高並列推論エンジンＰＩＥにおける並列処理の効率化手法ついて

On the Efficient Parallel Processing of the Highly Parallel Inference Engine －ＰＩＥ

後藤 厚宏，相田 仁，山崎 篤，丸山 勉，湯原 雅信，田中 英彦，元岡 達
A. Goto, H. Aida, A. Yamazaki, T. Maruyama, M. Yuhara, H. Tanaka, T. Moto-oka
（東京大学 工学部）
Faculty of Engineering, University of TOKYO

１．はじめに

論理型プログラムでは、試行錯誤による問題解決を積極的に進めることが可能であるため、非決定性を含む形で問題が記述される場合が多い。

一般に、このようなプログラムを単なる横型探索によって実行すると、その選択肢の数は非常に大きくなる。例えば、通常のProlog によって記述した例題プログラム８－Queens を横型探索に基づいて実行した結果を図１に示す。横軸は探索木の深度であり、縦軸はその深度における単一化の総成功数と総失敗数を示す。

ここで用いた８－Queens プログラムでは、全部で92の解が求まるが、選択肢の数は最大1500程度にまで達する。つまり、大部分の選択肢は失敗の確認を行っていることになる。

高並列推論エンジンＰＩＥ（Parallel Inference Engine ）は、論理型プログラムの非決定性によって生じる多数の選択肢の実行を並列に行うマシンであり、そのアーキテクチャは高並列処理に対応できる［Got83］。しかし高並列マシンといえども、そのハードウェア資源は有限であるため、単なる横型探索ではマシンのハードウェア資源（プロセッサ数）を容易に上回る。そこで、ＰＩＥにおいて有効な並列処理を実現するためには、積極的に並列処理の効率化手法を導入する必要がある。

並列処理の効率化は、

① プログラマによるメタ知識の記述
② 入力プログラムの前処理
③ 実行時における動的な判断

の組み合わせによって実現できると考えられる。

本報告では、現在、最も検討が進んでいる実行時の動的な判定による効率化手法について、シミュレーション結果に基づいた評価検討を行う。次章においては、ＰＩＥにおける効率化手法について整理する。続いて３章から５章にわたり、定義節の動的な絞り込み効果、ゴールフレームの選択の効果、ゴールリテラルの選択の効果、のそれぞれについて、シミュレーション結果に基づいた検討を行う。

高並列推論エンジンＰＩＥの概要、今回作成したソフトウェアシミュレータの概要および評価プログラムについては付録ならびに［Aid82］を参照されたい。

２．並列処理の効率化

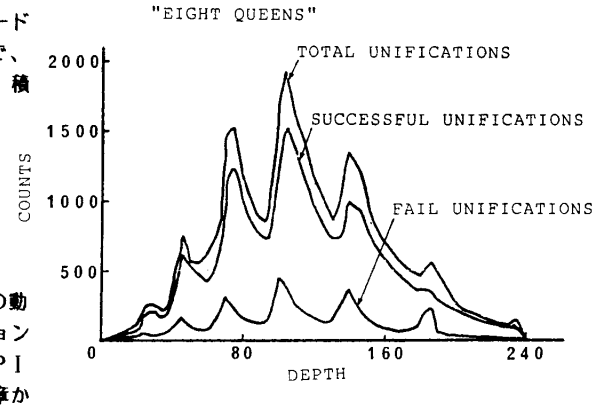
２．１ 処理の効率化を実現する要素

処理の効率化の実現は、大別して次の３要素の組み合わせとして考えることができる。

（１）プログラマによるメタ知識の記述

プログラマが解の探索ストラテジをメタ知識として記述するには、言語仕様としての記述方法が問題となる。

現在の逐次型Prolog における cut, mode 宣言, read-only annotation, bind-hook 等も一種のメタ知識の記述と考えられ、現在、それらを高い次元で包含するものの検討を進めている。



[Fig. 1]
Inter Goal-Frame Parallelism
in 8-Queens Program

このようなメタ知識が記述された場合は前処理を経てマシン内に取り込まれる。ＰＩＥでは原則としてアクティビティ制御機構（ＡＣおよびActivity Manager）によってメタ知識が利用される。ただし、bind-hook 等のように機械語としてゴールフレームの内部表現中に埋め込まれるものもある。

（２）入力プログラムの前処理

入力プログラムの前処理において、前述のメタ知識を内部に取り込むだけでなく、前処理の段階で積極的にストラテジを抽出することも可能である。

例えば、プログラム＜ｅｘ.1>でゴールリテラルgl１と定義（Ｄ３）およびgl２と（Ｄ２）の単一化はプログラムの入力時に失敗することが明白である。

また＜ｅｘ.2>では定義（Ｄ１）とゴールとの導出が実行され、ボディ部がゴールに取り込まれた時、引数＊ｘ，＊ｙのどちらが定数になっているかによってリテラルgl１，gl２の最適な実行順序が決ってくる。

この２つの例は、前者が両方向推論へ、後者は外部データベースマシン（または知識ベースマシン）との結合方式へ発展するものであり、重要な課題である。

ただし、このような前処理によるストラテジの抽出では、前者のような手法を徹底して行うとボトムアップ実行と同等になり、プログラムの実行以上に前処理のコストを必要とすることになりかねない。また、プログラムの動作がプログラマの意図しないものとなる可能性もある。このような点については、言語仕様を含め細心の注意が必要であろう。

（３）実行時における動的な判断

前述のメタ知識として記述されたストラテジや前処理において抽出されたストラテジに基いて、決定的な動作をする機械語を生成することが可能な場合もある。しかし多くの場合、実行時の動的な判断が処理の効率化を実現するために必要である。

そこで今回は、この実行時の動的な判断に基づく効率化手法について議論する。この手法は、

① 論理型プログラムの動作特性を明らかにする。
② 並列処理アーキテクチャの立場から有効な解探索ストラテジを検討する。
③ メタ知識の記述能力を含めた論理型言語の設計に対するマシン側からの指針を得る。

等の目的においても重要な意義を有している。

```
< ex.1 >                          gl1      gl2
        (D1)  P(a,*x) :- Q(b,*x), Q(*y,c).
                          |          |
                          |          |
        (D2)  Q(b,d).     |          |
                          |          |
              ------------------------
                          |
        (D3)  Q(a,c).
```

２．２　処理コスト

論理型プログラムの処理コストとしては、種々の要素が考えられるが、その１つとして単一化の回数が考えられる。

ひとつの解に注目するとき、入力ゴールからその解を得るまでに必要な単一化の回数は問題が与えられた時点で決定している。しかし実際の処理では解を求める過程において多数のゴールが分岐し、失敗や別解に達する。（停止しないこともある。）この間に行われる無駄な単一化の数は制御方式により大きな差がある。

そこで本報告では、“与えられた問題の解（一部または全部）を得るまでに必要な単一化の総数”を処理コストと考える。

２．３　動的な判断に基づく効率化手法

前述の実行時の動的な判断に基づく効率化手法は、ＵＰにおいて、単一化の負荷の一部をＤＭに肩替りさせ、ＵＰの負荷を軽くする手法と、ＡＣによりゴールプール中のゴールフレームの適切な管理を行う手法に分類できる。

このうち前者については第３章で検討する。後者については、有限資源のもとで、より早く解を求めるために有効な並列性を抽出する方法として、次の３つが考えられることを文献［Got82］において既に述べた。

① 解が求まりやすいゴールフレームから優先してプロセッサー（ＵＰ）を割り当てる。
② 解を持たないことがあらかじめ判定可能なゴールフレームを早く見つけ出し、消滅させる。
③ 数の爆発が起きないように、ゴールフレーム内では決定性の高いゴールリテラルを優先する。

これらのうち、①、③に関してそれぞれ第４章、第５章で検討を行う。②に関しては、その概略を以下に述べる。

ＰＩＥでは、実行時に、選択肢としての多数のゴールフレームがゴールプール内に存在する。

```
< ex.2 >
        grand-parent(*x,*z)
          :- parent(*x,*y),parent(*y,*z).
                  gl1            gl2

        parent(tohru,hidehiko).
        parent(tohru,tadashi).
        parent(junko,hidehiko).
        parent(hidehiko,fumiko).
                :
                :
```

このゴールプール内のゴールフレームを、他の特定のゴールフレームの導出結果に伴う副作用によって単一化の実行なしに消去可能な場合がある。ただし、一般的に並列処理の環境では、その操作が局所的な情報を用いるか、大域的な情報を用いるかによって実現の難易度が異なる。

前者の例には、guard 等により求める解がひとつでよいときに、あるゴールフレームの実行が成功した段階で他の選択枝の実行を停止させる場合などがある。このような局所的な情報を用いる操作は、 ＰＩＥの各ＡＣ（Activity Controller ）が持つ探索木情報と、ＡＣ間で送受するノード制御コマンドによって比較的容易に実現できる。

後者は逐次形システムにおけるremote cut［Tak83］や知的後戻りに相当するものである。

知的後戻りをゴールフレーム間並列処理の環境下に適用する場合は、ＵＰにおける全ての選択枝に対する単一化が失敗した時、失敗の原因を調べ、失敗の原因が最後に揃ったゴールフレームを祖先に溯って見出すことになる。このような大域的な情報を必要とする手法は、オーバーヘッドが大きいため、さらに検討が必要である。

３．　定義節の動的な絞り込み

（１）定義節の絞り込み

ＰＩＥでは、各ＵＰにおいて１つのゴールリテラルと、対応するすべての定義節との間で単一化が行われる。

ここで、ＵＰにおける単一化の失敗に注目すると、大部分の失敗が簡単な照合操作によって認識できると予想される。

一方、単一化が成功したものについては縮退操作が必要でおり、ＵＰにとって比較的重い負荷となる。ＵＰの負荷を低減するためには、単一化の早い段階で失敗を認識することが望ましい。

そこでＵＰが定義節を定義節メモリ（ＤＭ）から見つけ出すとき、入力ゴールフレーム内のリテラル引数の情報の一部をＤＭに送り、定義節を絞り込んでから単一化を行うことを考える。

このとき、ＤＭにおける照合操作では、より多くの情報を用いれば効果が上がることは自明である。ただし、ゴールフレームの内部構造やＤＭのハードウェア構造におけるオーバヘッドが少ないものでなければならない。

ＰＩＥのゴールフレームのデータ構造では、述語引数の第一レベルが述語記号とともにベクタ形式でまとまっているため（付録２　図７）、その情報を使って絞り込みを行うことが望ましい。そこで、シミュレーションでは絞り込みに用いる情報を深さ方向の２つのレベル

- level ＝０：　述語名，引数の第一レベルの atom（型と値），vector，list（型）
- level ＝１：　述語名，引数の第一レベルの atom（型と値），vector（vector名），list（car 部の型と値）

と、絞り込みに用いる述語引数の最大個数（ filtered args ＝ｎ）を組み合わせ、効果の違いを調べることにした。＜ｅｘ.3＞

（２）シミュレーション結果

定義節の絞り込みの効果についてシミュレーションで測定した結果を表１に示す。ここではＵＰ数等は充分大きいものとした。

この評価結果から次のことがわかる。

①［ＪＥ］、［equiv 1,2 ］などでは、評価に用いる述語引数の数を増しても絞り込みの効果は小さく、述語引数に現われる vector 名または list のcar を絞り込みの情報として用いる方（level ＝１）が有効である。

②［ＬＬ２Ｐ］、［6-queens］等では、逆に引数の数を増すことによって効果が上がり、［ＥＪ］では両方の効果が現われる。

（３）評価検討

今回のシミュレーションで用いたゴールフレームのデータ構造では、述語引数としての vector 名や list の car を構造体領域に置いた。しかし、表１の評価結果から、これらの情報をリテラル領域に置き、ＤＭ内の照合器へ送りやすいようにすることがよいことがわかる。ただし、 vector 名や list のcar のセルの全部をリテラル領域に移すことには無理がある。そこで図２に示すように、その一部をリテラル領域に持ち込むようなデータ構造が有効と思われる。一方、［ＪＥ］［ＥＪ］のように、両方向に実行させるプログラムでは評価する引数の数が多い程確実である。しかし述語引数の数は経験的にみてそれほど多いものはないため、評価する引数の数は、３程度に固定してかまわないと思われる。

< ex.3 >　P(a, f(*x,b), [c,d,e], 10)

MODE	CHECK	
level = 0 args = 4	(predicate name) (arg1: type, value) (arg2: type) (arg3: type) (arg4: type, value)	P SYM,a VECT LIST INT,10
level = 1 args = 3	(predicate name) (arg1: type, value) (arg2: type, func. name) (arg3: type, car)	P SYM,a VECT,f LIST,SYM,c

４．　ゴールフレームの選択

（１）ゴールフレームの決着の着き易さ

「プロセッサ（ＵＰ）数＜ゴールフレーム数」という環境における、プロセッサの割り当てストラテジとしては、一般に縦型探索（depth-first 処理）と横型探索（breadth-first 処理）が考えられる。　前者は、停止しないゴールがあるような時に危険であり、後者は最初の解を求めるまでのコストが大きい。

[Table 1]
Dynamic Pre-Filtering of Definition Clauses
- Unification Failures after Pre-Filtering -

Program Examples	total failures	filter level	filter args 1	2	3
[JE]	287	0	287	287	287
		1	71	71	71
[EJ]	39	0	39	39	39
		1	31	31	1
[equiv1]	1011	0	354	354	354
		1	98	98	98
[equiv2]	7431	0	3531	3531	3531
		1	737	737	737
[diff]	257	0	137	137	137
		1	50	50	50
[LL2P]	7331	0	7331	4989	4989
		1	7331	4989	4989
[6-Queens]	1595	0	1446	204	204
		1	1446	204	204

このためメタ知識によって各ゴールフレームの決着の着き易さが判断できれば、

・早く最初の解を求める、

・資源を解放する

という点において有利となる。

そこで次のようなメタ知識を考える。

ある実行ステップにおいて、図３に示すＧＦ１というゴールフレーム（Ａ～Ｄはゴールリテラル）があったとする。ゴールリテラルの実行順序として“左から右へ”という規則がある時、ＧＦ１からはＧＦ２，ＧＦ３のようなゴールフレームの子孫が派生すると考えられる。この場合、ＧＦ１で導入されたゴールリテラルの生き残りの数を比較するとＧＦ２の方が少ない。このためＧＦ２の方がＧＦ３に比べて成功／失敗の決着が付き易いと考えることができる。

このようなメタ知識を用いると、縦型探索のように無限ループに陥る危険も少なく、また横型探索よりも第一解を見つけるまでのコストを小さくできる。この手法をマシン内に実装するには、各ゴールリテラルに、それが導入された導出のステップ番号を付ければよい。しかし５章で述べるゴールリテラルの選択の手法と共に用いるにはさらに検討が必要であろう。

（２）ゴールフレームの選択効果の測定

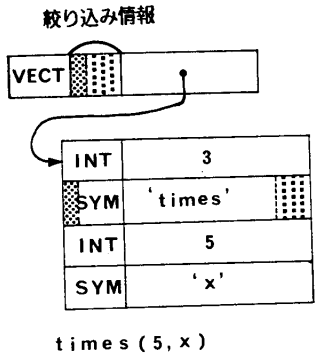
ここでは［6-queens］、［equiv２］の２つのプログラムについて、３種の選択ストラテジ

① 縦型探索、

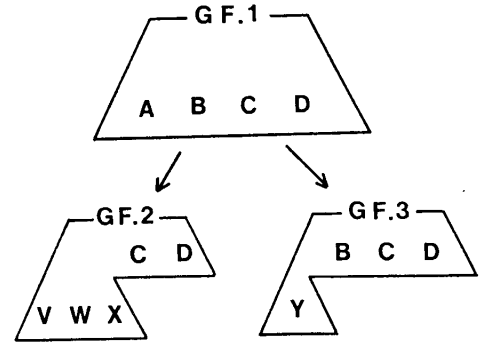
② 横型探索、

③ 決着の着き易さによる探索

による違いを調べた（図４，表２）。図４の実線はＵＰの稼動数を示し、点線は実行待ちを含めたゴールフレームの総数を示す。



[Fig. 2]
New Representation of Structures



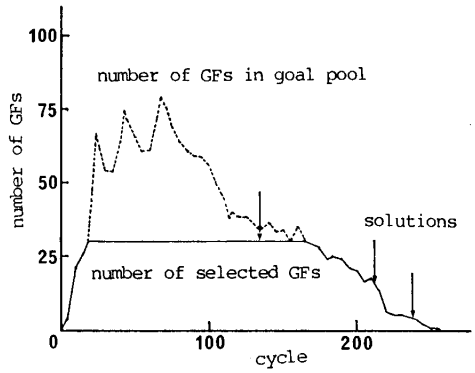
[Fig. 3]
Example of Goal-Frame Priority

[6-queens] の場合、最初の解を求めるまでの処理コスト（cycle 数）は、UPの台数（10,30,50）に係わらず、縦型探索と決着の着き易さによる探索の両方とも、UP台数＝∞のときと同程度である。さらに実行途中におけるゴールプール内の実行待ちゴールフレーム数は横型探索に比べて小さい。

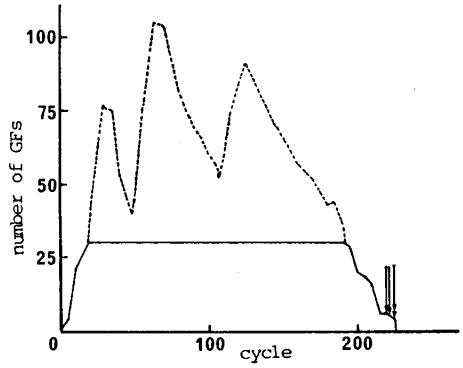
一方、[equiv 2] では [6-queens] ほどの効果が認められないが、これは最適解でないものが多数求まるためと考えられる。

[Table 2]　Selection of Goal Frames

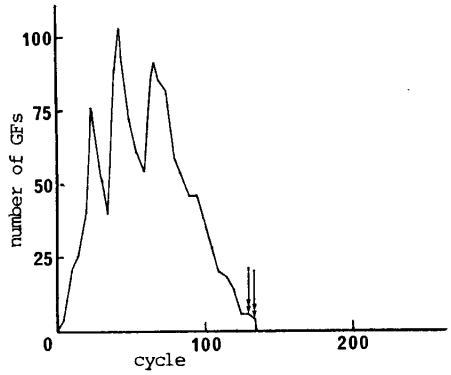
	Answers (cycle#) 1st	last	Total cycles	Waiting cycles Max	Total
[equiv2]	#UP = 10				
depth	1	40	52	42	748
breadth	1	41	43	8	947
proposed	1	42	44	44	895
[6-Queens]	#UP = 30				
depth	133	238	257	136	3178
breadth	220	224	225	6	6502
proposed	138	248	249	123	1942



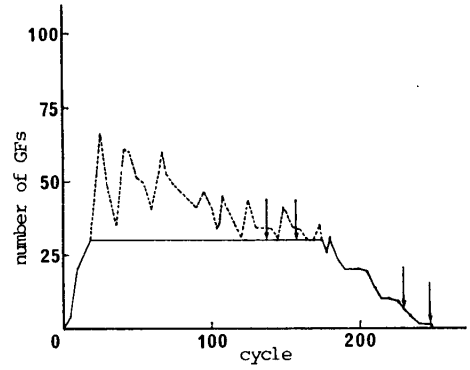
depth first search　(UP = 30)



breadth first search (UP = 30)



(UP = infinite)



proposed method (UP = 30)

[Fig. 4]　Selection of Goal-Frames

[6-Queens]

５．　リテラルの選択

（１）ゴールリテラルの選択

純Prolog 等では、ＡＮＤ関係にあるゴールリテラルには順序関係はない。しかし、ゴールフレーム間並列処理の環境では、実行途中におけるゴールフレームの数が異なる。すなわち探索木の形状が異なる。有限資源の環境において資源の節約を計るためには、枝が大きく広がりすぎないようにゴールフレーム内のゴールリテラルの実行順序を選択することが望ましい。

また、ゴールリテラルを選択して実行することは、ゴールリテラル間並列処理へのステップと考えることもできる。

このようなリテラルの選択基準としては次の２つが考えられる。

① リテラルに対応する定義節による選択

② ゴールリテラルの引数状況による選択

一般的傾向として、論理型プログラムの手続き（同じ頭部述語名を持つ定義節の集合）は引数パターンによる場合分けによって記述される場合が多い。そのためゴールフレーム内のリテラルでは、定数引数の割合が多いものほど早く成功または失敗に到ると予想される。

そこで、各ゴールリテラル中のそれぞれの引数に、その型に応じた点数を付け、点数（優先度）の最も高いリテラルを選択して単一化を行なうことにする。同一点数の場合には左側のリテラルを優先する。

ここで引数の評価点点は、原則として変数に負の点数（vpoint）、定数 atom に正の点数（cpoint）を与え、また構造体および構造体内に現われる引数については構造体の点（fpoint）とその深さに関する評価点（spoint）を設定することにする。＜ｅｘ.4＞

ただし、組み込み述語については次のような特別な優先度を与えることにした。

print　：　常に低い優先度

算術述語：　実行に必要な引数（integer ）が揃うまでは低い優先度。ただし、引数が揃った場合や失敗が明白な型を持つ値（たとえば構造体）が変数に束縛された段階で最優先。

```
< ex.4 >     P(a, f(*x,b), *y)

  priority score =
        cpoint +
        fpoint + spoint * (vpoint + cpoint) +
        vpoint
```

（２）シミュレーション結果

表３に４個のプログラムについて測定した結果を示す。

これから、［ＪＥ］では、変数に対して負の点を与えることによって実行が最適化されることがわかる。また［ＥＪ］［equiv 1,2 ］等で改善効果がないのは、左から右への実行が最も効率的になるように定義節が記述されていたためであり、実際、プログラムの定義節体部のリテラルの順序を無作為に並べ替えると、表４のような効果が現われる。

[Table 3] Selection of Goal Literals
  --- Total Unifications (cost) ---

point (c,v,f,s)	[JE]	[EJ]	[equiv1]	[equiv2]
left to right	145	23	86	513
(0, 0, 0, 0)	145	27	86	513
(10, 2, 0, 1)	36	27	-	-
(10, 0, 0, 1)	36	27	-	-
(10, -20, 0, 1)	36	27	86	-
(10, -50, 0, 1)	36	27	86	-
(10, -60, 0, 1)	36	27	86	513
(10, -80, 0, 1)	36	34	86	513
(10,-100, 0, 1)	36	34	86	513
(0, -10, 0, 0)	189	31	86	513
(0, -10, 0,0.5)	192	34	86	513
(0, -10, 0, 1)	193	34	86	513
(0, -10, 0, 2)	-	-	86	513

" - " means Over Flow

[Table 4] Selection of Goal Literals
(Ordering of Definition body is randam)
  --- Total Unifications (cost) ---

point (c,v,f,s)	[JE]	[EJ]	[equiv1]	[equiv2]
left to right	-	-	-	-
(0, 0, 0, 0)	-	-	-	-
(10, -50, 0, 1)	36	27	86	-
(10,-100, 0, 1)	36	34	86	513

" - " means Over Flow

また、表3の結果から、リテラルの選択において引数の評価点の感度が相当高いこと、さらにリテラルの選択順序によってプログラムの動作が大きく変化し、解が求まらないことも多いことがわかる。この理由のひとつとして、再帰的な定義節の記述がある。特に［equiv 1,2 ］の定義節は相互再帰を含んでいるため、より注意をはらってリテラルの選択を行わないと、無限ループに陥りやすいと思われる。

6．　おわりに

本報告では、主に実行時の動的な判定による並列処理の効率化と、シミュレーション評価を行った結果について述べた。

その成果を以下に整理しておく。

① 定義節の絞り込み

ＰＩＥでは、各ＵＰにおいて定義節テンプレート間並列処理を実行するため、定義節メモリ（ＤＭ）からＵＰへ入力される定義節の量が多い。

このような状況では、ＤＭ側において予め失敗が明白な定義節をふるいにかけることは重要である。

今回の評価では、定義節の絞り込みの効果が充分に大きいことがわかるとともに、定義節の絞り込みに適するゴールフレームのデータ構造や、ＤＭのハードウェア構造を検討する指針が得られた。

② ゴールリテラルの選択による処理の効率化

ゴールリテラルの選択による実行順序の効率化は、問題の非決定性に対応する並列処理において重要であり、ゴールリテラル間並列処理へのステップと考えることができる。

今回のシミュレーションでは、述語引数のデータ型の評価が、効率的な実行順序を動的に見い出す要素となり得ることがわかった。ただし一方では、再帰的に記述された定義節等に注意を要することもわかった。

今後は、ゴールリテラルの選択に必要な他の要素について検討することによって、実行を効率化するとともに、ゴールリテラル間並列処理への発展を考える予定である。

③ ゴールフレームの選択

今回のシミュレーションでは、縦型探索、横型探索、決着の着き易さ優先の３種について評価を行った。この結果、決着の着き易さに関するメタ知識の一例を導入できること、および、その改善効果が得られることがわかった。

ＰＩＥでは、この種のメタ知識は主としてアクティビティ制御機構によって実現する。今後は、様々のメタ知識について、アクティビティ制御機構上の実現手法を検討していく予定である。

［参考文献］

［Tak83］高木　他，“拡張制御構造のProlog への導入”，２６回情処全大，４Ｄ－１１．

［Got83］後藤　他，“高並列推論エンジンＰＩＥについて”，Logic Prog. Conf. ８３．

［Got82］後藤　他，“推論向き高並列計算機システムの基本アーキテクチャ”信学技報ＥＣ82-43

［Aid82］相田　他，“並列ＰＲＯＬＯＧシステム“Paralog”の性能測定”２４回情処全大会５Ｄ－５．

付録１．高並列推論エンジンＰＩＥ

現在設計を進めている高並列推論エンジンＰＩＥ第一次モデル（図５）の特徴は以下のとおりである。

［Got83］

① 書き換えモデルによる実行：

書き換えモデルとは、プログラムの実行を“初期ゴールに対応するテンプレートに定義節テンプレートを接続し（単一化）、中間結果としてのゴールを生成する過程”として捉えるた処理モデルである。この中間結果としてのゴール中には、単一化を必要とするゴールリテラルと共に、それまでの単一化によって生成された変数の束縛情報（単一化の環境）がすべて含まれており、これを特にゴールフレームと呼ぶ。

② ＵＰによる定義節テンプレート間並列処理と高並列アクティビティの生成：

ＰＩＥにおける基本処理要素は、ハードウェア照合器を有するＵＰ（Unify Processor）である。ＵＰは、ゴールフレームを入力すると、その中のひとつのゴールリテラルについて対応するすべての定義節テンプレートとの単一化を行ない、新たなゴールフレームを（複数）導出する。

各ＵＰには定義節テンプレートのメモリ（ＤＭ）が付随しており、各ＵＰのＤＭがすべての種類の定義節テンプレートを持つモデルを現在考えている。

③ 多数のＵＰによるゴールフレーム間並列処理：

ＵＰによって導出された複数のゴールフレームは多数のメモリモジュールから成るゴールプール（goal pool ）に蓄えられ、ＵＰに対する新たな入力となる。ＰＩＥでは、多数台のＵＰによってゴールフレーム間並列処理を実現する。

④ ゴールフレームの独立性と縮退：

ＰＩＥでは、ゴールフレーム全体をＵＰとゴールプール間で送受することにより、各々のゴールフレームの独立性を高める。これにより、操作の集中化を避け、ゴールフレーム単位の高並列性を活かす。

このとき、記憶コストおよび転送コストを押えるためにゴールフレームの縮退操作を導入する。縮退操作は、導出された新ゴールフレームから不要な情報を取り除き、ゴールフレームの大きさを必要最小限にする。

⑤ 探索木情報によるアクティビティ制御：

論理型プログラムの計算は、原則としてORノードから構成される探索木として表現できる。ＰＩＥでは、実行時に生成される多数のゴールフレームを、計算の途中状態として探索木上に位置付けることによって、各々の親子関係を明確にする。

⑥ ノード属性とノード制御コマンド：

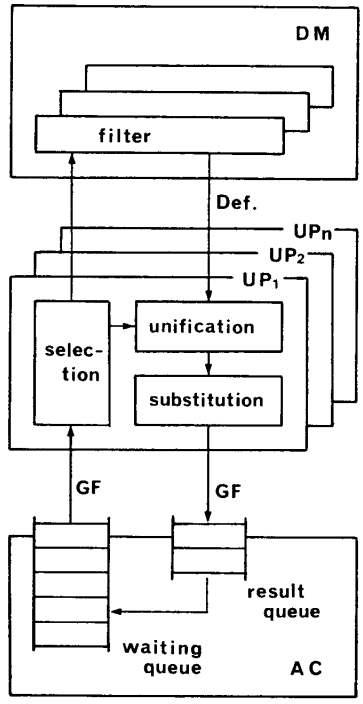
探索木の各ノードに、種々のノード属性を導入し、さらにノード制御コマンドを設定して、これをノード間で送受することによって、探索木の伸長、不要ノードの刈り込み等の基本的なアクティビティ制御と、探索ストラテジや言語上の拡張機能を実現する。

付録２．ソフトウェアシミュレータの構成

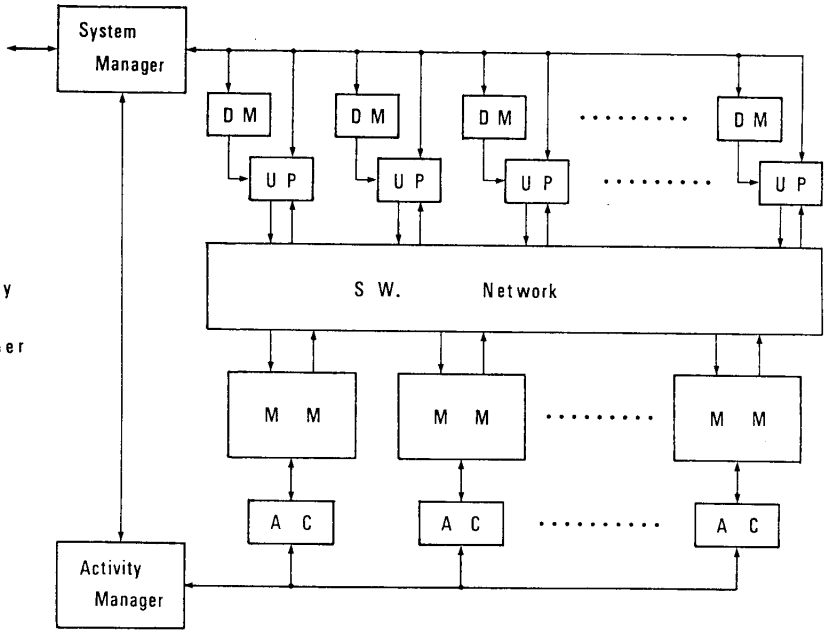
（１）シミュレータ概要

今回作成したシミュレータは、Unix 上のC言語で記述されており（約3600行）、図６で示すようにＵＰ部とＡＣ部から成る。

ＡＣ部はＰＩＥにおけるＡＣとMM（ゴールプール）に相当する。シミュレートするＡＣは現在１台であり、ゴールプールの管理を行なう。



[Fig. 6]
Organization of Software Simulator for PIE



UP : Unify Processor
DM : Definition Memory
MM : Memory Module
AC : Activity Controller

[Fig. 5] Hardware Organization of PIE (model 1)

UP部は、PIEのUPとDMに相当し、任意の台数のUPをシミュレートできる。UPはゴールフレーム内の選択されたリテラルについて定義節テンプレート間並列処理を行なう。すべてのUPは同一構造を持ち、ひとつの入力ゴールフレームに対する処理時間は定義節の数や単一化の成功・失敗によらず一定としている。

また本シミュレータではMM－UP間のNetworkの影響については考慮していない。

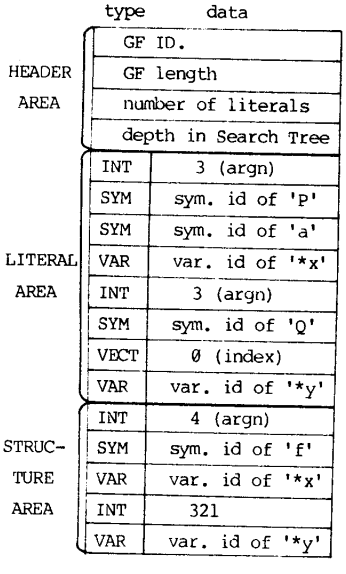
ゴールフレームはタグ付きの固定長セルが連続した構造に成っており次の3つの部分から成る。
（表5，図7）

① ヘッダ：ゴールフレームの属性や区分を示す。
② リテラル部：ベクタ型のリテラルの領域。
引数は各々1つのセルで表現され、1セルでは表現できないもの（functor,listなど）は構造部へのポインタとなる。
③ 構造部：1セルで表わせないデータ構造を収納する。

今回、評価を行なった各効率化手法は組み込み述語による動作モード設定として実装されている。（表6）

[Table 5] Data Types in Simulator

Atoms : INT, SYM
Variable : VAR
Structures : VECT, LIST



?- P(a,*x),Q(f(*x,321,*y),*y).

[Fig. 7] Internal Representation of Goal-Frame

（2）UP部

UP部は以下の手順でゴールフレームの処理を行なう。

① リテラルの選択
② 定義節の絞り込み
③ 単一化
④ 縮退および新しいゴールフレームの生成

ゴールフレーム、定義節中の各変数は通し番号によって識別される。単一化における変数の束縛は、図8に示すように置換メモリ上で行なう。置換メモリは、単一化の前にすべて未定義（undef ）に初期化される。

[Table 6]
System predicates for setting execution modes

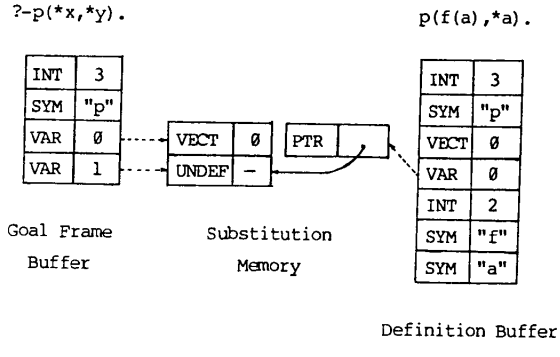
processor(n_up)
Sets number of UPs to n_up.

point(cpoint,vpoint,fpoint,spoint)
These points are used in case of selecting a literal to be unified.

selectgf(mode)
mode = 1: depth first search
mode = 2: breadth first search
mode = 3: proposed method

selectl(mode)
mode = 1: system selects one out of many goal literals, according to the priority scores calculated.
mode = 0: standard PROLOG mode (left to right) (default)

filter(fil_level,fil_argn)
mode = 1: used when filtering definitions in DM
mode = 0: does not filter



[Fig. 8] Unification

選択されなかったリテラルおよび定義節の体部のリテラルから、新しいゴールフレームを生成する。このとき、リテラル中の変数については置換メモリの値を参照しながら書き変えを行なう。置換メモリ上の値が未定義のままならば、新たな変数番号に書き変える。束縛されているときにはその値に書き変える。この手順によりどのセルからも参照されていないセルは自動的に取り除かれる。

（3）AC部

ゴールプールは waiting queue とresult queueからなる。waiting queue は優先度に従ってsortされた実行待ちゴールフレームの queue である。ACは waiting queue の先頭からUP台数分のゴールフレームを取り出しUPへ送る。UPによって生成されたゴールフレームは result queue につながれる。ACは result queue からゴールフレームを取りだすとその優先度を評価して waiting queueに挿入する。

付録3．　評価用プログラム

```
[JE]
  ?-translat([watashi,wa,watashi,no,te,no,
        naka,ni,ikutsuka,no,tamago,wo,
        motteiru],E),print(E).
[EJ]
  ?-translat([i,have,some,eggs,in,my,hands],
        J),print(J).

    translat(J,E):-sentence(J,[],E,[]).
    translat(E,J):-sentence(J,[],E,[]).
    sentence(J1,J3,E1,E3)
      :-np(J1,[wa|J2],E1,E2),vp(J2,J3,E2,E3).
    np(J1,J2,E1,E2):-noun(J1,J2,E1,E2).
    np(J1,J3,E1,E3)
      :-det(J1,J2,E1,E2),noun(J2,J3,E2,E3).
    vp(J1,J4,E1,E4)
      :-vt(J3,J4,E1,E2),np(J2,[wo|J3],E2,E3),
        pp(J1,J2,E3,E4).
    vp(J1,J3,E1,E3)
      :-vi(J2,J3,E1,E2),pp(J1,J2,E2,E3).
    pp(J1,J1,E1,E1).
    pp(J1,J3,E1,E3)
      :-prep(J2,J3,E1,E2),np(J1,J2,E2,E3).

    prep([no,naka,ni|J1],J1,[in|E1],E1).
    vt([motteiru|J1],J1,[have|E1],E1).
    vi([aru|J1],J1,[are|E1],E1).
    det([ikutsuka,no|J1],J1,[some|E1],E1).
    det([watashi,no|J1],J1,[my|E1],E1).
    det([anata,no|J1],J1,[your|E1],E1).
    noun([watashi|J1],J1,[i|E1],E1).
    noun([anata|J1],J1,[you|E1],E1).
    noun([te|J1],J1,[hands|E1],E1).
    noun([tamago|J1],J1,[eggs|E1],E1).
    noun([ringo|J1],J1,[apples|E1],E1).
```

```
[equiv 1]
  ?- equiv(times(3,times(2,power(x,1))),F),
       print(F).

[equiv 2]
  ?- equiv(plus(times(times(2,power(x,1)),
       exp(power(x,2))), times(power(x,2),
       times(times(2,power(x,1)),
       exp(power(x,2))))), F),  print(F).

/*  Definitions of equiv and simple
    are found in [Aid82].              */

[diff]
  ?- diff(times(power(x,2),exp(power(x,2))),
       x,F),print(F).

    diff(C,X,0):-number(C).
    diff(power(X,N),X,times(N,power(X,N1)))
        :-plus(N1,1,N).
    diff(exp(X),X,exp(X)).
    diff(times(C,Y),X,times(C,YY)):-
        number(C),diff(Y,X,YY).
    diff(plus(U,V),X,plus(UU,VV)):-
        diff(U,X,UU),diff(V,X,VV).
    diff(times(U,V),X,
        plus(times(UU,V),times(U,VV))):-
            diff(U,X,UU),diff(V,X,VV).
    diff(X,X,1).
    diff(times(A,Y),X,times(C,Z)):-
    diff(Y,X,times(B,Z)),times(A,B,C).
    diff(Z,X,times(YY,ZZ)):-
        dif(Z,Y,ZZ),diff(Y,X,YY).
    dif(power(X,N),X,times(N,power(X,N1))):-
        plus(N1,1,N).
    dif(exp(X),X,exp(X)).
```

```
[ LL2P ]
  ?-main(LISP,LOGIC,PROLOG).

  main([L,I,S,P],[L,O,G,I,C],[P,R,O,L,O,G]):-
    sel(P,[1,2],XX),
    sel(P,[0,1,2,3,4,5,6,7,8,9],R1),
    sel(C,R1,R2),
    onedigit(P,C,G,0,CAR1),sel(G,R2,R3),
    sel(S,R3,R4),sel(I,R4,R5),
    onedigit(S,I,O,CAR1,CAR2),sel(O,R5,R6),
    onedigit(I,G,L,CAR2,CAR3),
    sel(L,[4,5,6,7,8,9],YY),
    sel(L,R6,R7),
    onedigit(L,O,O,CAR3,CAR4),
    onedigit(L,0,R,CAR4,P),
    sel(R,R7,R8),
    print([L,I,S,P],[L,O,G,I,C],[P,R,O,L,O,G]).

    onedigit(A,B,C,CI,CO):-plus(A,B,AB),
        times(AB,2,AB2), plus(AB2,CI,DG),
        divmod(DG,10,CO,C).
    sel(X,[X|R],R).
    sel(X,[Y|R],[Y|Z]):-sel(X,R,Z).
```

```
[ 6-Queens ]
  ?-queens([1,2,3,4,5,6],[],ANS),print(ANS).

    queens([],Y,Y).
    queens(X,Y,Z):-sel(U,X,V),safe(U,Y,1),
        queens(V,[U|Y],Z).
    safe(U,[],_).
    safe(U,[P|Q],N):-nodiag(U,P,N),
        plus(N,1,M),safe(U,Q,M).
    nodiag(U,P,N):-plus(P,N,T1),
        minus(P,N,T2),neq(U,T1),neq(U,T2).
```