第 5 章
共有部分がある複数データのベクトル処理法

要旨

従来のベクトル処理法は基本的にはまったく独立な複数のデータの処理にだけ適用される．これに対してこの章では，同一データのかきかえをふくむ処理をベクトル計算機や SIMD 型並列計算機で並列処理する上書きラベル・フィルタ法についてのべる．この方法により，共有部分がある複数の動的データ構造や単一データ構造の同一要素やへの複数の並列かきこみをふくむベクトル処理が可能になった．たとえば複数データのハッシングやある種のソート，記号処理 (非数値処理) でよく使用される共有要素をもつ線形リスト，グラフ，木などのデータ構造が処理可能になった．この方法を数個のアルゴリズムに適用し，ベクトル計算機 S-810 において高加速率がえられることを確認した．

5.1 はじめに

内蔵データベース処理機構「HITAC M-680H IDP」(Integrated Database Processor) [Kojima 87, Kojima 90] はデータベース処理のために設計され，いくつかの記号処理に応用されている [Torii 88a, Torii 88b, Mishina 89]．しかし，Cray X-MP，HITAC S-820 など，他のほとんどのベクトル計算機はほとんど数値計算だけにつかわれている．応用範囲が数値計算から記号処理に拡大されるのをさまたげていたひとつの理由は，リスト，木，グラフなどのポインタで結合された動的データ構造をベクトル処理するための広範囲に適用可能な方法がいまだ確立されていないことにあるとかんがえられる．

第 3 章では，複数の動的データ構造をあつかうプログラムをプログラム変換 (ベクトル化) にもとづいてベクトル処理することを可能にした．この方法においては，処理対象データはインデクス・ベクトルすなわち複数のデータへのポインタやインデクスから構成されるベクトルによってアクセスされる．この方法をこの論文では SIVP (simple index-vector-based vector-processing method) とよぶことにする．SIVP においては，ベクトル計算機のリスト・ベクトル処理機能やほかの条件制御機能が使用される．

しかし，この方法は，基本的にはまったく独立な複数データの処理をベクトル処理するばあいにだけ適用される．すなわち，後述するように，共有部分がある複数の動的データ構造へのかきこみをふくむグラフのかきかえのような処理，あるいは同一データ要素への複数のかきこみをふくむハッシュ表への複数データの登録のような処理をベクトル処理することはできない．この問題を解決するのが，この章でしめす上書きラベル・フィルタ法 (Filtering-Overwritten-Label Method, FOL) である．

5.2 節では上記の問題点をさらにくわしく説明する．5.3 節では上書きラベル・フィルタ法の原理とアルゴリズムとをしめす．5.4 〜 5.6 節では上書きラベル・フィルタ法の応用とその評価結果とをしめす．5.7 節では，上書きラベル・フィルタ法と従来の 2 つのアルゴリズムとの関係，SIMD 型並列計算機における上書きラベル・フィルタ法の応用の可能性についてのべる．5.8 節では関連研究についてのべる．

5.2 共有部分があるデータのベクトル処理の問題点

第 3 章のベクトル記号処理法すなわち SIVP においては，データはインデクス・ベクトルをつうじてよみこまれ，またかきこまれる．**図** 5.1 は 2 種類のインデクス・ベクトル，すなわちデータの添字または変位からなるベクトルとデータへのポインタからなるベクトルとを例示している．インデクス・ベクトルを使用することにより，いわゆるリスト・ベクトル命令または間接ベクトル・ロード / ストア命令によって記号データの一部がベクトル・レジスタに収集され，また主記憶に拡散される．

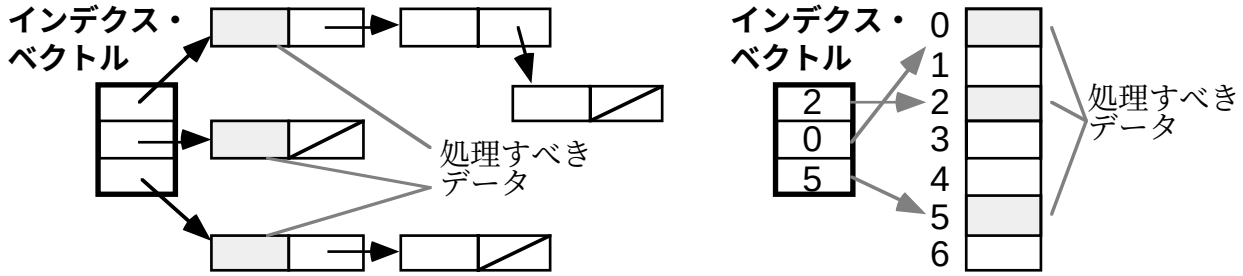
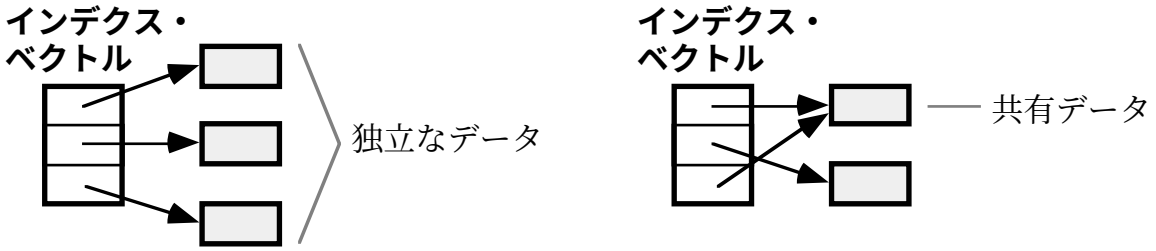


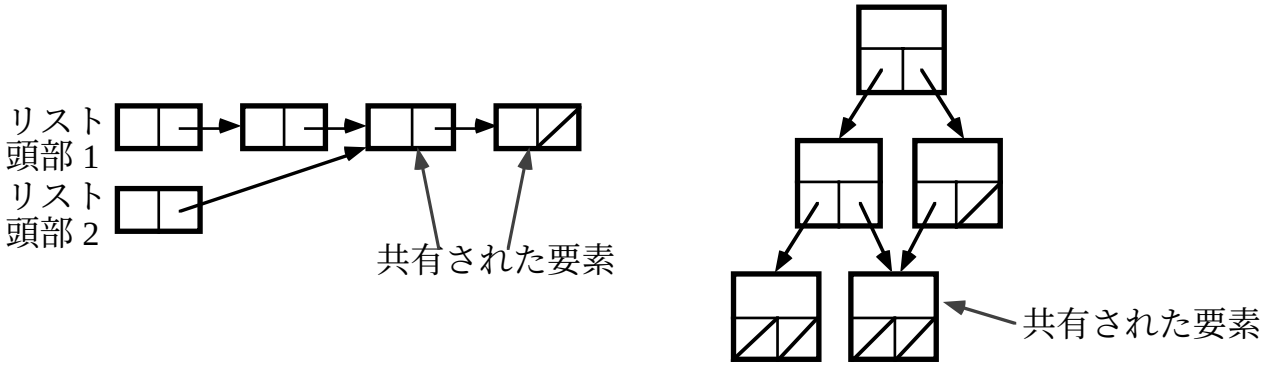
図 5.1　2 **種類のインデクス・ベクトル**

つぎに，SIVP が適用できる処理と適用できない処理を，**図** 5.2 を使用して説明する．SIVP は，基本的にはまったく独立な複数データの処理をベクトル処理するばあいに適用される．いいかえれば，処理対象のデータをさすインデクス・ベクトルの要素のなかに同一データへのポインタまたはインデクスがあらわれないことが保証されているような複数データの処理に適用される (図 5.2 a)．ただし，この方法は同一データよみだしをふくむがかきこみはふくまないベクトル処理にはそのまま適用することができる．すなわち，並列処理の対象となるインデクス・ベクトルの複数の要素が同一のデータをさすばあい (図 5.2 b) でも，そのデータをかきかえないばあいにはただしくベクトル処理をおこなうことができる．しかし，この方法を図 5.2 b のようなインデクス・ベクトルの複数の要素がさす同一のデータをかきかえる処理に適用すると，本来は逐次実行しなければならない処理を並列に実行することになって結果が不正になるので，このようなベクトル処理はできない．したがって，SIVP は**図** 5.3 にしめすような部分的に共有されたデータ構造のかきかえ処理には適用することができない．



(a) **無条件にベクトル処理できるデータ**　　(b) **かきこみがなければベクトル処理できるデータ**

図 5.2　SIVP **によりベクトル処理できるデータ**



(a) **共有のあるリスト**　　(b) **共有のある** 2 **分木**

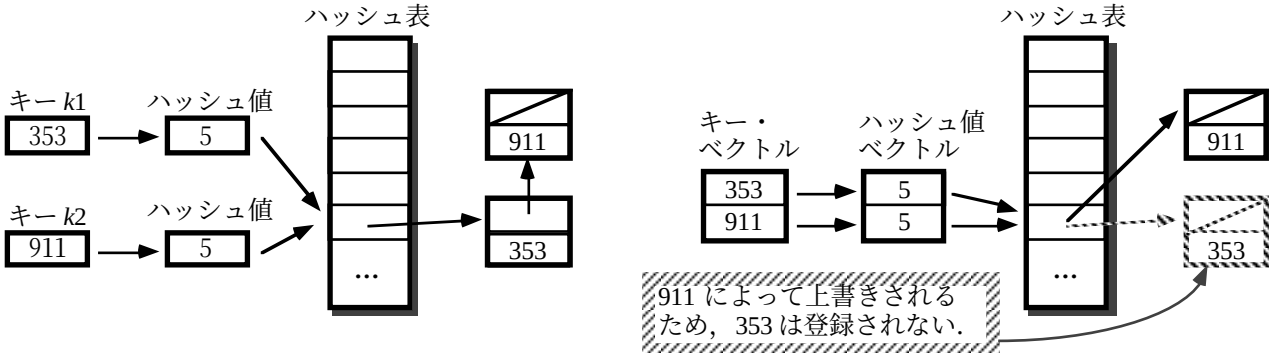
図 5.3　**部分的に共有されたデータ構造**

いいかえれば，上記のようなベクトル処理においてかきかえるデータに共有部分があると，ベクトルの各要素の処理のあいだにベクトル化に適さないデータ依存が生じる．このような共有がまれにしか存在しなければこのデータ依存は実際にはまれにしか生じないが，生じる可能性があるかぎりは従来の方法によりベクトル処理することができない．このように各要素の処理のあいだにデータ依存があっても，その依存が規則的であり，かつベクトル計算機にあらかじめ用意された命令で処理できるばあいにはベクトル化することができる．すなわち，累和 (Vector Element Sum)，内積，等差数列生成，一次巡回演算などの演算はベクトル計算機にあらかじめ用意されている命令で実行することができる．しかし，上記のような部分的に共有されたデータの処理のための命令はベクトル計算機に用意されておらず，またそこにおけるデータ依存は不規則であるから，それに対する命令をベクトル計算機に用意することも困難だとかんがえられる．

このような処理の例を 2 つあげる．第 1 の例として，**図** 5.4 にはハッシュ表への複数データの登録処理をしめす．このような処理を，この論文では**複ハッシング**とよぶ．図 5.4 ではハッシングの方法としてはチェイン・ハッシング (5.4 節参照) をつかっているため，登録データはハッシュ表からさされるリストに格納されている．図 5.4 a は逐次処理

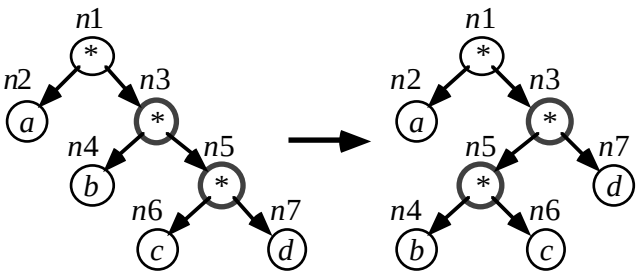
のばあいであり，キー 353 および 911 がこの順で処理される．これらのキーのハッシュ値はともに 5 となるが，これらは逐次に処理されるため，2 つのキーはともにただしく登録される．図 5.4 b は逐次処理のプログラムを強制的にベクトル化したばあいの処理をあらわしている．このばあい，2 個のキーをひとつのベクトルの要素とすることによってベクトル処理可能にしている．もしすべてのハッシュ値がことなるばあいには，このようなベクトル処理方法でもただしく処理することができる．図 5.4 のばあいでもハッシュ値ベクトルをもとめるところまではただしく処理されるが，ハッシュ表の同一エントリへのかきこみが並列におこなわれるため，一方のかきこみが他方を無効にしてしまう．図 5.4 b においては，ハッシュ表にはまず 353 をふくむリスト要素へのポインタが格納されるが，その直後に 911 をふくむリスト要素へのポインタが格納されるために，353 は未登録状態にもどってしまう．

第 2 の例として，**図 5.5** には入力された木にかきかえ規則を適用することによって等価な形式に変換する「木のかきかえ」をしめす．図 5.5 は，演算木に結合法則を適用してかきかえるプログラムの動作をしめしている．ここで結合法則は $X * (Y * Z) \rightarrow (X * Y) * Z$ とあらわされる．ここで，矢印がかきかえの向きをあらわす．図 5.5 におけるかきかえ前の演算木は $a * (b * (c * d))$ であり，これに対する結合法則の適用のしかたには 2 とおりある．これらを図 5.5 a および 5.5 b にしめしている．図 5.5 a におけるかきかえは演算ノード $n3$ および $n5$ に対しておこなわれ，図 5.5 b におけるかきかえは演算ノード $n1$ および $n3$ に対しておこなわれている．すなわち，これらのかきかえ操作はノード $n3$ をともにかきかえる (共用している)．したがって，これらのかきかえをベクトル処理によって並列に実行すると，もとの演算木とは等価でないあやまった木を生成したり，存在しないノードをアクセスしようとして異常終了する可能性がある．なお，この例では，単位処理すなわち 1 回のかきかえで木の複数のノードがかきかえられていることに注意を喚起しておく．

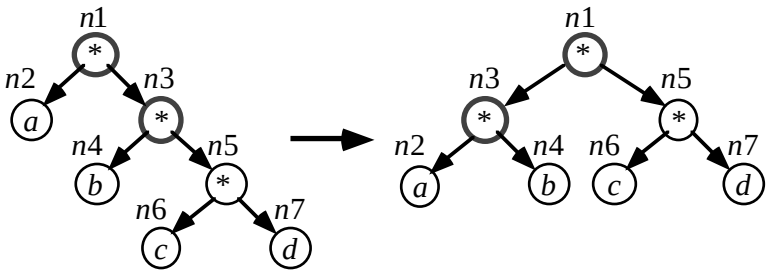


(a) **逐次処理による登録**　　(b) **強制ベクトル処理による登録と問題点**

図 5.4　ハッシュ表登録のベクトル処理の問題点



(a) **候補** 1



(b) **候補** 2

図 5.5　**演算木の** 2 **とおりのかきかえかた**

5.3 解決策としての上書きラベル・フィルタ法

5.2 節でしめしたような共有部分がある複数のデータのかきかえ処理に適用することができるベクトル処理法である**上書きラベル・フィルタ法** (Filtering-Overwritten-Label Method, FOL) を開発した．この節では，上書きラベル・フィルタ法の原理とアルゴリズムとを説明する．

5.3.1 上書きラベル・フィルタ法の原理

5.2 節で説明した問題点は，**図** 5.6 に例示するようなデータの並列処理可能な集合への分割によって解決することができる．すなわち，インデクス・ベクトルすなわちベクトル処理すべきデータをさすポインタまたはインデクスの集合 (*S*) を並列処理可能な集合 (*S*1, *S*2, *S*3) に分割してそれぞれを独立のベクトルとし (*V*1, *V*2, *V*3)，これらのベクトルを順次ベクトル処理する．いいかえれば，複数のデータのなかから並列処理できないものを検出して削除したうえで処理をおこない，削除されたデータに関してはあとであらためて処理する．あるいはこの方法は，共有部分がある複数のデータの処理において，共有のない部分は並列に処理し共有部分は逐次に処理するための方法だということもできる．

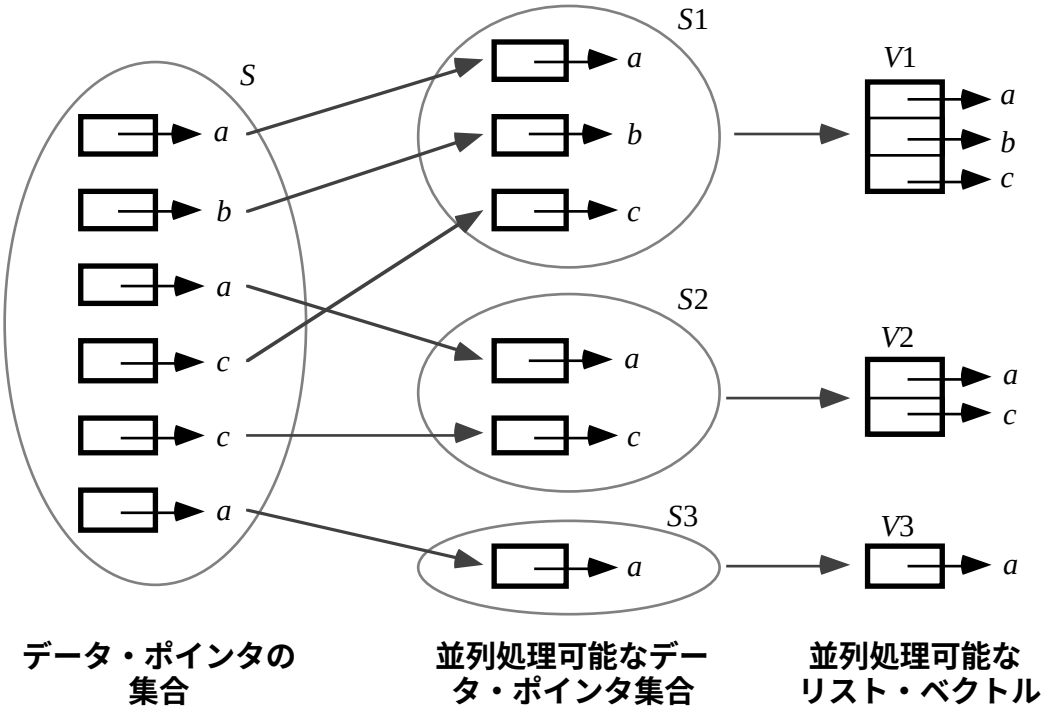
上記の方法を実現するためには，データを並列処理可能な集合に効率よく分割する方法をみいだす必要がある．並列処理できないのは同一データをさすポインタであるから，任意の 2 個のポインタのくみあわせに関してその値を比較すれば，並列処理できないデータを分離することができる．しかしそのためには，データの個数を N とすると $O(N^2)$ の比較回数が必要であり，実用的でない．並列処理できないデータの分離を $O(N)$ でおこなうことを可能にするのが上書きラベル・フィルタ法である[注1]．

FOL のアルゴリズムは次節以降でしめすが，この節では複ハッシングを例として，FOL による共有部分があるデータ処理の実現法をかんたんに説明する (**図** 5.7 参照)．複ハッシングにおいては，キーに衝突がないかぎりは各データの登録処理が独立におこなわれる．しかし，キーが衝突するデータの処理はハッシュ表の同一エントリへのかきこみをおこなうため，データ依存が生じる．このようなばあいには，衝突するキーを検出して依存がないキーだけをとりだせば，ベクトル処理することができる．ベクトル処理のためにキーはあらかじめベクトル (キー・ベクトル) に格納しておく．また，ハッシュ表には本来の登録領域すなわちエントリのほかに，各エントリに対して，「ラベル」を格納するための 1 個の作業領域をもうける．

衝突の検出 (上書きラベル・フィルタ処理) はつぎのようにしてすべてベクトル処理で

[注1] 複ハッシングのために特化された FOL を金田 [Kanada 90a] は Overwrite-and-Check Method とよんでいる．

おこなう．**ステップ** 1. でハッシュ値を添字としてキー・ベクトルのインデクス (配列添字) を「ラベル」としてハッシュ表の作業領域にかきこむ．そして**ステップ** 2. でハッシュ値を添字として作業領域からラベルをよみだす．ハッシュ値のかきこみとよみだしはともにベクトル計算機のリスト・ベクトル命令によって実行する．つぎに，よみだしたラベルをもとのラベルと比較する．衝突がなければ両者は一致するが，衝突したばあいはハッシュ表上でラベルが上書きされているために両者は一致しない．比較結果をマスク・ベクトルすなわち論理型のベクトルに格納する．対応するマスク・ベクトルの要素が *true* であるようなキー・ベクトルの要素が並列処理可能なデータである．図 5.7 の例においてはマスク・ベクトルの 2 ～ 4 番目の要素が *true* なので，キー・ベクトルの 2 ～ 4 番目の要素が最初の並列処理可能なデータ集合を構成する．これらのキーをマスクつきベクトル命令によってハッシュ表に登録する (**ステップ** 3.)．**ステップ** 4. では，すべてのデータが並列処理可能な集合に分割され，すべてのデータがハッシュ表に登録されるまで，**ステップ** 1. ～ 3. をくりかえす．なお，図 5.7 ではキーだけをハッシュ表に登録し，通常はキーとともに表に登録されるデータの登録は省略している．また，ベクトル処理によるハッシングのくわしい説明は 5.4 節にしるす．



a, b, c が処理すべきデータをあらわす。

図 5.6　共有部分があるデータのベクトル処理の基本原理 : 並列処理可能なベクトルへの分割

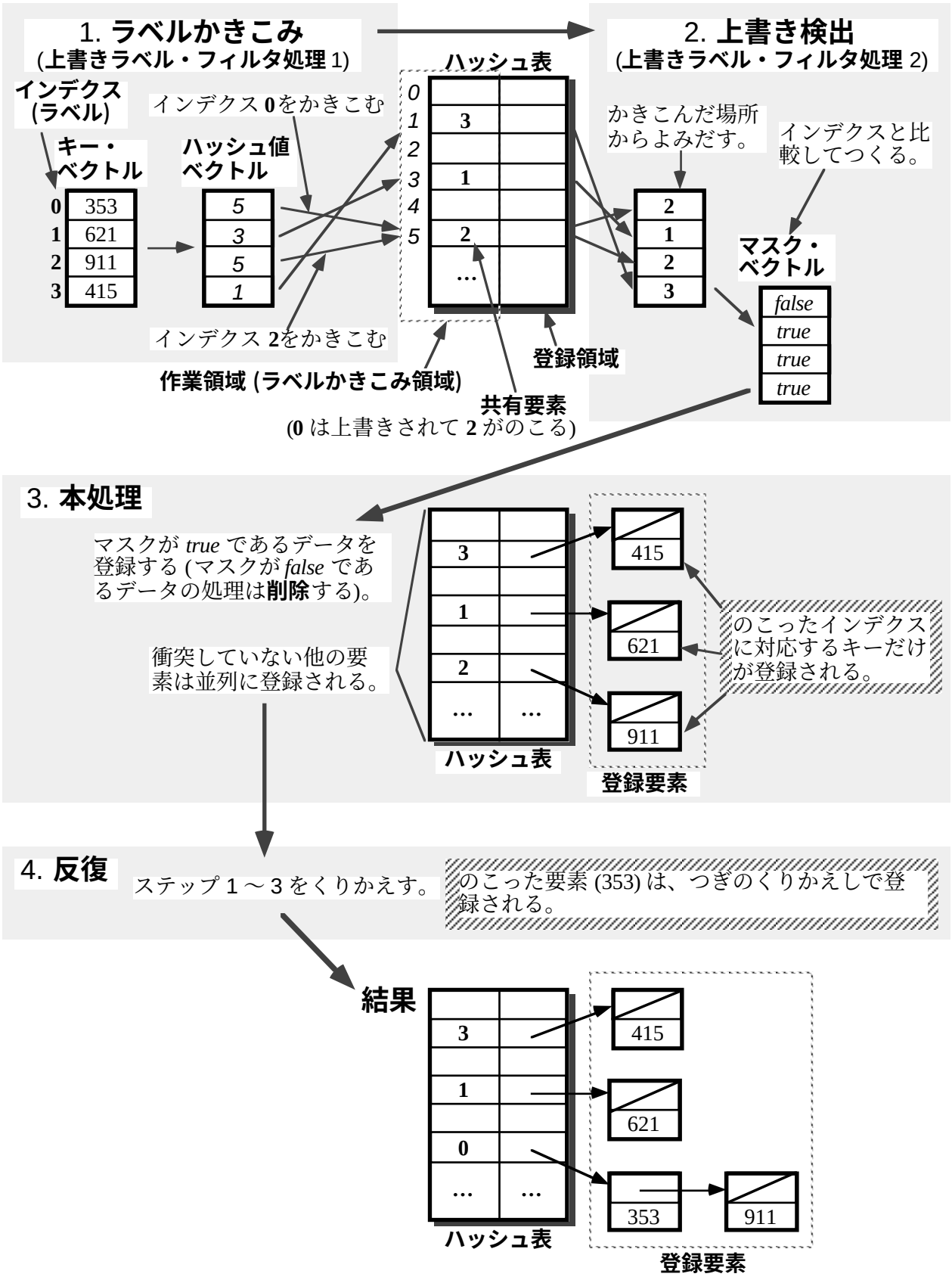


図 5.7　FOL にもとづくハッシングのベクトル処理法

5.3.2 単一データかきかえの上書きラベル・フィルタ法

前節の例における手順を抽象したベクトル処理方法が上書きラベル・フィルタ法 (FOL) である．FOL は，共有部分をふくむ可能性がある複数のデータをかきかえる広範な処理に適用することができる．この節では，ハッシングのように，ベクトル化すべき単位処理でそれぞれ 1 個だけのデータをかきかえるばあいの FOL のアルゴリズムをしめす．5.2 節で例示した演算木のかきかえのように単位処理で複数のデータをかきかえるばあいへの拡張は次節でしめす．

FOL による共有部分をふくむ可能性があるデータの並列処理可能な集合への分割のアルゴリズムをしめす．なお，S-820 のようなベクトル計算機においては，下記の過程はすべてベクトル処理によって実行することができる．

■ アルゴリズム : 上書きラベル・フィルタ法 1 (FOL1)

❑ 入力

インデクス・ベクトル V を入力する．V の各要素はデータ $d_1, d_2, \ldots, d_N$ のそれぞれをふくむ記憶領域へのポインタまたはインデクスである． このデータには重複があってもよい (すなわち，ひとつのデータが列 $d_1, d_2, \ldots, d_N$ のなかに複数回あらわれてもよい)．V の要素 v がさす記憶領域を $v\rightarrow$ とあらわし，そこに格納されたデータの値は $v\rightarrow d$ とあらわす．

❑ 出力

並列処理可能なデータからなる集合 $S_1, S_2, \ldots, S_M$ を出力する．ただし，$\bigcup_{i=1}^{M} S_i = \{d_1, d_2, \ldots, d_N\}$ かつ $S_1, S_2, \ldots, S_M$ はかさなりのない集合すなわち任意の i, j について $S_i \cap S_j = \varnothing$ がなりたつ (**独立分解条件**)[注2]．

❑ 処理条件

- データ d_i ($i = 1, 2, \ldots, N$) に対して，このアルゴリズムの適用後，ベクトル処理によって P_i (本処理) がそれぞれ実行される．ひとつの出力集合 S_j ($j = 1, 2, \ldots, M$) の要素であるすべてのデータに対する処理は並列に実行することができるし，また任意の順序で実行することができる．各出力集合の処理順序は任意でかまわないが，ことなる出力集合に属するデータの処理は並列に実行してはならない．すなわち，各出力集合に対する処理は逐次に実行しなければならない．なお，任意のデータどうしの実行順序は結果のただしさに影響してはならない．

[注2] すなわち，$S_1, S_2, \ldots, S_M$ は共通要素をもたない集合である．M の値はこのアルゴリズムの実行結果としてえられる．

- このアルゴリズムの適用前に V の任意の 2 つの要素 va, vb に関して $va\rightarrow$ と $vb\rightarrow$ とが同一の記憶領域であってもよい．すなわち，V の要素のなかには同一のポインタがあってもよい．ただし，このばあい va によってさされるデータ da と vb によってさされるデータ db とはことなるデータとみなす．
- このアルゴリズムで使用するために，V の各要素 v がさす記憶領域 $v\rightarrow$ 内にあらかじめ作業領域をわりあてておく．この作業領域を $v\rightarrow w$ とあらわす．記憶領域内にとるということは，$v1\rightarrow$ と $v2\rightarrow$ とが同一の記憶領域であれば $v1\rightarrow w$ と $v2\rightarrow w$ とも同一であることを意味する．

❑ 処理手順

(0) 前処理

$j = 1$ とする (j は変数)．インデクス・ベクトル V の各要素に重複のないようにつけることができる整数などのラベルをさだめておく．ラベルは実行時よりまえにきめておくことも可能である． V の第 i 要素 v_i のラベルとしてもっともかんたんに計算できるものは, V のなかでの v_i のインデクスすなわち要素番号 i または先頭からの変位 (バイト数など) である．

(1) ラベルかきこみ

すべての作業領域 $v_1\rightarrow w$, $v_2\rightarrow w$, … , $v_n\rightarrow w$ に V の要素 v_1, v_2, … , v_n のラベルをそれぞれかきこむ．ここで n は V の要素数をあらわす．かきこみ順序は任意であり，並列にかきこんでもかまわない．

(2) 上書き検出

すべての作業領域 $v_1\rightarrow w$, $v_2\rightarrow w$, … , $v_n\rightarrow w$ からそれぞれラベルをよみだし，v_1, v_2, … , v_n のラベルとの一致をとる．ただし，ラベルのよみだしのまえにステップ (1) が完了していなければならない．$v_i\rightarrow w$ ($i = 1, 2, \ldots, N$) からよみだされた値が v_i のラベルと一致しないときは v_i が上書きされたことを意味する．したがって，V の要素からさされるデータのうち不一致が生じたデータをのぞいたものからなる集合をつくれば, 並列処理可能なデータの集合 S_j がえられる．すなわち，V の要素 u_k ($k = 1, 2, \ldots, m$) のラベルを l_k とするとき，$u_k\rightarrow w = l_k$ という関係がなりたつような V のすべての要素 u_1, u_2, … , u_m をもとめ，$S_j = \{u_1\rightarrow d,\ u_2\rightarrow d,\ \ldots,\ u_m\rightarrow d\}$ とする．

(3) 変数値更新

j に 1 をくわえる．S_j にふくまれるデータへのポインタを V から削除してつめあわせることによってえられたベクトルをあらたな V とする．V の要素数は減少する．

(4) **反復**

Vが空になるまで上記の (1) ～ (3) のステップをくりかえす．停止時に変数 M に $j-1$ を代入する．■

複ハッシングの例では，効率をあげるために FOL のステップのなかに本処理 (登録) をまぜていたが，FOL1 (上記のアルゴリズム) においては，汎用性をもたせるために，それをのぞいてある．

上記のアルゴリズムがただしいためには，つぎの条件がなりたつ必要がある．

■ **ラベル排他格納条件** (The exclusive label storing condition, ELS **条件**)

ある 1 つの作業領域に複数のラベルがかきこまれたばあいには，そこにそれらのうちのいずれかのラベルがただしく格納される必要がある．すなわち，ラベル *la* と *lb* とが同一領域にかきこまれるばあい，格納された値の一部がラベル *la* とラベル *lb* が混合されたものであってはならない．かきこまれた複数のラベルのうちのいずれが格納されるかは任意でよい．■

パイプライン型ベクトル計算機に関しては，通常，ラベルが 1 語長以下のばあいにはラベル排他格納条件がなりたつ．以後この条件がなりたつことを仮定する．

上記のアルゴリズム FOL1 に関する 2 つの補題と 3 つの定理をしめす．

■ **定理** 1：**停止性**

FOL1 はかならず停止する．

証明

ラベル排他格納条件により，ある作業領域から FOL1 の過程でよみだされるラベルは，そこにかきこまれたラベルのなかの 1 つとかならずひとしい. したがって, ステップ (2) においてラベルの一致をとるときに一致するものがかならず存在する．ゆえに，任意の j について S_j は空ではない．これは，ステップ (3) においてインデクス・ベクトル V の要素数が毎回かならず減少することを意味するので，有限回のくりかえしで V の要素は空になり，FOL1 の停止条件がみたされる．したがって，FOL1 はかならず有限回のくりかえしで停止する．■

■ **補題** 1：**独立分解**

FOL1 の出力条件としてしめした独立分解条件がなりたつ．すなわち，FOL1 の停止時に $\bigcup_{j=1}^{M} S_j = \{d_1, d_2, \ldots, d_N\}$ かつ任意の i, j について $S_i \cap S_j = \varnothing$ がなりたつ．すなわち，出力される集合の要素の和は入力されたデータの集合にちょうどひとしく，

出力集合の要素にかさなりはない.

証明

つぎの条件がなりたてば独立分解条件がなりたつので，これらがなりたつことをしめせばよい.

(a) 出力集合 S_j ($j = 1, 2, \ldots, M$) のすべての要素が入力データのうちのひとつにひとしい.

(b) 入力集合のすべての要素が出力集合のいずれかにふくまれる.

(c) 出力集合に要素の重複がない.

まず条件 (a) をしめす．FOL1 の実行開始時には V はちょうど入力集合のすべての要素をさすポインタからなりたっている．任意の j について出力集合 S_j の要素 e はステップ (2) において V がさすデータのなかからえらばれたものであるから，e は入力データのひとつにひとしい.

つぎに，条件 (b) および (c) をしめす．実行開始時には V はちょうど入力集合のすべての要素をさすポインタまたはインデクスからなりたっている．そして，停止時には V は空ベクトルとなる．V から削除された要素は，それがさすデータがステップ (2) において S_j ($j = 1, 2, \ldots, M$) のいずれかにふくめられた直後に実行されたステップ (3) において削除された．そして，ステップ (3) がくりかえし実行されることによって V からすべての要素が削除された．これは，V にふくまれていたすべての要素が S_j のいずれかにふくめられたことを意味する．したがって，入力集合のすべての要素が出力集合 S_j のいずれかにふくまれることになる．しかも，S_j がふくむ要素をさすポインタまたはインデクスは S_j がもとめられた直後に V から削除されるから，S_m ($m > j$) にふくまれることはない．したがって出力集合に重複はない．■

集合 $S_1, S_2, \ldots, S_M$ の濃度 (要素数) をそれぞれ $|S_1|, |S_2|, \ldots, |S_M|$ とあらわす．すると，つぎの補題および定理がなりたつ.

■ 補題 2

d_k と d_l ($k \neq l$) を出力集合 S_j ($j = 1, 2, \ldots, M$) の任意の要素とするとき，d_k と d_l はことなる領域に存在する (d_k および d_l はことなるデータである).

証明

背理法で証明する．もし d_k と d_l が同一領域に存在すれば，V の要素である d_k, d_l へのポインタまたはインデクスは同一の値である．これらの要素にわりあてられたラベルは同一の作業領域に格納される．これらのラベルのうちの一方は他方によって上書

きされるから，d_k と d_l はステップ (2) においてことなる出力集合にふくめられる．これは仮定に矛盾している．したがって，補題は証明された．■

■ 定理 2：正当性

FOL1 の停止時に出力条件がなりたつ．

証明

この定理は補題 1 および補題 2 から証明される (補題 2 が出力集合が並列処理可能であることを保証している)．■

さらにもうひとつの定理をあたえる．

■ 定理 3

関係 $|S_1| \geq |S_2| \geq \ldots \geq |S_M|$ がなりたつ．とくに，入力データに重複がないときは $|S_2| = \ldots = |S_M| = 0$ がなりたつ．

証明は省略する．■

上記のアルゴリズムにおいて使用している作業領域のわりあてについてのべる．通常，作業領域としては本処理においてかきこみをおこなう領域を共用することができる．なぜなら，その領域に格納されていた値をラベルのかきこみによって破壊してもかまわないし，この共用によって上書き検出時にラベルではない不正な値をよみだすことはないからである．その領域に格納されていた値を破壊してもかまわない理由は，上記のアルゴリズムにおいてラベルをかきこんだ場所には，本処理においてかならずかきこみをおこなうはずだからである．逆にいえば，この条件をみたすために，本処理においてかならずかきこみをおこなうという十分条件をみたすか，またはよりよわい条件がみたされていなければならない．不正な値をよみだすことがない理由は，ステップ (2) におけるラベルのよみだしにおいては，かきこみに使用したポインタまたはインデクスをつかってよみだしをおこなっているため，ラベルをかきこんでいない場所からよみだすことはなく，かつラベル排他格納条件がなりたっているからである．

作業領域としてはすくなくとも $\log_2 N$ bit の領域が必要であり，本処理で必要とする領域がこれよりちいさいばあいにはそれをひろげる必要がある．すくなくとも $\log_2 N$ bit の領域が必要なのは，N 個のことなるラベルが格納できなければならないからである．

つぎに，FOL1 の性能について考察する．FOL においては，逐次処理される部分はスカラ処理にくらべて加速されない．それどころか，並列処理できないデータの検出オーバヘッドのためにその部分はかえって実行がおそくなってしまう．したがって，並列処理できない部分が大半であるような複数データ処理においては，FOL1 を使用するとか

えって実行がおそくなってしまう．しかし，共有部分がまれにしか存在せず，したがってほとんどの部分が並列処理できるばあいには，FOL1 によって実行を加速できる可能性がたかい．

つぎの定理は，共有部分が十分すくないばあいに FOL1 の実行時間が $O(N)$ であることをしめしている．

■ 定理 4：最良実行時間

条件 $|S_1| \gg \sum_{i=2}^{M} |S_i|$ がなりたつなら，FOL1 の実行時間は $O(N)$ である．

証明

上記の条件がなりたつなら，$|S_i|$ ($i = 2, 3, \ldots, M$) に依存するステップ (4) の実行時間は，$|S_1|$ に依存するステップ (1), (2) の実行時間にくらべて無視することができる．ステップ (1), (2) の実行時間はいずれも $O(N)$ であるから，FOL1 の実行時間も $O(N)$ である．■

とくに，定理 3 および定理 4 により，入力データに重複がないばあいには FOL1 の実行時間はは $O(N)$ である．

補題と定理をひとつずつしめす．この定理は，条件 $|S_1| \gg \sum_{i=2}^{M} |S_i|$ がなりたたないばあいにも，FOL1 によってある意味で最高性能がえられることを保証している．

■ 補題 3

入力データに，オリジナル・データをふくめて M' 個の重複があり (すなわち M' 個の入力データが共有する記憶領域があるとし) M' 個をこえる重複はないとすると，すなわち重複数の最大値が M' であるとすると，出力集合の数 M は M' にひとしい．

証明

入力データに M' 個の重複があるとき，インデクス・ベクトル V は，FOL1 の実行開始時に重複したデータをふくむひとつの記憶領域をさす M' 個の要素をもつ．定理 1 の証明でのべたように，ステップ (2) においてラベルの一致をとるときに，一致するラベルがかならず存在する．しかも，各入力データに対してはことなるラベルがつけられていて，そのなかのただ 1 個が M' 回重複してよみだされるから，重複した M' 個の V の要素につけられたラベルのなかで一致するものはただひとつである．したがって，M' 個の要素のなかからちょうど 1 個が，ステップ (3) が実行されるごとに V から削除される．ところで，FOL1 のくりかえし回数は M にひとしい．したがって，M' は M にひとしい．■

■ 定理 5：最少分割

$T_1 \cup T_2 \cup \ldots \cup T_{M''}$ を入力データ $\bigcup_{i=1}^{M''} T_i = \{d_1, d_2, \ldots, d_N\}$ の任意の並列処理可能な分割とし，FOL1 の出力集合の数を M とするとき，分割数 M'' とのあいだに関係 $M'' \geq M$ がなりたつ．すなわち，FOL1 は最少個数に分割された集合を出力する．

証明

重複したデータは同一の出力集合には属さない．なぜなら，そうでなければ出力集合が並列処理できないからである．したがって，もし入力データに M' 個の重複があれば，任意の分割において，並列処理可能な集合の数は M' 個よりすくなくはない．FOL1 の出力集合の数は補題 3 によって M' 個であるから，FOL1 は最少個数に分割された集合を出力する．■

つぎに，最悪のばあいの実行時間について考察する．

■ 定理 6：最悪実行時間

条件 $|S_1| = |S_2| = \ldots = |S_M| = 1$, $M = N$ がなりたつなら (すなわちすべてのデータが重複データならば)，FOL1 の実行時間は $O(N^2)$ である．

証明

FOL1 における 1 回のくりかえしの実行時間はステップ (1), (2) の実行時間によってほぼきまり，ステップ (1), (2) の実行時間はほぼその時点での V の要素数に比例する．また，上記の条件のもとではくりかえし 1 回ごとにちょうど 1 個ずつ V の要素数が減少していく．したがって，1 回あたり 1 要素あたりのステップ (1), (2) の実行時間の合計を t とすれば j 回目のくりかえしの実行時間はほぼ $(N-j+1)t$ となるから，

実行時間の総計はほぼ $\sum_{j=1}^{N}(N-j+1)t$ であり，したがって $O(N^2)$ である．■

つぎに，FOL1 における「各出力集合の処理順序は任意でかまわない」という処理条件についてかんがえる．複ハッシングのばあいにはこの条件がなりたっている．なぜなら，このばあいは，まず衝突がおこらないばあいには処理順序はまったく問題にならないし，衝突がおこったばあいについても，衝突したデータのうちのいずれが格納されるかによって各登録データのハッシュ表中での格納位置はかわりうるが，それは問題にならないからである．

しかし，FOL を適用する対象のアルゴリズムによっては，同一領域にふくまれるデータに関する複数の処理に関して処理順序を保存しなければならないばあいがある．上記の処理条件をつけずにすむように FOL1 を修正することによって，このようなアルゴリ

ズムのベクトル化版を構成することは可能である．すなわち, 処理 P_i の対象データ d_i (= $v_i \rightarrow d$) と処理 P_j の対象データ d_j (= $v_j \rightarrow d$) とが同一記憶領域に存在し，逐次処理において処理 P_i が処理 P_j に先行して実行されるならば，ELS 条件をよりつよい条件でおきかえることによって，データ d_i, 属する出力集合 S_k と d_j が属する出力集合 S_l についてかならず $k > l$ がなりたつようにアルゴリズムを構成すればよい．

ベクトル計算機 S-820 においては，FOL1 を実行するには，ラベルのかきこみにおいて，ELS 条件をみたす高速なリスト・ベクトル格納命令 `VIST` (Vector Indirect STore) を使用することができる．一方，$k > l$ という条件をなりたたせて上記の処理条件をなくすためには，`VIST` 命令のかわりにやや低速だが要素の格納順序が保証された (ELS 条件よりつよい条件をみたす) `VSTX` (Vector STore indeXed) 命令を使用すればよい．

つぎに，上記のアルゴリズムの応用範囲について考察する．FOL1 はかなり一般性があるベクトル処理 (並列処理) 技法である．複ハッシングは共有部分がすくない複数データ処理として代表的だということができるが，リスト処理などにおいてもこの条件がなりたっているばあいがおおいとかんがえられる．したがって，FOL1 の応用範囲はハッシングにとどまらず，たとえば，番地計算ソート，リストや木の並列かきかえや，図 5.3 にしめしたような部分的に共有されたリストや木 (正確には DAG すなわち Directed Acyclic Graphs) のようなデータ構造のかきかえなどの処理にも適用することができるとかんがえられる．

最後に，FOL1 をより単純化することによってその実行速度を向上させるための方法をかんがえる．この方法は，インデクス・ベクトル V からさされたデータにかきこむべき値が一意であるすなわち重複がないばあいに適用される．すなわち，このようなばあいには，本来データにかきこむべき値をラベルとして使用することにより，並列処理可能な集合への分割と本来の処理とをかねておこなうことが可能である．複ハッシングの例では登録すべきキー値に重複がなければ，キーをラベルとして使用することでこの条件がみたされる．前期のアルゴリズムを単純化したものはしめさないが，この単純化された方法をハッシングに適用したばあいの登録過程を**図 5.8** にしめす[注3]．ただし，単純化されたアルゴリズムは V からさされたデータにかきこむべき値すなわちラベルとして使用する値が一意であるという条件がなりたたないときには適用することができない．

[注3] そのアルゴリズムは第 4 章でしめす．

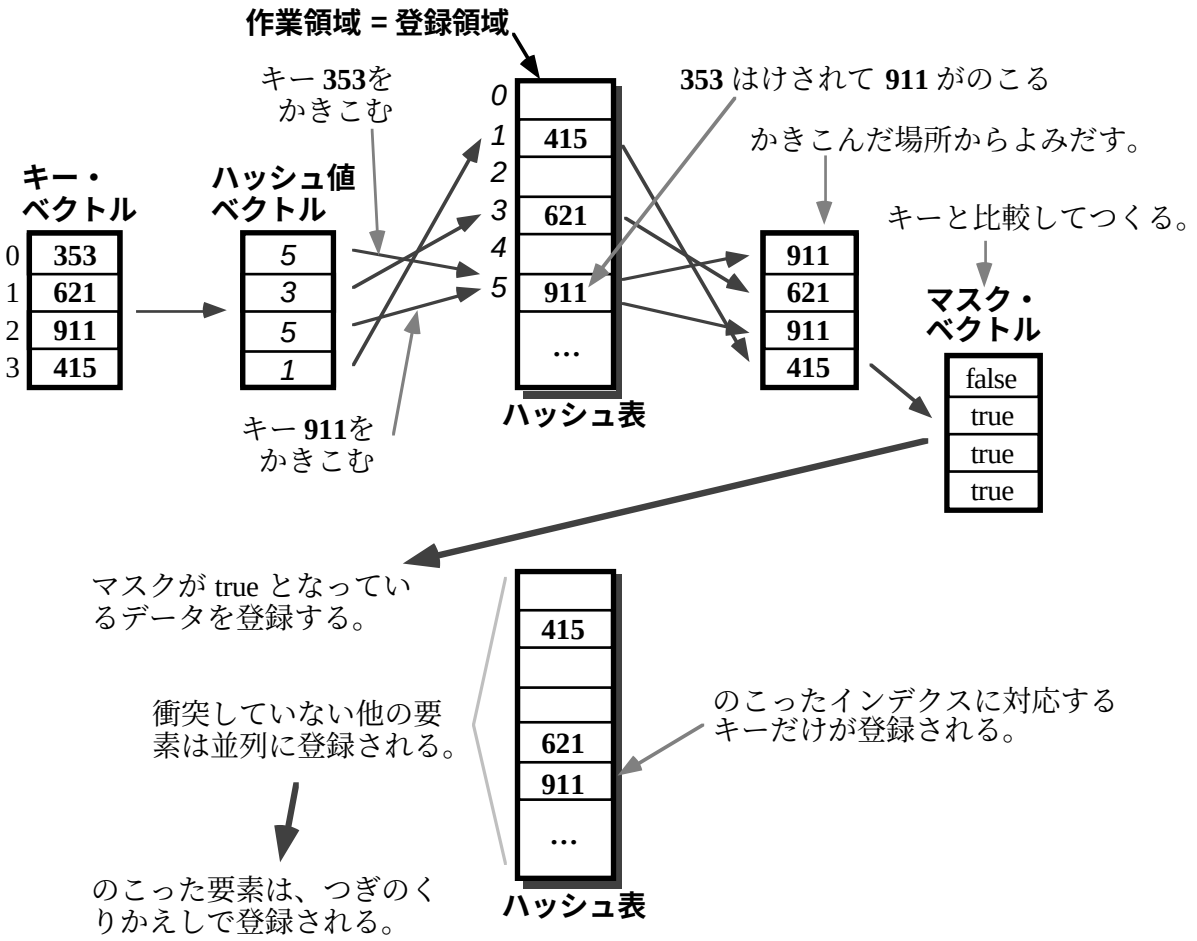


図 5.8　キーが一意なハッシングにおける上書きラベル・フィルタ法

5.3.3 複数データかきかえの上書きラベル・フィルタ法

前節でしめしたアルゴリズムは単位処理でかきかえるデータがただ 1 個のばあいだけに適用することができるが，この節では，5.2 節で例示した演算木のかきかえのように，複数のデータを単位処理でかきかえるばあいに適用できる上書きラベル・フィルタ法のアルゴリズムをしめす．

■ アルゴリズム : 上書きラベル・フィルタ法 2 (FOL*)

❑ 入力

インデクス・ベクトル $V_1, V_2, \ldots, V_L$ を入力する．これらのベクトルの要素は，すべてのデータ $d_{i1}, d_{i2}, \ldots, d_{iL}$ $(i = 1, 2, \ldots, N)$ のそれぞれをふくむ記憶領域へのポインタまたはインデクスである．V_k の要素 v がさす記憶領域を $v\rightarrow$ とあらわし，そこに格納されたデータの値は $v\rightarrow d$ とあらわす．

❑ 出力

データの組 $\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle$ $(i = 1, 2, \ldots, N)$ からなる並列処理可能な集合 $S_1, S_2, \ldots, S_M$ を出力する (M の値はこの処理の結果としてえられる)．ことなる組に属するデータは並列処理可能である．ただし，$\bigcup_{j=1}^{M} S_j = \{\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle \mid i = 1, 2, \ldots, N\}$ かつ $S_1, S_2, \ldots, S_M$ はかさなりのない集合とする．すなわち任意の i, j について $S_i \cap S_j = \varnothing$ がなりたつ (**独立分解条件**).

❑ 処理条件

- データの組 $\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle$ $(i = 1, 2, \ldots, N)$ に対して，このアルゴリズムの適用後，ベクトル処理によって処理 P_i $(i = 1, 2, \ldots, N)$ がそれぞれ実行される．ひとつの出力集合 S_j $(j = 1, 2, \ldots, M)$ にふくまれるすべてのデータは並列に，または任意の順序で処理することができるものとする．しかし，もし $k < l$ がなりたてば S_k にふくまれるすべてのデータは S_l にふくまれるどのデータよりもさきに処理される．上記の条件をみたすかぎり，処理の実行順序は結果のただしさに影響してはならない．

- このアルゴリズムの適用前に任意の $f, g \in \{1, 2, \ldots, L\}$ について V_f の任意の要素 va と V_g の任意の要素 vb に関して $va\rightarrow$ と $vb\rightarrow$ とが同一の記憶領域であってもよい．すなわち，$V_1, V_2, \ldots, V_L$ の要素のなかに同一のポインタがあってもよい．ただし，このばあい va によってさされるデータ da と vb によってさされるデータ db とはことなるデータとみなす．

- このアルゴリズムで使用するために，$V_1, V_2, \ldots, V_L$ の各要素 v がさす記憶領域内にあらかじめ作業領域をわりあてておく．この作業領域を $v\rightarrow w$ とあらわす．

❑ 処理手順

(0) 前処理

$j = 1$ とする (j は変数)．また，インデクス・ベクトル V_k $(k = 1, 2, \ldots, L)$ の各要素に重複のないようにつけることができる整数などのラベル (識別子) をさだめておく．すなわち，V_k の任意の要素 va, vb $(va \neq vb)$ のラベルをそれぞれ la, lb とするとき，$la \neq lb$ でなければならない．ただし，ことなる k に関しては同一のラベルを使用することができる．すなわち，$f \neq g$ がなりたつとき，V_f の任意の要素 va と V_g の任意の要素 vb には同一のラベルをつけてよい[注4]．V_k の第 i 要素 v_{ki} のラベルとしてもっともかんたんなものは，V_k のなかでの v_{ki} のインデクスすなわち要素番号 i または先頭からの変位 (バイト数など) である．

[注4] ラベルは実行時よりまえにきめておくことも可能である．

(1) **ラベルかきこみ**

v_{k1}, v_{k2}, … , v_{kn} をインデクス・ベクトル V_k の要素とする．ここで n は V_1, V_2, … , V_L の要素数である (これらのベクトルはすべて同数の要素をもつ)．$k = 1, 2, \ldots, L$ についてつぎの処理をおこなう．すべての作業領域 $v_{k1}\rightarrow w$, $v_{k2}\rightarrow w$, … , $v_{kn}\rightarrow w$ に V_k の要素 v_{k1}, v_{k2}, … , v_{kn} のラベルをそれぞれかきこむ．かきこみ順序は任意であり，並列にかきこんでもかまわない[注5]．

(2) **上書き検出**

$k = 1, 2, \ldots, L$ についてつぎの処理をおこなう．作業領域 $v_{k1}\rightarrow w$, $v_{k2}\rightarrow w$, … , $v_{kn}\rightarrow w$ からラベルをそれぞれよみだし，v_{k1}, v_{k2}, … , v_{kn} のラベルとの一致をとる．ステップ (1) はラベルのよみだしのまえに完了していなければならない．ラベルのかきこみはこのよみだしのまえに終了していなければならない．もし $v_i\rightarrow w$ が v_i のラベルと一致しないときは上書きがあったことを意味する．したがって，V_1, V_2, … , V_L からさされるデータの組 $\langle d_{i1}, d_{i2}, \ldots, d_{iL} \rangle$ のなかで，いずれかの k ($k = 1, 2, \ldots, L$) に関して V_k の第 i 要素からさされるデータのうち不一致が生じたデータをふくまない組からなる集合をつくれば，並列処理可能なデータの組からなる集合 S_j がえられる．すなわち，つぎのようにして集合 S_j をさだめる．V_1, V_2, … , V_L の第 i 要素をそれぞれ v_{1i}, v_{2i}, …, v_{Li} とし，これらの要素に対してさだめられたラベルをそれぞれ l_{1i}, l_{2i}, … , l_{Li} とするとき，$v_{1i}\rightarrow w = l_{1i} \wedge v_{2i}\rightarrow w = l_{2i} \wedge \ldots \wedge v_{Li}\rightarrow w = l_{Li}$ という関係がなりたつような要素 v_{1i}, v_{2i}, … , v_{Li} からさされるデータによって構成されるすべての組 $\langle v_{1i}\rightarrow d, v_{2i}\rightarrow d, \ldots, v_{Li}\rightarrow d \rangle$ ($i = i_1, i_2, \ldots, i_m$) を要素とする集合をつくって S_j とする．

(3) **変数値更新**

j に 1 をくわえる．$k = 1, 2, \ldots, L$ について S_j にふくまれるデータへのポインタを V_k から削除してつめあわせることによってえられたベクトルをあらたな V_k とする．

(4) **反復**

V_k が空になるまで上記の (1) ～ (3) のステップをくりかえす (V_k ($k = 1, 2, \ldots, L$) の要素数はすべてひとしいので，そのうちのいずれかひとつをテストすればよい)．停止時に変数 M に $j - 1$ を代入する．■

ステップ (1) においては，FOL1 のばあいと同様に ELS 条件がなりたたなければならない．さらに，ステップ (1) でのべたようにラベルのかきこみ順序などに関してさらに条件を追加しなければ，デッドロックが発生する可能性がある．すなわち，各ラベルのか

[注5] ただし，このままではデッドロックが発生する可能性があるため，その対策を要する．対策については後述する．

きこみに関して適当な順序性を保証することができないばあいには，各インデクス・ベクトルのどの要素に関してもステップ (2) における関係がなりたたず，S_j が空集合になることがありうる．そして，そのために FOL2 におけるくりかえしから永久に脱出できないという事態が生じうる．

このようなデッドロックをふせぐためには，たとえばつぎのようにすればよい．各インデクス・ベクトルの最後の要素以外はベクトル命令で並列にかきこむが，最後の要素だけはベクトル命令の実行後にスカラ命令で逐次にかきこむようにする．また，各インデクス・ベクトルの最後の要素がさすデータのあいだでの共有はない (同一領域には存在しない) とする．これにより，すくなくとも各インデクス・ベクトルの最後の要素に関してはステップ (2) における関係がなりたち，S_j が空集合になることはふせぐことができる．ただし，この方法ではデッドロックの発生はふせげるものの，並列度が極端に低下して加速率が 1 以下になることもありうるので，よりよい方法をくふうする必要があるとかんがえられる．

なお，FOL* の停止性と独立分解条件がなりたつことは，単一データかきかえのばあいと同様にして証明することができるので，その証明は省略する．

FOL* の性能について考察する．単位処理でかきかえるデータの数 L がおおきくなるとオーバヘッドがおおきくなり，スカラ処理と比較しての加速率が低下するとかんがえられる．したがって，実用的なのは L が数個程度のばあいであろう．5.2 節でしめした演算木のかきかえにおいては $L = 2$ なので，実用的な性能がえられるとかんがえられる．

FOL* は，演算木のかきかえ以外に，リストや木の複数の要素を同時にテストすることによってそれらのデータ構造のとなりあう要素を同時にかきかえるような処理，たとえば木の再バランス (rebalancing) などにも適用できるとかんがえられる．

5.4 複ハッシングへの応用

5.4.1 複ハッシングのアルゴリズム

ハッシングにはオープン・ハッシングとチェイン・ハッシングという 2 種類の方法がある [Knuth 73]．オープン・ハッシングとは，図 5.8 のようにハッシュ値がひとしい 2 個のデータをハッシュ表のことなるエントリに登録する方法である．また，チェイン・ハッシングとは，図 5.4, 5.7 のようにそのようなデータのうちの一方をハッシュ表とはことなる領域に登録する方法である．ここでは，固定長のテーブルをつかうオープン・ハッシングによるハッシュ表への登録のアルゴリズムを**図** 5.9 にしめし，それを Fortran でコーディングした例を**付録** 1 にしめす[注6]．チェイン・ハッシングではなくオープン・ハッシングをためしたのはおもにコーディングの容易さのためであるが，チェイン・ハッシングも同様にして実現可能である．

図 5.9 では，かんたんのため，キーと対にするべきデータの登録などははぶいてある．また，効率向上などのため，5.4.2 節で測定結果をしめす Fortran でコーディングしたプログラムにおける実行順序は図 5.9 のとおりではない部分がある．なお，ハッシュ表のエントリが使用されているかどうかをくべつするために，未使用のエントリにはあらかじめ未登録であることをあらわす特別の値 (キー値としては使用しない値) をかきこんでおく．登録が終了するたびに，再度登録する必要がある要素だけをあつめている．すなわち，圧縮方式で条件制御している．最初のくりかえしで登録に失敗したキーについては添字値をかえて再度登録をこころみるが，このときの添字の増分値をキーの値からきめている (その方法については図 5.9 参照)．これを一定値にせずキー・ベクトルの要素ごとにことなる値になるようにきめているのは，キー・ベクトルにふくまれるキーどうしが衝突するばあいの収束をよくする (再度の衝突をふせいで，くりかえし回数を減少させる) ためである．

以下，例題を使用して，ベクトル処理によるハッシング表への複数キーの登録を，従来の方法すなわちスカラ処理による登録と比較しながらしめす．**図** 5.10 にスカラ処理，**図** 5.11 にベクトル処理をしめす．いずれにおいても入力データおよび入力時のハッシュ表の状態はひとしい．ハッシュ表のサイズとしては，説明の都合により 6 というちいさな値をつかっているが，ベクトル処理による大幅な性能向上のためには数 10 以上のキーの並列登録が必要であり，したがってハッシュ表も数 10 以上である必要がある．ハッシュ関数 *hash*(*x*) としては x **mod** 6 を使用する．上書きラベル・フィルタ法においてはラベルとしてキーを使用し，ハッシュ表上に作業領域はとらない．登録するデータは 353, 621, 415, 911 という 4 個の整数値であり，これらをそのままキーとして使用する．入力

[注6] このアルゴリズムは [Kanada 90a] にしめしたアルゴリズムを一部修正したものである．

データはこれらのキー値からなる配列である．ハッシュ表にはあらかじめ 103 が登録されている．

まず，図 5.10 をつかって従来のスカラ処理による登録法について説明する．キーをふくむ配列の第 1 要素から順に処理していく．第 1 要素および第 2 要素は最初に計算したハッシュ値をそのままハッシュ表の添字として登録することができる．しかし，第 3 要素はハッシュ表にあらかじめ登録されていた 103 と衝突し，第 4 要素はこの処理のなかで登録した 353 と衝突する．したがって，それぞれ添字を計算しなおして登録する．

つぎに，図 5.11 をつかってベクトル処理による登録法について説明する．キーをふくむ配列の各要素のハッシュ値を計算する．スカラ処理のばあいとはちがって，各要素についての計算は並列に実行する．この点は，以下の計算においても同様である．この時点で各ハッシュ値をそのまま添字値として登録することができるかどうか，すなわちハッシュ表にすでに登録されているデータと衝突がおこらないかどうかを判定する．このような，ハッシュ表に登録されているデータとの衝突を**第 1 種衝突**とよぶことにする．第 1 種衝突の検出結果はマスク・ベクトル (論理型の配列) のかたちでえられる．キーをふくむ配列の第 *i* 要素をつかってえられる検出結果がマスク・ベクトルの第 *i* 要素の値となる．したがって，マスク・ベクトルの値は，衝突がおこる第 3 要素だけが *false* になり，ほかの要素は *true* になる．

ラベルかきこみ兼キーの登録は，このマスク・ベクトルによるマスク制御のもとでおこなう．キー配列の第 1 要素 353 および第 4 要素 911 のハッシュ値はいずれも 5 になるため，ハッシュ表の第 5 要素にはまず 353 がラベル兼キーとしてかきこまれたあと，911 がかきこまれて，353 は消される．したがって，上書き検出すなわちラベル兼キーがただしくハッシュ表にかきこまれているかどうかの判定をおこなう．この判定を**第 2 種衝突**検出とよぶ．第 2 種衝突検出の結果をマスク・ベクトルとしてえると，その第 1 要素は第 3 要素とともに *false* になる．したがって，値が *false* であるこれらの 2 要素のハッシュ値とキー値がそれぞれ配列に連続に格納され，再登録の手続きが実行される．このばあいは 2 回の登録ですべての要素の登録が完了しているが，一般には，すべての登録が終了するまで，くりかえし第 1 種衝突検出，登録と第 2 種衝突検出がくりかえされる．

入力：　ハッシュ表 *table*（すでにキーが登録されていてもよい）.
　　ハッシュすべきキー集合 *key*[1 : *n*].
出力：　入力したキー集合が登録ずみのハッシュ表 *table*.

```
local     hashedValue[1 : n], entered[1 : n], increment[1 : n].          /* 局所変数の宣言 */

/* ハッシュ値の計算，第 1 種衝突検出とかきこみ */
hashedValue[1 : n] := hash(key[1 : n]);
    /* ハッシュ値を計算する (たとえば hash(x) = x mod size(table)).                    */
where table[hashedValue[1 : n]] = unentered  do                          /* 第 1 種衝突検出 */
    table[hashedValue[1 : n]] := key[1 : n];
        /* 未登録 (unentered) の部分だけかきこむ．同一エントリに重複してかきこむ可能性がある．*/
end where;

for i in 1 .. size(table)  loop
        /* テーブルのあふれによる無限ループをさけるためにくりかえしを有限回にしている．    */
        /* (本来はテーブルのあふれを検出すべきだが，ここでは無視している．)              */

/* 第 2 種衝突検出と登録ずみ要素のフィルタ (未登録要素の収集) */
    entered[1 : n] := (key[1 : n] = table[hashedValue[1 : n]]);    /* 第 2 種衝突検出 (上書き検出) */
    nrest := countTrue(not entered[1 : n]);              /* 上書きされた要素の数 を nrest とする．*/
    hashedValue[1 : nrest] := hashedValue[1 : n]  where not entered[1 : n];
    key[1 : nrest] := key[1 : n]  where not entered[1 : n];
        /* 上書きされた要素のハッシュ値とキーとを，それぞれ，よりみじかい配列 につめる．    */

/* 登録終了判定 */
    if nrest = 0  then  exit loop;                                  /* ループを脱出する．*/
    n := nrest;                                                     /* n の値を更新する．*/

/* つぎの添字値の計算，第 1 種衝突検出とかきこみ */
    increment[1 : n] := key[1 : n] & 31 + 1;
                                  /* 増分値の計算 ("&" は bitwise and.  size(table) > 31 を仮定する). */
    hashedValue[1 : n] := (hashedValue[1 : n] + increment[1 : n]) mod size(table);
    where table[hashedValue[1 : n]] = unentered  do                  /* 第 1 種衝突検出 */
        table[hashedValue[1 : n]] := key[1 : n];
            /* 未登録の部分だけかきこむ．同一エントリに重複してかきこむ可能性がある．    */
    end where;

end loop;
```

図 5.9　ベクトル処理によるハッシングの登録アルゴリズム

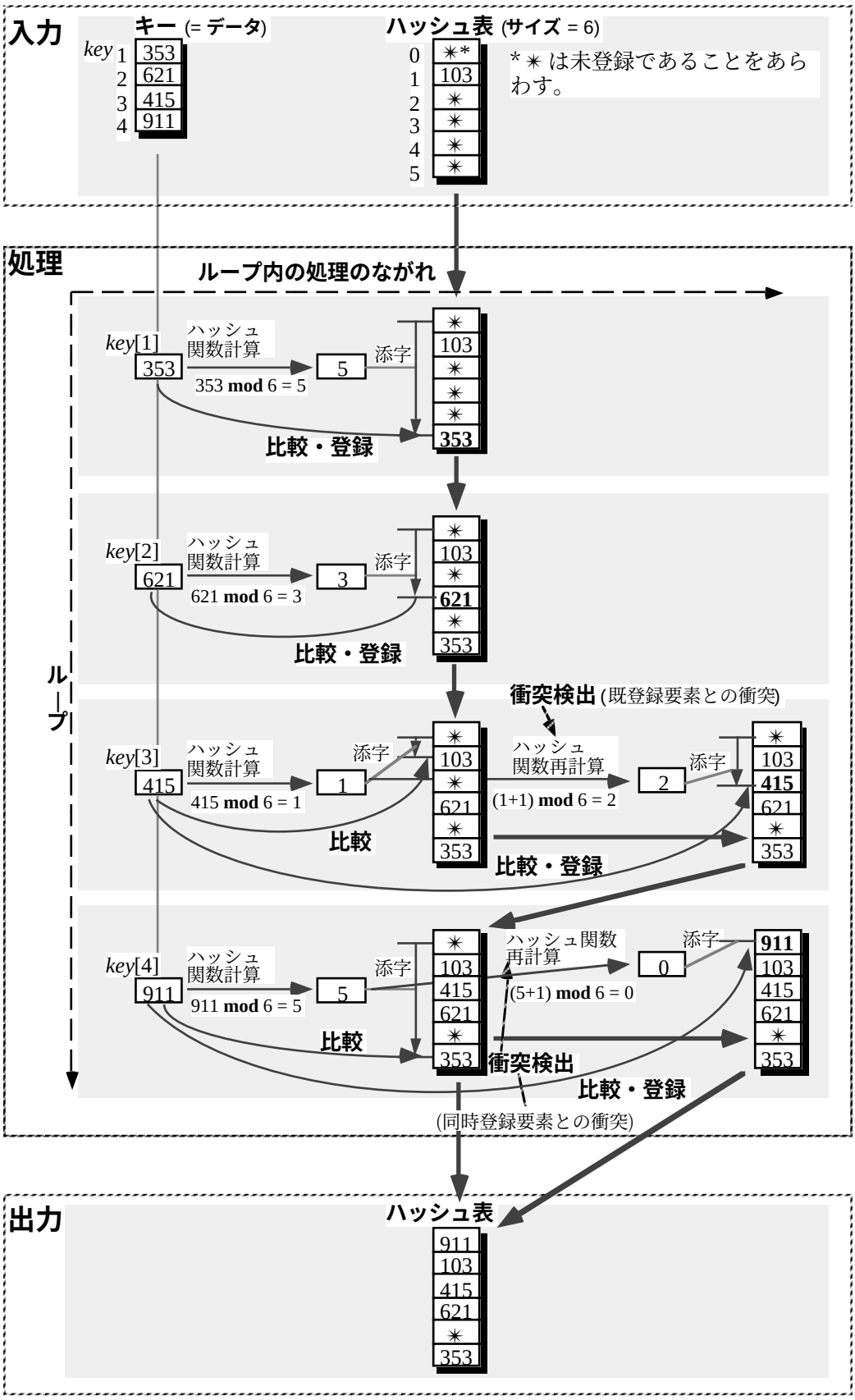


図 5.10　スカラ処理によるハッシュ表への複数データ登録例

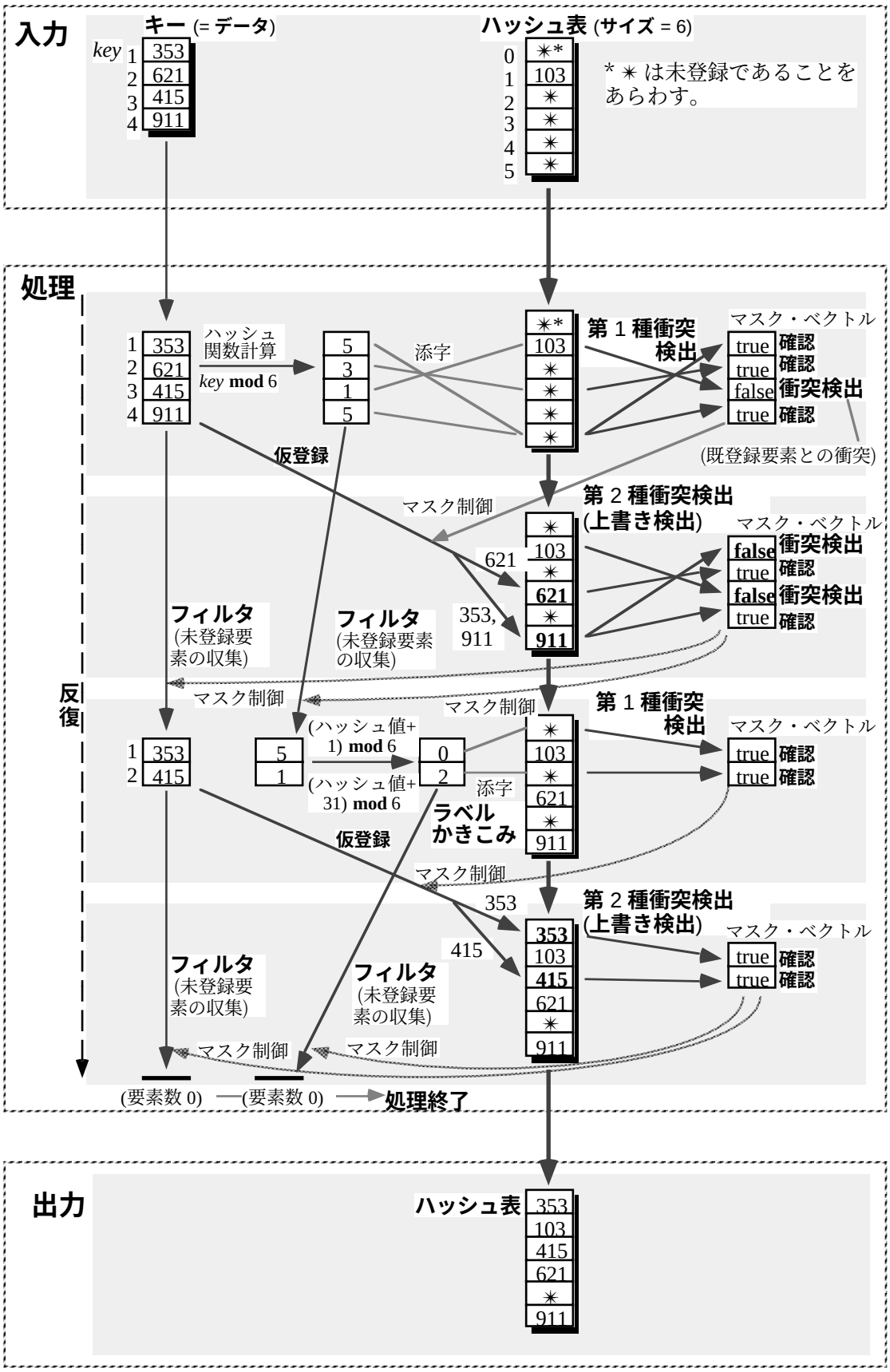


図 5.11　ベクトル処理によるハッシュ表への複数データ登録例

なお，S-810 などのベクトル計算機においては，1 個の命令で同一の番地に 2 回以上のかきこみをおこなったばあい，上記の例のように最後の要素が有効になる[注7]．しかし，S-820 の VIST 命令のようにそれが保証されないばあいもある．しかし，図 5.9 のアルゴリズムは，同一番地にかきこまれたいずれの要素が有効になるばあいでもただしく動作する．ただし最後以外の要素が有効になるばあいは，当然ながら，ハッシュ表に登録される順序は，最後の要素が有効になるばあいとはことなる．同一アドレスへのかきこみにおいて最初の要素が有効になるばあいには，ハッシュ表の内容は逐次処理のばあいとひとしくなる．

なお，図 5.9 のアルゴリズムには，同一の処理が 2 箇所に存在するなど，一見むだにみえる部分がふくまれている．このようにしたのは，このアルゴリズムをベクトル処理する際に，ベクトル長がひとしいループをまとめて処理することによって実行速度を向上させることをねらったためである．

5.4.2 実行結果

5.4.1 節でしめしたアルゴリズムの S-810 版とベクトル化前のアルゴリズムを Fortran でコーディングし，S-810 で性能を測定した．すべての最内側ループがベクトル処理されている．エントリ数 521 および 4099 の空ハッシュ表へのランダム・キーの一括登録に要した時間と加速率を**図** 5.12 ～ 5.13 にしめす．なお，より単純なアルゴリズムによる性能測定結果を [Kanada 90a] にしめした．

図 5.12 ～ 5.13 の横軸にしめした登録後のロード・ファクタすなわちハッシュ表の使用エントリの割合が 0.5 のときに加速率は最大値 5.2 ないし12.3 をとっている．したがって，上記の条件のもとではベクトル化による効果はおおきいということができる．ロード・ファクタが 0.5 以下のときにロード・ファクタ増大につれて加速率が増大するのは，一括登録をおこなっているためにロード・ファクタ増大につれてベクトル長が増大するからである．ロード・ファクタが 0.5 から 1 にちかづくにつれて加速率が低下するのは，ベクトル長の増大による加速効果よりも逐次性が増大することによる減速効果のほうがおおきいためである．しかし，ロード・ファクタを 1 にちかづけたとき加速率は 1 よりおおきい値に収束している．すなわち，上記の測定条件のもとでは，ロード・ファクタが 1 にちかづいてもなお並列性がかなりのこっているためベクトル処理のほうが有利だということがいえる．

また図 5.12 ～ 5.13 の測定結果は，金田 [Kanada 90a] の結果にくらべるとロード・ファクタが 0.5 ～ 0.98 のときの加速率が向上している．これは，5.4.1 節でのべたように，衝突時の添字の増分値として定数値のかわりにキーに依存する値をつかっているために連

[注7] S-820 においても VSTX 命令を使用するかぎりこの条件がみたされる．

続的な衝突がふせがれる効果のあらわれだとかんがえられる.

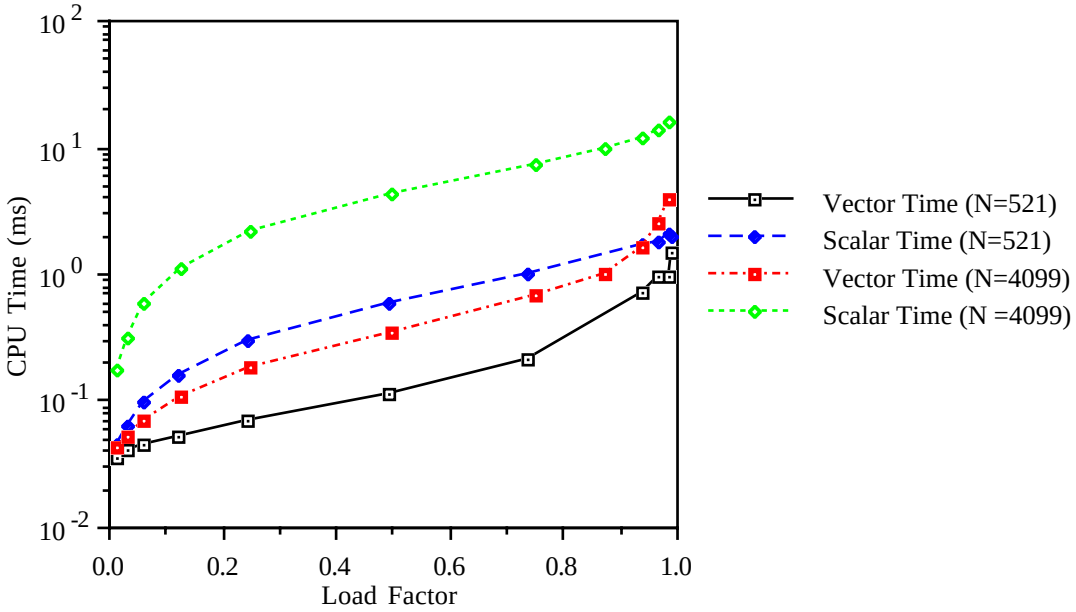


図 5.12　S-810 による空ハッシュ表への登録時間
(ハッシュ表サイズ *N* = 521 / 4099)

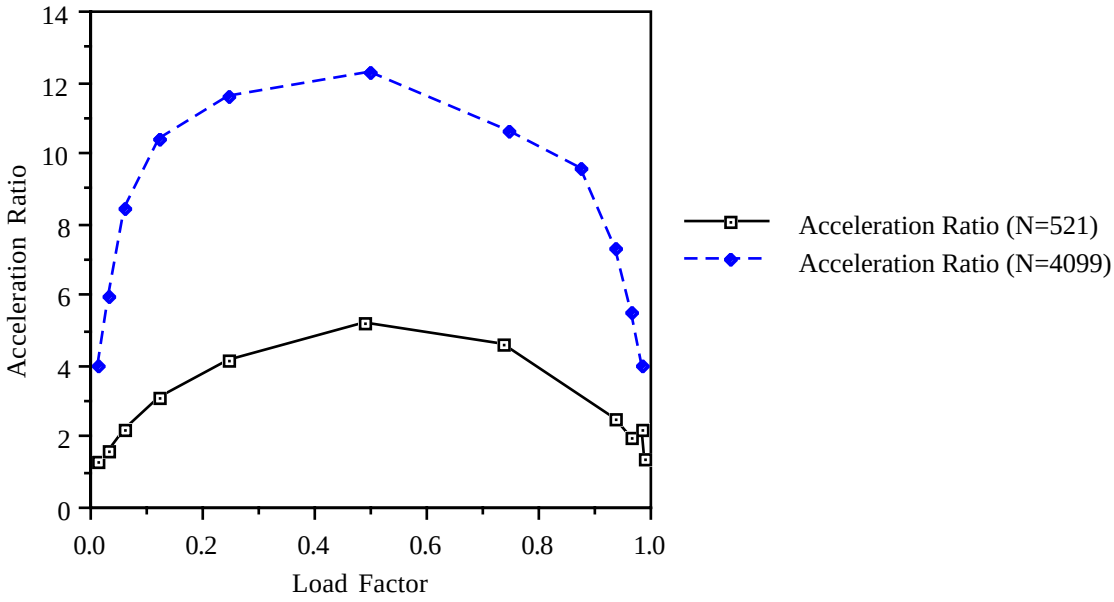


図 5.13　S-810 による空ハッシュ表への登録における加速率
(ハッシュ表サイズ *N* = 521 / 4099)

5.5 番地計算ソート等への応用

5.5.1 番地計算ソートのアルゴリズム

番地計算ソート (address calculation sorting) [Knuth 73] はハッシングに似た方法でデータを配列に登録し検索することによって，最善のばあい N 個のデータのソートを $O(N)$ でおこなうことができるソート方法である．番地計算ソートにはさまざまな変種があるが，そのなかに**線形探査ソート** (linear probing sort) [Gonnet 84] がある．ソートすべきデータをふくむ配列を A とする．線形探査ソートでは作業配列を使用するが，この配列を C とする．データは“ハッシュ”されて C に格納される．このときの“ハッシュ関数”としてはつぎのような性質をもつものがつかわれる．

$$data[i] \le data[j] \;\Rightarrow\; hash(data[i]) \le hash(data[j]); \qquad (1 \le i \le n,\ 1 \le j \le n)$$

この性質があるゆえに，この関数は正確にはハッシング関数ではない．しかし，ハッシングと同様に上書きラベル・フィルタ法をつかってベクトル処理することができる．上記の性質のゆえに，配列 C に格納されるときにデータはソートされた状態で格納される．ただし，C にはデータが不連続に格納され，途中に空領域が存在する．したがって，全データの登録がおわったあと，データを他の配列につめかえる．つめかえるための配列として，もとの配列 *data* を使用することができる．逐次処理による線形探査ソートのアルゴリズムを**図** 5.14 にしめし，その Fortran によるコーディング例を**付録** 2 にしめす．このアルゴリズムでは，未登録であることをあらわす値 *unentered* を配列 C のすべての要素にあらかじめ格納しておく．*unentered* は，C へのデータの挿入の際に番兵 (sentinel) として使用するために，配列 A にふくまれるどのデータよりもちいさい値でなければならない[注8]．

[注8] もちろん，番兵をつかわないアルゴリズムをつくることもできる．

```
input   A[1 : n]: array to sort {the element values must be in [0, Vmax)}.
output  A[1 : n]: sorted array.

local   C[0 : 3*n – 1], x, w, count, hashedValue;

for i in 0 .. size(C) – 1  loop
    C[i] := unentered;
end loop;                                              /* initialize C. */

/* Scatter the data into C. */
for i in 1 .. n  loop

/* A. Computing a "hashed" value of A[i]. */
    hashedValue := int( float(2 * size(C) * A[i]) / Vmax);

/* B. Finding the table entry to insert the new data A[i]. */
    while C[hashedValue] ≤ A[i]  loop
        hashedValue := hashedValue + 1;
    end while;

/* C&D. Inserting the new data and shifting the data in C. */
    w := C[hashedValue];
    C[hashedValue] := A[i];                            /* write the new data. */
    while w ≠ unentered  loop
        hashedValue := hashedValue + 1;
        x := C[hashedValue];
        C[hashedValue] := w;
        w := x;
    end while;
end for;

/* F. Packing the sorted data into A. */
count := 0;
for i in 0 .. size(C) – 1  loop
    if C[i] ≠ unentered  then
        count := count + 1;
        A[count] := C[i];
    end if;
end for;
```

図 5.14　逐次処理による番地計算ソート (線形探査ソート) のアルゴリズム

線形探査ソートのベクトル処理を上書きラベル・フィルタ法をもとにして実現したアルゴリズムを**図 5.15** にしめす．このアルゴリズムでは，ソートすべきデータをふくむ配列 A のインデクスをラベルとして使用している．ソートすべきデータをラベルとして使用すればオーバヘッドはすくなくなるが，そうすると配列 A が同一のデータを複数ふくむばあいにただしく動作しなくなるので，それはさけている．また，このアルゴリズムでは上書きラベル・フィルタ法が必要とする作業領域として配列 C そのものをつかっている．逐次処理のアルゴリズムと同様に，未登録であることをあらわす値 *unentered* を配列 C のすべての要素にあらかじめ格納しておく．図 5.15 のアルゴリズムは 6 個の部分 **A** ～ **F** から構成されている．これらの部分は，上書きラベル・フィルタ法に特有の部分である **E** をのぞいては，逐次処理によるアルゴリズム (図 5.14) における同名の部分と対応

している．

このアルゴリズムは，配列への並列代入文があり，また Fortran 90 のような **where** 文がある言語で記述されている．各配列代入文は並列に実行することができる．したがって，1 つの文で 1 つの配列要素に複数の代入をおこなえば，そのうちのいずれの値が格納されるかは不確定である (ただし，もちろん ELS 条件がなりたつ必要がある)．図 5.15 のアルゴリズムにおいては実際にこのような代入がおこなわれる可能性があるが，上書きラベル・フィルタ法にしたがっているため，そのばあいでもただしく動作する．しかし，この言語では，どの 2 つの文をとってもそれらが逐次に実行されたのとことなる結果をもたらしてはならない．

where 文はマスク・ベクトルすなわち論理型配列による実行の制御をあらわす．たとえば，$A = (1, 2, 3)$, $B = (10, 11, 12)$ かつ $M = (true, false, true)$ のときにつぎの文が実行されたとする．

where *M* **do** *A* := *B*; **end where**;

このとき *A* の値は $(10, 2, 12)$ となる．

この言語はまた，*countTrue* 関数と **where** 演算子をもつ．*M* がマスク・ベクトルであれば，式 *countTrue*(*M*) は *M* にふくまれる *true* の数をかえす．たとえば，$M = (true, false, true)$ のときの *countTrue*(*M*) の結果は 2 となる．式 *A* **where** *M* は，*M* の *true* である要素に対応する *A* の要素だけからなる配列をあらわす．たとえば，$A = (1, 2, 3)$ かつ $M = (true, false, true)$ ならば *A* **where** *M* の結果は $(1, 3)$ となる．さらに，式 $A[x : y]$ は *A* の部分配列 (subarray, array slice) $(A[x], A[x+1], \ldots, A[y])$ をあらわす．

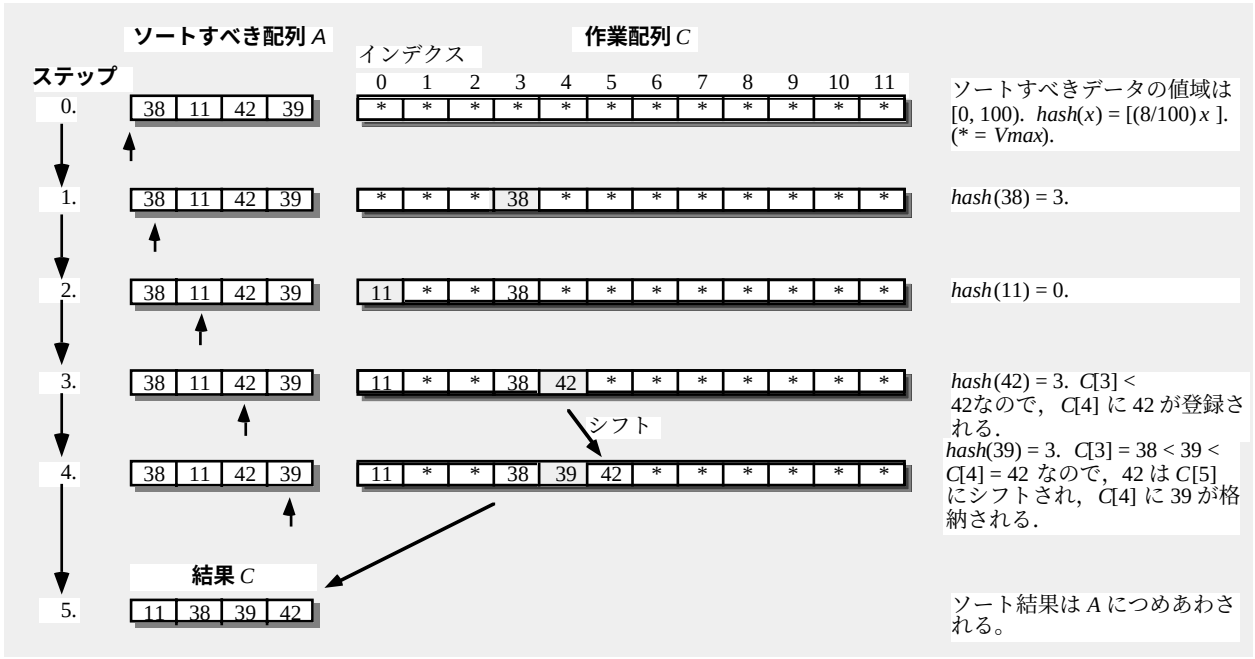
ハッシングのばあいと同様に，並列処理可能なデータの集合が複数になったばあいにはこれらに対して逐次的に処理をおこなうが，このくりかえし処理においては，まだ登録されていないデータは部分 **C** において配列 *C* に挿入される．しかし，ここでデータを登録しようとした場所にもしすでに以前に登録されているばあいには，すでに登録されているデータは部分 **C** において配列 *work* に退避される．そして，配列 *C* におけるつぎに利用可能な場所に部分 **D** において再格納される．

図 5.16 には逐次処理とベクトル処理それぞれによる線形探査ソートの例をしめす．

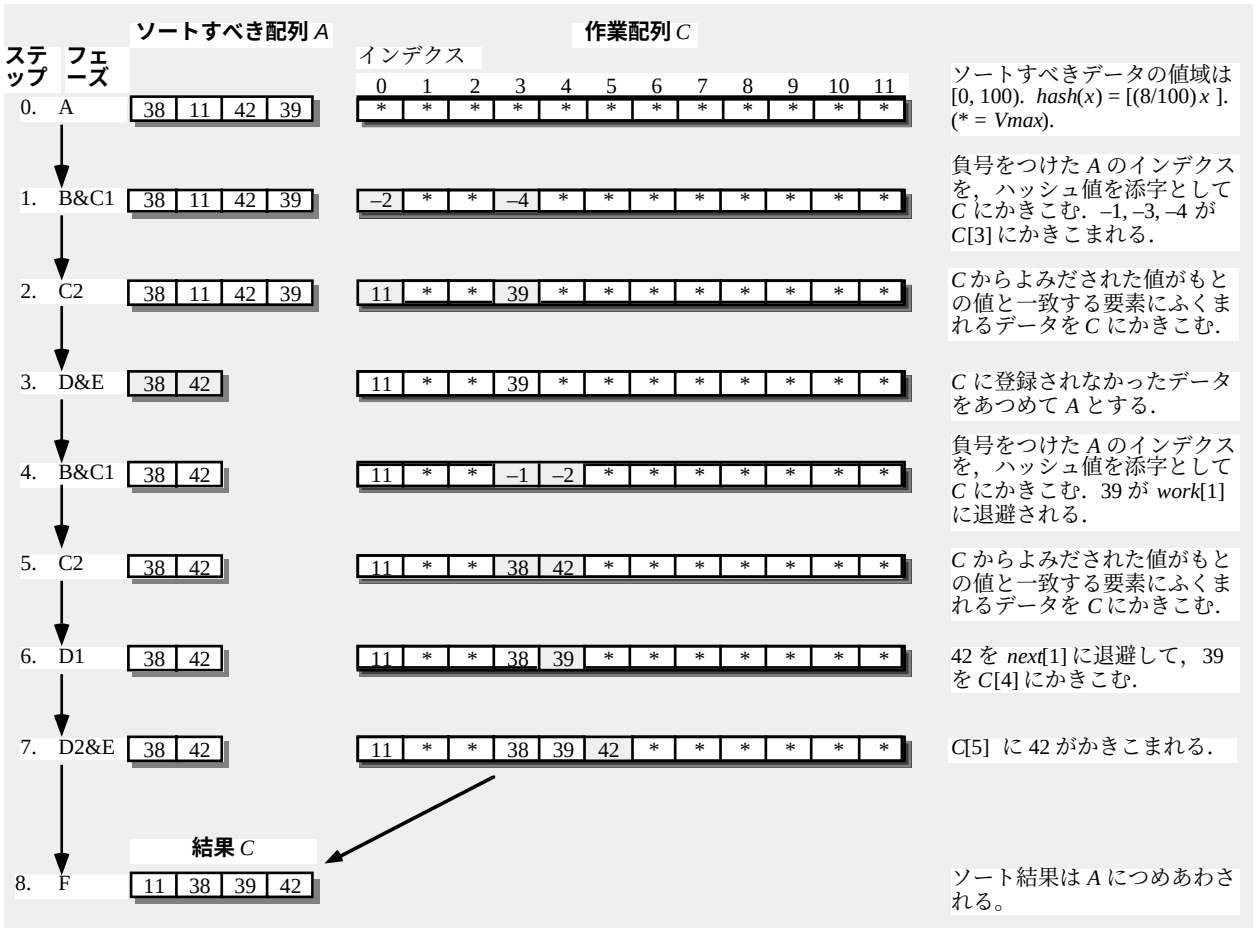
頻度ソート (distribution counting sort) [Knuth 73] (またはバケツ・ソート) も線形探査ソートと同様に，上書きラベル・フィルタ法によってベクトル処理することができる．そのアルゴリズムはここではしめさない．

```
input   A[1 : n]: array to sort {the element values must be in [0, Vmax)}.
output  A[1 : n]: sorted array.

local   C[0 : 3*n – 1], uninsertable[1 : n], work[1 : n], entered[1 : n],
        toShift[1 : n], index[1 : n], next[1 : n], nonempty[1 : n].

C[0 : size(C) – 1] := unentered;                    /* initialize C  (unentered = Vmax) */

/* A. Computing "hashed" values. */
hashedValue[1 : n] := int( float(2 * size(C) * A[i]) / Vmax);
nrest := n;
repeat

 /* B. Finding the table entries to insert the data. */
    repeat
        uninsertable[1 : nrest] := (C[hashedValue[1 : nrest]] ≤ A[1 : nrest]);
                /* check the first type of collision with stored data.  If hashedValue[i] ≠ unentered,   */
                /* the right-hand side condition holds, i.e., there is a first type of collision.       */
        Nuninsertable := countTrue(uninsertable[1 : nrest]);
                /* count the number of uninsertable (colliding) data.                                   */
        where uninsertable[1 : nrest]  do
            hashedValue[1 : nrest] := hashedValue[1 : nrest] + 1;
        end where;
    until Nuninsertable = 0;
            /* repeat until there is no first type of collision.                                        */

/* C. Inserting the data. */
    work[1 : nrest] := C[hashedValue[1 : nrest]];
            /* save the original values of C to work.                                                   */
    C[hashedValue[1 : nrest]] := ι;
            /* store the identifiers to check {ι is array (1, 2, …, nrest)}.                            */
            /* An entry of C may be written twice or more (overwritten).                                */
    entered[1 : nrest] := C[hashedValue[1 : nrest]] = ι;
            /* check the second type of collision between newly entered data.                           */
    where entered[1 : nrest]  do
        C[hashedValue[1 : nrest]] := A[1 : nrest];
    end where;     /* enter */

/* D. Shifting the work array elements {only for successfully inserted data}. */
    toShift[1 : nrest] := entered[1 : nrest] and (work[1 : nrest] ≠ unentered);
    NtoShift := countTrue(toShift[1 : nrest]);
    work[1 : NtoShift] := work[1 : nrest]  where toShift[1 : nrest];
    index[1 : NtoShift] := (hashedValue[1 : nrest] + 1)  where toShift[1 : nrest];
    while NtoShift > 0  loop
        next[1 : NtoShift] := C[index[1 : NtoShift]];
        C[index[1 : NtoShift]] := work[1 : NtoShift];
        nonempty[1 : NtoShift] := next[1 : NtoShift] ≤ unentered;
        count  := countTrue(nonempty[1 : NtoShift]);
        work[1 : count] := next[1 : NtoShift]  where nonempty[1 : NtoShift];     /* pack work. */
        index[1 : count] := index[1 : NtoShift] + 1  where nonempty[1 : NtoShift];   /* pack index. */
        NtoShift := count;
    end while;

/* E. Collecting the uninserted data for the next iteration. */
    irest := countTrue(not entered);
    hashedValue[1 : irest] := hashedValue[1 : nrest]  where not entered[1 : nrest];
    A[1 : irest] := A[1 : nrest]  where not entered[1 : nrest];
    nrest  := irest;
until nrest = 0;        /* until all the data are inserted. */

/* F. Packing the sorted data into A. */
A[1 : n] := C[0 : size(C) – 1]  where (C[0 : size(C) – 1] ≠ unentered);
```

図 5.15　ベクトル処理による番地計算ソート (線形探査ソート) のアルゴリズム



(a) **逐次版**



(b) **ベクトル化版**

図 5.16　逐次版およびベクトル化版の番地計算ソートによるソート例

5.5.2 実行結果

線形探査ソートと頻度ソートの両方を Fortran でコーディングして S-810 によって実行性能を測定した[注9]．**表 5.1** にその測定結果をしめす．線形探査ソートにおいては $N = 2^{10}$ のとき 7.65，$N = 2^{14}$ のとき 12.84 という高加速率がえられた．頻度ソートの加速率はそれにくらべるとひくい．

表 5.1　ベクトル化された $O(N)$ ソートの実行速度と加速率

アルゴリズム	N	S-810/20 CPU 時間		加速率
		スカラ処理 (逐次実行)	ベクトル処理	
線形探査ソート (番地計算ソート)*	2^6	289	110	2.62
	2^{10}	4,286	560	7.65
	2^{14}	66,955	5,215	12.84
頻度ソート**	2^6	12,206	1,522	8.02
	2^{10}	13,072	1,738	7.52
	2^{14}	30,089	5,667	5.31

* 作業配列 C のおおきさは $3n$ とした．
** 作業配列のおおきさは 2^{16} とした (これはデータの値域にひとしい)．

[注9] ただし，本来は Fortran では並列代入が記述できないので，このプログラムは Fortran の規格にしたがっているとはいえない．この章で上書きラベル・フィルタ法によるプログラムを Fortran などの逐次的な手続き型言語で記述しなかったのも，それが不可能だからである．

5.6 2 分探索木への登録への適用

5.6.1 2 分探索木への登録のアルゴリズム

複数のデータを 2 分木に登録するベクトル処理アルゴリズムを FOL1 をつかって開発した．このアルゴリズムでは再バランスはおこなわない．アルゴリズムの記述は省略するが，Fortran によるプログラムを**付録 3** にしめす．

5.6.2 実行結果

図 5.17 に S-810 における性能測定結果をしめす．水平軸は木に登録した一様乱数であるキーの数をあらわし，垂直軸は加速率をあらわす．木は登録開始前に *Ni* 個の乱数キーをもつ要素をふくむ (*Ni* をパラメタとして実験している)．すなわち，木ははじめから空ではない．空の木をつかわなかった理由は，そうするとベクトル処理にとってあまりに不利だからである．なぜなら，木が空のときには登録すべきすべてのキーが衝突するからである．図 5.17 のデータは各測定点ごとにただ 1 個の試行をおこなっただけであり，十分信頼性が高いとはいえなず，より精密な実験をおこなうことがのぞましい．しかし，この実験からつぎのように結論することはできるだろう．木の初期サイズがちいさすぎず，登録データ数がすくなすぎなければ，平均加速率は，10 倍のオーダではないが，1.0 より高い．

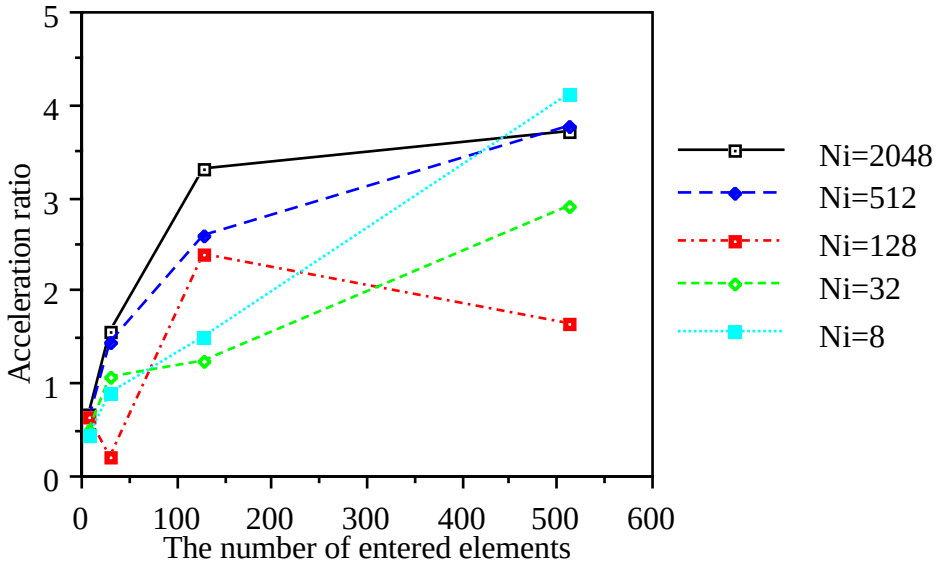


図 5.17　2 分木への複数データ登録における加速率

5.7 マーキング・PV 操作との類似点

いくつかの古典的なアルゴリズムと上書きラベル・フィルタ法とのあいだに類似点をみいだすことができる．これらは，上書きラベル・フィルタ法が SIMD 型並列計算機における一般性のある並列処理技法となりうることを示唆しているようにおもわれる．これらの点に関して指摘しておきたい．

❑ マーキングにもとづくアルゴリズムとの類似点

共有部分があるデータに一種のマークをつけながら処理するという点においては，上書きラベル・フィルタ法はマーキングにもとづくグラフの検索法やガーベジ・コレクションの方法ににているということができる．ただし，マークをつける目的はことなっている．上記のようなマーキングにもとづくアルゴリズムにおいては各データを複数回処理しないようにするためにマーキングをおこなうのに対して，上書きラベル・フィルタ法においては各データを複数回処理するために「マーキング」をおこなう．複数回処理するためという点では，つぎにのべる並列処理のばあいと共通している．

マーキングにもとづくアルゴリズムにおいてもまた上書きラベル・フィルタ法においても，処理対象のデータ構造を構成する各要素データ上に作業領域をとることを必要とする．この作業領域のおおきさについていえば，逐次のグラフ検索法のばあいにはマーキングのために 1 bit あれば十分 (アルゴリズムによっては 2 bit 必要とする) だが，上書きラベル・フィルタ法においてはベクトル長 (プロセス数) の対数に比例する単位容量の作業領域が必要である．

❑ PV 操作にもとづく並列処理との類似点

一種のフラグをつかって並列処理をおこなうという点では，上書きラベル・フィルタ法はセマフォをつかった PV 操作 (send /receive 操作) にもとづく並列処理にているということができる．ただし，PV 操作にもとづく並列処理は MIMD 型計算機のための技法であるのに対し，上書きラベル・フィルタ法は SIMD 型計算機のための技法である．

MIMD 型計算機とはちがって SIMD 型計算機においては同期がハードウェアによってとられるため，上書きラベル・フィルタ法によって，パイプライン計算機にはオーバヘッドがおおきい非同期の PV 操作にもとづく処理より効率的な同期処理が可能になっているということができるだろう．すなわち，第 1 に上書きラベル・フィルタ法においては共有部分へのかきこみにおいて排他制御をおこなう必要はない (ただし，そのかわりに ELS 条件だけはなりたっている必要がある)．また，第 2 に処理開始前に上書きラベル・フィルタ法処理をおこなうだけでよく，処理終了時に特別の操作をおこなう必要はない．

5.3.3 節でしめした複数データかきかえの上書きラベル・フィルタ法は，複数のセマフォをつかって複数資源を同時に獲得して処理する方法に相当する．

5.8 関連研究

共有部分がある複数データのベクトル処理をめざした研究として，本研究と水準を比較しうる研究はないものとかんがえられる．ただし，つぎのような関連研究がある．

Appel ら [Appel 89] は Lisp などにおける記憶管理のためのガーベジ・コレクションをベクトル処理でおこなう方法を開発している．使用中のデータのマーキングをベクトル処理するには共有されたデータをあつかえる必要があり，そのためにこの章でのべたのと類似の方法を使用している．また，三木ら [Miki 91] は LSI の配線アルゴリズムである迷路法をベクトル処理するために，上書きラベル・フィルタ法と類似の方法をわれわれとは独立に開発している．しかし，これらのいずれにおいても，上書きされたデータに関しては処理をおこなう必要がないという点で上書きラベル・フィルタ法が対象としている処理とはことなっている．すなわち，これらの方法では上書きラベル・フィルタ法が出力する集合のうち最初のもの (S_1) 以外をもとめる必要がない．したがって，これらは上書きラベル・フィルタ法と水準を比較しうる研究ではないとかんがえられる．

5.9 まとめ

この章でしめした上書きラベル・フィルタ法をつかうことより，共有のあるデータをかきかえる処理がベクトル処理可能になることがしめされた．また，性能評価の結果，ハッシュ表への登録と番地計算ソートおよび頻度ソートに関しては，上書きラベル・フィルタ法の適用によって，いちじるしく実行の高速化をはかれることがわかった．上書きラベル・フィルタ法はハッシングやソートにとどまらず広範囲に応用できる方法であることがわかっているから，リスト，木，グラフの処理をはじめとする各種の記号処理のベクトル計算機または SIMD 並列計算機による実行を可能にするために，上書きラベル・フィルタ法は有望なベクトル処理方法だとかんがえられる．

これまでに上書きラベル・フィルタ法の適用をこころみたのは，共有のあるデータをかきかえる処理をふくむ各種のアルゴリズムのなかのほんの一部である．この方法を他のさまざまなアルゴリズムに適用することが今後の最大の課題である．