並列 Prolog 処理システム "Paralog" について†

相田 仁†† 田中英彦†† 元岡 達††

Prolog は一階述語論理式をそのままプログラムとして解釈することを基礎としているので，いくつかの優れた素質を有しているが，現在の Prolog は探索順序を固定した逐次処理を行っているため，その素質が十分に活かされていない．そこで，Prolog の並列処理は，その素質を活かすことと処理速度の両面からの利点があると考えられる．Prolog の並列処理の方法としては，AND 並列，OR 並列，引数間並列の三つが考えられるが，AND 並列処理は無矛盾性検査の点で，また，引数間並列処理は処理単位が小さい点で，OR 並列処理に比べ実装がむずかしい．われわれは，まず OR 並列処理をとりあげることにして，既存のマルチマイクロプロセッサの上に，Prolog の並列処理を行うシステム "Paralog" の第1版を実装した．Paralog 第1版では，PM (処理モジュール) 5台の範囲内で，PM 台数にほぼ比例した性能向上が測定され，Prolog の並列処理の有効性が確認された．オーバヘッドの測定結果から，このシステムの方式で CM (制御モジュール) 1台当り PM 50 台程度まで拡張可能と思われる．また，プログラムの実行を追跡することにより，Prolog プログラムの実行には，十分大きな並列性が内在することも確認された．

1. はじめに

知識を計算機の上に表現し，それを用いて知的な情報処理を行うための言語として Prolog[1] が注目を浴びている[2),3)]．Prolog は一階述語論理式をそのままプログラムとして解釈することを基礎としているので，

① 静的な意味解釈によりプログラムを理解することが容易である，

② プログラムの各部分の正当性がプログラム全体としての正当性に結びつく，

などの優れた素質を有しているが，反面，純粋な Prolog をそのままプログラミング言語として用いるには2種の大きな弱点が存在する．

第1の弱点は否定的内容を含んだ知識をとり扱うことができない点で，Prolog が論理式の範囲をホーン節に限定したことに由来している．そこで，その解決には，より広い範囲の論理式を扱えるように論理的な裏づけを強化するか，あるいは論理外の機能を導入しなければならない．

第2の弱点は，プログラム中の式の配列順により実行の効率が大きく影響を受け，場合によって無限ループに陥ることもある点で，これは解の探索順序を記述順に固定したことに由来している．そこで，その解決のためには，各時点において適切な順序を定めて処理を行えばよいが，さらに一歩進めて，プログラムの各部分の処理を並列に行えば，処理速度が向上すると同時に，記述順による影響がまったくなくなり，Prolog 本来の素質が十分発揮できるようになると考えられる．

上記のような観点から，現在われわれは，論理プログラミング言語の並列処理に関する研究を進めているが，その第1段階として，既存のマルチマイクロプロセッサハードウェア上に，Prolog の並列処理を行う処理系 "Paralog" の第1版を実装した．本論文では，まず，論理プログラミング言語の並列処理方式について一般的な考察を行った後，今回実装した処理系の概要を紹介し，その性能測定結果に基づいて評価・検討を行う．

2. 論理プログラミング言語の並列処理方式

論理プログラミング言語で書かれたプログラムは，本来静的な論理式であるから，どの部分から処理しなければならないという制限はなく，並列に処理可能な部分が多数あると考えられる．たとえば純粋な Prolog で書かれたプログラムの場合，次の三つの観点から並列処理を行うことが可能である．

i) あるゴール節に含まれる複数のリテラルを切り離して別々のゴール節とし，それらを並列に処理する．

ii) あるゴール節中の特定のリテラルと単一化可能な定義節が複数ある場合，それらを用いて複数の

† On Parallel Prolog Processing System "Paralog" by HITOSHI AIDA, HIDEHIKO TANAKA and TOHRU MOTO-OKA (Department of Electrical Engineering, Faculty of Engineering, University of Tokyo).

†† 東京大学工学部電気工学科

```
+zero(0).
+zero(*x-*x).
+zero(*x・*y)-zero(*x).
+zero(*x・*y)-zero(*y).
-zero((*a-2)・(*a-3))-zero((*a-3)・(*b-5)).
```

図 1　簡単なプログラム例
Fig. 1　A simple program example.

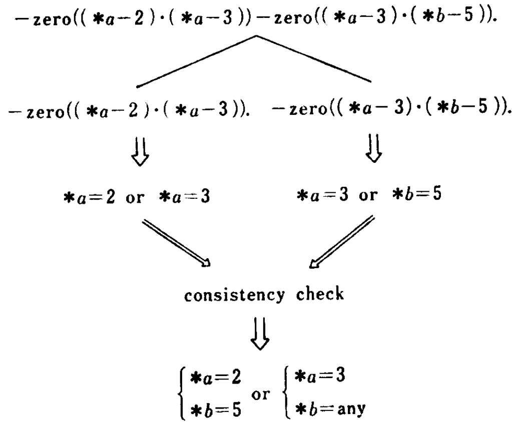

図 2　AND 並列処理
Fig. 2　AND parallel processing.

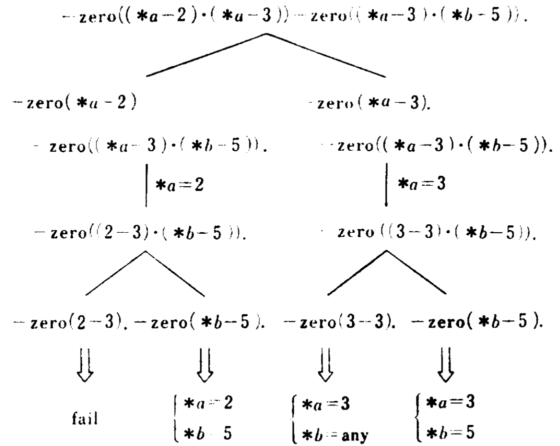

図 3　OR 並列処理
Fig. 3　OR parallel processing.

ゴール節を導出し，並列に処理する．

iii）あるゴール節のリテラルと，特定の定義節頭部との単一化操作において，各引数の単一化を並列に行う．

これらのうち，i）では，並列に処理するサブゴールの結果をすべて満たすものが全体としての解になるので，AND 並列処理と呼ぶ．**図 1** の簡単な数式処理プログラムにおける AND 並列処理の例を**図 2** に示す．AND 並列処理において，分割したリテラルの間で変数が共有されていた場合，各リテラルを並列に処理した後に，それぞれから得られた解のうちでリテラル間の共有変数に関して矛盾のない組合せを選び出す必要がある(無矛盾性検査)．いま，並列処理した二つのリテラルからそれぞれ m, n 通りの解が得られた場合，原則としては $m\times n$ 通りのすべての組合せについて単一化を行ってみる必要があり，その負荷はかなり重い．AND 並列処理を実現し，それにより処理速度を向上させるためには，この無矛盾性検査を要領よく行う方法が開発されなければならない．

これに対し ii）では，並列に処理する各ゴール節から得られる解が全体としての解にもなるので，OR 並列処理と呼ぶ．先と同じ例題における OR 並列処理の様子を**図 3** に示す．OR 並列処理においては，並列処理したゴール節間で，一方が他方の特別な場合であるなど，重複した解が得られることがある．この重複をとりのぞくためには，それぞれから得られた解の単一化を行ってみる必要があり，AND 並列処理における無矛盾性検査と同様な手間を要する．しかし，解の重複は存在してもかまわないことが多く，その場合には各ゴール節の処理を完全に独立して行うことができるので，高い並列性をとり出すことが可能である．また，従来の逐次形 Prolog と同様に，カット機能を実装して，解の重複の防止をプログラマに任せてしまう方法も考えられる．これには，たとえばあるゴール節の処理結果に基づいて他のゴール節を処理しているプロセッサにメッセージを送り，それらの処理を中止させる機能を実現すればよい．このように OR 並列処理は，実現の観点において最も有望な並列処理方式である．

iii）の単一化操作時における各引数の並列処理については，

① 各引数の単一化がすべて成功した場合に限り，全体としての成功とみなされる，

② 引数間に共通な変数がある場合に特別の操作が必要である，

などの点において AND 並列処理と共通の性質を有している．しかし，

③ 各引数の単一化から得られる結果は常に 1 通りに限られる，

④ ゴール側，定義側の引数がともに構造体である場合を除き，並列処理が入れ子になることはない，

などの点においては，AND 並列処理よりも単純である．また，この方式では，並列処理の単位が AND 並列処理や OR 並列処理の場合に比べて小さいので，オーバヘッドの問題から，複数プロセッサに分配するような処理形態には適さないが，ハードウェア単一化器として実現できれば，各引数を逐次処理する場合に比べて数倍の速度向上が期待できる．

3. Paralog 第1版の実装

3.1 実装方針とハードウェア

Paralog 第1版は，論理プログラミング言語の並列処理の研究の第1段階として，

① 並列処理の有効性を確認するとともに，

② 並列処理における問題点を実地に見いだすこと，

に重点を置き，既存のハードウェアを用いた小規模な処理系として作成したものである．

Paralog 第1版の基本的な枠組としては，次のような点があげられる．

③ 言語としてとり扱う論理式は，カット機能等を含まない純粋な Prolog（ホーン節）の範囲とする．

④ 並列処理方式としては，すべての選択肢を（プロセッサ数の範囲内で）並列に処理する OR 並列処理のみを行う．各ゴール節内のリテラルは，当面，通常の Prolog と同様に，左から右の順で処理する．ただし，上記のリテラルの処理順序は，適当な評価ルーチンを組み込むことにより，容易に変更可能である．

⑤ 並列に処理可能な選択肢の数がプロセッサ数を越えた場合には，プログラム記述順の影響を受けにくい，広さ優先の原則に従って順に処理する．

実装に用いたハードウェアは，先にわれわれの研究室において設計した高レベルデータフローマシン用マルチマイクロプロセッサシステム“TOPSTAR-II”である．ハードウェアの詳細については，文献4)を参照されたい．

3.2 基本制御方式

Paralog 第1版における基本的な実行制御方式は，これまでに TOPSTAR-II に実装された，データ駆動型システムの場合と同様とした．すなわち，CM（制御モジュール）は，処理すべきゴール節を蓄えるプールをもち，その管理を行うほか，端末およびフロッピデスクとの入出力を受け持つ．一方，各 PM（処理モジュール）は，定義節のコピーをもち，単一化・導出操作および組込み述語の処理を行う（図4）．

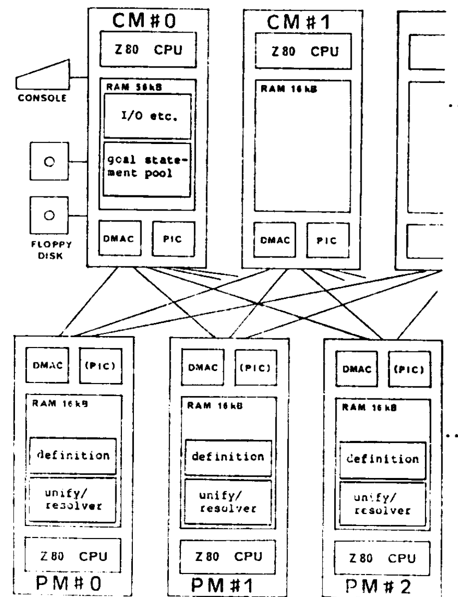

図 4 システム全体像
Fig. 4 System overview.

実行開始後，アイドル状態の PM は，適当な CM に割込みをかける．その CM のゴールプール内に，処理可能なゴール節があれば，CM はそのうちの一つをとり出して PM に転送する．PM は，受けとったゴール節内の適当なリテラルについて，その述語のすべての定義節と単一化を試み，単一化可能なものすべてについて，ゴール節を導出する．これを終えると，PM は再び CM に割込みをかけ，新たに得られたゴール節を CM に転送する．次いで CM から PM に，次に処理すべきゴール節が送られ，上記の過程が繰り返される．

CM のゴールプールに処理可能なゴール節がない場合には，PM は他の CM に割込みをかけるか，多少の時間の待ち合せをしてから，同じ CM に割込みをかけなおす．

今回の Paralog 第1版では，簡単のため，CM については，端末が直接に接続されている CM #0 のみを使用し，PM についても，CM #0 に直接結合している5台のみを用いた．

また，定義節に関しては，入力された時点で，すべての PM に対して同一のコピーを配った．これは TOPSTAR-II のハードウェアが，モジュール間に共有なメモリをもたないためで，モジュール間に共有で PM からは読出し専用の定義節メモリに対し，各 PM

にキャッシュが設けられている状態を模擬している.

3.3 データ構造

Prolog を逐次型計算機上に実装する場合には，構造体共有[5]の方式が，メモリ効率，データのコピーの必要量などの点で優れており，後戻りにも比較的適しているので広く採用されている[6].

しかし，並列処理を行う場合には，並列に処理を行うプロセス間での干渉をさけるため，束縛環境はゴール節ごとにもつ必要があり，コピーまたは deep binding などの方式をとらざるをえないため，構造体共有の利点は少なくなる.

また，TOPSTAR-II においてはモジュール間に共有メモリがなく，メモリ対メモリの DMA 転送によって情報を伝達するので，1回に伝達される情報は，メモリ上で連続した領域にまとまっていることが望ましい.

そこで Paralog 第1版では，構造体共有は用いず，ゴールリテラルと定義節頭部との単一化が成功する都度，単一化により値を束縛された変数を参照しているセルの内容を書き換えながらゴール節をコピーし，端末から入力された時点と同一のデータ構造のゴール節を再構成する方式をとった.

Paralog 第1版におけるゴール節および定義節のデータ構造を図5に示す．これらの構造の単位となるメモリセルは，1セル当り16ビットで，4ビットのタグと12ビットの値として用いた．また，各述語，関数の引数の数は固定とし，別の表に登録しておくことにより，図5の構造における領域の縮小を図った.

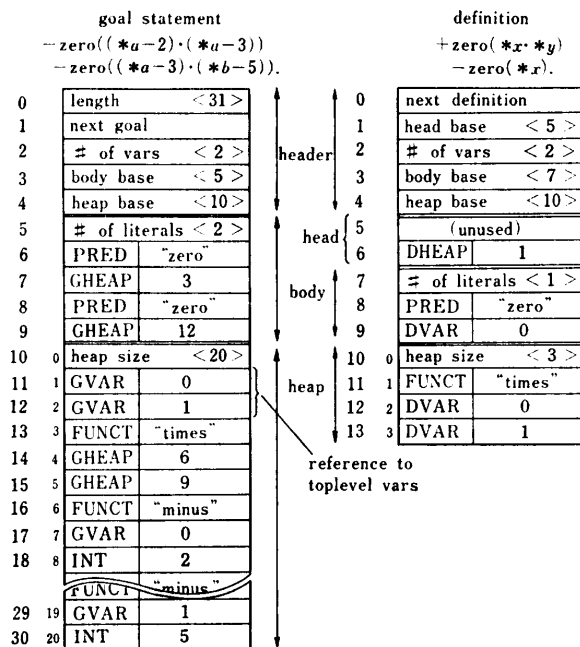

図5 データ構造
Fig. 5 Data structure.

3.4 ゴール節の管理方式

選択肢の実行順序を広さ優先としたため，CM に設けるゴールプールはたんなる FIFO バッファでよい．しかし，リングバッファ構造は CM-PM 間の DMA 通信に適さない点があるので，さらに単純なセルの1次元配列として先頭から使用してゆき，後方の空き領域が不足した時点で PM からの割込みを禁止し，使用中の領域を前につめる方式をとった.

ゴール節が成功に至った場合の解の表示方法としては，表示のための擬述語を設けることはせず，そのゴール節が入力された時点で含んでいた変数の最終的な値を，実行開始からの経過時間とともに表示することにした．このため，あらかじめ図5に示すように，各変数を参照するセルをゴール節の中にうめ込んでおき，成功に至ったときにこのセルの値を読み出している.

OR 並列処理の特徴を活かすため，選択肢のうちで成功に至ったものがあっても，他の選択肢の実行は中断せず，そのまま他の解の探索を継続する．すべての探索を終了すると，実行開始からの経過時間を表示し，入力待ち状態にもどる．実行を途中で中止したい場合には，端末のキーをたたけば CM はただちに入力待ち状態にもどり，各 PM も CM に割込みをかけた時点で実行の中止を知らされてアイドル状態になる.

3.5 使用言語その他

Paralog 第1版の処理系は，割込み時点でのレジスタ退避などごく一部をアセンブラに頼ったほかは，処理内容の明確化，開発期間の短縮などをめざし，C言語により記述した．処理系の大きさは，CM 側約 18 kB，PM 側約 10 kB である．処理速度を評価する際の基準とするため，単一プロセッサ上で動作する同一構造の処理系を，通常の縦型探索のものと，Paralog 第1版と同じく横型探索のものの2通り作成した.

4. 性能測定および評価

4.1 性能測定

Paralog 第1版の性能を測定するため，自然言語処理，論理回路シミュレーション，数式処理などの分野における簡単な応用プログラムを実行させた．このうち，数式簡単化のプログラムの例を付録に掲げる．プ

ログラム中，♯DECLARE で始まる行は，述語/関数がいくつの引数をもつかを宣言するものである．またARITH は整数を扱うための組込み述語で，四つの引数 a, b, c, d がそれぞれ整数であるか変数であるかのパタンに応じて，$a*b+c=d$ が満たされるように各種の演算を行うものである．定義した述語の意味は equiv が式の同値変形を行うもの，simpl が式の簡単化を行うものである．

このプログラムにより

$$-\text{equiv}\ (\ (2\cdot x^1)\cdot \exp(x^2) + x^2(\ (2\cdot x^1)\cdot \exp(x^2)\ ),\ *y).$$

を実行した場合の実行時間を**図 6** に，その逆数すなわち実行速度を**図 7** に示す．図 6，7 において，横軸は稼動させた PM の台数であり，比較基準として，いちばん左側に単一プロセッサ版における実行時間を掲げた．

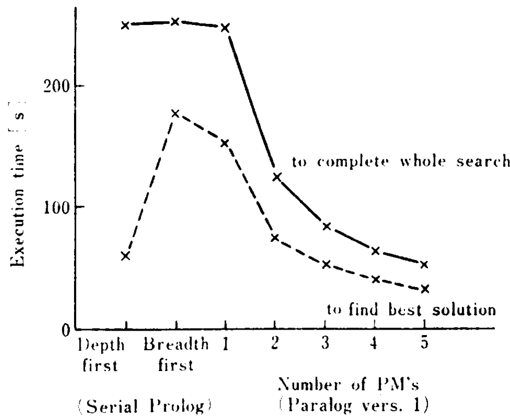

図 6　実行時間
Fig. 6　Execution time.

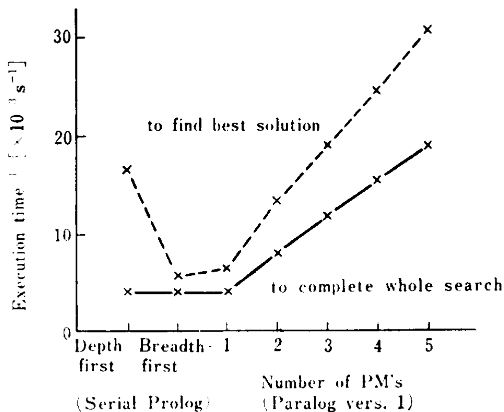

図 7　実行速度
Fig. 7　Execution speed.

このプログラムにおいては，通常の Prolog 処理系のように，定義節は宣言順に，式の中のリテラルは左から順に処理して縦型探索を行った場合に，最初に最も簡単化された解が得られる．これに対し Paralog 第1版では横型探索を行っているので，複数得られる解のうちでどれが最簡形であるかは，すべての探索が終了してからでないとわからない．このような探索方式の違いを考慮するため，図6，7では，すべての探索を終了するまでの時間を実線で，そのうち最簡形を得るまでに要する時間を破線で示した．

図のように，Paralog 第1版の処理速度は PM 台数にほぼ比例して向上し，5台の PM を用いることにより，単一プロセッサの縦型探索で最初の解（＝最簡形）を得るよりも少ない所要時間ですべての探索を終了することが可能であった．

興味深い事実として，最簡形を得るまでの時間に関しては，PM 台数に対して比例以上の速度向上を示す場合のあることが観察された．これは，選択肢のうちでより簡単化の進んだゴール節のほうが処理時間が短くなり，簡単化のあまり進んでいないゴール節の処理を追い越すことによるものらしい．この事実は，たんなる横型探索ではなく，有望そうな選択肢を優先した探索を行い，カット機能と組み合わせることにより，さらに処理速度が向上することを示唆するものである．

4.2　処理系のオーバヘッド

Paralog 第1版においては，前述のように，PM 5 台の範囲に限定して実装を行ったが，今後さらにシステムを拡張した場合の性能を評価するため，PM での処理に要する時間と，CM-PM 間通信および CM での割込み処理に要する時間（処理系のオーバヘッド）の両者を測定した．先の例題の実行時における，これらの時間の分布はそれぞれ**図 8，9** のようになり，平均値はそれぞれ 455 ms および 9.6 ms であった．このように Paralog 第1版の処理方式では，PM における単一化操作等の負荷が重く，並列処理に伴うオーバヘッドの割合は十分小さいことが確認された．上記の平均値から，CM 1台当り約 40〜50 台の PM を接続可能であると考えられる．

オーバヘッド時間のうち，CM-PM 間通信に要する時間は，平均で 0.93 ms(転送量 465 B)，最悪の場合でも 2.56 ms（1,280 B）にすぎず，割込み処理に要する時間に比べ，1桁小さい．処理する問題が大きくなり，ゴール節の大きさが先の例題の場合に比べ 10 倍程度大きくなれば，通信時間と割込み処理時間とが

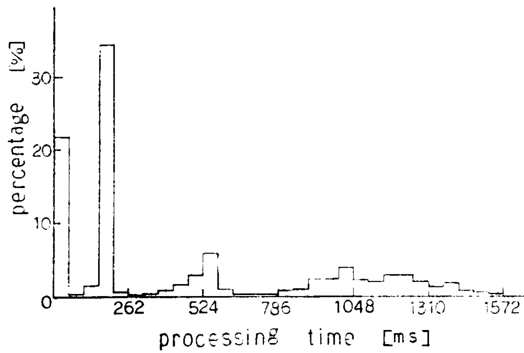

図 8 PM における処理時間の分布
Fig. 8 Distribution of processing time in PM.

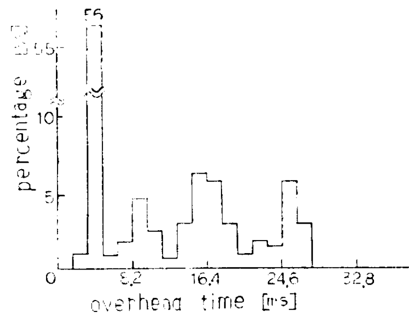

図 9 オーバヘッド時間の分布
Fig. 9 Distribution of overhead time.

同程度になる．その場合でも，PM での処理時間がゴール節の組替えに要する時間等でかなり長くなるため，オーバヘッドの割合としては大きくなることはないと考えられる．さらにそれ以上大きな問題を扱うことは，TOPSTAR-II のアドレス空間の大きさからみて現実的ではない．

また，Paralog と同様な制御方式に基づく，これまでの TOPSTAR-II 上のシステムでは，問題の有する並列性があまり高くない場合に，PM 台数をむやみに増やすと，アイドルな PM が増加し，それらからの無用な割込みにより CM が忙殺されて，かえって性能が低下することがあった．しかし，Paralog 第1版においては，CM のゴールプール内に処理可能なゴール節がない場合に，割込み処理に要する時間は約 1 ms にすぎないため，多数の PM を接続した場合でも問題は生じないと思われる．

5. 並列性の大きさに関する考察

5.1 OR 並列処理の並列性の大きさ

2章で述べたように，論理プログラミング言語の処理には大きな並列性が内在すると考えられ，前章では，その並列性を効率よくとり出しうることを示した．しかし，プログラムのもつ並列性の大きさを一般的に評価した例はほとんど知られていない．ここでは，その一つの目安として，前章で扱った例題について，OR 並列性の大きさを評価する．

図10は，プログラムの実行を追跡することにより得た，例題の探索木である．図の中で○印をつけたノードは成功したノード，そのうち◎印をつけたものは，解として最簡形の得られたノードである．また，図の左側の数字は，木の深さおよびその深さにあるノードの総数である．例題を実行するにあたって逐次的に処理しなければならないゴール節(探索木上のノード)の数を数えたものが**表1**である．表より，全探索については，平均約 19 倍の並列性をとり出すことが可能で，そのとき必要なプロセッサ数は 50 台と評価できる．

この例題のような小さなプログラムの場合でさえ，これだけの並列性があり，一般に論理的プログラミング言語の並列処理に内在する並列性は，十分に大きいと考えることができよう．

5.2 数の爆発の防止

Paralog 第1版において，プログラムに内在する並列性を効率よくとり出しうることを述べたが，大きな問題を解く場合には，逆に，とり出す並列性の大きさを適切に制限しないと，いわゆる数の爆発をひきおこし，管理不能に陥る可能性がある．数の爆発を防止するための方法としては，二つのアプローチが考えられる．

一つは，実行しなくても結果のわかるゴールの実行を行わないようにして，処理の総量を減少させる方法である．これには，①ハッシュなどを用いて，同一のゴールが複数のノードに現れた場合にそれを検出し，処理を1度しか行わないようにする，②逐次 Prolog における知的後戻り[7]と同様に，引数の設定状況に基づいて，失敗に至るゴール節を見いだす，などの方法が考えられる．

もう一つは，各時点において，処理の結果新たに生成されるゴールを最小限におさえて，処理の負荷を平坦化する方法で，OR 並列処理においては，③ゴールプール内のどのゴールから処理するか，および，④ゴール内のどのリテラルに関して導出を行うか，の二つの選択方法が問題となる．③に関しては，たんなる深さ優先や広さ優先ではなく，リテラル数が少なく，失敗する確率の高そうなゴールを優先するのがよいと考えられる．また，④は，探索木の形状を変える機能

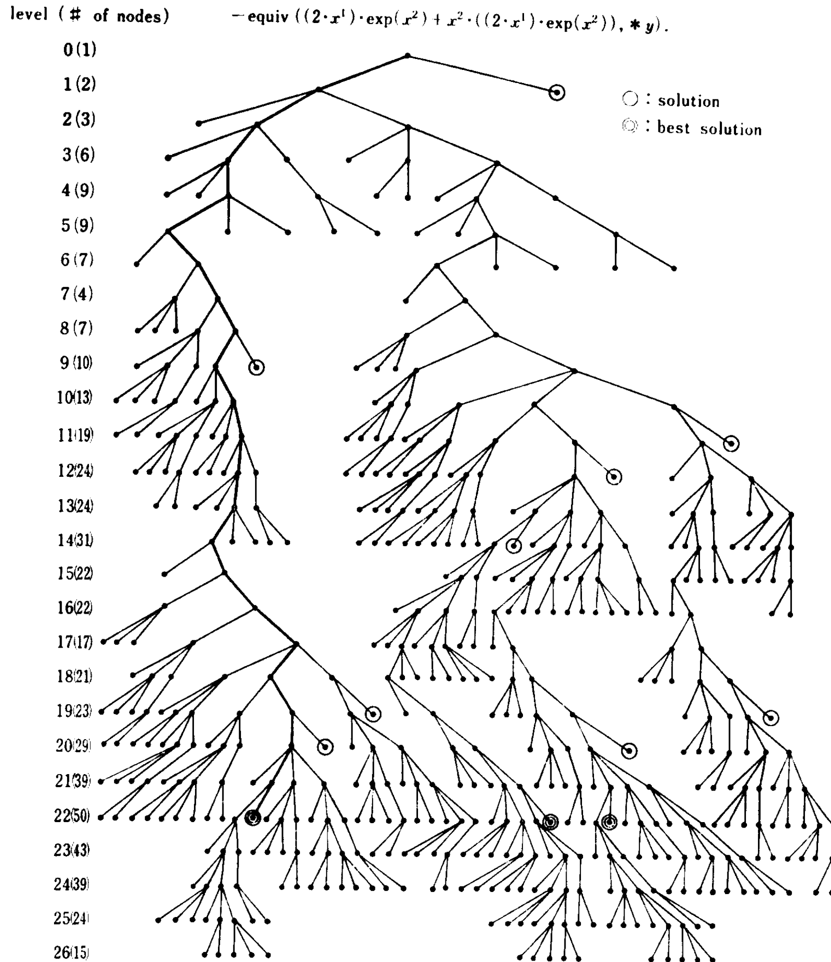

図 10 例題の探索木
Fig. 10 Search tree.

表 1 逐次処理の必要なノード数
Table 1 # of nodes for sequential processing.

To find best solution			Whole search	
Parallel	Serial		Parallel	Serial
	Depth-first	Breadth-first		
[a] 23	[b] 127	[c] 353	[d] 27	[e] 513
Maximum width 39	$\left[\frac{b}{a}\right]$ 5.5	$\left[\frac{c}{a}\right]$ 15.3	Maximum width 50	$\left[\frac{e}{d}\right]$ 19.0

があり，逐次 Prolog において sidetracking[7] と呼ばれているもので，決定性の高いリテラルを選択するのがよいと考えられる．

また，このような防止策とは別に，資源が実際に使いつくされた場合に備えて，あらかじめチェックポイントを設けてそこでのゴールを記憶しておく方法も考えられる．純粋な Prolog においては，解を求めるのに必要な環境はおのおののゴール節のなかに閉じているので，これさえ記憶しておけば，いったん，適当なチェックポイント以降の処理を放棄して資源を再配分し，処理の区切りがついたところで，放棄したチェックポイント以降の処理をやりなおすことが可能である．

6. おわりに

本論文では，Prolog に代表される論理プログラミング言語の並列処理方式に関する一般的な考察を行い，それらの方式のうち OR 並列処理を用いて，マルチマイクロプロセッサ上で Prolog の並列処理を行うシステム“Paralog”を紹介した．例題の実行速度および処理系のオーバヘッドに関する測定結果から，このシステムが Prolog に内在する並列性を効率よくとり出していることを示し，また，例題の実行を追跡することにより，論理プログラミング言語の並列処理

には，十分大きな並列性が内在していることも確認した．最後に，数の爆発に対処するための方法について簡単な考察を行ったが，その詳細な検討および実装は今後の課題である．

参 考 文 献

1) Kowalski, R.: Predicate Logic as Programming Language, *Proc. IFIP* 74, pp. 569-574, North-Holland, Amsterdam (1974).
2) McDermott, D.: The Prolog Phenomenon, *ACM SIGART Newsl.*, No. 72, pp. 16-20 (1980).
3) Moto-oka, T. (ed.): *Fifth Generation Computer Systems*, p. 287, North-Holland, Amsterdam (1982).
4) Suzuki, T. et al.: Procedure Level Data Flow Processing on Dynamic Structure Multimicroprocessors, *JIP*, Vol. 5, No. 1, pp. 11-16 (1982).
5) Boyer, R. S. and Moore, J. S.: The Sharing of Structure in Theorem-Proving Programs, *Mach. Intell.*, Vol. 7, No. 6, pp. 101-116, (1972).
6) Warren, D. H. D.: Implementing Prolog—Compiling Predicate Logic Programs, *D.A.I. Res. Rep.*, No. 39-40 (1977).
7) Pereira, L. M. et al.: Intelligent Backtracking and Sidetracking in Horn Clause Programs —Theory, CIUNL 2/79, Centro de Informatica da Universidade Nova de Lisboa (1979).

(昭和58年 2 月28日受付)
(昭和58年 5 月10日採録)

付録　実行時間測定に用いた数式簡単化プログラム

```
#DECLARE equiv 2.
#DECLARE simpl 2.

#DECLARE plus 2.
#DECLARE times 2.
#DECLARE power 2.
#DECLARE exp 1.

+equiv(*x,*y)-simpl(*x,*y).

+simpl(exp(0),1).

+simpl(power(*x,0),1).
+simpl(power(*x,1),*y)-equiv(*x,*y).

+simpl(times(0,*x),0).
+simpl(times(1,*x),*y)-equiv(*x,*y).
+simpl(times(*x,0),0).
+simpl(times(*x,1),*y)-equiv(*x,*y).

+simpl(plus(0,*x),*y)-equiv(*x,*y).
+simpl(plus(*x,0),*y)-equiv(*x,*y).

+simpl(times(*a,*b),*c)-ARITH(*a,*b,0,*c).
+simpl(plus(*a,*b),*c)-ARITH(1,*a,*b,*c).

+simpl(times(power(*x,*a),power(*x,*b)),*y)
  -equiv(power(*x,plus(*a,*b)),*y).

+simpl(plus(times(*a,*x),times(*b,*x)),*y)
  -equiv(times(plus(*a,*b),*x),*y).

+simpl(plus(times(*x,*a),times(*x,*b)),*y)
  -equiv(times(*x,plus(*a,*b)),*y).

+equiv(times(*a,times(*b,*c)),*x)-equiv(times(times(*a,*b),*c),*x).
+equiv(plus(*a,plus(*b,*c)),*x)-equiv(plus(plus(*a,*b),*c),*x).

+simpl(exp(*x),*z)-simpl(*x,*y)-equiv(exp(*y),*z).
+simpl(power(*x,*y),*z)-simpl(*x,*u)-equiv(power(*u,*y),*z).
+simpl(power(*x,*y),*z)-simpl(*y,*v)-equiv(power(*x,*v),*z).
+simpl(times(*x,*y),*z)-simpl(*x,*u)-equiv(times(*u,*y),*z).
+simpl(times(*x,*y),*z)-simpl(*y,*v)-equiv(times(*x,*v),*z).
+simpl(plus(*x,*y),*z)-simpl(*x,*u)-equiv(plus(*u,*y),*z).
+simpl(plus(*x,*y),*z)-simpl(*y,*v)-equiv(plus(*x,*v),*z).

+equiv(*x,*x).
```