プログラミング言語 14－3
（1987. 12. 18）

スーパーコンピュータ上の並列論理型言語処理系

辰口和保　村岡洋一
早稲田大学理工学部

スーパーコンピュータによる論理型言語の高速実行のための引数の突き合わせによるユニフィケーションのベクトル処理手法を提案し、これを用いた２種のインタプリタ、ＯＲ並列インタプリタと（ＯＲ並列＋制限ＡＮＤ並列）インタプリタについて報告する。現段階では、両処理系とも満足のいく速度を得られてはいないが、突き合わせ操作の際には十分なベクトル長が得られており、ユニフィケーションのベクトル処理の可能性が確認できた。

Parallel Logic Programming Language Interpreters on Supercomputers

Kazuyasu TATSUGUCHI　Yoichi MURAOKA
Department of Science and Engineering,WASEDA University
3-4-1, Ohkubo, Shinjuku-ku, Tokyo, 160, Japan

For fast execution of a logic programming language on supercomputers, we are studying a vector processing oriented unification method.In This paper,we introduced the "argument based vector unification method" and present two interpreters of a parallel logic programming language - an OR parallel interpreter and an "OR parallel+RAP" interpreter.At present,they are not much fast.But there remains the posibility for improving the efficiency as the length of vectors obtained are long enough to take advantage of the vector facility.

１．はじめに

スーパーコンピュータの高いパフォーマンスを数値計算以外の分野にも利用しようという研究が始まっている[1],[2]。 そのような研究の中に、並列論理型言語の高速処理系の研究がある[3],[4],[5]。 金田[4]ではPrologをベクトル処理向きの中間言語にコンパイルし、これをスーパーコンピュータ上で高速実行する方法を提案している。並列処理方式としてはＯＲ並列処理の立場をとっている。またNilsson[4]では、インタプリタ方式のベクトル処理向き並列論理型言語FLENG-Prologを提案している。並列処理方式としては、committed choiceによるＡＮＤ並列処理の立場をとっている。

本論文ではユニフィケーションのベクトル処理手法を提案し、これに基づく並列論理型言語のＯＲ並列インタプリタと（ＯＲ並列＋制限ＡＮＤ並列）インタプリタをＦＯＲＴＲＡＮでプログラムし、これをベクトル化コンパイラでコンパイルした場合の並列処理効率について述べる。

２．論理型言語のベクトル処理

スーパーコンピュータは、大量データに対する一様な処理を高速で実行（ベクトル処理）する。これをＦＯＲＴＲＡＮでみれば

```
      ＤＯ　１０　Ｉ＝１，１０００
        Ａ（Ｉ）＝Ｂ（Ｉ）＊Ｃ（Ｉ）
        Ｄ（Ｉ）＝Ｂ（Ｉ）＋Ｃ（Ｉ）
   １０　ＣＯＮＴＩＮＵＥ
```

のようなＤＯ文による配列データの処理が高速化される。ＤＯ文の繰り返し回数（ベクトル長）が多いほど汎用機に対する処理速度は向上する。反面、ベクトル長が短いと、かえって処理速度を落とす。また、配列要素の依存関係によってはベクトル処理できない場合がある。スーパーコンピュータ上で実行するプログラムを作成する際には、ベクトル処理向きの性質の良いＤＯループを数多く見出さねばならない。この点から論理型言語をスーパーコンピュータ上に実装する場合を考えてみる。

逐次型Prologの場合、その実行機構の大きな位置を占めるのは、ユニフィケーションとバックトラック機構である。ユニフィケーションは各種データ型に対するタグ判定、変数のデレファレンスや束縛操作からなり、バックトラックは各種スタックの複雑な操作からなっている。逐次型Prologは１つのgoalに対してユニフィケーションを行ない、スタック操作を行なって、全ての候補が失敗した時にバックトラックを起こす。この過程においてベクトル処理向きのＤＯループによる操作を見出すのは困難であり、逐次型Prologをスーパーコンピュータ上にそのまま実装することは困難である。

１つのgoalに対してではなく、複数のgoalに対しての処理を考えることでＤＯループ使用の可能性が出てくる。これは、並列論理型言語のスーパーコンピュータ上への実装である。並列論理型言語では、処理の複雑さ、並列処理ハードウェアからの要請などから、バックトラック機構は排除されている。したがって、バックトラック機構を排除した並列論理型言語を採用すれば、問題となるのはユニフィケーションをいかにしてスーパーコンピュータ上に実装するかである。

３．ベクトル処理向きユニフィケーション

ユニフィケーション動作はさまざまなデータ型のタグ判定、デレファレンス、変数の束縛操作からなる。goalごとのユニフィケーション動作は異なり、単純にユニフィケーション動作をＤＯループにしてgoalの数だけ繰り返してもベクトル処理することはできない。細部にいたってベクトル処理方法を検討しなくてはならない。

データ型の豊富さは逐次型Prologのユニフィケーションの効率的実行に役立つ。しかし、ベクトル処理を考える時、タグごとに比較ループを回すのは効率的ではない。また、インタプリタの場合にはgoal側と候補節側の引数がunify可能であるかどうかが判定できればよい。そこで言語の扱うデータ型を極力減らし、goal側と候補節head側の引数を一列に並べて突き合わせ操作することによってunify可能性を判定することを考える（図３－１）。この突き合わせ操作によってマスクベクトルを作り、これをもとにその後の処理を行なう。データ型としては、

アトム　変数　構造

の３種とする。リストは構造に含め、データ型としては採用しない。

引数を一列に並べての突き合わせ操作をする際には、

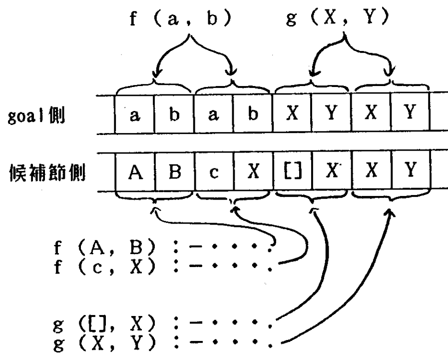


図３－１：引数の整列

変数の値のデレファレンス
構造引数

の２点が問題となる。

・変数の値のデレファレンス
構造共有方式の場合、デレファレンスのためのポインタの手繰りは厄介な仕事でありベクトル処理には不向きである。これを回避するため変数に直接に値を書き込んでしまうと、構造共有方式はとれなくなる。そこで今回の処理系では構造コピー方式を採用し、変数に直接に値を書き込むことにする。

・構造引数
ごく一般的なPrologのデータ構造で、構造データは図３－２ａのように表わされる。これでは引数の突き合わせのためにポインタを手繰らねばならない。しかもその先にどのような大きさの構造があるかわからず、要素同士の対応がとれるよう引数を並べることはできない。しかしその構造のfunctor/arityは決まった長さの識別記号（シンボルテーブル・アドレス）であるから、これなら要素の対応がとれるように並べることができる。そこでこの部分を上に引き上げ、functor/arityと引数のあるアドレスへのポインタとによって構造の引数を表わすことにする（図３－２ｂ）。

構造引数の持つ引数のユニフィケーションは、そのfunctor/arityのユニフィケーションが成功したものについてのみ行なえばよい。このためにはポインタを手繰り引数を並べる必要があるが、１回の突き合わせ操作で複数のgoalのユニフィケーションが

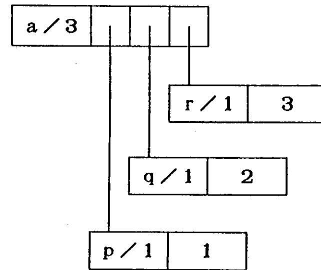


図３－２ａ：従来のデータ構造

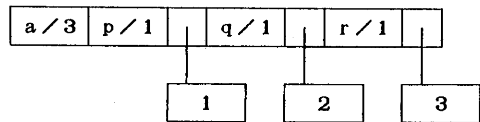


図３－２ｂ：ベクトルユニフィケーション向きデータ構造

行なわれており、成功・失敗の確定したgoalもあるはずなので、それらに対する処理を先に行なう。そして、次のユニフィケーション・サイクルで他の新しいgoalの引数とともに突き合わせ操作を行なう。したがって、構造を引数として持つgoalは、その構造のネストの分だけユニフィケーションの完了が遅延される。

これでタグ判定とデレファレンスの問題は回避できた。残りは変数に対する値の束縛である。コピー方式を採用しているので値の束縛は、直接変数位置に束縛されるものを書き込むことになる。これをベクトル処理するには同じ変数に対して複数の書き込みがあってはならない。つまり一列にならべた引数配列に同一の変数が含まれてはならず、したがって、一つのgoal内でも同一の変数が複数存在してはならない。このため、

f（X，X）：－・・・・.

は、

f（X，Y）：－X＝Y，・・・・.

と書き換えておく必要がある。さらに変数同士のユ

ニフィケーションによって実行中にこの制限が破られる可能性があり、これに対するチェックを行なわねばならない。

以上のようなデータ構造、構造コピー方式、シンタックス上の制限と変数チェック機構を導入することで、ユニフィケーションのベクトル処理を行なうことができる。

4．処理系の概要

本論文で述べる２種の処理系に共通の特長には次のものがある。

- ベクトル処理向きユニフィケーション
- データ構造
- ground instance共有の構造コピー方式
- bounded queue
- 組み込み述語は"＝"（unify）のみ

さらに制限ＡＮＤ並列処理（ＲＡＰ）のために次のような特長を持つ。

- テーブルによる並列性管理
- 実行時制限ＡＮＤ並列処理

ベクトル処理向きユニフィケーションについては３で述べたので、ここではそれ以外のものについて説明する。

・データ構造

述語と引数の構造は図４－１のようにする。述語の引数は４個までとし、それよりも多い時は構造引数を作る。節に関しては、

’：－’（head、body）．

の形の述語とする。bodyは２引数の’，’のネストとはせず、並び順に直接ポインタでつなぐ。

・ground instance共有の構造コピー方式

全ての構造をコピーしていたのでは、その手間は莫大なものとなる。そこでground instanceはgoal間で共有することとし、コピーは行なわない。

述語の構造

ｼﾝﾎﾞﾙﾃｰﾌﾞﾙ ｱﾄﾞﾚｽ	引数 1	引数 2	引数 3	引数 4	残りのgoal列へのポインタ

引数の構造

アトム	ｼﾝﾎﾞﾙﾃｰﾌﾞﾙ ｱﾄﾞﾚｽ	アトムタグ
構造	ｼﾝﾎﾞﾙﾃｰﾌﾞﾙ ｱﾄﾞﾚｽ	構造のアドレス
変数	変数タグ	変数番号

図４－１：データ構造

・bounded queue

並列処理されるgoalのスケジューリング戦略としてはbounded queue方式を採用する。明示的なgoal数の制限は行なわない。

・組み込み述語は’＝’（unify）のみ

’＝’（unify）は、変数の存在制限を解消するために用いられ、第一引数をgoal側ユニフィケーション配列に、第２引数を候補節側ユニフィケーション配列に入れる操作をする。それ以外の組み込み述語については、現在実装されていない。

・テーブルによる並列性制御

（ＯＲ並列＋ＲＡＰ）インタプリタでは、２種の並列性が混在する。これを管理するため、図４－２のようなテーブルを用いる。ＡＮＤテーブル、ＯＲテーブルともにプロセスカウンタ、状態フラグ、親テーブルへのポインタの３つのフィールドを持っている。プロセスカウンタは、そのテーブルに属して現在実行中のgoal列の数を表わす。状態フラグは、ＡＮＤテーブルにおいてはＡＮＤ関係にあるgoal列が失敗しているか否かを表わし、ＯＲテーブルにおいてはＯＲ関係にあるgoal列が成功しているか否かを表わす。

・実行時制限ＡＮＤ並列処理

制限ＡＮＤ並列処理[6]は、共有変数を持たないgoal列同士を並列に処理する方式である。このための変数によるgoal列のクラスタ化を実行時に全て行

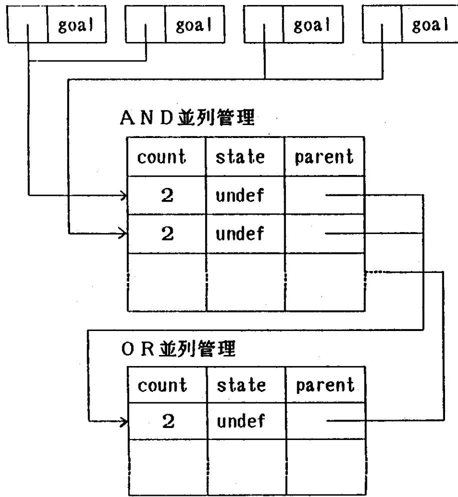


図４－２：並列性管理テーブル

なう。非常に負荷が重いが、並列に実行できるgoal数が増えるため、ユニフィケーションのベクトル処理にはプラスに働く。

５．処理系の動作

本処理系はおおよそ図５－１のような実行サイクルで動作する。この動作を（ＯＲ並列＋ＲＡＰ）インタプリタを例にとり少し詳しく説明する。

・goalの取り出しと引数の整列

まず、処理系はqueueから１個のgoal列を取り出し、その先頭goalとfunctor/arityの一致する候補節headの引数をユニフィケーション用の２つの配列に整列させる。goal側の引数は、候補節の数だけコピーされる。この時、各goal（のコピー）と候補節ごとに識別番号が与えられ、ユニフィケーション用配列のどの要素がどのgoal列に属するのか、残りのgoal列、及び節のbody部へのポインタなどの各種情報を保持する配列も同時に作られる。さらに並列性管理のため、同一のheadをも持つ候補節群に対してＯＲテーブル上に１つの位置が与えられ、候補節ごとにＡＮＤテーブル上に１つの位置が与えられる（図５－２）。

唯一の組み込み述語’＝’はここで例外処理される。’＝’が取り出されると第１引数をgoal側のユニフィケーション用配列に、第２引数を候補節側のユニフィケーション用配列に入れ、arity １の述語のように処理を行なう。

以上の部分は、逐次的に行なわれる部分が多い。queueが空になるか、ユニフィケーション用配列が一定の長さを超えそうになると整列を打ち切り、マスクベクトルの生成に移行する。打ち切りの長さは１２８としている。

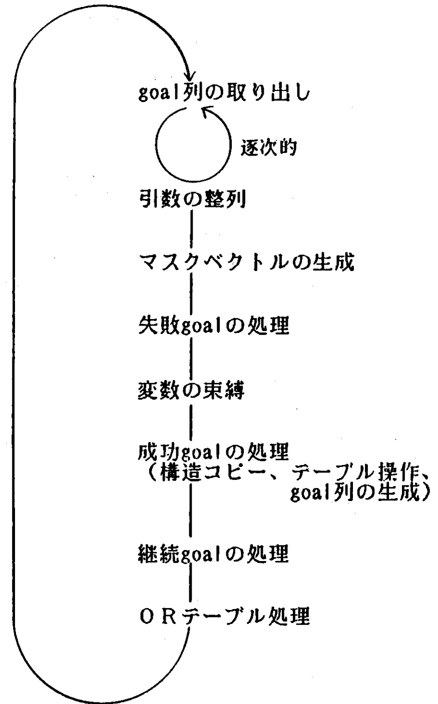


図５－１：処理系の動作サイクル

・マスクベクトルの生成

ユニフィケーション用配列ができると配列同士または変数タグとの比較によって次の４種の論理値ベクトルを作る。

MVAR1	goal側が変数
MVAR2	候補節側が変数
MEQ	CAR部が一致
MSTRCT	CDR部が一致

この論理値ベクトルからどの引数でユニフィケーションが失敗したか、どの引数のユニフィケーションが次のサイクルまで継続されるかがわかる。これと引数のgoalに対する従属情報からどのgoalのユニフィケーションが失敗したかを示すマスクベクトルFAILID、どのgoalのユニフィケーションが継続されるかを示すマスクベクトルCONTIDができる。ユニフィケーションが成功したgoalはこの２つのマスクの論理和の否定をとることによって知ることができる。

以上のFAILID、CONTIDを用いて以後の処理を進める。

・失敗goalの処理

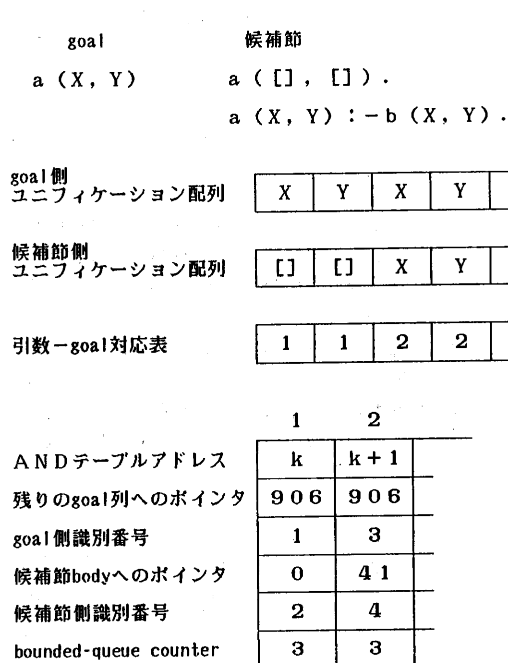
FAILIDがTRUEのgoalはユニフィケーションが失敗したものである。この場合には、そのgoalが属するＡＮＤテーブルのプロセスカウンタを１減じ、状態フラグをFAILにする。もしプロセスカウンタが０となれば、さらにその親にあたるＯＲテーブルのプロセスカウンタを１減じる。

・変数の束縛（変数のコピー）

変数には固有のメモリー領域が割り当てられている訳ではなく、変数に与えられた固有の変数番号と引数整列の際にゴールと節に与えられた識別番号の２つを用いてハッシュ・テーブルのキーを計算し、そのキーの場所に変数の値が書き込まれる。同時にそのキーがその変数のコピーに与えられる変数番号となる。キーの初期値の計算はベクトル処理できるが衝突の回避は逐次的に行なわれる。値の束縛されていない変数は、キーそのものがテーブルに書き込まれる。このようにして作られたテーブルを構造コピーの際に参照することでgoal列内にある全ての変数に値を書き込む。

・成功goalの処理

成功goalの処理には構造のコピー、テーブル操作、goal列の生成の３種がある。



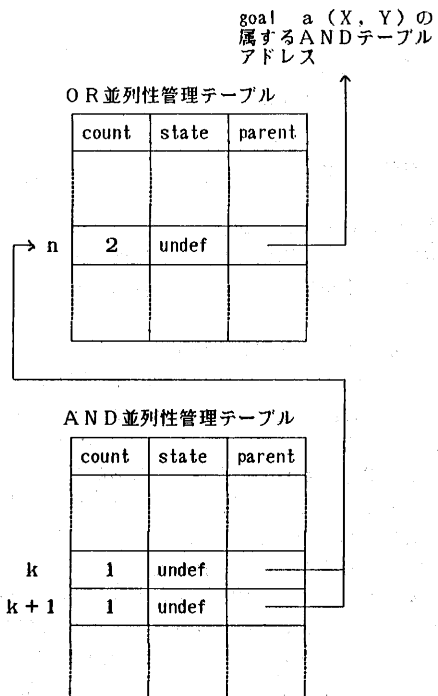


図５－２：各種の情報のセットアップ

A)構造のコピー

構造のコピーは、ユニフィケーションが成功したものに対して行なわれる。ユニフィケーションが継続されているものは、変数に束縛される構造引数のコピー操作のみが行なわれる。コピーのため、残りのgoal列、及び節のbody部の全てを探索し、構造引数の位置や変数の位置についての情報を集める。この操作は逐次的であるが、一旦情報が集まれば実際にコピーをメモリー上に構築することはベクトル処理することができる。

B)テーブル操作

ユニフィケーションが成功したものに対して、残りのgoalも節のbodyも空である場合には、そのgoal列の実行が成功して終了したことを示している。そこで自分の属するＡＮＤテーブルのプロセスカウンタを１減ずる。プロセスカウンタが０となれば、状態フラグを見て、FAILであればなにもせず、FAILでないならそのＡＮＤテーブルの親であるＯＲテーブルのプロセスカウンタを１減じ、その状態フラグをSUCCESSにする。

C)goal列生成

ここでは、変数チェック、ground instanceの識別、およびＲＡＰ化を行なう。

変数チェックはユニフィケーションにおけるgoal列内の変数についての制限を満足させるためのものであり、これに抵触する変数を発見すると、その位置の変数番号を変え、新たにgoal列の末尾にunify述語を挿入する。

ground instanceの識別は、値の束縛の起こった変数を含むgoalに対して行なわれ、groundとなった全ての構造にフラグを立てる。特に別領域をとってコピーはしない。

ＲＡＰ化はgoal列の順序関係を保存しながら共有変数によってgoalをクラスタ化する操作である。これにはまず、図５－３のようなgoalと変数の関係を示す配列を作る。配列の添字のもっとも若いgoalをとり、それに含まれる変数がどのgoalに含まれるかを探し、これを記録しておく。記録されたgoalに含まれる全ての変数に対する探索が終了すれば、記録されたgoalを配列添字の順に整列し、お互いをポインタで結び１つのgoal列ができる。次のgoal列を作るには、まだ使われていない最も若い添字のgoalから同様の操作を行なう。

このような操作によってできたgoal列をqueueに格納する。queueingにはbounded queue方式を採用しており、goal列はそれを作る元となったgoal列からカウンタを引き継いでいる。bound値としては100を用いており、カウンタの値がこれ以下ならgoal列をqueueのtopに付け加え、大きければカウンタを初期化してqueueのtailに付け加える。

・継続goalの処理

構造引数同士のユニフィケーションが起きた場合には、そのユニフィケーションの進行は次のサイクルに持ち越される。まず、ユニフィケーションの完了していない部分構造を手繰ってユニフィケーション用配列に先頭から格納する。さらにgoalに関する各種の情報を保持する配列を詰め、あらためてユニフィケーション配列の要素との対応がとれるようにする。

・ＯＲテーブル処理

１サイクル中にＯＲテーブルへの書き込みが起こった場合に、ＯＲテーブルの状態を更新する。状態フラグがsuccesならＯＲ関係にある節のどれかが成功したことを示す。プロセスカウンタが０なら親のＡＮＤテーブルのプロセスカウンタから１減じる。また、状態フラグがsuccessでない場合には、親のＡＮＤテーブルの状態フラグをfailにする。ＡＮＤテーブルの書き込みでさらにＡＮＤテーブルの状態が変われば、その親への書き込みを行なう。こうして状態が変わらなくなるまでテーブルへの書き込みを繰り返す。

最終的に初期goalの属するＯＲテーブルのプロセスカウンタが０となって実行が終了し、その時の状態フラグが実行の成否を表わす。

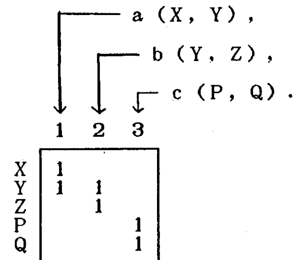


図５－３：ＲＡＰ化用配列の構成

ＯＲ並列インタプリタでは、テーブル処理とＲＡＰ処理の部分が不要となる。goal列の生成は、節のbodyとgoal列とをポインタでつなぎ、変数チェックを行なっただけでqueueに格納する。

６．マルチタスク

現在、早稲田大学情報科学研究教育センターに設置されている計算機は、IBM-3090-200型で２つのＣＰＵを持ち、両方にベクトル演算機構が付加されている。3090にはＭＴＦと呼ばれるマルチタスキング・サポートソフトウェアがある。これは、サブルーチンレベルのタスクの並列処理を支援するもので並列タスクの起動と、その終了の同期を行なう。これを用いて今回の処理系のマルチタスク処理を考える。

マルチタスク処理を考える場合、２つの処理系をそれぞれ別々のＣＰＵ上で動作させ、協調的に問題を解く方法と、１つの処理系の中で並列実行できる部分を探しだし、それらを並列に実行する方法とが考えられる。ＭＴＦの提供するマルチタスク処理は、並列に動作する２つのタスクは同一の変数にアクセスすることができないので、前者のような処理系を書くことは困難である。このため、今回は後者の立場で簡単なプログラムの変更を試みる。具体的には、（ＯＲ並列＋ＲＡＰ）インタプリタにおいてＲＡＰ処理部分とground instance識別部分の並列処理を行なう。

７．性能評価

今回作成した処理系の記述言語は、パーザ部分がProlog、実行系がVS-FORTRANVer.2.1.1と一部アセンブラである。VS-FORTRANは、自動ベクトル化コンパイラであるが、ベクトル化の性能はあまり良くない。これを補うため、アセンブラによる記述を行なっているが、まだベクトル処理できる部分が残っている。

今回の処理系上で小さな例題を走らせたところ、表６－１のような結果が得られた。ＯＲ並列インタプリタと（ＯＲ並列＋ＲＡＰ）インタプリタの両方について、

・ＯＲ並列の方が（ＯＲ並列＋ＲＡＰ）より約４倍ほど速い。

・コピー部分は大きいが予想したほどではない。

	ＯＲ並列	ＯＲ並列＋ＲＡＰ
ＳＣＡＬＡＲ	１４２ｍｓ	５６０ｍｓ
ＶＥＣＴＯＲ	１１２ｍｓ	４５２ｍｓ
ＭＴＦ (with VECTOR)	―	５０１ｍｓ
突き合わせのベクトル長	１１２．９	１１７．９
速度向上	１．２７	１．２４($^{MTF}{}_{1.12}$)

表６－１：例題ｃｏｌｏｒの実行結果

・逆に引数の整列、変数の束縛、変数のチェック部分が高負荷となっている。

などのことに気付く。ＯＲ並列インタプリタの方が速いのは、並列性の混在によるテーブル処理の必要性と実行時ＲＡＰによる高負荷を主因として、ＲＡＰによって増加したgoal列が、逐次的な整列部分に影響したからと考えられる。ＲＡＰはコンパイラ化によってかなり負荷を低減できると考えられる。整列部分では多種のデータを配列に詰め込む作業をしており、この数を減らす必要がある。変数の束縛では、ハッシュ・テーブル操作に逐次的な部分が多く負荷が大きくなっている。キーの衝突回避をベクトルレジスタ上で行なえば性能はいくらか改善されると考えられる。変数チェックではループからの飛び出しが多く、これらがベクトル処理されていない。これをアセンブラで記述することでベクトル処理できれば、やはり負荷は低減される。コピーが必ずしも大きくないのは、問題の性質によるところが大きい。コピー部分の改善にはコピー向きのデータ構造を再考察する必要がある。

ベクトル化の点から見ると、

・スカラー部分が多くベクトル化率が低い。

・突き合わせに対してはベクトル処理は有効に働いている。

・引数の整列部分において各種データの詰め込み作業は一部ベクトル化されており、突き合わせと同程度の平均ベクトル長が得られるが、この部分だけの速度向上を見ても５０％程度である。

突き合わせ部分のベクトル長を大きくとることができ、ユニフィケーションのベクトル処理の可能性が

確認できたと考えられる。しかし、他の部分では平均ベクトル長は余り大きくはない（突き合わせ部分の１／２から１／５程度）。このためベクトル化されないコードが実行される部分が多く、速度向上は３０％弱にとどまっている。また、引数整列部分のベクトル化は、多重にネストしたリストベクトルを用いており、インターリーブのない仮想記憶装置の影響が出たものと考えられる。

マルチタスク処理の対象としてground instanceの識別部分とＲＡＰ部分を選んだのは、まとまりのあるコードで並列に動かせるものがその程度しかなったからである。他にいくつか小さなプログラム部分を並列動作させせることを試みたが、いずれの場合もベクトル処理を併用してもスカラー処理より遅くなってしまった。マルチタスク処理は非常に大きなオーバーヘッドを生じ、小さなプログラムをマルチタスク処理しても意味がない。また、データの共有を許さないため、そのようなプログラム部分を見つけるのが困難である。データの共有を解消するには、各所でコピーを作れば良いが、コピー作成部分は並列に実行できるとは限らず、現在の処理系では、プログラムの単純な変更でマルチタスク処理が処理系の速度向上に寄与するとは言えない。

８．おわりに

ユニフィケーションのベクトル処理手法を提案し、それに基づく２つの並列論理型言語インタプリタ、ＯＲ並列インタプリタと（ＯＲ並列＋ＲＡＰ）インタプリタについて述べ、さらにそのマルチタスク処理についても触れた。思うような速度向上を得られなかったが、ユニフィケーションのベクトル処理の可能性は確認できた。今後は、処理系の速度向上を計り、ＲＡＰ部分のベクトル化、さらにはコンパイラ化を実現したい。また、実用上重要な組み込み述語の実装方法も考えなくてはならない。ユーザーインターフェースも重要な問題である。

９．謝辞

貴重な御助言をいただいたＮＴＴ梅村恭司氏、熱心に御討論いただいた東大Martin Nilsson氏に感謝致します。また、プログラミングを支援していただいた情報科学研究教育センターの皆さんに感謝致します。

１０．参考文献

[1]石浦、安浦、矢島：ベクトル計算機による論理シミュレーションの性能評価、信学技報CAS86-82、pp.25～32、1986.

[2]加賀谷、高木、矢島：ベクトル計算機向き論理関数の素項生成法について、第３４回情報処理学会全国大会、3N-1、pp.27～28、1987.

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

[4]金田　泰：ベクトル計算機による論理型言語プログラムの高速実行をめざしてー各種ＯＲベクトル実行方式の実現と性能ー、情報処理学会プログラミング研究会資料、87-PL-12-2、1987.

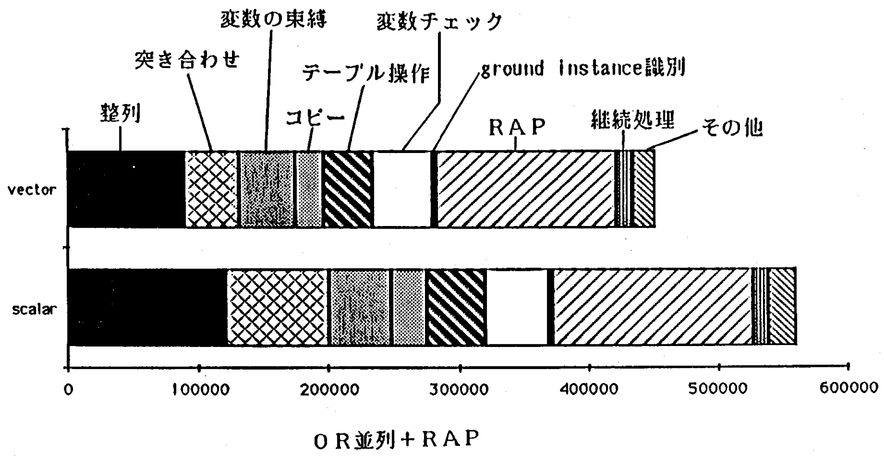
[5]Nilsson,M.:-FLENG Prolog-The Language which turns Supercomputers into Parallel Prolog Machines、Logic Programming Conference'86、pp.209～216,1986.

[6]DeGroot,D.：Restricted And-Parallelism、International Conference on Fifth Generation Computer Systems 1984、pp.471～478、1984.

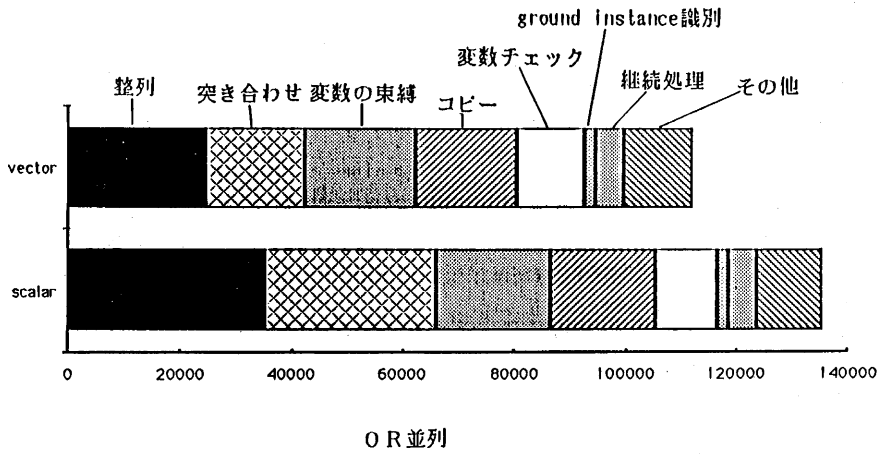
[7]辰口、村岡：ベクトル計算機上の並列論理型言語処理系、情報処理学会第３５回全国大会、5Q-1、pp.753～754、1987.

```
color([A,B,C,D,E]):-
    next(A,B),
    next(C,D),
    next(A,C),
    next(A,D),
    next(B,C),
    next(B,E),
    next(C,E),
    next(D,E).
next(green,yellow).
next(green,red).
next(green,blue).
next(yellow,green).
next(yellow,red).
next(yellow,blue).
next(red,green).
next(red,yellow).
next(red,blue).
next(blue,green).
next(blue,yellow).
next(blue,red).
```

例題ｃｏｌｏｒ



ＯＲ並列＋ＲＡＰ



ＯＲ並列

図６－１：例題ｃｏｌｏｒにおける各部の処理時間