Vol. 26 No. 3 情報処理学会論文誌 May 1985

グループ化された ground assertion における Prolog の AND 並列パターンマッチング†

中川 裕志††

人工知能向き言語として注目を集めている Prolog に関して，その処理能力の向上を目指した並列処理の研究がさかんである．従来の並列 Prolog は，事実を記述した ground assertion と推論規則を記述したルールを同様に実行時に解釈し推論するものである．しかし，データベースへの知的アクセスへの応用においては，データベースの内容である ground assertion が頻繁に変化する．一方，質問は定型的なものが多く，むしろ同一の質問を異なった ground assertion に適用する場合もある．このような場合は，質問をあらかじめ推論規則により展開し，ground assertion に対応するリテラル列を求めておく方法が有効と考えられる．この方法では，質問から得られたリテラル列と ground assertion 集合間でのパターンマッチングは見通しのよいものであり，並列化が容易である．本論文では，Prolog プログラムを ground assertion に対応するリテラル列へ展開する方法について，再帰と並列処理の関係，カットオペレータの意味論的な扱いについて述べる．次に，リテラル列と ground assertion 間での並列パターンマッチング方式として，並列処理の初期の段階で高い並列度が得られる AND 並列方式を提案する．以上の検討により，たとえばデータベースへの知的アクセスにおけるユーザインタフェースとして Prolog を用いた場合の並列アクセス方式について有効な枠組を提案している．

1. はじめに

Prolog 言語は一階述語論理という明確な数学的基礎をもっていること，宣言的プログラミングが可能なこと，などから知識ベースシステム，自然言語処理，データベースアクセスにおける知的インタフェースなどの応用分野をもつ．ただし時間，空間のいずれにおいても効率の低さは問題であり，これを解決するために並列処理の研究がさかんである[1)~6)]．

Prolog の並列処理には，AND 並列，OR 並列，ストリーム並列などがある．これらはプログラムに対応する探索木上をどのような戦略で探索するかに由来している．一方，プログラムは事実を記述する ground assertion と推論規則を記述するルールから構成されている．さらにルールは head と body から成る．このような構造に着目した並列処理の研究もある[3),4)]．ただしいずれにしてもルールを用いた推論は実行時に行われる．このことはプログラム言語の並列処理という観点からは望ましいものである．ところで Prolog プログラムに対応する AND/OR 探索木の葉は ground assertion に対応しており，葉の直上のノードは OR ノードである．その上方の部分はルールを展開した部分である．そこでまずルールの部分のみを用いて探索木を作ってしまい ground assertion とのパターンマッチングは探索木の葉の直上ノードで示される ground assertion の集合上で行う方法が提案されている[7)]．この方法はルールも含めたモジュール性の良さという Prolog 言語の特長を活かしていないこと，探索木作成に手間がかかることから，通常のプログラミングやプログラム実行においては優れた方法とはいえない．しかし，(i)ルール部の変更がなく，ground assertion のみが変化する場合，(ii) ground assertion が膨大な量である場合，にはむしろ好ましい方法と考えられる．たとえば update の頻繁な関係データベースへの定型的質問などにおいては柔軟かつ効率よいアクセス法を提供するであろう．後で述べるようにルール部分を用いてあらかじめ探索木を作ってしまう方法では，ground assertion とのパターンマッチングにおける変数の binding は，一つの変数については1回しかせずにすむ．このため，パターンマッチングは簡単で見通しのよいものとなり，並列化方式も理解しやすいものとなる．このような考察に基づき，本論文では，2章で Prolog プログラムのルール部を展開し，質問をプログラムの探索木における ground assertion の直上リテラルの列（以下これを CPGA と呼ぶ）に変換する方法について述べる．3章で，プログラムが再帰を含む場合の展開と並列処理について，4章でカットオペレータの扱いについて述べる．5章で CPGA と ground assertion の集合との間での並列パターンマッチング，とくに AND 並

† AND Parallel Pattern-Matching for Grouped Ground Assertions by HIROSHI NAKAGAWA (Department of Computer Engineering, Faculty of Engineering, Yokohama National University).

†† 横浜国立大学工学部情報工学科

列のアルゴリズムについて述べ，6章で本論文で提案した方法に対する簡単なシミュレータについて述べる．なお，本論文で提案する並列処理の枠組みは文献7)を土台にしているが，再帰を含む場合の展開法とこれに基づくシミュレーションおよびカットの意味論については，その後の検討によるものである．

2. 探索木と並列処理

本章では，本論文で提案する方法が，Prologプログラムに対応するAND/OR探索木のどの部分を抽出して並列処理するかについて述べる．まず，次式に示すプログラムについて考える．

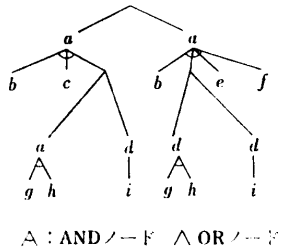
$$\begin{aligned} &a(*X,*Y)\leftarrow b(*X*Z),c(*Z,*Y),d(*Y).\\ &a(*X,*Y)\leftarrow b(*X,*Z),d(*Z),e(*X),\\ &\qquad\qquad f(*Z,*Y).\\ &d(*P)\leftarrow g(*P,*Q),h(*Q).\\ &d(*P)\leftarrow i(*P). \end{aligned} \tag{1}$$

ただし，$*X$, $*Y$ などの＊の付く文字は，変数を表すとする．また，リテラル b,c,e,f,g,h,i に関しては，これ以上ルールで展開させず，おのおのに対して次のようなground assertion（以下GAと略記する）が定義されているとする．

$$\begin{aligned} &b(x_1,y_1).\\ &b(x_2,y_2).\\ &\vdots \end{aligned} \tag{2}$$

x_i, y_i は定数である．

以下 $b(*X,*Y)$ のようにGAに直接対応するリテラルをparent literal of GA：PGAと呼ぶ．探索木との対応で考えれば，PGAは葉の直上ノードである．(1)に対応する探索木は**図1**のようになる．なおGAの部分は省略してあるが，たとえば最左のノード b には，(2)で示される各GAがORの関係で子供としてぶらさがっている（$c,g,h,\cdots$についても同様）．この木の探索法にはdepth-first, breadth-first等が知られている．通常のProlog処理系ではdepth-first である．従来の並列処理では探索木の展開はプログラム実行時に動的に行われていた．ここで提案する方法では，探索木への展開をあらかじめ行っておく．そして探索木の適当なPGAの集合をとり出し，これに対して並列パターンマッチングを行う．このためにはまず並列パターンマッチングの単位となるPGA集合を決める必要がある．



∧：ANDノード ∨：ORノード

図1 (1)の探索木
Fig. 1 The search tree of (1).

(CPGA)

言語としてのPrologのsemanticsを保存することはユーザインタフェース上意味あることである．そこで現在のPrologのdepth-first-searchを考えてみると，これは探索木のORノードにおいてつねに左優先の選択を行うことに対応する．この結果得られるPGAの集合は，図1の場合 $\{b,c,g,h\}$ である．この集合から解が得られないときはバックトラックがおこり，いちばん深いORノードで左から2番目の葉の直上ノードを選ぶ．つまり，図1では $\{b,c,i\}$．これらの集合の各要素はANDで結合されていることに加えて，上記の左優先の選択による順序が与えられている．この集合を以下ではその性質に由来してconjunction of PGA(CPGAと略記する）と呼ぶ．図1におけるCPGAは，最終的goalを a とすれば次式のような順に並ぶものである．

$$\begin{aligned} &a\leftarrow b,c,g,h.\\ &a\leftarrow b,c,i.\\ &a\leftarrow b,g,h,e,f.\\ &a\leftarrow b,i,e,f. \end{aligned} \quad \text{（変数は省略）} \tag{3}$$

(変数の名前換え)

探索木への展開でもう一つ問題になるのはリテラル間での共有変数の名前の統一化である．これは通常のPrologにおけるunificationに相当するものである．ここでは，親ゴールで用いられている変数名を展開の際に子へ引き継ぐ．たとえば前出のプログラム(1)では，親ゴール a の変数 $*X$, $*Y$ などが子の a へ引き継がれ，展開結果得られるCPGAは(4)のようになる．

$$\begin{aligned} &a(*X,*Y)\leftarrow b(*X,*Z),c(*Z,*Y),\\ &\qquad\qquad g(*Y,*Q),h(*Q).\\ &a(*X,*Y)\leftarrow b(*X,*Z),c(*Z,*Y),i(*Y).\\ &a(*X,*Y)\leftarrow b(*X,*Z),g(*Z,*Q)h(*Q),\\ &\qquad\qquad e(*X)f(*Z,*Y).\\ &a(*Y,*Y)\leftarrow b(*X,*Z),i(*Z),e(*Z),\\ &\qquad\qquad f(*Z,*Y). \end{aligned} \tag{4}$$

再帰のない場合はこの例からも明らかなように，たんなる変数名の読換えのみでよい．

(並列処理)

以上の結果得られた CPGA に対して，従来の Prolog の semantics を保存する並列処理は次のようになる．すなわち，先頭の（最左の）CPGA をとり出し，CPGA を構成する各リテラル間で並列パターンマッチングを行う．この CPGA から解が求まらなければ，次の CPGA をとり出して同様に並列パターンマッチングする．これを解が見つかるまでくり返す．CPGA における並列パターンマッチングについては5章で述べる．

3. 再帰の扱い

CPGA が以下のように与えられる再帰的プログラムについて考えてみる．

$$a(*X,*Y)\leftarrow \underbrace{b(*X),c(*Y)}_{①}.$$

$$a(*X,*Y)\leftarrow \underbrace{d(*X),e(*X,*Z)}_{②},\ a(*Z,*W),\underbrace{g(*W,*Y),h(*Y)}_{③}. \qquad (5)$$

この場合は，まず①の CPGA で並列パターンマッチングし，fail した場合は第2の CPGA を並列パターンマッチングする．その際，まず②の部分を並列パターンマッチングし，得られた解（$*X,*Z$ に unify された instance）を $a(*Z,*W)$ に渡す．つまり，a が再帰的に呼び出される．再帰的に呼ばれた a で，①の CPGA のみから解が求まればこれを呼び出し側へ返し，なければ第2の CPGA をパターンマッチングし，再び再帰がかかる．第2の CPGA では a から戻ってきた解を③の部分に与えて並列パターンマッチングする．以上の操作においては，通常の Prolog の処理と同じような処理を行わなければならない．また変数の値の環境の保存も必要である．以上説明した方法では，再帰を含む場合は処理が複雑なわりには並列度が上がらない．

ここでは，上記の問題点の解決策として(5)のような再帰を含む CPGA をさらに展開する方法を考える．つまり，第2の CPGA のリテラル a を展開すれば(6)のような CPGA が得られる（変数は省略）．

$$a\leftarrow b,c.$$
$$a\leftarrow d,e,b,c,g,h.$$
$$a\leftarrow d,e,d,e,a,g,h,g,h. \qquad (6)$$

もちろん(6)においては，一つの CPGA 内に複数回出現するリテラル (d,e,g,h) の変数名は，すべて異なるように名前変えする．このように展開してしまうと，並列パターンマッチングの単位は，第1 CPGA の b,c，第2 CPGA の d,e,b,c,g,h，第3 CPGA の d,e,d,e と g,h,g,h となる．(5)の CPGA の場合，再帰する a を n 回展開すると，並列サーチの単位としては，(d,e の n 回くり返し)，b,c，(g,h の n 回くり返し）が得られ，n を大きくすると高い並列度が得られることが予想される．展開する回数 n は，並列処理を実行するハードウェアを過不足なく使えるように選べばよい．あらかじめ展開してある CPGA のみで解が求まらない場合は，$d,e,\cdots,d,e$ から得られた解によって a を再帰的に呼び出すか，あるいはさらに a を展開して上記の処理を行うことになる．

4. カットオペレータの扱い

以上述べてきた枠組だけではカットは実現できない．並列 Prolog にカットは不要という意見もあるが，現在の Prolog の semantics の保存を目的としてカットを組み込みたければ，次のようにすればよい．次のプログラムを考えてみる．

$$a\leftarrow b,c.$$
$$a\leftarrow d.$$
$$b\leftarrow e,!,f.$$
$$b\leftarrow g. \quad (!はカットシンボル) \qquad (7)$$

これを展開した CPGA は次のようになる．

$$a\leftarrow e,!,f,c.$$
$$a\leftarrow g,c.$$
$$a\leftarrow d.$$

ここでカットオペレータの semantics を考えれば，第1 CPGA の f,c が失敗すれば $a\leftarrow g,c.$ をとばして $a\leftarrow d.$ をパターンマッチングすることになる．これを実現するには，カットを含む CPGA のパターンマッチングを次のようにすればよい．

（i） カットの前後は別々に並列パターンマッチングする．

（ii） カットの前で失敗した場合は元のプログラムの semantics に従って，次にパターンマッチングすべき CPGA へジャンプする．なお，ジャンプ先の CPGA は元のプログラムの CPGA への展開時に決定できる．

換言すればこの方法は，カットの operational semantics をあらかじめ抽出しておくことにほかならない．

5. 並列パターンマッチング

元のプログラムを展開して得られたCPGAとground assertionとの間でパターンマッチングを行えば解を求めることができる．この操作は直列に行ってもよい．しかしデータベースアクセスに見られるように大規模なground assertionの集合がある場合，効率化のためには並列パターンマッチングを行うことが必須である．この章では，ground assertionの集合とCPGAとの間での並列パターンマッチングの方法について述べる．

まず，ground assertion全体を，同一リテテル名をもつものごとにグループ化する．これを以下assertion group（AGと略記する）と呼ぶ．ここでは，CPGAの各リテラルに対応するAGから同時に部分解を出力し，これらがAG間を移動する間に淘汰されて，正しい解のみが生き残るモデルを考える．ここでいう部分解とは，CPGAに現れる全変数と，これにbindされたinstanceの対からなるリストである．たとえば，((＊A1)(＊B Scott)(＊C Eliza))のように表される．

一つのCPGAにおける並列パターンマッチングの方法としては，たとえば部分解が一方向に流れ，AGを通過するたびに変数がinstantiateされていく方法が考えられる．この場合は，部分解がCPGA中の全AGを通過して，全変数が矛盾なくinstantiateされた場合にのみ完全な解が求まる．この方法はストリーム並列に対応している．一方，各AGから独立に部分解を発生し，AG間を動き回りながら淘汰されて完全解のみが生き残る方法も考えられる．これは，AND並列に対応すると考えられるので，以下AND並列と呼ぶことにする．ストリーム並列においては，左側のAGから出発した部分解が最右のAGに到着し，パイプライン的処理になるまでは，並列度が上がらない．一方，AND並列においては，各AGから部分解が発生するため，並列処理の初期段階から高い並列度が得られることが期待される．したがって，最初の解が早く見つかる可能性が高い．一方，各AGが無制限に部分解を発生，あるいはパターンマッチングすると解の重複がおこり，効率低下等の問題があるので，何らかの制限を加える必要がある．以下に，CPGAでのAND並列パターンマッチングにおける解の重複回避アルゴリズムの一例を提案する．

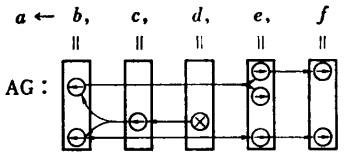


図2 部分解の流れ
Fig. 2 A flow of partial solution.

（部分解の流れ）

CPGAの各リテラルに対応するAGから発生された部分解は，**図2**に示すように，初めは左方向にAGを一つずつ進み，最左のAGまで行きつくと，次は自分の発生したAGの一つ右のAGへ戻り，以後右方向にAGを一つずつ進む．したがって最右のAGまで行きついた部分解は全AGを通過している．部分解は，AG中の一つのground assertion（GA）から発生される．図2では，AGを□で，GAを○で示す．部分解はあるAGに到着すると，現在の変数のbind状況をかんがみてパターンマッチングできるGAをさがす．パターンマッチングに失敗すると，その部分解は消滅する．パターンマッチングに成功すると，部分解には新たに変数にbindされたinstanceの情報が付加され，次のAGへ進む．ある部分解に対してパターンマッチングに成功するinstanceがn個（$n>1$）ある場合は，部分解は各instance対応に作られる．すなわちn個の部分解に分裂し，次のAGへ進む．図2では⊗は部分解の発生源であるGA，⊖は左方向へ流れる部分解とのパターンマッチングに成功した状態のGA，⊖は右方向に流れる部分解とのパターンマッチングに成功した状態のGAを表す．また→は部分解の径路を示す．

（重複回避アルゴリズム）

解の重複は，**図3**の＊1，＊2，＊3に示すように，1個以上の部分解が同一のGAの対の上を流れる場合におこる．＊1においては，Cの⊗で発生し，bの⊖，dの⊖，eの⊖と実線上を流れる部分解がまずあったとする．その後にdの⊖で発生した部分解が上記と同じように破線に沿って流れると重複解になって

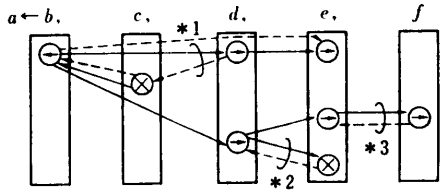


図3 重複解の発生
Fig. 3 The cases of duplicate solutions.

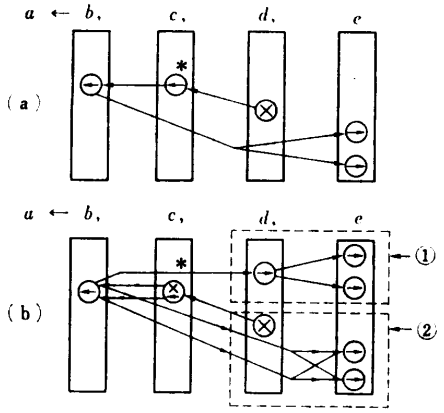


図 4 別のタイプの重複解発生
Fig. 4 Another case of duplicate solution.

		GAの状態			
		⊗	⊖	⊖	○
部分解の発生		×	○	×	○
パターンマッチングの可否	右に流れる	○	○	○	○
	左に流れる	×	○	×	○
	⊖から発生して右に流れる	×	○	○	○
	⊖から発生して左に流れる	×	○	×	○

表 1 部分解の発生，パターンマッチングの制限
Table 1 The condition for generation and pattern matching of partial solution.

しまう．＊2においては，まず d の⊖，e の⊗と実線上を流れる部分解があったとする．この解が e の⊗へ到着する前に，e が部分解を発生し，d の⊖で流れると，これが重複解となる可能性がある．＊3は，すでに e の⊖から f の⊖で流れた部分解があり，その後に f の⊖が部分解を発生する場合であり，やはり後者は重複解となる可能性がある．これらを防ぐために部分解を発生したGA，部分解とパターンマッチングに成功したGAにはそれ以後，部分解の発生，他の部分解とパターンマッチングに制限を課す．

まず部分解の発生については，すでに部分解を発生したGA（すなわち⊗）は当然再び部分解を発生できない．また図3の＊3のような重複を防ぐために，右方向へ流れる部分解とすでにパターンマッチングしたGA（すなわち⊖）も部分解の発生を禁止される．

次にパターンマッチングの制限について述べる．図3の＊1，＊2のような場合を防ぐために，⊗，⊖のGAは左方向へ流れる部分解とのパターンマッチングは禁止する．ところで，図4 a のように，d で発生し，c の⊖*，b, e と流れる解があったとする．しかしながら，後に c の⊖* が部分解を発生すると，図4 b の①で示す破線内のように，GA d, e で別の解が求まる可能性がある．したがって，⊖ のGAからは部分解を発生しなくてはならない．ところが c の⊖* が部分解を発生すると，図4 b の②で示す破線内のように先に図4 a で求まっていた解との間に重複が発生する．そこでこの種の重複解を防ぐために，⊖から発生した部分解は，右方向へ流れる際には⊗のGAとのパターンマッチングを禁止する．こうすれば，図4 bにおいて，c の⊖* から発生した部分解は，d の⊗にぶつかった時点で消滅し，②の重複はおこらない．以上の制限を表1にまとめた．もちろん，あるGAが×，←，→のうち2個以上の状態をもってしまうこともおこりうる．この場合は，どれか一つの状態によって部分解発生やパターンマッチングが禁止されれば，そのGAでの部分解発生やパターンマッチングはまったく禁止される．

部分解の発生とパターンマッチングに制限がなければ，重複解も含めてすべての解が求まる．この方法は，上記の制限により重複解の発生のみを制限している方法であるといえる．また，$a(*X) \leftarrow b(*X, *Y)$, $c(*Y, *Z), d(*Z, *X)$ のような場合は，$*X$ にinstance（たとえば ab）が bind されると，発生される部分解は，$((*X\ ab)(*Y\ ?)(*Z\ ?))$（? は任意の値）に制限されることになる．

6. 簡易シミュレータ

本論文で述べてきた並列処理方法の確認をするために，汎用計算機上に，簡易なシミュレータを試作した．シミュレータは二つの部分からなる．第1の部分では2章で述べた方法で Prolog プログラムのルールを展開して CPGA を求める．第2の部分は，一つの CPGA と各 assertion group : AG との間での5章で述べた AND 並列パターンマッチングをシミュレートする．図5に第2の部分の流れ図を示す．

（動作）

（1） 可能ならば各 AG から部分解を一つずつ発生する．

（2） CPGA と5章で述べた制御方法により部分解を次の AG へ送る．表1の制約条件下で，各 AG で到着した部分解のパターンマッチングを行う．この結果の部分解を次の AG へ送る．

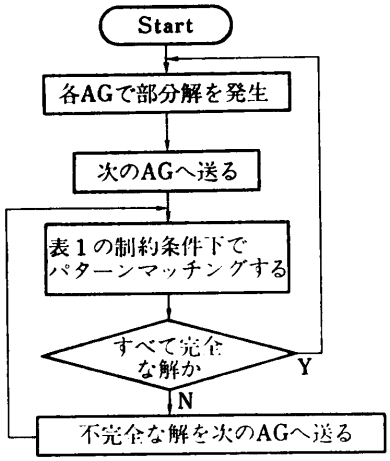


図5 AND並列パターンマッチングシミュレータの概略
Fig. 5 The configuration of AND parallel pattern matching simulator.

(3) (2)の操作を，すべての部分解が消滅するか，完全な解になるまでくり返す．

(4) (1)へ戻る．

ここで，(2)，(3)のくり返しで各AGから発生された部分解が全部処理を終了するまでを，フェーズと呼ぶ．このような簡易シミュレータにより，提案したCPGAへの展開，フェーズ単位でのAND並列動作が確認できた．次に，簡易シミュレータにおける並列度について考える．ここで，$N_j(i)$は，第iフェーズにおいて，第jAGで試みた(成功，失敗にかかわらず)パターンマッチングの回数とする．また，第iフェーズの実行時間をt_iとする．ただし，パターンマッチングの試行1回を1単位時間とする．1個のCPGAの処理にかかる時間tは，$t=\sum_i t_i$．並列度の平均値pは

$$p=(\sum_i \sum_j N_j(i))/t \qquad (8)$$

で与えられる．また逐次処理においてかかる時間は，およそ$p \cdot t$となる．

以下に示すような簡単なプログラムによる結果について述べる．

$$a(*A,*F)\leftarrow b(*A,*B),c(*B,*C),d(*C,*D),e(*D,*E),f(*E,*F). \qquad (9)$$

GA名	b	c	d	e	f
GA内のassertion数	7	4	4	4	4

ただし，解の数は6個，$t=129$，$p=1.775$

$$a(*A,*K)\leftarrow b(*A,*B),c(*B,*C),d(*C,*D),e(*D,*E),f(*E,*F),g(*F,*G),h(*G,*H),i(*H,*I),j(*I,*J),k(*J,*K). \qquad (10)$$

GA名	b	c	d	e	f	g	h	i	j	k
GA内のassertion数	12	12	11	12	9	5	3	4	4	7

ただし，解の数は7個，$t=701$，$p=2.083$

また再帰のある場合については次の例を行った．

$$a(*P,*S)\leftarrow b(*P,*Q),c(*Q,*R),d(*R,*S).$$
$$a(*P,*S)\leftarrow e(*P,*Q),f(*Q,*R),a(*R,*S) \qquad (11)$$

GA名	b	c	d	e	f
GA内のassertion数	4	5	4	3	3

ただし，解の数は6個

(10)については，CPGAとして次のものを実行した．

(変数名は省略)

$a\leftarrow b,c,d.$ $\cdots t=41$，$p=1.537$

$a\leftarrow e,f,b,c,d.$ $\cdots t=39$，$p=2.179$

$a\rightarrow e,f,e,f,b,c,d.$ $\cdots t=40$，$p=2.900$

ただし，このシミュレータは，簡便さのためフェーズ単位で動作しており，あるフェーズにおいて，空いているAGは次の部分解の発生をフェーズ終了まで待つことになる．提案した並列処理方法は，フェーズ単位の動作を強要してはいない．したがって，空きAGはただちに次の部分解を発生できるため，並列度は，上で示したものより高くすることが可能である．ただし，正確な並列度は，並列マシンのアーキテクチャに依存する部分が多いので，ここでは述べない．

7. おわりに

Prologプログラムを推論規則の部分を用いてCPGAへ展開する方法において，再帰の扱いと並列処理の関係，カットオペレータの扱いについて述べた．さらに，CPGAとground assertionとの間でAND並列パターンマッチング方法を提案し，簡易シミュレータを作成した．本論文は，以上のようなProlog並列処理の一つの枠組みを提案するるものであり，インプリメンテーションの詳細については，多く

の検討課題が残されている．たとえば次のようなものが今後の課題となる．

（1） 本提案に沿った並列処理マシンのアーキテクチャの設計

（2） assertion group のパターンマッチングにおける効率化に関する諸問題．たとえば，assertion group の大きさがふぞろいな場合の統合や分割などの問題

（3） CPGA におけるリテラルの順序を変更することによって，assertion group とのパターンマッチングの効率化を向上させる方法の検討，などである．

参 考 文 献

1) 相田，田中，元岡：並列 Prolog 処理システム "paralog" について，情報処理学会論文誌，Vol. 24, No. 6, pp. 830-837 (1983).
2) Conery, J. S. and Kibler D.: Parallel Interpretation of Logic Programming, Proc. of Conf. on Functional Programming Language and Computer Architecture, ACM (Oct. 1981).
3) Eisinger, N., Kasif, S. and Minker, J.: Logic Programming: A Parallel Approach, Proc. of 1'st International Logic Programming Conf. (1982).
4) Taylor, S., Lowry, A., Magulre, G. Q., Jr. and Stolbo, S. J.: Logic Programming Using Parallel Associative Operations, Proc. of 1984 International Symposium on Logic Programming (1984).
5) 田村，松田，金田，前岡：*K*-prolog のインプリメンテイーション，Proc. of Logic Programming Conference '83 (1983).
6) 後藤，相田，丸山，湯原，田中，元岡：高並列度推論エンジン PIE について，Proc. of Logic Programming Conference '83 (1983).
7) 中川：AND Parallel Prolog with Devided Assertion Set, Proc. of 1984 International Symposium on Logic Programming (1984).

(昭和59年 5 月 24 日受付)
(昭和59年11月 15 日採録)