと一致している．

変換後の手続き v_select のプログラムをしめす．

(2) ベクトル化後のプログラム

```
v_select(AL, X, Y, MI, BI, BO) :-
    v_select_1(AL, X1L, Y1L, MI, ML),
    v_merge([X1L, Y1L], [X, Y], [BI], [BO], ML).

v_select_1(_, [], [], MI, []) :-
    v_finished(MI), !.
v_select_1(AL, [A'|X1L], [L'|Y1L], MI, [M1'|ML]) :-
    v_car(AL, A', MI, M0'),
    v_cdr(AL, L', M0', M1'),
```

```
v_car(AL, A, MI, M0),
v_cdr(AL, L, M0, M1),
v_select_1(L, X1L, L1L, M1, ML1),
mapcar(v_cons(A), L1L, Y1L, ML1, ML).
```

(1) から (2) への変換の手順は省略するが，その対応関係を表 3.2 にしめす．

v_select の第 1 ～ 3 引数は select の第 1 ～ 3 引数に対応している．v_select の入力時にも，出力時と同様のベクトル・インタフェースをとる．すなわち，v_select の実行開始以前に複数の解をもつ手続きが実行されていれば，それらを要素とするベクトルが第 1 引数として入力される．入力する解が 1 個のばあいも，唯一の要素をもつベクトルが第 1 引数として入力される．

表 3.2　手続き **select** における，原始プログラムとベクトル化後のプログラムとの対応

原始プログラム	ベクトル化後のプログラム	意味
-*	v_select_1(_,[],[],MI,[])	空のマルチ・ベクトルを生成する．
-*	v_finished(MI)	手続きの実行を終了するかどうか (再帰を停止させるかどうか) を判定する．
-*	v_select1(AL, [A'|X1L],[L'|Y1L], MI,[M1'|ML])	各マルチ・ベクトルに要素を追加する．
[A'|L']*1	v_car(AL,A',MI,M0'), v_cdr(AL,L',M0',M1')	ベクトルの要素であるリストを頭部と尾部とに分解する．
[A|L]*2	v_car(AL,A,MI,M0), v_cdr(AL,L,M0,M1)	ベクトルの要素であるリストを頭部と尾部とに分解する．
select(L,X,L,1)	v_select_1(AL,X1L,L1L,M1,ML1)	再帰よびだしをする．
[A|L1]	mapcar(v_cons(A), L1L,Y1L,ML1,ML)	マルチ・ベクトル L1L の各部分ベクトルの要素に関してリストを合成する．
-*	v_merge([X1L,Y1L], [X,Y],[BI],[BO],ML)	マルチ・ベクトル X1L, Y1L をそれぞれ 1 個のベクトル X, Y に併合し，ベクトル BI の要素と対応づけて BO を生成する．

* 原始プログラムに対応する部分がない．
*1 原始プログラム第 1 節に対応している．
*2 原始プログラム第 2 節に対応している．

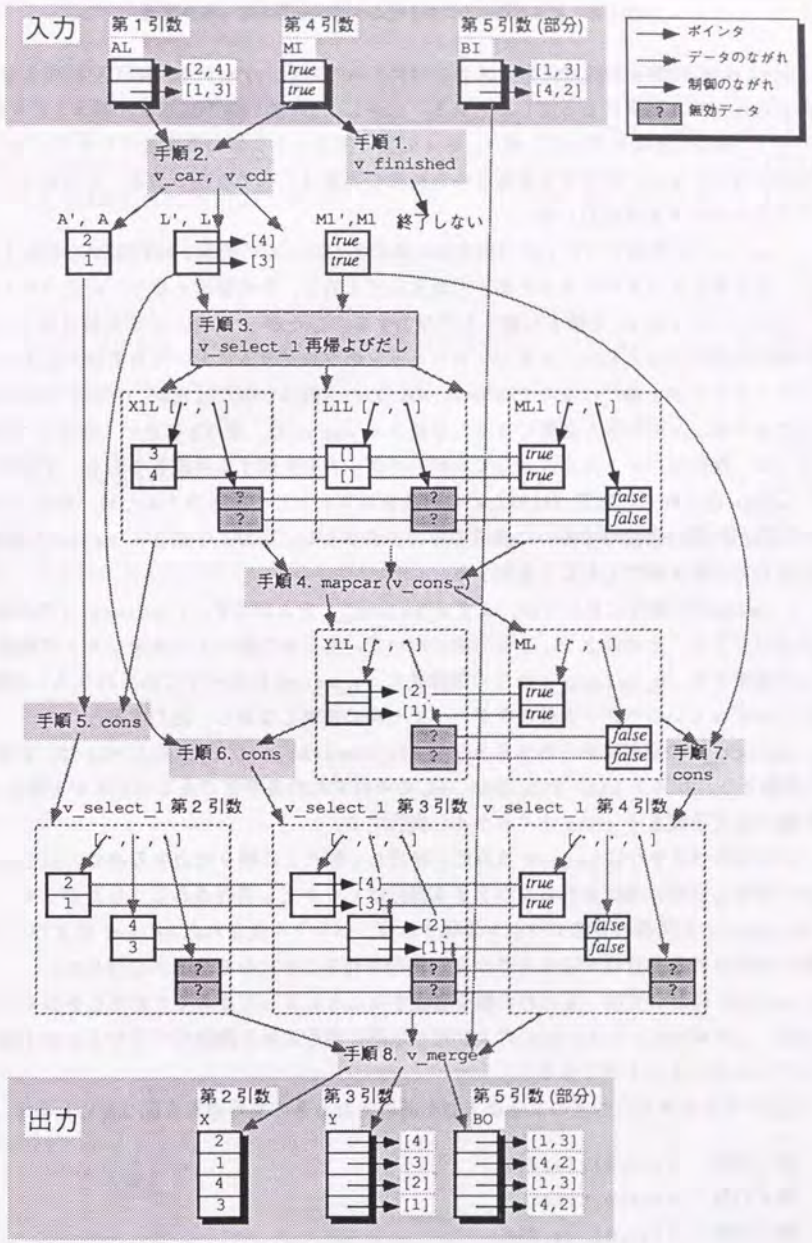


図 3.8　手続き **v_select, v_selec1_1** の実行例

v_select の第4引数は入力マスク・ベクトルである．すなわち，第1引数の各要素の有効性が第4引数によってしめされる．しかし，3.4 節の各手続きや手続き v_not_take1 とはちがって，第1，第4引数と第2～3引数の要素数はひとしくなく，要素は対応しない．出力である第2～3引数の要素はすべて有効である．したがって，マスク・ベクトルは出力しない．

v_select の各出力ベクトルの要素は，各入力ベクトルの要素とは対応がくずれるため，対応をとるべきベクトルを第5引数として入力し，その要素を複写・対応づけしてえられたベクトル BO を第6引数として出力する．したがって，BO の要素数は第2～3引数の要素数とひとしい．エイト・クウィーンのプログラムにおいては要素対応をとるべきベクトルが1個だけなので対応づけのための引数は2個だけだが，プログラムによっては4個以上の引数が必要になる．手続き v_merge は，表 3.2 において説明しているように，複数のベクトルを併合してながいベクトルを生成する手続きである．手続き v_merge はこれらの対応づけのための引数をリストとして入出力するため，対応のための引数が任意の個数のばあいに使用することができる．なお，手続き v_merge の機能に関しては第6章でくわしく説明する．

v_select の実行においては，まず v_select_1 をよびだす．v_select_1 の再帰よびだしごとに，その第2，3，5仮引数において，もとめた解ベクトルをリストの要素として蓄積する．v_select_1 からの復帰後に，v_merge においてこれらのリストの要素をそれぞれ1つのベクトル長のながいベクトルに格納しなおし，出力する．v_select_1 の定義にあらわれる mapcar(v_cons(A),L1L,Y1L,ML1,ML) は，A を第1引数とし，リスト L1L，Y1L，ML1，ML のそれぞれの各要素であるベクトルを第2～5引数として手続き v_cons をくりかえし実行する．

すでにのべたように select は再帰よびだし1回ごとに解を出力するので，select のよびだし以降に実行されるプログラム部分はくりかえし実行される．したがって，not_take1 と同様の方法でベクトル化すれば，ベクトル化された select のよびだし以降に実行される部分はやはりくりかえし実行されることになる．それに対して v_select においては，もとめた解を蓄積することによってこのくりかえしをなくしている．したがって，v_select のよびだし以降に実行される部分のくりかえしを1重化しているということができる．

(2) のプログラムにつぎのような入力をあたえたときの実行過程を図 3.8 にしめす．

第1引数： #([2,4],[1,3]).
第4引数： #(true,true).
第5引数： #([1,3],[4,2]).

上記の入力をあたえて v_select をよびだせば，つぎのような2つの手続きよびだしを実行することとほぼ等価である．

```
?- select([2,4],X,Y).  ……(3.5.3)
?- select([1,3],X,Y).  ……(3.5.4)
```

(3.5.3) を実行するとつぎのような解がえられる．

```
X=2, Y=[4].  ……(3.5.5)
X=4, Y=[2].  ……(3.5.6)
```

また，(3.5.4) を実行するとつぎのような解がえられる．

```
X=1, Y=[3].  ……(3.5.7)
X=3, Y=[1].  ……(3.5.8)
```

図 3.7 においては，(3.5.5) ～ (3.5.8) をベクトルに格納した形の解がえられているので，ただしくベクトル化されていることがわかる．

3.6 評価

3.4 節でしめした方法にしたがってエイト・クウィーン問題の Prolog プログラムを手動でベクトル化し，さらに手動で Fortran と Pascal とによるプログラムに変換したあとコンパイルした．Fortran 部分はベクトル・コンパイラでコンパイルした．Fortran コンパイラのベクトル化オプションをかえることによってベクトル処理用のコードとスカラ処理用のコードとを生成して，両者をベクトル計算機 S-810 で実行した．

実行時間を測定した結果を表 3.3 にしめす．3.4 節でしめしたマスク演算方式だけではなく，3 つの条件制御方式のそれぞれをおもに使用するプログラムをコーディングして測定した[注9]．プログラム全体としては，ベクトル処理速度はスカラ処理速度の 8 ～ 9 倍となっている．この結果のよりくわしい分析は第 6 章でしめす．

表 3.3　ベクトル計算機 S-810 むきにハンド・コンパイルした
エイト・クウィーン問題のプログラムの実行性能

プログラムの版 (主要な条件制御方式)	S-810 ベクトル処理時間 (ms)	S-810 スカラ処理時間 (ms)	加速率
マスク演算版	18	167	9.3
インデクス版	18	140	7.8
圧縮版	19	160	8.4

エイト・クウィーン問題のプログラムの各部分ごとの実行時間を測定することによって，リスト処理基本演算の加速率すなわちスカラ処理時間とベクトル処理時間の比をもとめ，表 3.4 にしめした．`v_car`, `v_cdr` および `v_null` の加速率と，`v_cons` の加速率とをしめしている．前 3 者のそれぞれの時間を個別にしめしていないのは，測定したプログラムにおいては，高速化のためにこれらが 1 個のループのなかに共存していて，個別に測定することが困難だったためである．

前 3 者の加速率は 10 倍をこえていて，満足すべき加速率だとかんがえられる．しかし，`v_cons` は 3 倍程度であり，十分な加速率とはいえない．加速率がひくいおもな原因は，`v_cons` が 1 要素あたり 3 つのポインタのストアを必要とするのに対して，S-810 の主記憶のストア・スループットが十分でないことだとかんがえられる．これは，現在のスーパコンピュータが数値計算むきに最適化されているために，とくに S-810 においてはロード・スループットにくらべてストア・スループットがひくいことの結果だとかんがえられる．

[注9] ただし，実際には各プログラムのなかで複数の方式をまぜて使用している．これは，1 つの方式だけでは実現不可能な部分があるなどの理由による．

表 3.4　エイト・クウィーン問題のプログラムにおける基本演算種別ごとの加速率

プログラムの版 (主要な条件制御方式)	`v_car`, `v_cdr`, `v_null` における加速率	`v_cons` における加速率
マスク演算版	12.1	3.1
インデクス版	13.6	3.3
圧縮版	13.7	3.1

3.7 まとめ

ベクトル計算機を使用し，プログラム変換にもとづいてリスト処理を高速に実行するための「くりかえし構造の交換」および「くりかえし構造の1重化」という戦略を提案した．この方式をエイト・クウィーンのPrologプログラムに適用して，ベクトル計算機S-810において逐次処理の約9倍の実行速度をえた．かぎられた例題に適用しただけではあるが，この結果はこれらのプログラム変換方式の有効性をしめすものとかんがえられる．したがって，これらの方法に関するより具体的な研究をおこなうとともに，論理型言語プログラムの自動ベクトル化などへの応用を検討する必要があるとかんがえられる．そこで，次章ではさらにこの章でしめした残存要素検出法とならぶくりかえし構造の交換を具体化するもうひとつの方法をしめし，実測結果にもとづいてこれらを比較する．また，これらの方法の論理型言語プログラムのベクトル化への応用に関しては3.5節でその概要をしめしたが，それについては第6章でくわしくのべる．

第4章
制御構造変換にもとづくベクトル化—2 くりかえし構造交換法の比較評価

要旨

リスト処理やデータ変換などのプログラムにおける可変長の内側くりかえしをふくむ2重のくりかえし構造(ループ，再帰よびだしなど)に，変換前の内側くりかえし回数の最大値をあらかじめもとめることによって前章で定義したくりかえし構造の交換を適用する方法を開発し，それをS-810において他の方法と比較した．その結果，交換によって十分なくりかえし回数がえられるときには，おおくのばあいこの方法による実行が他の方法より高速であることなどがわかった．この結果は，今後くりかえし構造の交換をつかった具体的なベクトル記号処理アルゴリズムの開発の参考になるとともに，論理型言語などの，最適化をともなう自動ベクトル化処理系の設計の基礎になるとかんがえられる．

4.1 はじめに

数値計算プログラムにおいて，ループ内の処理にループ依存性があるためにベクトル化できないかまたはベクトル化しても加速率がひくいばあいがある．このようなループをふくむ2重ループをベクトル化可能にするプログラム変換技法として，前章でのべたループ交換がある．ループ交換がおこなえるための必要条件は，変換前の内側ループのくりかえし回数がループの実行開始前に確定することである．数値計算におけるループは，通常この必要条件をみたしている．

一方，記号処理プログラムにおいても，ループ交換やそれを再帰よびだしなどに拡張したプログラム変換戦略であるくりかえし構造の交換は，ベクトル化において有効な方法である．ここでくりかえし構造の交換とは，前章でのべたように，多重のくりかえし構造における内側のくりかえしと外側のくりかえしとをいれかえることによってベクトル処理できないプログラムをベクトル処理可能にするための，またはベクトル処理性能を向上させるためのプログラム変換戦略のことである．くりかえし構造の変換によって，図3.2 (1.1) に PAD (Problem Analysis Diagram) をつかってしめしたようなプログラムが図3.2 (1.2) にしめしたようなプログラムに変換される．もちろん，くりかえし構造の交換ができるためにはデータフロー等に関して一定の条件がなりたっていなければならないが，それについてはここでは省略する．

ところで，リストのような可変長のデータ構造，木やグラフのような可変構造のデータ構造をあつかう記号処理においては，リストをたぐりながらの処理のように内側のくりかえしがそのままではベクトル化できず，しかもそのくりかえし回数が実行開始前に確定しないばあいがおおい．このようなばあい，交換前の内側ループが可変長であるとすると，すなわち内側ループのくりかえし回数が外側ループのくりかえしごとにことなるとすると，内側ループと外側ループの制御変数をいれかえるだけの単純なループ交換ではプログラムが不正になってしまう．第3章ではくりかえし構造の交換を適用してこのようなリスト処理プログラムのベクトル化をはかっているが，この報告ではそこで暗につかわれている具体的なベクトル処理技法を明確化するとともに，その代案をしめして，配列の線形検索に関する実測データにもとづいて両者を比較検討する．この比較結果は，今後くりかえし構造の交換をつかった具体的なベクトル記号処理アルゴリズムの開発の参考になるとともに，論理型言語などの自動ベクトル化処理系の設計の基礎になるとかんがえられる．

4.2節では，くりかえし構造変換が必要な可変長くりかえしをふくむ応用の例をしめす．4.3節では，可変長くりかえし構造の交換にもとづく2つのベクトル化のためのプログラム変換方法をしめす．4.4節では，4.3節の2つの方法を配列の線形検索に適用し

たプログラムをしめし，4.5節でその実測データをしめし，それにもとづいて両者を比較する．4.6節では関連研究についてのべる．

4.2 くりかえし構造交換が必要な可変長くりかえしの例

この節では，この報告の主題であるくりかえし構造交換の有用性をしめすため，もとのままではベクトル化に適さないがくりかえし構造を交換することによってベクトル化に適するようになる可変長くりかえし構造をふくむ応用の例をあげる．ここで可変長のくりかえし構造とは，Fortran の DO ループなどの固定長くりかえし構造とはちがって，可変長のくりかえし処理をおこなうくりかえし構造すなわち **while** ループや再帰よびだしなどのことをいう．ベクトル化に適さない理由としては，最内側のくりかえしがベクトル化に適さない，あるいはベクトル長がみじかいためにベクトル化しても十分な加速率がえられないなどの理由がありうる．

上記のような応用として，つぎのようなものがある．

❑ データベースにおけるデータ変換

レコード長が可変長でかつみじかいデータベースにおいて，データの暗号化や圧縮・伸長などの変換をベクトル処理でおこなうばあいには，2重ループの交換またはくりかえし構造の交換をともなう可変長2重ループのベクトル化をおこなうのが有効である (図 4.1 参照)．なぜなら，1つのレコードをベクトルとして表現してベクトル化しようとしても，おおくのアルゴリズムにおいてデータ依存のためにベクトル化ができないまたはできてもたかい性能がえられないからである．データ依存が存在するのは，レコードのある部分の処理にそれ以前の部分の結果がつかわれるばあいであるが，暗号化や圧縮・伸長のアルゴリズムにはこのような依存があるばあいがおおい．また，ベクトル化してもたかい性能がえられない理由は，レコード長がみじかいためにベクトル長もみじかく，たかい加速率がのぞめないからである．一方，2重ループの交換をおこなえば，ベクトル要素間の依存関係はなくなってベクトル化に関する障害がなくなるうえ，レコード数が十分におおければ十分なベクトル長を確保することができ，したがって実行を加速することができる．

データベースにおいては，上記のような操作以外にもくりかえし構造の交換の適用によってベクトル化が可能になったりベクトル処理性能が向上する処理があり，その一部は M-680H IDP においても実現されている．

❑ 複数のリストの各要素への処理

前節でもふれたように，リストの各要素に同種の処理をほどこすばあいにおいては，リストを逐次的にたぐらなくてはならないため，1つのリストに関する処理をベクトル処理することは困難である．しかし，このような処理を多数のリストに対しておこなうばあいには，前章でしめしたように，2重ループの交換 (または2重のくりかえし構造の交換) によってベクトル化し，実行を加速することができる．

❑ 複数の収束計算

ニュートン法で代表される収束計算は，直前のくりかえしの結果を使用する．したがって，1つの収束計算をベクトル処理することはできない．しかし，リスト処理のばあいと同様に，同種の収束計算を多数のデータに対してくりかえし実行するばあいには，2重ループの交換または2重のくりかえし構造の交換によってベクトル化し，実行を加速することができる．ただし，性質がよい関数に関するニュートン法のように収束がはやいばあいは定数回の反復で十分であり，このようなばあいには内側ループを展開してしまえばよい．ループ交換またはくりかえし構造の交換を適用するのが適当なのはより収束がおそい計算である．

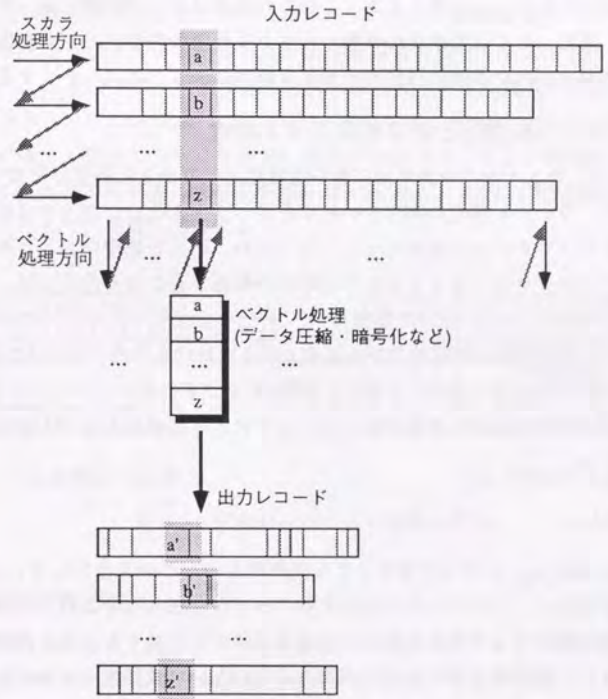


図 4.1　くりかえし構造交換によりベクトル処理好適となる例: データ変換

4.3 可変長くりかえし構造交換のための技法

この節では，可変長くりかえし構造の交換によるベクトル化において終了判定コード生成とオーバラン無効化という2つの処理をおこなう必要があることを説明し，残存要素検出法および最大回数反復法という，これらの処理に関してことなる方法をもちいた2つのくりかえし構造交換法をしめす．

そのまえに，記述の便宜のためにこの章で使用するいくつかの用語をここで定義しておく．くりかえし構造の交換において，対応している交換前の外側くりかえしと交換後の内側くりかえしとを原外くりかえしとよぶことにする．同様に，対応している交換前の内側くりかえしと交換後の外側くりかえしとを原内くりかえしとよぶことにする．また，原外くりかえしのくりかえし回数を原外くりかえし回数，原内くりかえしのくりかえし回数を原内くりかえし回数とよぶ．これらの名称はループ交換においても使用することにする．また，とくに交換前の外側ループと交換後の内側ループとを原外ループ，交換前の内側ループと交換後の外側ループとを原内ループとよぶことにする．

4.3.1 オーバラン無効化と終了判定コード生成

4.2節で例示したようなプログラムにおいては原内くりかえし回数が可変であるため，単純に2重のくりかえし構造の内外をいれかえてベクトル化しただけでは原内くりかえし回数がもとのプログラムとは変わってしまうため，不正な処理がおこなわれるようになる．もとのプログラムとちょうどおなじだけの処理をおこなうためには，つぎのような2つの処理が必要である (図4.2参照)．

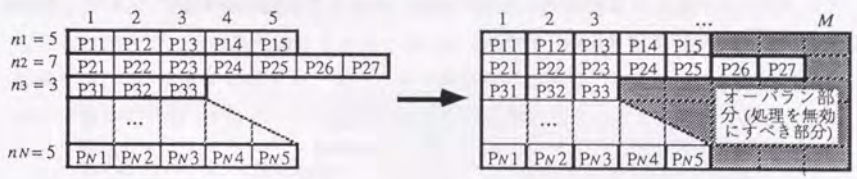
第1に，ベクトル化前の原外くりかえしの i 回目における原内くりかえし回数を m_i とするとき，ベクトル化後の原内くりかえし回数 M をつぎの条件をみたすように適当にきめ，それをこえるくりかえしをおこなわないようにする必要がある．これを終了判定コード生成とよぶ．

$$M \geq max_i(m_i)$$

ここで，$M = max_i(m_i)$ とするのがもっとも効率がよい．このようにくりかえしをうちきるコードは変換前のプログラムには対応するコードが存在しないため，それを明に生成しないと，変換後のプログラムにおいてはばあいによってはくりかえしが停止しない．たとえば，リスト処理のように最大長がおさえられない可変長データのばあいには，くりかえしが停止しなくなる．したがって，M 回をこえるくりかえしをおこなわないようにするコードを生成する必要がある．これに対して，4.4節でしめすような配列を操作するプログラムのばあいには，変換前のプログラムにくりかえし回数を配列の最大長でおさえるコードがあるために問題はおこらない．したがって，このばあいにはかならずしも終了判定コードを生成する必要はない．

第2に，ベクトル処理においてベクトルの第 i 要素であって $M > m_i$ をみたすものに対する処理は余分であって，この部分を実行すると不正な結果がえられるおそれがあるので，無効にする必要がある．この無効化すべき部分をオーバラン部分とよび，この無効化をオーバラン無効化とよぶことにする．オーバラン無効化のためには，ベクトル計算機に用意された3種類の条件制御方式 [Kamiya 83] すなわちマスク演算方式，収集拡散方式，あるいは圧縮伸長方式をつかえばよい．これらを利用した記号処理のための条件制御方式を，われわれはそれぞれマスク演算方式，インデクス方式，圧縮方式とよんでいる．リスト処理においてこれらの条件制御方式をどのように使用すればよいかについては前節でのべたとおりであるが，同様にしてリスト処理以外へも容易に適用することができる．したがって，ここではオーバラン無効化の方法をくわしくはのべないが，マスク演算方式のばあいだけについてかんたんにのべると，処理対象のベクトルと要素数のひとしいマスク・ベクトルを用意して，無効な要素に対応するマスク・ベクトルの要素は *false* とし，このマスク・ベクトルの制御のもとでベクトル処理を制御すればよいということになる．

次節以降では，上記のようなくりかえしの終了判定とオーバラン無効化に関してことなる方法をふくんだくりかえし構造交換法を2つしめし，配列の線形検索に関する実測データをもとにして，両者を比較する．



(a) 交換前の処理　　(b) 交換後の処理

図4.2　可変長くりかえし構造の交換

4.3.2 残存要素検出法

残存要素検出法による2重ループの変換前および変換後のプログラムを図4.3にしめす (変換後のプログラムもスカラ処理のままのかたちでしめしている)．この方法は，2重ループの実行を開始するまえに原外くりかえし回数がもとめられるばあいに適用することができ，原内くりかえし回数の最大値はあらかじめわからなくても適用することができる．残存要素検出法は従来から部分的にしられており，4.2節でふれたM-680H

IDP の可変長データ処理機能もインデクス方式 (リスト・ベクトルを使用する方式) にもとづく残存要素検出法の使用を前提として設計されている.

残存要素検出法においては, 変換後の内側ループはベクトル処理とし, その実行においてくりかえしごと, すなわちベクトルの要素ごとに条件 $M > m_i$ がみたされているかどうかの判定をおこない, その結果をマスク・ベクトルとする. そのマスク・ベクトルによってオーバラン無効化をおこなうとともに, 有効な要素 (残存要素とよぶ) が存在するかどうかを判定し, 残存要素がなくなるまで外側ループを実行する. 図 4.3 の変換後のプログラムにおいては内側ループに条件文がふくまれているが, このプログラムに自然なベクトル化をおこなうと, その条件式がマスク・ベクトルを生成する命令にコンパイルされる.

残存要素がなくなったかどうかの判定にはいくつかの方法がかんがえられるが, ベクトル計算機 S-810 / S-820 においてはこのためにマスク計数命令をつかうことができる. 図 4.3 のプログラムはマスク計数命令の使用を前提としている. 図 4.3 では変数 *count* によって上記の条件式が真となる内側くりかえし回数すなわち *true* であるマスク・ベクトルの要素の数がかぞえられる. なお, 図 4.3 においては内側ループのくりかえし回数はつねに M 回としているが, ここで残存要素だけを処理するようにすればくりかえし回数をへらすことができる. また, このばあいはくりかえし回数が残存要素数にひとしいので, マスク計数命令を使用せずに終了判定をおこなうことができる. 条件制御のためにインデクス方式または圧縮方式を採用すればこのような最適化が可能になるが, その具体的な方法については 4.5 節でのべる.

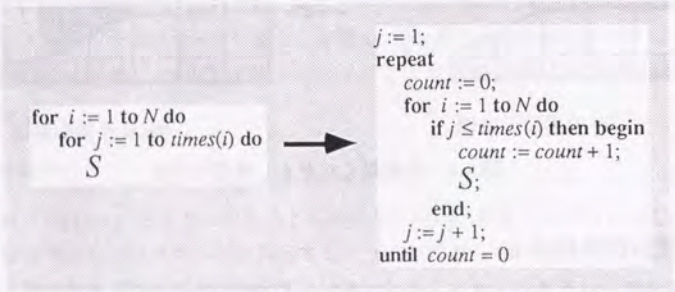


図 4.3　残存要素検出法による可変長 2 重ループのベクトル化
(マスク計数によるインプリメント)

4.3.3 最大回数反復法

上記のように, インデクス方式や圧縮方式にもとづく最大回数反復法においてはループのくりかえし回数が残存要素数にひとしいため, 残存要素検出法を余分のオーバヘッドなしに実現することができる. しかし, インデクス方式においてはデータのアクセスにインデクス・ベクトルを使用するため, もともとそのオーバヘッドがおおきい. また, 圧縮方式は外側ループのくりかえしごとにすべてのベクトルの圧縮をおこなうためにオーバヘッドがおおきい. 一方, マスク演算方式と残存要素検出法とをくみあわせると, 原内ループのくりかえしごとに, 残存要素をかぞえるためにマスク・ベクトルの *true* という値をもつ要素の数をかぞえる必要が生じる[注1]. この演算は各要素の処理のあいだにデータ依存 (ループ依存) があるために比較的オーバヘッドがおおきいとかんがえられる. この高価な演算をおこなうことなしにマスク演算方式をつかったくりかえし構造交換を可能にするあらたな方法が, 最大回数反復法である.

最大回数反復法による 2 重ループの変換前および変換後のプログラムを図 4.4 にしめす[注2]. この方法は, 2 重ループの実行を開始するまえに原外くりかえし回数がもとめられるとともに, 変換前の原内くりかえし回数の最大値 (または最大値よりおおきい適当な値) *Mmax* があらかじめもとめられるばあいに適用することができる.

最大回数反復法においては, 変換後のプログラムにおいて *Mmax* をループの実行前にもとめて, *Mmax* を変換後の原内くりかえし回数とする. 図 4.4 においては *times*(i) が原外ループの i 回目のくりかえしにおける原内ループのくりかえし回数をあらわしている. したがって, その最大値をもとめて *Mmax* の値としている. 変換後の内側ループ (原外ループ) はベクトル処理する. オーバラン無効化の方法については 4.3.1 節でのべたとおりであるが, 図 4.4 においては原内ループ内の条件文によってオーバラン無効化がなされている.

[注1] S-810 / S-820 においては VMCO (Vector Mask Count Ones) 命令を実行する.

[注2] 変換後のプログラムも, ベクトル化前のかたちすなわちループ分散 (3.2 節参照) をほどこさないスカラ処理のままのかたちでしめしている. しかしながら, 変換前のプログラムはベクトル化に適さず, 変換後のプログラムはベクトル化に適する.

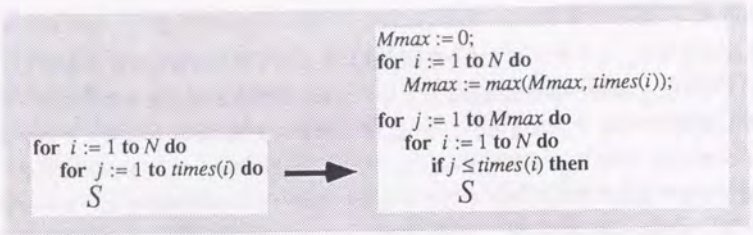


図 4.4　最大回数反復法による可変長 2 重ループのベクトル化

4.4 配列線形検索への適用例

この節では，最大回数反復法と残存要素検出法を使用したアルゴリズムの実例として，配列の線形検索のアルゴリズムをしめす．これらのアルゴリズムにもとづくプログラムの性能比較を次章でおこなう．

まず，配列の線形検索を例題としてとりあげる理由をのべる．第 1 の理由は，線形検索はアルゴリズムが単純であり，したがって最大回数反復法，残存要素検出法の本質がみきわめやすいとかんがえられることである．第 2 の理由は，これらの方法に直接の関係がない部分がしめる割合が実行のうえでもひくいため，性能上もこれらの方法の比較に適しているとかんがえられることである．第 3 の理由は，このアルゴリズムはループ交換なしでもベクトル化できるため，ループ交換せずにベクトル化したばあいとの比較もできることである．第 4 の理由は，みじかい配列の線形検索は実用的なアルゴリズムであり，この章でとりあげるベクトル処理アルゴリズムも実用性があり，その測定データは実用的な用途をかんがえるときに直接参考にすることができるとかんがえられることである．ただし，それにしても配列線形検索はひとつの例にすぎず，これだけで最大回数反復法と残存要素検出法の完全な比較がなされるわけではない．この点に関しては 4.5 節でふたたび考察する．

なお，4.5 節でしめす実測には Fortran によって記述したプログラムを使用するが，この章では Pascal 風の言語でアルゴリズムを記述する．その理由は，Fortran ではその言語上の制約のためにアルゴリズムの記述が複雑化する部分があるからである．Fortran のプログラムは付録 1 にしめしているので，必要に応じて参照されたい．

4.4.1 スカラ処理アルゴリズム

スカラ処理のための配列線形検索のアルゴリズムを図 4.5 にしめす．このアルゴリズムの機能を図 4.6 に例示する．入力は検索表を要素とする配列 *table*（2 重の配列または 2 次元の配列）と検索すべきデータからなる配列 *data_to_search* であり，出力は検索表への添字からなる配列 *found* である．*data_to_search* と *found* の配列としてのおおきさはともに N であり，N も引数として入力するものとする．このアルゴリズムは，ベクトル処理アルゴリズムとあわせるため，通常の検索アルゴリズムとはことなるインタフェースを採用している．すなわち，検索すべきデータは複数個あり，それらを配列 にいれてわたしている．検索データ *data_to_search*[i] が検索表 *found*[i] に登録されているときは，その検索表エントリの添字（図 4.6 ではイタリック体の数字でしめしている）を *found*[i] に代入する．登録されていないときは，*found*[i] の値は 0 とする．図 4.6 にしめしたように，検索表の複数のエントリに検索データとひとしい値がふくまれるときには，そのなかの最初のエントリの添字だけが *found*[i] に代入される．

なおこのアルゴリズムにおいては，むだな計算をはぶくため，検索データとひとしい値が検索表にみつかったときは内側ループを脱出するようにしているが，そのために自然なベクトル化(ループ分散技法だけによるベクトル化) はできなくなっている．

```
procedure s_search (table,data_to_search, found,N);
begin
  found[1..N] := 0;                    — 配列 found の全要素を 0 に初期化する．
  for i := 1 to N do begin             — 検索データに関するくりかえし．
L:  for j := 1 to size(table[i]) do    — 検索表のエントリに関するくりかえし．
      if data_to_search[i] = table[i,j] then begin
        found[i] := j;                 — みつかった検索表エントリの添字を found[i] に代入する．
        exit L;                        — むだな計算をはぶくため，内側ループを脱出する．
      end;
  end;
end s_search;
```

図 4.5　配列線形検索スカラ処理アルゴリズム

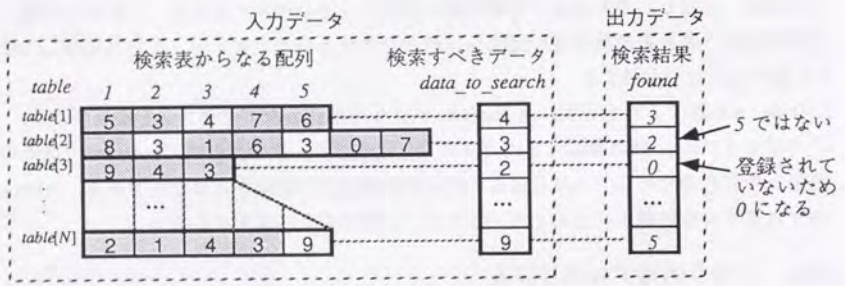


図 4.6　配列線形検索アルゴリズムの機能

4.4.2 ループ非交換版のベクトル処理アルゴリズム

図 4.6 のアルゴリズムに対しては自然なベクトル化ができないので，ループ脱出をなくしてベクトル処理可能にした配列線形検索アルゴリズムを図 4.7 にしめす．このアルゴリズムを Fortran でコーディングすれば，ループ交換などの特別なプログラム変換なしに自然にベクトル化することができる．すでにのべたように，スカラ処理版のアルゴリズムにおいては，検索表の複数のエントリに検索値がふくまれるときには，この **exit** 文があることによって最初のエントリの添字だけが *found*[*i*] に代入される．しかし，この **exit** 文をなくすとこれらの複数のエントリの添字がすべて *found*[*i*] に代入されるため，*found*[*i*] には最後に代入された添字の値がのこる．したがって，ベクトル処理版のアルゴリズムにおいてスカラ処理版とおなじ結果をえるためには，ループの実行の向きを逆

転させる必要がある[注3]．こうしてえられたアルゴリズムが図 4.7 である．

```
procedure vm_search (table,data_to_search, found,N);
begin
  found[1..N] := 0;
  for i := 1 to N do
    for j := size(table[i]) to 1 by -1 do
                                       — スカラ処理版とひとしい結果をえるため，逆順に実行する．
      if data_to_search[i] = table[i,j] then
        found[i] := j;                 — ベクトル化可能にするため，ループ外脱出を削除する．
end vm_search;
```

図 4.7　ループ非交換版の配列線形検索ベクトル処理アルゴリズム

4.4.3 マスク演算方式最大回数反復法によるベクトル処理アルゴリズム

最大回数反復法を使用してループ交換をおこなった配列線形検索アルゴリズムを図 4.8 にしめす．このアルゴリズムも特別なプログラム変換なしに自然にベクトル化することができる．このアルゴリズムにおいては，変数 *mx* にループ非交換版の原内ループのくりかえし回数の最大値をもとめている．原外ループの *i* 番目のくりかえしにおける原内くりかえし回数は，スカラ処理アルゴリズムのようにループからの脱出をしないかぎりは検索表 *table*[*i*] のおおきさにひとしいから，その最大値をもとめて *mx* に代入している[注4]．このアルゴリズムでは，条件制御の方法としてはマスク演算方式によっている．すなわち，最内側ループにふくまれる条件文が，自然なベクトル化によってマスクつき演算に変換される．

インデクス方式による最大回数反復法はためさなかった．その理由はつぎのとおりである．インデクス方式においてはいずれにしても有効要素の数をかぞえる必要がある．なぜなら，それがインデクス・ベクトルのベクトル長であり，したがってベクトル演算に必要だからである．有効要素数がわかれば残存要素検出法をつかうことができ，そのほうがむだな計算をしなくてすむ．したがって，インデクス方式を採用したばあいには，最大回数反復法よりも残存要素検出法のほうが有利である．これに対して，マスク演算方式においては有効要素の数をかぞえる必要はなく，2 重ループの外であらかじめくりかえし回数の最大値をもとめておくほうがむだがすくないとかんがえられる．したがっ

[注3] したがってこのプログラムにおいてはオーバラン部分をさきに実行している．

[注4] スカラ処理においてループから脱出する内側くりかえし以降のくりかえしを (どのアルゴリズムであろうと) 実行するのはむだであるから，できればスカラ処理におけるくりかえし回数の最大値を *mx* に代入するのがのぞましい．しかし，スカラ処理におけるくりかえし回数は実行してみてはじめて確定するものだから，それをあらかじめもとめるのは不可能である．したがって，最大回数反復法においては，むだを承知でループ非交換版におけるくりかえし回数の最大値をくりかえし回数 *mx* とするほかはない．

て，マスク演算方式を採用したばあいには，最大回数反復法のほうが有利になりうる．

```
procedure vmx_search (table,data_to_search, found,N);
begin
  found[1..N] := 0;
  mx := max(size(table[1..N]));
       — 変換前の内側ループのくりかえし回数の最大値 mx をもとめる．
  for j := mx to 1 by -1 do          — ループ非交換版における内側ループ．
    for i := 1 to N do               — ループ非交換版における外側ループ．
      if data_to_search[i] = table[i,j] and j ≤ size(table[i]) then
                                     — j ≤ size(table[i]) は余計な処理をおさえるための条件．
        found[i] := j;
end vmx_search;
```

図 4.8　マスク演算方式最大回数反復法による配列線形検索ベクトル処理アルゴリズム

4.4.4 インデクス方式残存要素検出法によるベクトル処理アルゴリズム

残存要素検出法を使用してループ交換をおこなった配列線形検索アルゴリズムを図 4.9 にしめす．このアルゴリズムも特別なプログラム変換なしにベクトル化することができる．条件制御の方法としてはインデクス方式によっている．このアルゴリズムにおいては，配列 *index* がインデクス・ベクトルであり，配列 *table*，*data_to_search* および *found* のすべての処理すべき要素の添字 (インデクス) が格納されている．また，配列 *index* のおおきさが *ndata* に格納されている．

index には，最初はすべての配列要素の添字 $1, 2, \ldots, N$ が格納される．すなわち，初期値設定部においてこのように *index* および *ndata* の値が設定される．検索処理がすすむにつれて，すでに検索が終了した配列要素の添字は *index* からとりのぞかれていく．すなわち，さらに検索すべき要素の添字だけが *index* につめかえられる．内側ループのベクトル長は *ndata* であるから，外側ループのくりかえしがすすむにつれてしだいに内側ループのベクトル長はみじかくなっていくことになる．そして，*index* に格納された添字がなくなると，すなわち *ndata* の値が 0 になると，処理は終了する．

なお，マスク演算方式によって残存要素検出法を実現することも可能だが，そのアルゴリズムの記述はここでは省略する．

4.5 配列線形検索における性能実測結果とその検討

前節でしめした各アルゴリズムおよびマスク演算方式による残存要素検出法を Fortran でコーディングして，その性能をパイプライン型ベクトル計算機 S-810 で実測した．その主要な結果を図 4.10 ～ 4.12 にまとめる．図 4.10 ～ 4.12 で表現しきれない部分については，補足的にしめす．最大回数反復法および残存要素検出法の性能をうまく説明するモデルの記述に成功していないため，ここではそれらの測定データのなかから代表的とかんがえられるものをえらんでしめしている．

4.5.1 測定内容および条件

図 4.10 ～ 4.12 は，それぞれ検索データ数 N が 16, 128, 1024 のときの配列線形検索の S-810 における検索時間をしめしている (数値を付録 2 にしめした)．これらの図に関する測定条件および表示条件はつぎのとおりである．検索データおよび検索表の数すなわち原外ループのくりかえし回数 N を横軸にとっている．図 4.10 に $N = 16$ のばあい，図 4.11 に $N = 128$ のばあい，図 4.12 に $N = 1024$ のばあいをしめしている[注5]．原内ループのくりかえし回数すなわち検索表のおおきさ m_i $(i = 1, 2, \ldots, N)$ は $[0, M)$ すなわち $0 \leq m_i < M$ という範囲の整数値をとる一様乱数によってきめた．図 4.10 ～ 4.12 の横軸にとったのが M の値である．登録データおよび検索データも $[0, M)$ という範囲の整数値をとる一様乱数を使用している (データの値域をかえると性能が変化するが，この変化については後述する)．検索表のおおきさが上記の乱数であたえられるとき，最大回数反復法における $Mmax$ の推定値は $M(N+1)/(N+2)$ となる (付録 3 に証明をしめす) から，N が十分おおきいばあいには，$M \approx Mmax$ とみなしてさしつかえない．

[注5] この測定をおこなうまえに，さまざまな条件のもとでの測定をためしにおこなった．その結果，データを効果的にまとめるには $N = 16$, 128, 1024 のばあいをそれぞれ 1 枚のグラフにするのがよいという結論に達したため，これらのばあいのデータをとりなおした．これらの図で表現しきれない原内くりかえし回数への実行時間の依存性などについては，図 4.14 以下で補足的にしめす．

```
procedure vic_search (table, data_to_search, found, N);
  var index, count, ndata, ix, j;
begin
  found[1..N] := 0;

/* 配列 index および変数 ndata の初期値設定: */
  count := 0;
L1:
  for i := 1 to N do
    if 1 ≤ size(table[i]) then begin
      count := count + 1;          — 検索すべきデータの数をかぞえる.
      index[count] := i;
          — 処理すべき要素の添字を配列 index に (S-810 / S-820 のばあい, ベクトル圧縮命令
          — VSTC (Vector STore Compressed) によって) 格納する.
    end;
  ndata := count;                  — くりかえし回数の初期値設定.

/* 検索: */
  j := 1;
  repeat                           — スカラ処理版における内側ループ.
    count := 0;
    for i := 1 to ndata do begin   — スカラ処理版における外側ループ.
      ix := index[i];
      if j ≤ size(table[ix]) then begin
        if data_to_search[ix] = table[ix, j] then
          found[ix] := j
        else begin
          count := count + 1;      — さらに検索をつづけるべきデータの数をかぞえる.
          index[count] := ix;
            — 検索をつづけるべき要素の添字だけを, ベクトル圧縮命令によって
            — 配列 index につめかえる.
        end;
      end;
    end;
    j := j + 1;
    ndata := count;                — くりかえし回数の更新.
  until count = 0;                 — 残存要素数が 0 になるまでくりかえす.
end vic_search;
```

図 4.9　インデクス方式残存要素検出法による配列線形検索ベクトル処理アルゴリズム

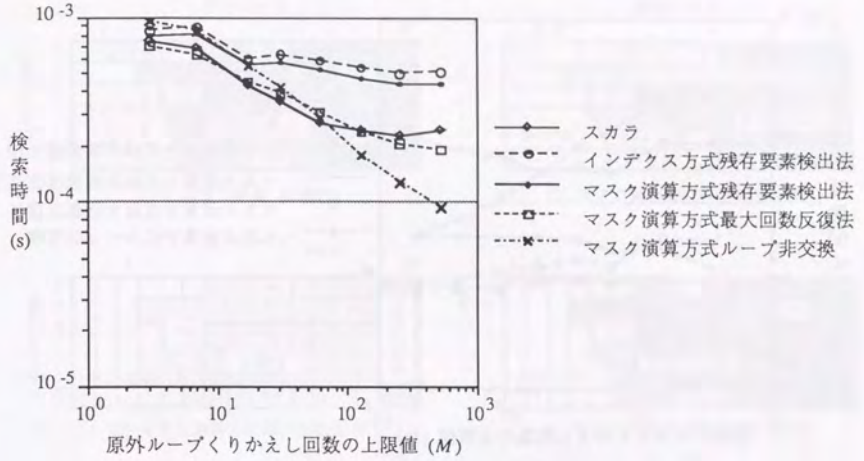


図 4.10　検索データ数 $N = 16$ のときの検索時間

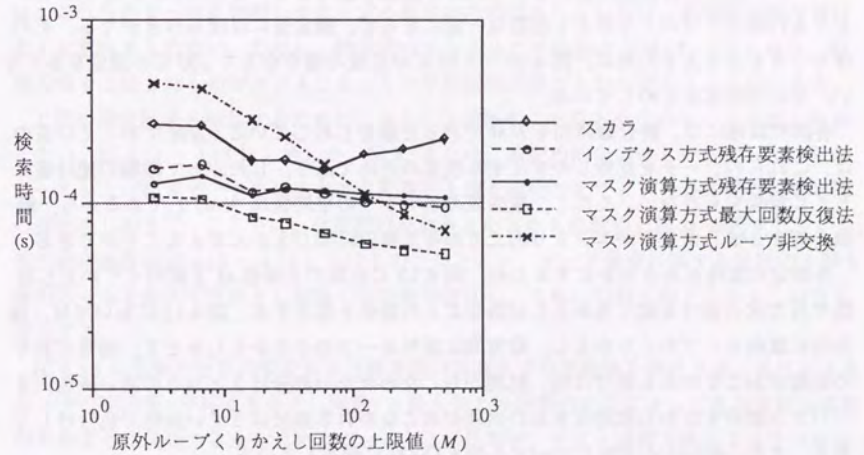


図 4.11　検索データ数 $N = 128$ のときの検索時間

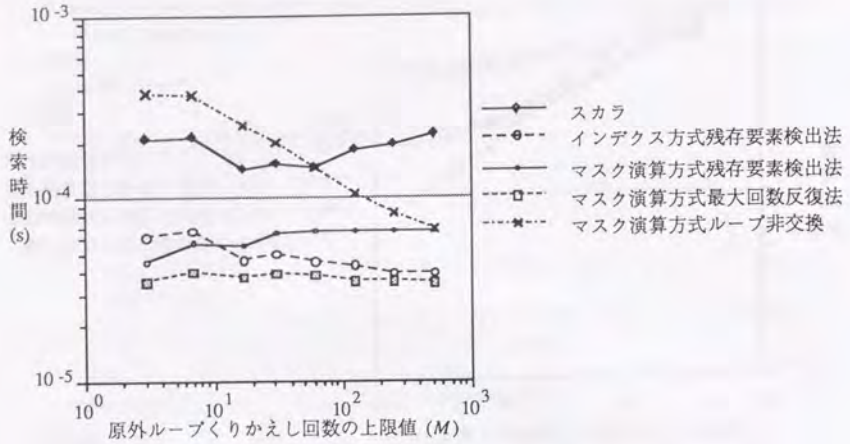


図 4.12　検索データ数 $N = 1024$ のときの検索時間

データの生成と検索表のおおきさの決定に乱数を使用しているために，スカラ処理における内側ループのくりかえし回数は一定にならず，測定値にはばらつきがでる．そのばらつきをおさえるために，図 4.10 ～ 4.12 には乱数の値をかえて 100 回の測定をおこない，その平均値をしめしている．

各図の縦軸には，総登録時間を NM でわった値をしめしている．NM でわっているのは，これらのデータを比較しやすくする便宜のためであり，したがって縦軸の絶対値はとくに意味をもたない．ただし，検索表のエントリの平均値は $NM/2$ であるから，縦軸の値を 2 倍したものがエントリあたりの平均検索時間だとかんがえることができる．

各測定の意味をあきらかにするため，図 4.13 に乱数の上限値 M と原内くりかえし回数や各方式における総くりかえし回数などとの関係を図示する．図 4.13 においては，横方向に原内ループのくりかえし，縦方向に原外ループのくりかえしをとり，通常どおりの処理がおこなわれる部分は白，処理がおこなわれない部分は 2 とおりの濃い灰色，オーバラン部分すなわち無効化された処理がおこなわれる部分はうすい灰色で色分けしてある．また，$Mmax$ の意味についても図 4.13 (3) に図示している．

データの生成と検索表のおおきさの決定に乱数を使用しているために，原内ループのくりかえし回数は一定にならず，測定値にはばらつきがでる．そのばらつきをおさえるために，図 4.10 ～ 4.12 には乱数の値をかえて 100 回の測定をおこない，その平均値をしめしている．

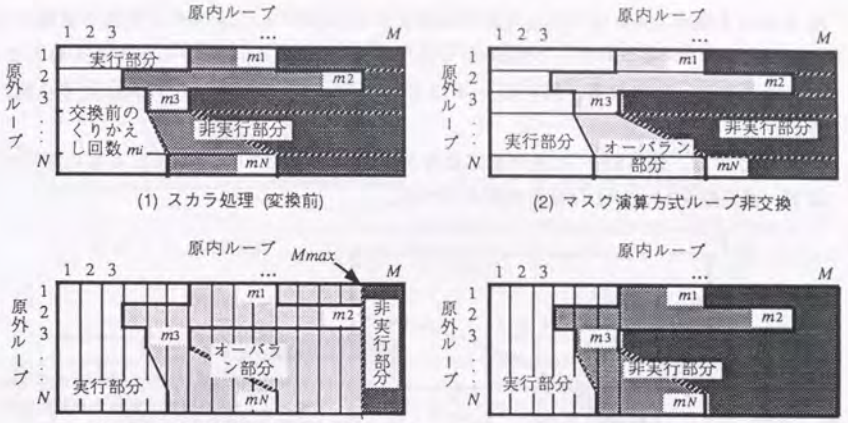


図 4.13　各方式における検索処理の実行部分と非実行部分

各図の縦軸には，総登録時間を NM でわった値をしめしている．NM でわっているのは，これらのデータを比較しやすくする便宜のためであり，したがって縦軸の絶対値はとくに意味をもたない．ただし，検索表のエントリの平均値は $NM/2$ であるから，縦軸の値を 2 倍したものがエントリあたりの平均検索時間だとかんがえることができる．

上記の測定結果を補足するために，さらに 2 種類のことなる条件のもとでの測定をおこない，図 4.14 ～ 4.15 にしめした．図 4.14 には，前記の測定条件とはちがって，登録データおよび検索データを発生させる乱数の上限を変化させたときの各方式の検索時間の変化をしめす[注6]．ここでは，検索表のおおきさをきめる乱数の上限 M は固定している．他の測定条件は図 4.10 ～ 4.12 におけるのとひとしい．データ発生に関する乱数の上限を増加させると原内くりかえし回数の平均値が増加し，それにつれて総くりかえし回数も増加する．

図 4.15 は乱数の分布の変化により検索表のおおきさの平均値を変化させ，それによるループの平均長 (平均くりかえし回数) と最大長との比率の変化によって各方式の加速率がどのように変化するかをしめしている．この比率が，マスク演算方式によってつねに M 回のくりかえしをおこなうばあいのマスク成立比率すなわちマスク・ベクトルにおける真である要素の比率となる．ただし，マスク演算方式最大回数反復法においては，通常は M 回よりすくないくりかえししか実行しないので，実際のマスク成立比率はこれよりややたかい．測定条件は図 4.10 ～ 4.12 のばあいとほぼおなじだが，乱数の分布だけがことなる．ループの平均長と最大長との比率は，つぎのようにして変化させた．一様乱

[注6] ただし，マスク演算方式による残存要素検出法の検索時間は測定していない．

数 X のべき乗をとることによって分布を変化させ，結果としてえられる乱数の値域は [0, M) に固定する．これにより，乱数 X^i を使用したばあいにはループの平均長と最大長との比率は $1/(i+1)$ となる．図 4.10 ～ 4.12 と同様の $i=1$ のばあいには成立比率は 0.5 である．

図 4.16 には，図 4.15 ～ 4.16 の測定条件を図 4.10 ～ 4.12 の測定条件とともに図示する．以下，測定結果およびその検討内容をのべる．

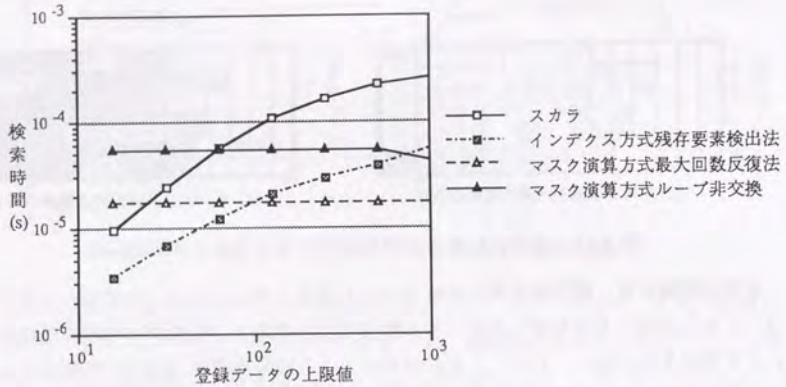


図 4.14　登録データ (一様乱数) 上限値の変化による検索時間の変化例
(N = 1024, M = 511)

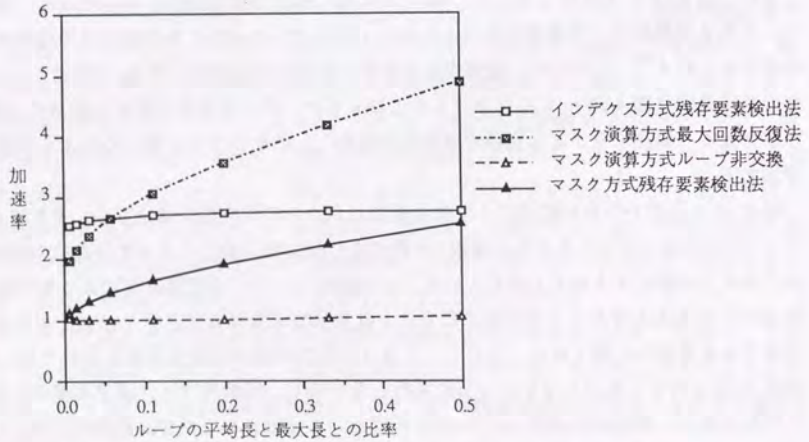


図 4.15　ループの平均長と最大長との比率による加速率の変化例
(N = 1024, M = 61)

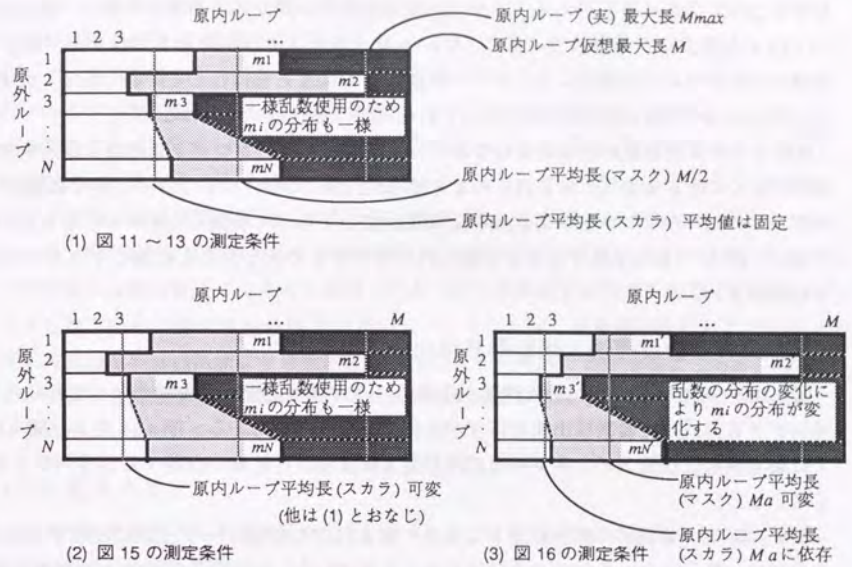


図 4.16　図 4.14 ～ 4.15 の測定条件

4.5.2 ループ交換にもとづくベクトル化の実用性

線形検索が実用的なのは原内ループがみじかいばあい，したがってループ非交換にもとづく方法にくらべて 2 つのループ交換にもとづく方法が有利なばあいである．図 4.11 ～ 4.12 は，このようなばあいにループ交換にもとづくベクトル化法によって高い加速率がえられ，したがって実用になりうることをしめしている．一方，線形検索じたいが実用的でない内側ループがながいばあいのデータは，線形検索のデータとしてはやくにたたないが，他の応用における最大回数反復法と残存要素検出法との関係およびこれらの方法と他の方法との関係を示唆しているという点において，示唆をあたえているとかんがえられる．すなわちこの結果は，ループ交換なしにはベクトル化できない応用においてはもちろん，ループ交換をしなくてもベクトル化可能でありかつ原内くりかえし回数がながい応用においても，ループ交換によって性能をたかめることができるばあいがあることを示唆している．

4.5.3 原外くりかえし回数への実行時間の依存性

原外くりかえし回数がちいさいばあいには，それをベクトル長とするループ交換をと

もなう2つの方法は不利であり，ループ非交換版のアルゴリズムが有利である．図4.10～4.12の測定データに関していえば，$N=16$ のときはスカラ処理あるいはループ非交換版のプログラムが比較的よく，ループ交換をともなう方法はむしろ不利である．ただし，$N=16$ かつ $M \leq 16$ のばあいはいずれの方法でも性能に大差はない．

原外くりかえし回数が増加するにつれて，ループ交換したプログラムのほうが有利になっていくことがわかる．$N=128$ のときは $M \geq 256$ においてまだループ非交換版のプログラムのほうがインデクス方式残存要素検出法とくらべて性能がたかいが，$N=1024$ においては M の値によらずループ交換したプログラムのほうがともに他のプログラムより性能がよい．

4.5.4 最大回数反復法と残存要素検出法との比較

図4.10～4.12においては最大回数反復法のほうが残存要素検出法にくらべて有利だが，インデクス方式残存要素検出法のほうが有利になるばあいもある．図4.14および図4.15では最大回数反復法とインデクス方式残存要素検出法との性能が交差する測定例をしめしている．

まず，図4.14に関する観察結果をしるす．図4.14では登録データ生成に使用する乱数の分布は一様分布のままでその上限値を変化させている．一方，検索表のおおきさをきめる一様乱数の上限は511に固定している．スカラ処理プログラムの実行時間は総くりかえし回数に比例して増加する．すなわち，実行時間はデータ発生に関する乱数の上限に比例する．インデクス方式残存要素検出法によるプログラムにおいてはむだなくりかえしがないため，総くりかえし回数がスカラ処理におけるそれと一致している．したがってその実行時間も，スカラ処理プログラムのばあいよりはゆるやかだが，ほぼならんで増加する[注7]．これらの変化曲線が上に凸となっているのは，原内くりかえし回数が検索表のおおきさでおさえられるからである[注8]．一方，ループ非交換版およびマスク演算方式最大回数反復法によるベクトル処理プログラムの原内くりかえし回数は検索表のおおきさに固定されていて変化しないため，総くりかえし回数も変化しない．したがって，それらの実行時間は乱数の上限を変化させてもほぼ一定である．なお，図4.14にしめした $N=1024$, $M=511$ のばあいは上記の観察結果がもっとも顕著にあらわれるが，N, M としてよりちいさな値をとっても，結果は基本的にはかわらないことがたしかめられている．

[注7] これらが比例しないおもな理由は，インデクス方式残存要素検出法にベクトル処理オーバヘッドが存在するからである．

[注8] ここではデータはしめさないが，M すなわち検索表のおおきさの上限を増大させるとスカラ処理プログラムおよびインデクス方式残存要素検出法のプログラムの実行時間の変化はより直線的になることがたしかめられている．

つぎに，最大回数反復法と残存要素検出法とを比較する．図4.14においては，データ生成の乱数上限値が64以下のばあいにはインデクス方式残存要素検出法のほうが高速になっている．すなわち，スカラ処理で検索表のごく一部しか検索されないばあい(同一のデータが多数登録されているばあいなど)は，ループ非交換版およびマスク演算方式最大回数反復法にくらべてインデクス方式残存要素検出法のほうが有利である．しかし，検索表のおおきさにくらべてデータの値域がせまいという条件はまれであり，通常使用される条件では最大回数反復法のほうが有利である．また図4.15でしめした $N=1024$, $M=61$ のばあいは比較的インデクス方式残存要素検出法に有利なばあいだが，それでもそれが最大回数反復法にくらべて有利になるのはマスク成立比率が6%以下のばあいにかぎられている．図4.14～4.15をとおしていえることは，通常の条件のもとではインデクス方式残存要素検出法より最大回数反復法のほうが高速だということである．なお，インデクス方式残存要素検出法とマスク演算方式残存要素検出法との比較においては，いずれがよいかは一概にいえない．

4.5.5 結果のまとめ

以上の結果をまとめるとつぎのようになる．なお，ここで M は原内くりかえし回数，N は原外くりかえし回数をあらわす．

(1) ループ交換にもとづくベクトル化の実用性

(1a) 一般的に

M がちいさいばあいはもちろん，M がおおきいばあいでもループ交換をおこなわないベクトル化にくらべて有利なばあいがある．

例：配列線形検索においては，$N=128, 1024$ のとき，つねにマスク演算方式最大回数反復法のプログラムのほうがループ非交換のプログラムより高速である．

(1b) ベクトル線形検索は

線形検索が実用的である M がちいさいばあいにおいては，ループ交換にもとづくベクトル線形検索は加速率が3～6倍程度であるから実用性はあるが，けたちがいの性能向上はえられない．

例：加速率は，$M=3$ かつ $N=128$ のとき2.6，$M=3$ かつ $N=1024$ のとき6.0である．

(2) 原外くりかえし回数と有利な方式との関係

予想されるとおり，原外くりかえし回数が増加するにつれて，ループ交換したプログラムのほうが高速になる．

例：配列線形検索においては，$N=16$ のときはスカラ処理あるいはループ非交換のプログラムがループ交換したプログラムより高速であるが，$N=1024$ のときはルー

プ交換したプログラムのほうがつねに高速である．

(3) 最大回数反復法と残存要素検出法との比較

スカラ処理で非常に少数のくりかえし回数で原内ループから脱出するばあいは残存要素検出法がよいが，たいていの条件のもとで最大回数反復法のほうが高速である．

例：配列線形検索のばあい，インデクス方式残存要素検出法がマスク演算方式最大回数反復法より高速になるのは，$N = 1024$, $M = 61$ のばあいでループの平均長と最大長との比 (マスク成立比率) が 0.06 以下のばあいにかぎられる．

4.6 関連研究

ループ実行前にはくりかえし回数がわからない **while** 文で記述されたループの並列化 (ベクトル化) をはかった研究として Wu ら [Wu 90] がある．しかし，この研究は 1 重の **while** ループにおいてそのくりかえし回数を決定する部分以外の部分をベクトル化するための技法に関するものであり，くりかえし回数が可変の多重のくりかえし構造に関するくりかえし構造の交換にもとづくベクトル化に関する研究はこれまでのところみられない．Wu らの技法は，ベクトル化すべき **while** ループが交換可能なループ内にないとき，外側ループのベクトル長がみじかいときなどに有効だが，当該ループのデータフローにループ依存性があってベクトル化できないときや，ループ内の演算のほとんどがくりかえし回数決定に関与するばあいには適用することができない．このようなばあいにはこの研究におけるくりかえし構造交換法が有効である．