EC84－45

高並列推論エンジンＰＩＥの階層的構成とそのシミュレータ

The Architecture of A Highly Parallel Inference Engine - PIE - and its Simulator

丸山　勉　，　平田　圭二　，　相田　仁　，　田中　英彦　，　元岡　達

T.Maruyama, K.Hirata, H.Aida, H.Tanaka, T.Moto-oka

東京大学　工学部

Faculty of Engineering, University of TOKYO

１．はじめに

我々は、現在推論エンジンＰＩＥ（A　Parallel Inference　Engine ）の設計を進めている。現在までに、ＰＩＥの基本処理要素である単一化プロセッサの試作を行ない［１］［２］、そのデータを基にＰＩＥ第一次モデルについての基本的なシミュレーションを行なった［３］［４］［５］。それらのデータを基にして、ＰＩＥ第二次モデルの構成を決定し、現在、そのシミュレータを製作中である。本発表では、最初にＰＩＥ第二次モデルの概略について述べる。次に、ＰＩＥ第一次モデルから第二次モデルへの主な変更点である構造メモリ、およびＡＮＤ並列の導入について述べる。ＡＮＤ並列の実現方式について述べるに当って、bagof の実現方式についても述べる。最後に、現在製作中であるＰＩＥ第二次モデルのシミュレータについて述べる。

２．ＰＩＥ第一次モデル

ＰＩＥ第二次モデル（ＰＩＥ－Ⅱ）について述べる前に、ＰＩＥ－Ⅱの構成を決定する基となったＰＩＥ第一次モデル（ＰＩＥ－Ⅰ）、およびそのシミュレーション結果について簡単に述べる。

ＰＩＥ－Ⅰの構成図を図１に示す。主なモジュールの機能は以下の通りである。

▶ 定義節メモリ（ＤＭ）

定義節（ＤＴ）を保持するメモリであり、単一化が行なわれる引数の状況によって、動的なＤＴの絞り込みを行なう。

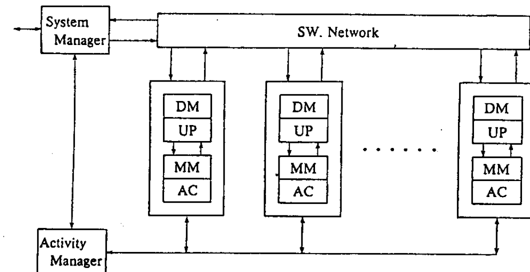

Fig.1　The System Organization of PIE-I

▶ 単一化プロセッサ（ＵＰ）

ゴールフレーム（ＧＦ）とＤＴとの間で、単一化、および縮退を行ない新しいＧＦを生成する。

▶ メモリモジュール（ＭＭ）

ＧＦを蓄えるメモリである。

▶ ＧＦ分配網（ＤＮ）

ＧＦの分配に用いられる網である。

▶ アクティビティ・コントローラ（ＡＣ）

ＧＦ間の関係を表わす推論木を保持し、ＡＣ間でコマンドを送受することによって、推論木の操作を行ない、ＧＦを管理する。

▶ アクティビティ・マネージャ（ＡＭ）

ＡＣの管理を行なう。

▶ システム・マネージャ

ＰＩＥ－Ⅰシステム全体の管理を行なう。

３．ＰＩＥ－Ⅰおよび構造データの共有方式についてのシミュレーション結果

ＰＩＥ－Ⅰについて行なったシミュレーション結果については、［３］［４］［５］［６］等で既に報告しているので、ここでは簡単に述べることにする。

３．１　試作単一化プロセッサ

試作単一化プロセッサの測定結果より、単一化および縮退はcell当り５～７step程度かかり、縮退時間はＧＦ長に比例することが解った。より詳しい結果については［１］を参照されたい。

３．２　シミュレーション結果

以下、ＰＩＥ－Ⅰについてのシミュレーション結果について述べる。

（１）ＰＩＥ－Ⅰの性能評価

<8-queens >を実行した場合、ＵＰ２５６台でＵＰ１台のときの約１７０倍の処理速度を実現することができることが確認された。更に並列性の大きなプログラムを実行すれば、より高い性能を発揮すると考えられる。

（２）ＧＦの分配ストラテジ

最も良いシミュレーション結果を示したストラテジempty-selfは、ＭＭが空ならば、ＵＰが導出したＧＦを

MMに戻し、そうでなければ、DNを通して他のユニットに送出するというものである。このような簡単な方法によって良い負荷分散を行なうことができることが解った。しかし、実際にはDNのトラヒックを考慮にいれて、GFをMMに戻すべきMM中のGFの数を動的に変えることが必要であろう。

（3）GFのMMからの選択ストラテジ

MM中からの単一化／縮退を行なうGFの選択方法をdepth-first , breadth-first 等に変えることによって、GFの数の増加を制御し、また最初の解を得るまでの時間を変えることができることが確認された。

（4）アクティビティ制御機構の評価

1台のACが処理しなければならないコマンドの数は、高々100～200stepに1つであることが解った。また、Command Networkによって転送されるコマンドの数は、AC1台当り、最高250stepに1つであることが解った。

3．3 構造データの共有方式についてのシミュレーション結果

試作単一化プロセッサの測定結果より、導出の処理時間全体に対して縮退操作が大きな比重占め、縮退時間はGF長に比例することが解った。一般にGF中、構造データが大きな比率を占めるので、ground instance となった構造データを構造メモリ（SM）に格納し、GF長を抑えることによって縮退時間の短縮を図ること考えた。SMに格納されたground instance は、単一化時に必要になると、その一部がlazy fetchと呼ばれる操作によってSMからUPに転送される。シミュレーション結果から、GF長がground instance の共有を行なわないときに比べ約半分程度になることが解った。また、lazy fetchが行なわれる単一化は、全単一化数の1～3割程度であることが解った。

4．PIE第二次モデル

前章で述べたデータを基に、PIE－Ⅱの構成を決定した。PIE－Ⅱの構成を図2、3に示す。PIE－Ⅱは、2階層（レベル－1および、レベル－2）より構成される。

レベル－1システムは、Inference Unit（IU），SM，AM，およびネットワークより構成される。レベル－1システム内では、SMを用いて構造データの共有を行なう。レベル－2システムは、複数のレベル－1システムからなる。レベル－1システム間では、構造データは共有されない。負荷分散のメカニズムによってレベル－1システム間でGFの転送が行なわれる場合には、

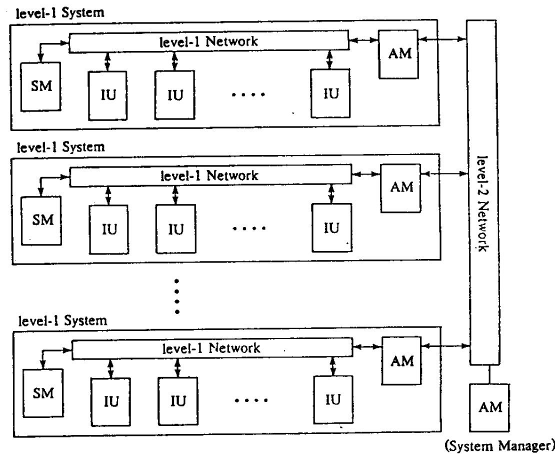

Fig.2 The Global Architecture of PIE-II'

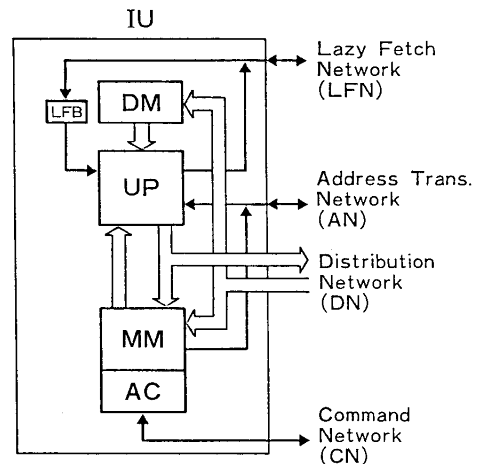

Fig.3 The Internal Architecture of IU

GFの転送に先立って構造データはSMからそのGF内にコピーされる。System Managerは、PIE－IIシステム全体の管理を行なう。

IUはDM，UP，MM，およびACより構成される。IU内には、高速なlazy fetch操作を実現するために、lazy fetchされたデータ用のcache であるLazy Fetch Buffer （LFB）が用意されている。単一化時にSMに格納されているデータが必要となり、lazy fetchが行なわれると、まずLFBの中が調べられ、見つからない場合にはLazy Fetch Network（LFN）を通してSMに格納されているデータの転送が行なわれる。構造データをSMに格納する場合には、SM中のどのアドレスにデータが格納されたかを示す返答を待っているとUPの処理速度が低下するため、UPにはAddress Trans. Network（AN）によってSMの空きアドレスがあらかじめ供給されている。UPはこの空きアドレスを用いてSMに対しデータの格納要求を出す。DNはIU間でのGFの分散、およびDefinition Structure Memory（DSM）からDMへのDTの転送に用いられる。DTは、まずDSMにロードされ、DMからの要求に応じて各DMに転送される。Command Networkは、コマンドの転送に用いられる。

lazy fetch操作が行なわれている間、UPは処理待ちとなるため、LFNの遅延時間は小さくなくてはならない。また、SMへのアクセス頻度が大きいとlazy fetch操作の時間が増加する。3．3で述べたシミュレーション結果より、レベル－1システム内のIU台数は16台程度が適当である。

レベル－2システムのネットワークは、DNとCNからなる。レベル－1システム間でのGF、およびコマンドのトラヒックはそれ程大きくないと考えられるので、レベル－1システム64台程度でレベル－2システムを構成することを考えている。図4にレベル－1システムとレベル－2システムのインタフェース部の構成を示す。

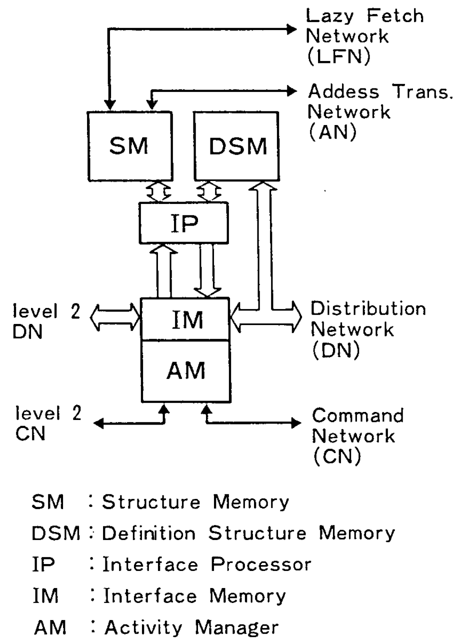

Fig.4 The Interface of level-1 and level-2 system

5．構造メモリ

PIE－IIでは、SMを用いて、構造データの共有を行なう。ground instance のGF間での共有方式については、既にシミュレーションが行なわれている［5］。さらに、ground instance 以外の構造データの共有方式［8］についても、現在検討中である。構造データの共有を行なう目的は、構造データをSMに格納することによって、GFから切り離し、GFの長さを抑え、縮退時間の短縮を図ることにある。

ここでは、構造データではなく、単一化に直接関係のないリテラルをSMに格納することによって、GF長を短縮し、縮退時間を短くする方法について述べる。

以下、例題の説明において、幾つかのシステム述語を導入するが、それらは、そのGFを実行するときにシステムがすべき処理を記述するために用いたものであり、実際にPIE－II上においてそれらの述語の実装を考えている訳ではない。それらのシステム述語のうち、#で始まるものは、UP内のみで処理される述語を、$で始まるものはSMに対してデータの書き込みおよび読み出しを行なう述語を示す。&で始まるものは、SMのアドレスを示す。例題として次のプログラム、nrevを考える。

```
nrev([X|L0], L) :-
        nrev(L0, L1),
        app (L1, [X], L).
nrev([], []).
app ([], X, X).
app ([X|A], B, [X|C]) :-
        app (A, B, C).
```

このプログラムに次のGF①が与えられたとする。

?－nrev（[a，b，c]，X）．　①

このとき、左側のリテラルから単一化を行なうとすると、GF①は次のように書き換えられる。

?－nrev（[b，c]，_0），
app（_0，[a]，_1）． ②

ここで、_0と_1はGF内における変数の通し番号である。このGFは、更に次のように書き換えられる。

?－nrev（[c]，_0），
app（_0，[b]，_1），
app（_1，[a]，_2）． ③

GF③において、3つ目のリテラルapp（_1，[a]，_2）は、変数_0を含まないため、GF③の第一リテラルnrev（[c]，_0）の単一化の際、その構造は変らない。縮退によって変数の通し番号が変るだけである。そこでGF②の縮退時に、このリテラルをSMに格納し、それ以降の縮退時間を短縮することを考える。リテラルをSMに格納すると、GF③は次のようになる。

▶ GF

?－nrev（[c]，_0），
app（_0，[b]，_1），
$lsave（&addr0，_1）．

▶ SMに格納された部分

&addr0→app（_0，[a]，_1）．

$lsave はGFの一部がSM上のアドレス$addr0に格納されていることを示すシステム述語である。&addr0は、ANによってUPに供給されたものである。第二引数以降は、SM中に格納された部分とGFとの共有変数の対応関係を示す。例えば、第二引数はSMに格納された部分の変数_0に、第三引数は_1に対応する。但し、SMに格納された部分の変数番号は、0番からに付け直されており、共有変数でない変数に対しては、共有変数より大きな番号が付けられている。この例では、GF③の第三リテラルの変数_1がSMに格納された部分の変数_0に対応していることを示している。GF③における変数_2は、共有変数ではないので、$lsave の引数とはならず、SMに格納された部分では、_1に書き換えられている。例えば、GF

?－p（_0），q（_0，_1，_2，_3），
r（_1，_4，_3）．

において3つ目のリテラル r（_1，_4，_3）が、SMに格納されると、次のようになる。

▶ GF

?－p（_0），q（_0，_1，_2，_3），
$lsave（&addr，_1，_3）．

▶ SMに格納された部分

&addr→ r（_0，_2，_1）．

GF②が、更に単一化／縮退され、次の単一化に直接関係のない部分がSMに格納されると、GFおよびSMに格納されている部分は次のようになる。

▶ GF

?－nrev（[]，_0），
app（_0，[c]，_1），
$lsave（&addr1，_1）．

▶ SMに格納された部分

&addr0→app（_0，[a]，_1）．
&addr1→app（_0，[b]，_1），
$lsave（&addr0，_1）．

このGFは、次の単一化／縮退で次のようになる。

?－app（[]，[c]，_0），
$lsave（$addr1，_0）．

このGFが単一化されると、変数_0の値は、[c]となる。引き続き行なわれる縮退において、システム述語$lsave がGFの先頭リテラルとなるので、lazy fetch操作によってSMからリテラルのlazy fetchが行なわれ、縮退後のGFは次のようになる。

?－app（[c]，[b]，_0），
$lsave（&addr0，_0）．

次に、このGFは、

?－app（[]，[b]，_0），
$lsave（&addr0，[c｜_0]）．

となり、更に次の単一化で変数_0に[b]が結合され、[c｜_0]→[c，b]となる。この値がSM上に格納されているGFの変数_0に対応しているので、その縮退操作でlazy fetchが行なわれ、新GFは、

?－app（[c，b]，[a]，_0）．

となる。このGF中の変数番号_0は、縮退操作による変数番号の付け換えによって生じたものである。

以上、リテラルをSMに格納することによってGF長を短縮する方式について述べた。この操作は、縮退時におけるGFとSMに格納されるべき部分の切り分け、およびSMへのリテラルの書き込み、読み出しのオーバーヘッドを伴なうため、どのような場合に行なうべきかシミュレーションを行なって決定したい。また、次章で述べるAND並列を導入した場合にも、ACによる推論木を用いたサブゴールの制御などの手間が増えるものの、縮退に要する時間は短縮されるため、どちらの方法がより効率的であるか慎重に評価する必要がある。

6．BagofとAND並列

PIE－IIにおける bagofとAND並列の実行メカニズムを考えるために、まず次のような例題を考える。

?－ p（X），q（Y），r（X，Y）．

この場合、サブゴール?－ p（X）．と?－ q（Y）．は共有変数を持たないので、並列に実行することができる。この間、サブゴール?－ r（X，Y）．の実行は保

留されている。サブゴール？－ｒ（Ｘ，Ｙ）．を起動するタイミングとして、次の２つが考えられる。
①各サブゴール？－ｐ（Ｘ）．と？－ｑ（Ｙ）．の実行が終了し、全ての解が求まってから？－ｒ（Ｘ，Ｙ）．を起動する。
②各サブゴール？－ｐ（Ｘ）．と？－ｑ（Ｙ）．が新たな解を見つけるごとにサブゴール？－ｒ（Ｘ，Ｙ）．を起動する。

①の方法は

？－bagof （Ｘ，ｐ（Ｘ），Ｓ），
　bagof （Ｙ，ｑ（Ｙ），Ｒ），
　#cprod （Ｓ，Ｒ，Ａ，Ｂ），ｒ（Ａ，Ｂ）．

と等価である（但し、#cprod （Ｓ，Ｒ，Ａ，Ｂ）は、リストＳとＲの要素のすべての直積をつくり出す述語であるとする）。従って、①はいかにしてbagof を実現するかという問題に帰着される。②は所謂ストリーム並列であり、基本的には①のbagof と同様な処理方式で実現できるが、サブゴール？－ｐ（Ｘ）．と？－ｑ（Ｙ）．が新たな解を見つけるごとに、？－ｒ（Ｘ，Ｙ）．にそれを引き渡し、起動するメカニズムとその解を用いた部分的な直積を作る方法が問題となる。

一般に並列処理の環境下でbagof 及びＡＮＤ並列を実現しようとする場合、

（１）サブゴール間の解の受け渡し
（２）サブゴール間の制御（activate　等）
（３）解の直積の生成

等が問題となる。

ＰＩＥ－Ⅱでは（１）はＳＭを用いて、（２）はＡＣおよび推論木を用いて実現することができる。（３）については、ＵＰを用いる場合とＳＭを用いる場合が考えられるが、ＳＭに対する負荷を考えるとＵＰを用いるのが適当であると思われる。

②の方法を用いたＡＮＤ並列の実現方式を述べる前に、ＳＭを用いた解の受け渡しおよびＡＣと関係木を用いたサブゴール間の制御等、類似した所が多いbagof の実現方式について述べる。

６．１　bagof の実現方式

bagof の実現方式について考えるために、まず次のような簡単な例題を考えることにする。

例１

？－ bagof（Ｘ，ｐ（Ｘ），Ｓ），ｑ（Ｓ）．　①

このＧＦは次のように前処理される。

？－（ ｐ（Ｘ）＆　$app （［Ｘ］，Ｓ））／／
　ｑ（Ｓ）．　②

②において、＆は逐次処理されることを示し、／／は実行時にＡＮＤ分割されることを示す。ＳはＳＭ上のアドレスを示す変数であり、このＧＦの実行時に、既にＡＮによりＵＰに供給されているＳＭの空きアドレスが代入される。$app （Ｘ，Ｓ）はＳＭに対する操作を示すシステム述語でありＳＭ上のアドレスＳにＸの値をappend することを示す。ＳＭのアドレスＳに格納されている値が［ａ，ｂ］であるときに、$app （［ｃ］，Ｓ）が実行されるとＳに格納されている値は［ａ，ｂ，ｃ］になる。ＧＦ②は次のようにして実行される。

（１）ＧＦ②は実行時に、次のように２つのサブゴールにＡＮＤ分割される。

？－ｐ（Ｘ），$app （Ｘ，&addr）．　③
？－ｑ（&addr）．　④

③、④において&addrは、実行時に変数Ｓに実際に割り振られたされたＳＭ上の空きアドレスを示す。

（２）まずサブゴール③を実行する。この時サブゴール④の実行は保留されている。
（３）サブゴール③は実行が進むに従って見つけた解を&addrにappendしてゆく。
（４）サブゴール③の実行が終了したことが推論木上で、success コマンドによって確認されるとＡＣは保留されていたサブゴール④を起動する。
（５）サブゴール④はlazy fetch操作を用いて&addrに格納されている解の集合を得て、述語q を実行する。

例２

次に、もう少し複雑な例として、ＧＦ⑤について考える。

？－ bagof（Ｘ，Ｘ likes　Ｙ，Ｓ），
　ｐ（Ｙ，Ｓ）．　⑤

このとき、述語likes に対する定義が以下のようであったとすると、ＧＦ⑤のbagof に対する解は次の２つとなる。

▶ 述語likes の定義
　dick likes beer .
　tom likes beer.
　bill likes cider.
　tom likes cider .

▶ ⑤のbagof に対する解
　Ｙ＝beer,　Ｓ＝［dick, tom ］；
　Ｙ＝cider ，Ｓ＝［bill, tom ］

⑤のbagof に対して上記のような２つの解を得るためには、サブゴール？－Ｘ likes　Ｙ．の解をＹの値に応じて分類しなくてはならない。

このためには、次のような方法が考えられる。

（１）サブゴール？－Ｘ likes　Ｙ．を逐次モード（ＧＦの単一化を行ないＯＲ関係にある子ＧＦが複数生成されたときに、ある１つの子ＧＦ以外の子ＧＦの実行を保

留しておくことによって実現される。実行が保留されているGFは必要に応じて起動される）で実行し、解が得られるごとにそのYの値に応じて？－X likes Y．を解きなおす。この方法では、Yの値に応じて解の分類を行なう必要がないものの、サブゴールを２回解くことになるため、効率的であるとは言えない。

（２）サブゴール？－X likes Y．の全ての解を求め、それらをSM中に格納する。このとき、どのモジュールを用いて解の分類を行なうかが問題となるが、

（a）UP
（b）SM

の２つが考えられる。

まず、UPを用いて解の分類を行なう場合について考える。GF⑤は実行時に次のようにAND分割される。

```
?－X likes Y,
    $app ((key(Y), X), &addr).          ⑥
?－#kcluster(&addr, Y, S),
    q (Y, S).                           ⑦
```

#kcluster(&addr, Y, S)は&addrに格納されている値を、key の値に応じて分類するシステム述語である。Yはkey の値であり、Sはkey に応じて分類された結果であるとする。サブゴール⑥と⑦の実行手順は例１の場合と同様である。サブゴール⑥の実行終了後、&addrの値は、

```
&addr→ [ ( key(beer), dick),
         ( key(beer), tom ),
         ( key(cider ), bill),
         ( key(cider ), tom ) ]
```

となる。但し、実際にSM中にこのような形でリストが作られる訳ではなく、

```
&addr→ [&addr0,&addr1,&addr2,&addr3 ]
&addr0 → ( key(beer), dick)
&addr1 → ( key(beer), tom )
&addr2 → ( key(cider ), bill)
&addr3 → ( key(cider ), tom )
```

といった形（実際には、更に複雑な形となる）で格納される［６］。

サブゴール⑦が起動されると、#kclusterが実行され、その結果次の２つのGFが実行される。

```
?－ q(beer, [dick, tom ]).
?－ q(cider , [bill, tom ]).
```

この様子を図５に示す。

次に、SMを用いて解の分類を行なう場合について述べる。GF⑤は、次のようにAND分割される。

```
?－X likes Y,
    $kcluster(&addr, key(Y), X).        ⑧
?－#member([Y, S], &addr),
    q (Y, S).                           ⑨
```

$kclusterは、XをSMに格納するときに、key Yの値に応じて分類することを示すシステム述語であり、#memberは&addrの要素を１つずつ取り出すことを示すシステム述語である。サブゴール⑧と⑨の実行手順は、解の分類にUPを用いた場合とまったく同様である。サブゴール⑧の実行後、&addrの値は、

```
&addr→ [[beer, [dick, tom ]],
        [cider , [bill, tom ]]]
```

となる。この後サブゴール⑨が実行され、その結果

```
?－ q(beer, [dick, tom ]).
?－ q(cider , [bill, tom ]).
```

の２つのGFが実行される。

以上、例１と例２を用いて、bagof の実現方式について述べた。例２では、key の値が１つだけであったが、述語bagof に含まれる自由変数が複数ある場合には、その数に応じて、key の引数を増やせばよい。UPを用いて解の分類を行なうか、SMを用いて解の分類を行なうかについては、各モジュールの負荷を考えて決定すべきである。

６．２　AND並列の実現方式

次に、AND並列の実現方式について考える。AND並列を導入することによって、OR並列性の小さな問題についても処理速度の向上が期待できる。

AND並列では、本章のはじめに述べた問題点以外に無矛盾性検査をどうするかという問題がある。無矛盾性

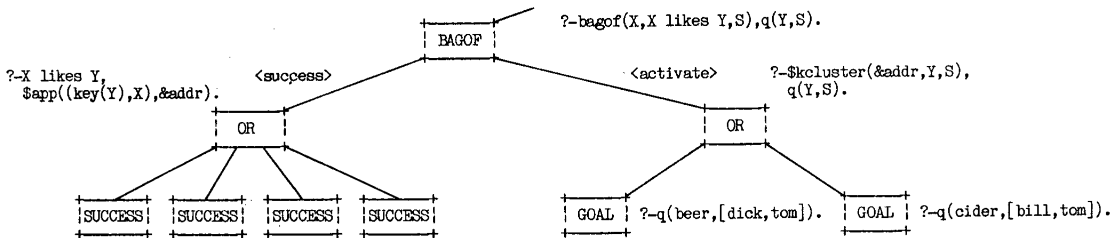

Fig.5 Execution mechanism of Bagof

検査の手間は一般に大きいため、共有変数を持つリテラルについては、そのリテラルに含まれる変数の関係によってリテラルの実行順序を制御すると有効であると考えられる。しかし、

?－p（X），q（X）．

のようなGFを考えた場合、各サブゴール？－p（X）．と？－q（X）．の見つけ出す解の個数が十分少なければ、無矛盾性検査の手間もそれ程大きいとは言えず、並列性を有効に生かすために無矛盾性検査を行なうことも考えられる。しかし、そのような場合を自動的に検出することは困難なので、無矛盾性検査が有効であると考えられる場合については、現在のところ

?－ p(X)， q(Y)，X＝＝Y.

のようにプログラマに陽に記述してもらうことを考えている。

各リテラルの実行順序を決定するためには、実行中に変数の結合関係が動的に変化するため、静的な解析だけでは不十分であり、単一化が行なわれるごとに変数の結合関係を調べ直す必要性がある。しかし、この操作の手間はかなり大きくなる可能性がある。実行時にどの程度まで変数の結合関係を調べるかは、それによって検出されるAND並列性による処理速度の向上とのトレードオフとなるため、十分に検討する必要性がある。

例1

AND並列の実現方式を考えるために、次のような簡単な例題を考える。

?－ p(X)， q(X)．　①

このGFを並列に実行するときの手順を以下に示す。

（1）GF①のサブゴールである？－ p(X)．と？－q(X)．は共有変数Xを持つため、同時に実行を開始することはできない。このような場合は、どちらかのサブゴールをproducerとみなし、他方をconsumerと見なすことによってストリーム並列を行なう。サブゴール？－p(X)．をproducerと見なすと、GF①は実行時に、次のようにAND分割される。

?－p（X），＄app（X，&addr）．　②

?－＃exec(q（&addr［N］））,
　N：＝N＋1，
　＃regererate.　③

&addr［N］は&addrに格納されているリストのN番目の要素を示す。変数Nの値は0に初期化されているものとする。＃execは、いわゆるcallとほぼ同様であるが、引数として与えられたGFを並列に実行する点が異なる。＃regenerateは、そのGF自身を複製し推論木上の同一ノードの同じ枝に繋ぐことを示すシステム述語である。サブゴール③の実行前の変数Nの値が0であったとすると、サブゴール③が実行されると次のように2つのGFが作られる。

?－q（&addr［0］）．

?－＃exec（q（&addr［N］）），
　N：＝N＋1，
　＃regenerate.　④
　（但し、N＝1）

このとき、サブゴール④は、サブゴール③と推論木上の同一ノードの同じ枝に繋がれる。

（2）サブゴール②はbagof のときと同様に、見つけた解を次々とSM上のアドレス&addrにappendしてゆく。bagof のときと異なる点は解が1つ見つかるごとに推論木上で解が見つかったことを示すコマンドfind-answerが伝達され、サブゴール③（実際には＃regenerateにより③が複製されたもの）が起動されることにある。

（3）上記のようにサブゴール③は、起動されるごとに、次々と新しく見つかった解に対して、GF？－q．を実行してゆく。

この例はconsumerが1つの場合であるが、複数の場合には、consumerごとにそれぞれに対応するサブゴール③が作られ、同じANDノードに繋がれ、解が新たに求まるごとに、それぞれが起動される。

例2

例1では、producerが1つであったために、consumerに対して解の直積を作る必要がなかった。今度の例では、1つのconsumerに対して複数のproducerが存在する場合について考える。

?－ p(X)， q(Y)， r(X，Y)．　⑤

この例では、サブゴール？－ p(X)．と？－ q(Y)．がproducerになり、サブゴール？－ r(X，Y)．がconsumerとなるものとする。このとき、producerであるサブゴール？－ p(X)．および、？－ q(Y)．が見つけた解X，Yの直積を作り、consumerであるサブゴール？－ r(X，Y)．をその解の直積の全てについて実行する必要がある。例えば、XおよびYの解が

X＝a，b
Y＝x，y

であるときには、解の直積

X，Y＝a，x
　　　a，y
　　　b，x
　　　b，y

の4つの組み合わせについて，？－r（X，Y）．を実行しなくてはならない。このとき、例1で述べたように、それぞれの解の組み合わせについてGFを複製することによって、全ての解の組み合わせに対してconsumerとな

るサブゴールを実行させる方法をとる。
解の直積を作り出す方法としては次のものが考えられる。

（a）推論木を用いる方法

新たな解が求まるごとに推論木を広げて、各ノードを解の組み合わせの1つ1つに対応させることによって解の直積を作り出すことができる。この方法では、推論木の構造が複雑になり、操作の手間が増えるとともに、新たに生成された解の組み合わせに対応する複数のGFがSM上の同一アドレスにアクセスすることになるので、好ましくない。

（b）UPを用いる方法

UPが解の直積を生成する場合は、GF⑤は実行時に、次のようにAND分割される。

```
?－p （X），$app （X，&ax）．                    ⑥
?－q （Y），$app （Y，&ay）．                    ⑦
?－$mkcprod （&ax，N，&ay，M，
                              r（_，_）），
    N：＝$ln（&ax），
    M：＝$ln（&ay），
    #regenerate.                                 ⑧
```

```
$mkcprod(&ax, N, &ay, M, G)
SMaddr &ax, &ay;
int N, M;
GOAL G;
{
    int i, j;

    for (i = N; i < $ln(&ax); i++)
        for (j = 0; j < M; j++)
            #exec( ?-G(&ax[i], &ay[j]). );
    for (j = M+1; j < $ln(&ay); j++)
        for (i = 0; i < $ln(&ax); i++)
            #exec( ?-G(&ax[i], &ay[j]). );
}
```

Fig.6 system predicate <$mkcprod>

$mkcprod は解の直積に対してconsumerであるサブゴールを実行するためのシステム述語である。その機能を図6に示す。M，Nは、0に初期化されているものとする。$lnはリストの要素の数を表わす。他のシステム述語は今までに述べたものと同様である。

これらのサブゴールは、次のようにして実行される。

（1）サブゴール⑥と⑦は、解を見つけるごとに、解をそれぞれ&ax，&ayにappendし親ノードに対してコマンド find-answerを送る。サブゴール⑧は、find-answerコマンドを受けとったANDノードによって起動される。この時、ANDノードはそのconsumerに対するproducerである全てのサブゴールが少なくとも1つ解を見つけるまでは、そのproducerであるサブゴールを起動しないものとする。この例では、サブゴール⑥と⑦が、それぞれ1つ以上の解を見つけるまで、サブゴール⑧は、起動されない。

（2）起動されたサブゴール⑧は、$mkcprod を実行するために、&ax，&ayに対してlazy fetchを行なう。この間にサブゴール⑥，⑦が、更に新たな解を見つけていれば、その解が既に&ax，&ayに格納されており、その解に対応するfind-answer コマンドは、推論木上を転送されているか、既にANDノードに到着しており、サブゴール⑧が処理中であるため、ANDノードにおいて処理待ちとなっている。サブゴール⑧は、それらの解に対する組み合わせについても、?－r （A，B）．を実行する。このため、#regenerateによって、そのANDノードに戻ったサブゴール⑧が起動されても、1つも?－r （A，B）．が実行されないことがあり得る。図6で、lazy fetchの対象を&ax［N＋1］まで等に限れば、このようなことはなくなる。処理効率上からは前者の方が適当であると考えられる。

この例において、&ax，&ayの値が

&ax→［a，b］

&ay→［x，y］

であり、サブゴール⑧中のN，Mの値がそれぞれ1であ

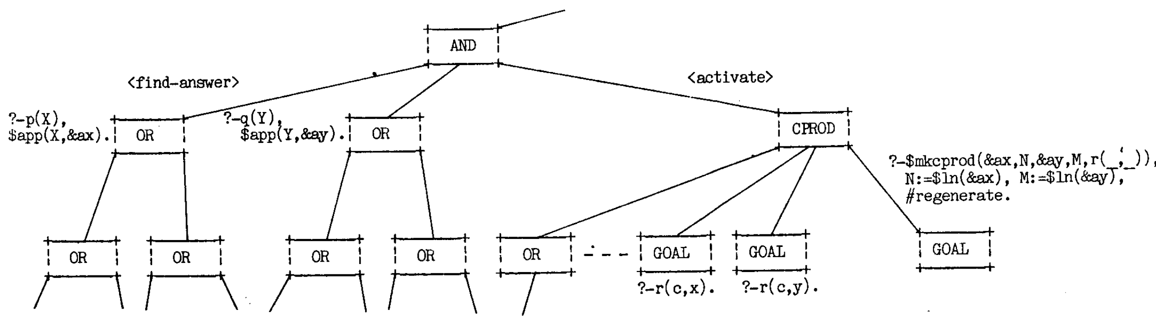

Fig.7 Execution Mechanism of AND Parallel

るとする。このとき、サブゴール⑥が新たな解X＝c を見つけると

&ax→［a，b，c］

となる。サブゴール⑧が起動され、サブゴール⑧が&ax, &ayに対してlazy fetchを行なうまでに、新たな解が&ax, &ayにappendされていなければ、

？－ r(c，x)．

？－ r(c，y)．

が実行される。この様子を図7に示す。

（c）SMを用いる方法

SMを用いて解の直積を作る場合は、基本的にはUPを用いた場合と同じである。SMを用いた場合にはGF⑤は、次のように分割される。

？－p (X)，$mkcprod (&addr, 0, X)．

？－q (Y)，$mkcprod (&addr, 1, Y)．

？－$execcprod (&addr, N, r(_, _))，

N：＝$ln(&addr)，

#regererate. ⑨

$mkcprod は、SM上のアドレス&addrに第二引数の値により第三引数の直積を作る述語である。$execcprod は、$mkcprod によって作られた解の直積に対してconsumer であるサブゴールを実行する述語であり、その機能を図8に示す。UPを用いた場合と同様な例を用いるならば、SM上には、次のように解の直積が作られる。

&addr→［［a，x］，［a，y］，

［b，x］，［b，y］］

他のシステム述語は、今までに述べたものと同じである。変数Nは0に初期化されているものとする。

これらのサブゴールを実行する手順は、上記のUPを用いて解の直積を作る場合と全く同じであるので、ここでは省略する。

以上、3種類の解の直積の作り方について述べた。これらの中では、SMを用いたものが最も簡単であり、lazy fetchによる転送量も少ないが、直積を作る操作は、かなりSMの負荷を重くすることが考えられるので、UPを用いる方法が適当であると思われる。

```
$execcprod(&addr, N, G)
SMaddr &addr;
int N;
GOAL G;
{
    int i;

    for (i = N; i < $ln(&addr); i++)
        #exec( ?-G($car(&addr[i]),
                   $cadr(&addr[i])). );
}
```

Fig.8 system predicate <$execcprod>

7．PIE第二次モデルのシミュレーション

PIE－Ⅱは図2に示したようにレベル－1システムとレベル－2システムとからなる。以下、それぞれのシステムのシミュレータについて述べる。

7．1 レベル－1システムシミュレーション

現在、レベル－1システムのシミュレータを作成中である。レベル－1システムは、基本的にはPIE－Ⅰと同様の構成であるが、SMが導入されている点が大きく異なる。レベル－1システムのシミュレータの主な評価対象について以下に述べる。

（1）AND並列の実現方式についての評価検討

AND並列の実現方式については6章で述べた。今後、AND並列の実現方式についてシミュレーションを通して評価する予定である。同様に、bagof の実現方式についても評価する予定である。

（2）UPの処理速度の向上

UPの処理速度を向上させるためには、次の2つの方法が考えられる。

（a）GF長を抑え縮退時間の短縮を図る。

これについては、構造データのSMへの格納、5章で述べたリテラルのレベルでのSMへの格納、および6章で述べたAND並列の導入などが考えられる。

（b）複数台のunifier ／reducer の設置

現在まで、unifier,reducer が各IU内に1台のモデルについてシミュレーションを行なってきた。しかし、単一化に比べ縮退に時間がかかるため、1台のunifier に対して複数台のreducer を、または複数台のunifier と更に多くのreducer を設けることによって処理速度の向上を図ることができる。今後、シミュレーションによってunifier とreducer の台数を決定する予定である。

（3）SMの構成についての評価検討

現在までに行なった構造データの共有方式についてのシミュレーションは1台のIUを対象としたものであり、AND並列について考慮されていなかった。よって、このシミュレーションによって実際に複数台のIUを稼動させたときのデータ、およびAND並列の導入によって増加するSMへのアクセス頻度等のデータを取り、SMの構成を決定するための目安とする。また、構造データの共有方式によってSMに要求される機能が大きく異なるため、それらについても評価を行なう予定である。

（4）各ネットワークの構成についての評価

レベル－1システムには、次の4つのネットワークがある。

（a）Distribution Network（DN）

（b）Lazy Fetch Network（LFN）

(c) Command Network (CN)
(d) Address Trans. Network (AN)

これらのネットワークは現在設計中であり[9]、レベル−1システムで実際に論理型言語を実行させて評価を行なう予定である。

(5) アクティビティ制御方式の評価検討

アクティビティ制御方式については既に評価を行なったが[4]、AND並列の導入等によって制御方式が若干変わるため、それらについて評価を行なう予定である。

(6) レベル−1システムでの負荷分散

現在までのシミュレーション結果より、負荷の状況によって動的に負荷分散のストラテジを変化させることがよいと考えられる。シミュレーションによって、どのように変化させるべきかを調べる。

(7) レベル−1システムにおけるIUの台数

構造データの共有についてのシミュレーションより、レベル−1システムのIU台数は、16台程度が適当であると考えられている。しかし、この台数は、各ネットワークのトラヒック、およびSMへのアクセス頻度によって決定されるべきであり、シミュレーションによって適当な台数について検討する。

(8) レベル−1システムの性能評価

上記の項目について評価検討を行ないながら、レベル−1システムによって実現され得る性能について評価を行なう。

以上、レベル−1システムのシミュレーションの主な評価対象について述べた。以下、シミュレーションモデルについて簡単に述べる。

(1) IU

IU内の各モジュールについて以下に述べる。

(a) DM

DMに要求される機能は

①高速なDTの絞り込み

②DTのUPへの高速な転送

の2つである。①については、現在、処理方式を検討中である。②については、DMを多バンク化してDTを各バンクに適当に配置することによって実現することが考えられる。

(b) UP

UP内の構成について考えるとき、既に述べたようにunifer, reducer の台数が問題になる。AND並列、およびリテラルの共有等の導入により、GFの長さが30cell程度(ただし、ヘッダ等を除く)に抑えられるならば、単一化時間のlazy fetch操作による増加を考慮に入れ、単一化と縮退時間の処理時間の比は、1対2〜4程度になると考えられる。この比率から考えてunifier 1台に対してreducer を2〜4台設ければよいことになる(ハードウェア的には、unifier とreducer は、それほど大きく異なる訳ではないので、複数台のunifier /reducer の組を設置することも考えられる)。このときのGFの処理の流れを図9に示す。図9においてGFは、まずMMからcontrollerに転送される。ついでDMによって単一化が行なわれるリテラルに対応するDTの絞り込みが行なわれる。絞り込まれたDTはDMから1つずつunifier に転送される。同時にcontrollerからunifierにGFが転送される(DTの絞り込みと同時に転送することも可能である)。unifier はDTとGFが揃うと単一化を開始し、その結果をreducer に転送する。reducer によって縮退が行なわれ新しいGFが作られる。それがI/Fに送られ、MMもしくはDNに送られる。このとき、このGFを直接controllerに戻し処理の高速化を図ることができる。

図9は、各モジュール間で、GFを転送する場合のものであるが、実際には、多数のGF用のメモリを用意し、各モジュール間で、アクセスするメモリを切り替えることによって、unifier とreducer の間、およびreducer とI/Fとの間のGFの転送を取り除くことができる。

図9に示した方法に対し、図10に示したようなモデルを考えることができる。図10では、controllerは各unifier /reducer の組に対してGFを同時に転送する。一方DMは多バンク化されており、unifier /reducer に同時にDTが送られてくる。DTの数が、unifier /reducer の数より少ない場合に、空いているunifier /reducer に、次の入力GFに対する処理を行なわせることも可能であるがその場合、controllerに要求されるunifier /reducer の制御はやや複雑なものになる。

以上、2種類のモデルについて述べた。両者を組み合わせたモデルも考えられる。UPの構成として、どのような構成を採用すべきか、シミュレーションによって評価する。

(c) MM

MMには高速なGFの転送が要求される。この処理速度としては、UPの処理速度にあわせてGFを送出し、かつUPおよびDNから送られてくるGFを格納できるものでなくてはならない。多バンク化等を含めて評価検討を行なう予定である。

(d) AC

ACには高速なコマンド処理が要求されるためコマンドの処理を行なう部分とコマンドの生成を行なう部分を分離したモデルを考えている[10]。

(2) SM

AND並列等を導入した場合のSMへのアクセス頻度についてはまだデータが得られていないため、SMにつ

いてはメモリへのアクセス時間を仮定するにとどめ、主にSMに要求される性能を調べることを目的とする。
（３）ネットワーク
各ネットワークついては［９］を参照されたい。

７．２　レベル－２システムシミュレーション

レベル－２システムのシミュレーションについては、使用する計算機の記憶容量等の制約により、レベル－１システムのシミュレータのように実際に論理型言語を実行しながらデータを取ることは困難であるので、レベル－１システムのシミュレーションによって得られたデータを基にして、主としてレベル－１システム間の負荷分散について評価を行なう予定である。

８．おわりに

本報告では、ＰＩＥ－Ⅱの構成および、ＰＩＥ－Ⅱに導入する予定であるＡＮＤ並列の実現方式について述べた。今後、レベル－１システムおよびレベル－２システムのシミュレーションを行ない、ＡＮＤ並列等の実現方式、各レベルのシステム等の性能評価を行なう予定である。

《参考文献》

［１］湯原他，“A　Unify　Processor　Pilot Machine　for ＰＩＥ”，Logic　Prog．　Conf．ＩＣＯＴ　´８４
［２］小池他，“ＰＩＥの試作ＵＰの性能評価”，第２９回情処全大２Ｂ－６　´８４
［３］丸山他，“高並列推論エンジンＰＩＥ～並列度のシミュレーションとその評価”　ＥＣ８３－３９
［４］丸山他，“A　Preliminary　Evaluation of the Activity Control　Mechanism in ＰＩＥ”，Logic　Prog．Conf．　ＩＣＯＴ　´８４
［５］平田他，“高並列推論エンジンＰＩＥにおける構造データの効率的な処理方式について”，　ＥＣ８３－３８
［６］平田他，“ＰＩＥにおける構造メモリの構成について”，　アーキテクチャワークショップインジャパン．´８４　情報処理学会
［７］Moto-oka 他，“The　Architecture of　A Parallel　Inference　Engine －ＰＩＥ－”，ＦＧＣＳ´８４　ＩＣＯＴ
［８］平田他，“ＰＩＥにおける構造メモリの構成法”第２９回情処全大２Ｂ－３　´８４
［９］坂井他，“高並列推論エンジンＰＩＥにおける相互結合網の構成”，ＥＣ８４，本研究会
［１０］濱中他，“ＰＩＥのＡＣの設計概容”第２９回情処全大２Ｂ－２　´８４

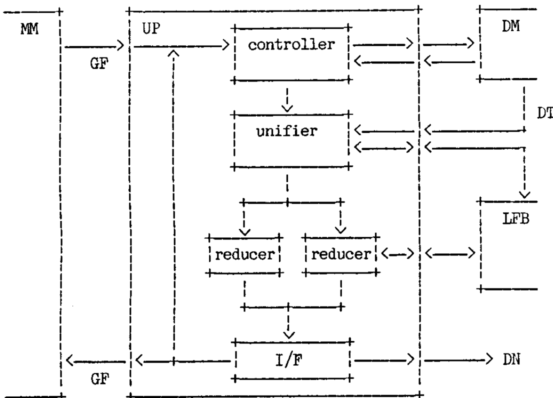

Fig.9 Simulation model of UP

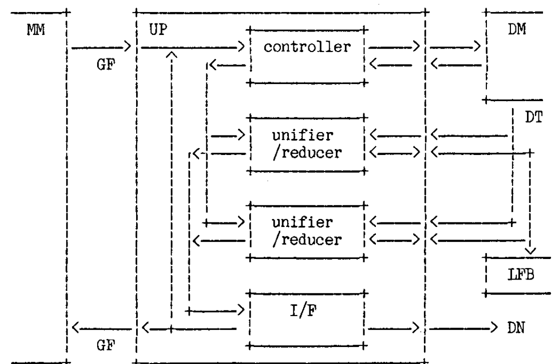

Fig.10 Simulation model of UP