TR-050

データフロー方式並列推論マシンのアーキテクチャ

伊藤徳義、益田嘉直、清水　肇
来住晶介（沖電気）
久野英治（沖電気）

1984. 3

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

データフロー方式並列推論マシンのアーキテクチャ

伊藤穂義　益田嘉直　清水肇　　来住晶介　久野英治
（（財）新世代コンピュータ技術開発機構）　（沖電気）

Ⅰ．はじめに

近年，一階述語論理をベースとした論理型言語Prologを実行する並列推論マシンの提案が各所で行われている．このPrologは単純であるが強力な記述能力を有している．また本質的に並列処理の可能性を秘めており，第５世代コンピュータの基礎言語として注目されている．

このPrologを並列推論マシンのためのプログラミング言語として考えた場合，いわゆる純Prologの機能だけでは不十分であり，並行プロセス等を記述するための能力が必要となる．後者の機能を実現する言語の１つとしてConcurrent Prolog［19］［20］が提案されている．純Prolog及びConcurrent Prolog のいずれも一階述語論理に基づくホーン節を基礎としているが，その狙いとするところはやや異なる．前者が与えられた問題の全解探索を行いたい場合に使用されるのに対して，後者は主として対象指向プログラミングとの整合性に主眼を置いている．

著者等はデータフローモデルに基づく並列推論マシンの研究を進めているが，このデータフローモデルは上述のような論理型言語との親和性が良くその並列処理を自然な形で実現できるという特徴がある．上記２つの言語は，その使用形態はかなり異なるが，いずれも基本機能である統一化処理においては共通の面を持つ．もう１つの基本機能である非決定性の面から見ると，純Prologに関してはこれに'don't know nondeterminism' の制御［６］［８］［11］が要求され，Concurrent Prolog に関しては'don't care nondeterminism' の制御［４］［13］［17］が要求される．著者等は本マシン上で，この２つの言語をサポートすることを検討してきたのでその詳細を報告する．

本稿では，まず純Prolog及びConcurrent Prolog の概要について述べた後，第３節で両言語を実行する際に重要となる変数の表現形式について述べ，統一化や非決定的制御機能を本マシン上で実現するためのプリミティブについて第４節で説明する．次に第５節で本マシンのアーキテクチャについて説明し，このマシン上で純Prologを実行した場合のシミュレーション結果について第６節で述べる．

Ⅱ．純PrologとConcurrent Prolog

純Prologプログラムは以下のような節(Clause)の集合から成る．なお後述するConcurrent Prolog におけるそれと区別するために，このようなガード棒を持たない節を特に非ガード節(Non-guarded Clause)と呼ぶ．

```
H1  <-  B1.
H2  <-  B2.
   ...
Hn  <-  Bn.
```

ここで，Ｈｉは頭部リテラルであり，Ｂｉは本体部である（但し，０≦ｉ≦ｎ）．記号'<-'は含意を表しており，その右辺が成立するならば，左辺が成立することを意味している．本体部はさらに任意個のリテラルから構成されるが，これは空であってもよい．本体部が空であるような節を単位節(Unit Clause) と呼んでいる．

統一化処理は，これらの節の集合に対して質問に相当するゴールリテラルが与えられたときに起動される．このとき，与えられたゴールリテラルの述語と頭部リテラルの述語が同一である節が統一化処理の対象となる．このような同一述語を有する節の集合をその述語に対する定義と呼んでいる．なお，ゴールリテラルを含むゴール文はその頭部が空である節として表現される．

あるゴールリテラルGが与えられたとき，Gの述語を有する定義が呼出され，その中の任意の節が選ばれてGとその節の頭部リテラルＨｉとの統一化が試みられる．一般に，定義中に複数の節が存在すれば，GとＨｉとの統一化は全ての節に対して並列に実行可能である．この並列性はＯＲ並列性と呼ばれている．

上記の統一化が成功した節は，それが単位節であれば，Gに対して統一化の結果（解）を返すし，そうでなければ，節の本体部を新たなゴールとみなして次の統一化処理を起動する．即ち，本体部のリテラルを新しいゴールリテラルと見做してその定義を呼出す．

このとき本体部中に複数のリテラルが存在すれば，それらはＡＮＤの関係にある．即ち，全てのリテラルの解が存在し，しかもそれらの解の間に矛盾がないとき本体部は満足される（本体部の解が存在する）．このＡＮＤ関係にあるリテラルも上述のＯＲ並列実行の場合と同様に，並列に実行制御することができるが，一般には，このＡＮＤ並列実行はいたずらに探索空間を広げたり，リテラル間の無矛盾性検査のオーバヘッドを大きくする恐れがある．ここではＡＮＤ関係にあるリテラルの間を逐次的に実行するか又

は並列に実行するかを指定するためのオペレータを導入し，そのいずれを使用するかはプログラマの判断に委ねるものとする．

本稿では，記号'&' はその両辺のリテラルを左から順に実行するように指定するオペレータであり，記号'//'はその両辺のリテラルを並列に実行するオペレータであるものとする．

これに対してConcurrent Prolog プログラムは以下のガード節(Guarded Clause)の集合で与えられる．

```
H1  <-  G1  |  B1.
H2  <-  G2  |  B2.
        ...
Hn  <-  Gn  |  Bn.
```

ここでＨｉは頭部リテラル，Ｇｉはガード部，Ｂｉは本体部である（但し，０≦ｉ≦ｎ），Ｇｉ及びＢｉは純Prologの場合と同様に任意個のリテラルから構成され，それらは空であってもよい．記号"｜"はガード棒(Guard Bar) と呼ばれ，DijkstraのGuarded Command と同様な排他的制御を行う［７］．

上記プログラムにおいて，ガード棒を通過するまでの処理は純Prologにおける場合と全く同様である．即ちＯＲ並列及びＡＮＤ並列実行を実現することができる．このガード棒は一種の逐次制御オペレータと見做すことができ，ＯＲ並列実行環境下においては，ガード部までの統一化処理が最初に成功した節のみがその本体部の実行を行うと解釈することができる．このとき他に実行されている節の結果は棄却される．このような排他的実行制御は後述するようにガード制御オペレーションによって実現できる．

ガード棒のもう１つの役割は，ガードを通過した時点で頭部リテラルに現れた変数に対する結合情報を他のプロセスに対して公開することである．この機構は後述するようにプロセス間でメモリセルを共有させることによって，実現できる．

Concurrent Prolog のもう１つの特徴は変数に対して読出し専用タグ (Read Only Annotation) を付与することができることである．例えばゴール文

```
<-  gen-list(X) //  print(X?).
```

が与えられたとしよう．ここで，記号？は読出し専用タグを示している．このとき２つのリテラルgen-list(X) 及びprint(X?) は並行して実行される．ここでprint(X?) の定義として，

```
print([H|T]) <-  |  ...
```

が与えられた場合，print(X?) は変数Ｘが変数以外の項(Term)に結合されるまで起動されない．即ち，読出し専用タグの付与された変数を変数以外の項と統一化しようとするとき，その変数が変数以外の項に結合されるまで統一化の処理は待ち状態となる．ここで項とは，プログラムで処理の対象とする任意のデータを意味している．項には，変数，シンボル，数値のような非構造データと，リストやベクタのような構造データがある．本マシンでは基本的に各々のデータにデータタイプを示すタグを付与したタグ付きアーキテクチャを採用しており，これらデータの間の区別はデータタイプタグによって行なわれる．

上例の場合，gen-list(X) の呼出しによって変数Ｘがリストに結合されたときに，述語print の頭部の統一化処理が起動されて節の頭部との統一化は成功する．このように変数に結合された項を変数のインスタンスと呼ぶ．Ｘが変数やリスト以外の項に結合されたときは統一化は失敗する．

このような機構は後述するようにメモリセルに非同期待合せの機能を用意することによって実現できる．

Ⅲ．　変数の属性

節に現れる変数(Variable)は任意の項と統一化可能である．但し節内に同一変数が２度以上現れる場合，その変数に対して同一の項が結合されなければならない．並行して実行される統一化処理の間で同一変数が共有されるような場合，その変数を共有変数(Shared Variable) と呼ぶ．これに対して，節内で唯一度だけ現れる変数であるか，又は，２度以上現れたとしてもそれらに対する統一化処理が逐次的に実行されるような場合，その変数を単純変数(Simple Variable) と呼ぶことにする．

後述するように，本マシンではリテラルに含まれる引数間の並列処理を実現する．また，統一化しようとしている２つの引数がいずれも同一のデータタイプを有する構造データである場合，それらの部分構造間の統一化の並列処理も実現する．従って，１つのゴールリテラル内に同一変数が２度以上現れしかもその変数がグランド項(Ground Term)に結合されてなければ，共有変数として取扱われる．ここでグランド項とはいかなる変数も含まない項を指す．

統一化処理において，単純変数と任意の項の間の統一化が行われた場合，単純変数を項に置換して出力すればよいが，共有変数と単純変数以外の項の間の統一化が行われた場合，共有変数に対する置換情報を出力する．この置換情報を結合環境(Binding Environment) と呼ぶ．結合環境は共有変数とそれに結合された項のペアを要素とするリストで表現できる．この結合環境は他の並列実行される統一化

処理における共有変数に対する結合環境との間で矛盾がないか否かを検出するために使用される．

共有変数は，純Prologにおける場合と，Concurrent Prologにおいてガード棒を通過する前の統一化処理における場合では全く同様に取扱われる．但し，Concurrent Prologにおいては全ての定義節がガード節であるため，１つのゴールリテラルが与えられたとき定義内でガード棒を通過して共有変数に対して最終的な項を結合させるのはたかだか１つである．従って，共有変数をメモリセルとして表現することができる．このような共有変数を特に共有大域変数(Shared Global Variable)，又は単に大域変数(Global Variable)，と呼ぶことにする．前述のガード棒は大域変数を表すメモリセルに対する書込み指令であると解釈することができ，読出し専用タグはこのメモリセルに対する読出し指令と解釈することができる．

これに対して，純Prologにおける共有変数は共有変数の間の識別が可能なようにするための変数識別子によって表現する．即ち，全ての共有変数には互いに異なる識別子が与えられる．

Concurrent Prolog において，読出し専用タグの付与された変数は読出し専用大域変数(Read Only Global Variable)，又は単に読出し専用変数(Read Only Variable)，として表現される．この読出し専用変数は後述するように大域変数から生成され大域変数と同様にメモリセルへのポインタとして表現される．

Ⅳ．　統一化処理

あるゴールリテラルが与えられ，その述語に対する定義が与えられているとき，統一化の基本部分はゴールと定義中の各々の節頭部リテラルとの統一化処理を行うことにある．

この統一化処理を逐次的に行うか並列に行うかの選択は取扱う応用に依存する面がある．一般的にはこのレベルでの並列実行はあまり多大な効果をもたらさないとの報告が発表されているが［22］，本マシンでは以下の理由によりこのレベルでの並列処理を実現する．

本マシンでは基本的に構造データ共有方式を採用している．即ち，プロセス間で構造データを共有し，必要が生じた際（オンデマンドで）構造データの内容を参照する．この方式のメリットは構造データコピーのためのオーバヘッドを最小化できることや，構造データの不必要な重複格納を回避できることにある．例えば，自然言語処理におけるＤＣＧ［18］やＢＵＰ［16］のような複雑な構造データ操作の並列処理を実現しようとする場合に，一般に，リテラルの引数は複雑な構造データとなる．今，多数の処理要素(Processing Element)から成る並列推論マシンにおいて統一化の処理をそれぞれの処理要素に分散させようとした場合を考えたとき，統一化処理プロセス起動の度に構造データを各処理要素にコピーしようとすれば，各処理要素は構造データコピーのためにかなりの処理時間を費やされることになり，ネットワークのトラフィックも増加する．また，起動されたプロセスの数だけのデータ格納領域を確保しなければならない．ここで注目すべきであるのは，各プロセスにおける統一化の処理においては，一般に構造の一部しか参照しないということである．従って，それぞれのデータコピーのかなりの部分は無駄になる恐れがある．

構造データ共有方式におけるもう１つのメリットは，Concurrent Prolog における共有メモリセルの概念をそのままの形で実現できることである．

このような構造データ共有方式において問題となるのは，構造データのアクセス遅延である．即ち，並列処理効果を向上させるために，処理要素の数を増加しシステムを大規模にすると，構造データをアクセスするためのネットワーク遅延や，構造データアクセス競合に起因する遅延が増大する．従って処理要素は構造データのリモートアクセスが完了するまで待ち状態になる．

前述の統一化処理において複数の構造データのリモートアクセスを必要とする場合，処理要素は１つのアクセス要求に対する結果が戻る前に並行して他のアクセス要求を発した方が処理効率は良くなる．このような多重アクセスを行う場合，ネットワークトラフィックの状態や構造データの格納されているユニットまでの距離によって応答結果は必ずしも要求を発した順番には返らない．従って，個々のアクセス要求毎に識別子を付与して管理する必要が生ずる．

データフローモデルはこのような制御を自然な形で実現できるという特徴がある［３］．

ここでは，まず純Prologを実行するための基本プリミティブについて説明した後，Concurrent Prolog 特有の機能をサポートするためのプリミティブについて説明する．

４．１　純Prologのための基本プリミティブ

（１）　統一化基本プリミティブ

統一化の処理は，１つのゴールリテラルが与えられたときゴールリテラルと同一述語を有する定義にその引数を渡すことによって起動される．起動された定義側は渡された引数を節の頭部の統一化処理に渡す．ＯＲ並列実行環境下においてはこの節頭部との統一化処理は定義に含まれる節の数だけ並列に実行できる．

この引数間の統一化処理はunify オペレータによって行われる．このときゴールリテラル及び節頭部の引数が複数であれば，対応する引数の数だけ並列に実行される．このオペレータは共通インスタンス出力ポート及び結合環境出力ポートの２つの出力ポートを持っており，それぞれのポ

ートに，両引数の間の共通インスタンス及び共有変数に対する結合環境を出力する．統一化が失敗したならばこれら２つの出力はいずれも特殊シンボル'fail'となる．なお，共有変数に対する置換情報が存在しなければ結合環境は'nil' で表現される．

図１にこのオペレータの使用例を示す．同図は，本マシンの機械語に相当するデータフローグラフによって表現されており，以下に示すようなゴールリテラル及び節頭部リテラルが与えられた場合を示している．

```
        <-  p([a | X],b,c).
p([Y,b],Z,c) <- ...
```

各unify オペレータは述語ｐの３つの引数毎に用意され，それぞれ，ゴールリテラル及び節頭部リテラルの対応する引数間の統一化処理を行なう．このunify オペレータは再帰的に定義されており，上例の両リテラルの第一引数のように引数がいずれも構造データ（上例はリストを示している）である場合には，それらを部分構造に分解してその部分構造毎にunify オペレータを並列起動させる．なお，変数とは任意の項が統一化可能でありその共通インスタンスは与えられた項になることから，この例の第二引数におけるように，節頭部の引数が変数である場合には，同図破線で示したように，第二引数のunify オペレータは省略できる．

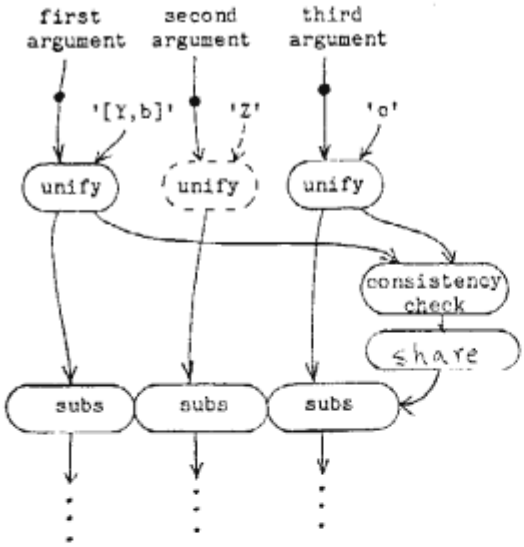

図１．　統一化処理のデータフローグラフ表現

ここで問題となるのは，それらの引数の間で互いに共有変数を持つ場合である．このようなときに，それぞれのunify オペレータによって共有変数に何らかの項が結合されたならば，それらの結合の間で矛盾がないか否かを調べるためのオペレーションを必要とする．この無矛盾性検査は同図におけるconsistency-check オペレータが行う．このオペレータは，矛盾がなければ共有変数に対する最終的な結合環境を出力するし，そうでなければ特殊シンボル'fail'を出力する．この無矛盾性検査処理において，入力した２つの結合環境内に同一共有変数に対する置換情報が含まれていれば，共有変数に結合されている項の間の統一化を試みるために，unify オペレータを呼出し，この統一化処理によって得られた最終共有インスタンスを結合環境の形式で出力する．得られた最終インスタンスに単純変数が含まれていればそれを共有変数に更新するためにshare オペレータが実行される．share オペレータについては後述する．

consistency-check オペレータの結果は前記unify オペレータによって得られた共通インスタンスに含まれる共有変数を置換するのに使用される．この置換はsubstituteオペレータによって行われる．

以下，ゴールリテラル又は節頭部リテラルに共有変数を含む場合及びそうでない場合に分けてこれらのオペレータの動作を説明する．

①　共有変数を含まない場合

たとえば以下に示すようなゴールリテラル，及び，定義節頭部リテラルが与えられた場合，データフローグラフは図２で示される．

```
<-  p(f(X),Y).
p(f(g(W)),g(b))  <-  ...
```

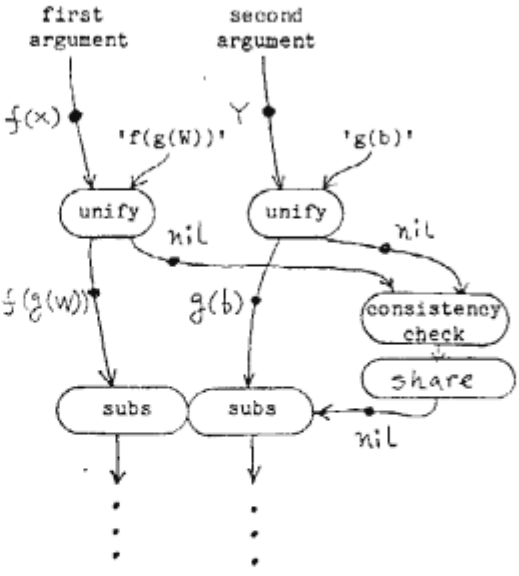

図２．　共有変数を含まない統一化処理

この例においてはゴールリテラル及び節頭部リテラルのいずれも共有変数を含んでなく，しかも統一化は成功するので２つの引数間の統一化を行うunify オペレータはその結合環境としていずれも'nil' を出力する．従って，consistency-check オペレータは結果として'nil' を出力し，substituteオペレータはunify オペレータの共通インスタンス出力ポートの結果をそのまま出力する．

これらの結果は節に本体部が存在すれば本体部に渡されるし，そうでなければ節の結果としてゴール側に返される（このオペレーションについては後述する）．

② ゴールリテラルに共有変数を含む場合

次に，ゴールリテラル側に共有変数を含む場合を考えよう．例えば，前述の定義節に対してゴールリテラル

```
<-  p(f(Z), Z).
```

が与えられたとする．両リテラルの第一引数の間のunify オペレータはその共通インスタンス及び結合環境として，それぞれ，f(Zs) 及びZs=g(W) を返す．これに対して第二引数に対するunify オペレータはそのインスタンス及び結合環境として，それぞれ，Zs及びZs=g(b) を返す．ここで添字ｓは変数Ｚが共有変数であることを示している．

この共有変数の生成はshare オペレータが行う．上述のようなゴールリテラルが与えられたとき，そのデータフローグラフは図３で示される．share オペレータはその入力オペランドが単純変数であるか又は単純変数を含む構造データであるとき，オペランド内の全ての単純変数を共有変数に更新する（なお，このとき生成された共有変数にはユニークな変数識別子が付与される．この識別子は共有変数相互を区別するために使用される），入力オペランドが上記以外であるとき，share オペレータは入力オペランドの内容をそのまま出力する．

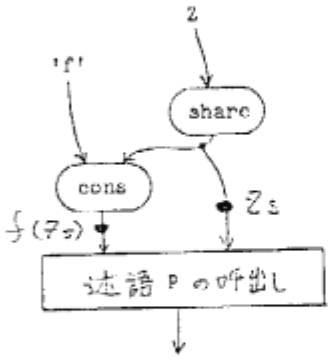

図３． 共有変数を含むゴール呼出し

入力オペランドが単純変数を含む構造データであるときのオペレーションを高速化するため，構造データにはその部分構造として単純変数を含むか否かを示すＶタグが用意される．share オペレータは，入力オペランドが構造データである場合，そのＶタグを調べＶタグがオフであればオペランドをそのまま出力するし，Ｖタグがオンであれば構造を部分構造に分解してそれぞれの部分構造毎にふたたびshare オペレータを実行する．

consistency-check オペレータは上記２つの結合環境の間の統一化を行い，最終的な結合環境Zs=g(b) を出力する．この場合，共有変数に対する最終インスタンスには，単純変数が含まれないので，consistency-check オペレータに続くshare オペレータは実質的な処理を行わず，結合環境をそのまま出力する．

この最終結合環境はsubustitute オペレータに送られそれぞれのインスタンス入力オペランドに含まれる共有変数を結合環境で示される置換に従って置換して，２つの最終インスタンスf(g(b)) 及びg(b)を出力する．substituteオペレータの実行を高速化するため前述のＶタグと同様に，構造データにはその部分構造として共有変数を含むか否かを示すＳタグが付与される．substituteオペレータは，その結合環境入力に何らかの共有変数に対する置換が存在し，そのインスタンス入力オペランドが構造データである場合まずＳタグを調べ，Ｓタグがオフであれば構造データをそのまま出力するし，Ｓタグがオンであれば構造を部分構造に分解してそれぞれの部分構造毎にsubstituteオペレータを並列実行する．

上例は，consistency-check オペレータの結果においてZsに結合されるインスタンスがグランド項になる場合を示したが，例えば，節の頭部として以下のようなリテラルが与えられた場合，consistency-check オペレータの出力は，Zs=g(W) となる．

```
p(f(g(W)), g(Z))  <-  ...
```

このような場合，上記share オペレータはこの結合環境をZs=g(Ws)のように更新して出力する．

上の２例は統一化が成功する場合を示したが，以下に示すようなゴールリテラルが与えられると２つのunify オペレータによって生成される結合環境は，それぞれ，Zs=f(g(W))及びZs=g(b) となりconsistency-check オペレータは統一化が失敗したことを知らせる結果'fail'を出力する．これによってsubstituteオペレータも'fail'を出力する．

```
<-  p(Z, Z).
```

③　節頭部リテラルに共有変数を含む場合

節頭部に変数が２度以上現れる場合も前述の②で示したような手法が適用できるが，consistency-check におけるオーバヘッドを軽減するため以下に示すようにshare オペレーションの実行を遅らせるようにすることができる．

例えば，以下のようなゴールリテラル及び節が与えられたとしよう．

```
        <－  p（X，Y）．
    p（f（Z），g（Z））  <－  ．．．
```

このようなとき，節頭部に現れる変数Ｚを統一化処理の実行前に共有変数に更新するのでなく，Ｚの２度の出現が同一インスタンスに結合されることを保証するためにunify オペレータを実行する．即ち，ゴールリテラルと節頭部リテラルの間の統一化を行う２つのunify オペレータは，ゴールリテラルに含まれる２つの引数がベクタであるかもしくは変数であるかを調べた後，それらをベクタ要素に分解し，その第二要素（即ちベクタの引数であり，この場合はＺのインスタンスとなる）の間で前記unify オペレーションを行う．このとき，ゴールリテラルに含まれる引数が変数（単純変数であるかもしくは共有変数）であるならば，得られたベクタ要素は変数（それぞれ，単純変数かもしくは新たに生成された共有変数）となる．

節に本体部が存在し，このＺが節の本体部で使用されているならば，Ｚのインスタンスに本体部に継される．即ち，本体部における統一化処理においては，Ｚそれ自身を共有変数としてではなく，単純変数として処理する．これによって，本体部における統一化の処理は一般にＺを共有変数として取扱う場合よりも単純になる．この本体部における処理が完了してＺに対する最終インスタンスが求められた後に，share オペレーションが実行される．

この結果は与えられたゴールリテラルの引数Ｘ及びＹの部分構造になる．即ち，上記share オペレーションの結果に共有変数が含まれていれば，変数Ｘ及びＹがそれ以降の処理で使用される場合，それらのインスタンスは互いに共有変数を持つことになる．

（２）　非決定的マージプリミティブ

ＯＲ並列実行環境下においては，１つのゴールリテラルに対する結果（解）が複数個現れる場合，それらを非決定的にマージする制御を必要とする［２］．この場合解の現れる順序は非決定的である．即ち，定義節側では求められた順序に解を出力する．ゴールリテラルはそれらが出力される度に次々に取出して次の処理を進める．この非決定性は'don't know nondeterminism' と呼ばれる．

このような非決定的制御を実現するために，本マシンではストリームと呼んでいるnon-strictなデータ構造を導入した．ストリームは差分リスト形式で表現され，ストリームの消費者側（即ち，ゴール側）及びストリームの生産者側（即ち，定義節側）で非同期に通信する手段を提供する．

図４にストリームの表現を示す．同図に示すように，ストリームはいくつかのセルから構成される．構造データを格納するための基本単位であるセルはfirst 部及びrest部から構成される．このfirst 部及びrest部は，いずれも，１語分のデータを格納するためのDAT(Data) フィールド，そのフィールドの内容が有効であるか否か（即ちDAT フィールドにデータの書込みがすでに行われたか否か）を示すR(Ready)フラグ，及び，データに対する読出し要求待ちのオペレーションが存在するか否かを示すP(Pending)フラグから構成される．P フラグは，R フラグがオフの状態の時にデータに対する読出し要求が発生したときにセットされ，DAT フィールドには，読出し要求待ちとなったオペレーション群に対するポインタが格納される．R フラグ及びP フラグは後述するようにストリームの生産者側と消費者側の間の非同期通信手段を提供する．この他，セルを参照しているポインタの数を格納するためのRC(Reference Count) フィールドを各セル毎に用意する．RCフィールドは，後述するようにメモリ管理のために使用される．

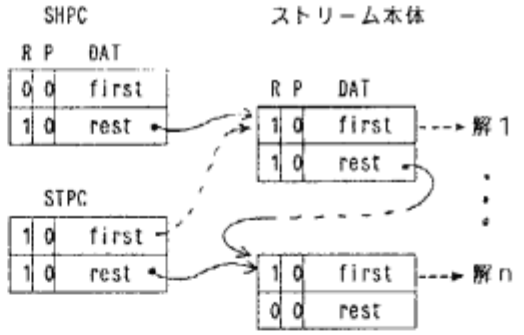

図４．　ストリームの表現

SHPC(Stream Head Pointer Cell)はストリーム本体の先頭を指すポインタをそのrest部に持つセルであり，STPC (Stream Tail Pointer Cell)は現時点でのストリームの最後尾を指すポインタをそのrest部に持つセルである．SHPC はストリームの消費者側で参照され，STPCはストリームの生産者側で共有される．ストリーム本体セルはそのfirst 部に解の内容を，rest部にストリームの残りへのポインタを持つ．解の内容は後述するように，節の頭部リテラルに現れた引数に対するインスタンスの内容及び結合環境から構成されるベクタで表現される．

ゴール呼出しが発生すると，定義節側はまずcreate-

streamオペレータを実行して２つのセルSHPC及びSIPCを生成する．このときSTPCのrest部のポインタの内容はSHPCを指すように初期化され，SHPCのrest部の内容は空(empty)の状態に初期化される（即ち，rest部のR フラグ及びP フラグはいずれもリセットされる），定義側はSHPCのポインタ（アドレス）をこの時点でただちにゴール側に返す．ゴール側はSHPCを受取るとそのrest部の内容（即ち，ストリームの先頭アドレス）を読出す．このときSHPCのrest部が空の状態であれば（即ち，生産者側でストリームに対する解の書込みがまだ行われていなければ），この読出しは待ち状態となる．そうでなければ，STPCのrest部をもとにストリーム本体を辿り，解の内容を次々に取出す．

create-stream オペレータによって生成されたもう１つのセルSTPCのポインタは定義内のそれぞれの節に渡される．節は統一化が成功すると，append-stream オペレータを実行する．このオペレータは解をSTPCのポインタをもとに以下の手順でストリームに書込む．まず新たな解を格納するためのセル（これをストリーム本体の新最後尾セルと呼ぶ）を確保し，STPCのrest部の内容を読出しそのポインタの指すセル（これをストリーム本体の旧最後尾セルと呼ぶ）のrest部に新最後尾セルのアドレスを書込む．このとき，すでに消費者側の読出し要求待ちのオペレーションが旧最後尾セルのrest部に存在していたならば（即ち，そのP フラグがオンであれば），それらを起動する．この後，STPCのrest部の内容を新最後尾セルのアドレスに更新する．このとき，STPCの読出しから更新までの間STPCはロックされる．この処理と並行して新最後尾セルのfirst 部への解の書込みも行われる．

ストリームの生産者から消費者へ解が尽きたか否かを知らせる制御は以下のように行われる．本マシンでは構造データを格納する構造メモリの管理法として参照数法［５］を使用している．即ち，すべてのメモリセルにそのセルを参照しているポインタの数を格納するためのRCフィールドを設けている．append-stream オペレータは上記の処理と並行してSTPCの参照数を１だけ減少させる．この参照数の減少は節における統一化の処理が失敗したときにも行われる．従って，STPCの参照数がゼロに達したとき，全ての定義節の実行が終了したことを意味する．このとき最後にSTPCの減少を行ったappend-stream オペレータはSTPCのrest部の指すセル（即ち，ストリームの最後尾セル）のrest部にシンボル'end-of-stream'（実際には，シンボル'fail'を使用する）を書込む．消費者側はストリームを辿る際にセルのrest部にこのシンボルを検出したならば，ストリーム内の解が尽きたことを知る．

また，このappend-stream オペレータにおいて以下に示すように同一の解が重複するのを回避するように制御することもできる．

解の重複を避けるように制御する場合，STPCのfirst 部もそのrest部と同様SHPCのポインタに初期化される．append-stream オペレータはまずSTPCのfirst 部の内容を読み出しそのポインタをもとに，ストリーム本体に格納された解の内容を次々に調べて，新たに付与しようとしている解と同一のものが存在するか否かを調べる．重複する解が存在しなければ上記に示した解をストリーム本体の最後尾に付与する処理を行うし，そうでなければ単に参照数の制御を行う．但し，このような重複検査は一般に大きなオーバヘッドをもたらす恐れがあり，解の重複を除去した方が効率の向上が期待できる部分にのみ使用するのが望ましい．

（３）　ＡＮＤリテラル実行制御

次に節の本体部が存在する場合の処理について，本体部内のリテラルが逐次的に実行される場合と並列に実行される場合に分けて説明する．

①　ＡＮＤ逐次実行

例えば，１つの定義節として以下のようなものが与えられた場合を考えよう．

```
p([a | X],Y) <- q(X,Z) & r(Z,Y).
```

このとき，節に現れる全ての変数の渡される様子は図５に示すグラフで表現される．

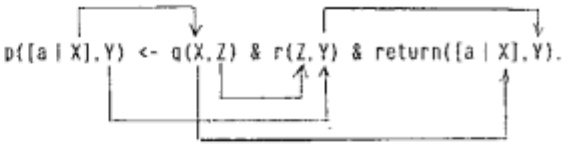

図５．　ＡＮＤ逐次処理

同図において return([a | X],Y) は統一化結果をゴール側に返すための処理である（実際にはこの処理において前述のappend-stream オペレーションが実行される），ここで返すべき結果は節頭部の引数と同一であるが，現在試作しているコンパイラにおいては，DEC-10 Prolog やParlog［４］にみられるような節頭部の入出力を指定できるようにインプリメントしており，この指定があれば，返すべき引数は節頭部のサブセットになる．

例えばこの指定によって，述語pの第一引数が入力に，第二引数が出力に指定されたとすれば，節の結果として返すのは変数Yに対する結果だけでよく，前述のグラフは図６のようになる．

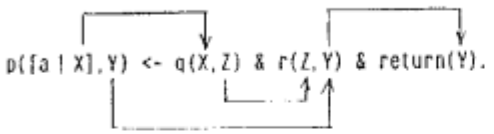

図６．　節頭部の第二引数のみが出力される場合

この節に対してゴールリテラルが与えられたとき，ゴールリテラルと節頭部の統一化が成功すれば，変数Ｘ及びＹに対するインスタンスが得られる．このうちＸに対するインスタンスは述語ｑのリテラルの第一引数として使用され，Ｙに対するインスタンスは述語ｒのリテラルの第二引数として使用される（即ち，Ｙに対するインスタンスは述語ｑのリテラルをバイパスして述語ｒのリテラルに渡される）．

このような場合，変数Ｘ及びＹの間で互いに共有変数を持つようなとき，q(X,Z)の呼出し処理においてＸに含まれる共有変数に何らかのインスタンスが結合されたとすれば，変数Ｙに含まれる共有変数に対してもＸに対して行ったと同様な置換を施す必要がある．このためq(X,Z)の呼出し処理においてはその入力引数に対するインスタンスと共に入力引数にもし共有変数が存在すればその共有変数に対する結合環境も出力する．この結合環境を記号Eqで表すことにする．

同様に，この定義節の述語ｐは定義節の本体部として使用される可能性があるため，定義節はその結合環境を出力しなければならない．今，節頭部における統一化処理で生成された共有変数に対する結合環境をEpとしよう．

ここで注意しなければならないのは，q(X,Z)の呼出しによって得られるのはストリームであることである．即ち，q(X,Z)の呼出しは変数Ｘ及びＺに対するインスタンスの系列と共に結合環境の系列も出力する．従って，ストリームのｉ番目の要素（解）の内容は(Xi,Zi,Eqi)のように表現される．定義節側は，実際にはこのストリームを要素に分解して，それぞれの要素毎に後述する置換の処理及び次のリテラルr(Z,Y)の呼出し処理を行わなければならない．この様に全ストリーム要素に対して下記の処理を行なうように制御するのは，再帰呼出しによって表現できるが，ここではその説明は省略する．

q(X,Z)によって生成されたEqはＹに対するインスタンスを置換するのに使用されると共に，節頭部からの結合環境Ｅｐとの間で無矛盾性検査を行うのにも使用される（置換は前述のsubstituteオペレータによって行われ，無矛盾性検査はconsistency-check オペレータによって行われる）．同様に，r(Z,Y)の呼出しも結合環境Erを返すが，このErはr(Z,Y)をバイパスする引数の置換及びEpとの間の無矛盾性検査に使用される．

この場合，図５及び図６ではバイパスされる引数が異なる．即ち，図５においてはバイパスされてreturnの処理に渡される引数が変数Ｘ及びＹに対するインスタンスと環境Epであるのに対して，図６においては変数Ｙに対するインスタンスとEpのみである．

②　ＡＮＤ並列実行

次にＡＮＤリテラルが並列実行される場合を考える．この場合例えば前述の図６においてq(X,Z)とr(Z,Y)が並列に実行される場合，図７のようなグラフで表現される．

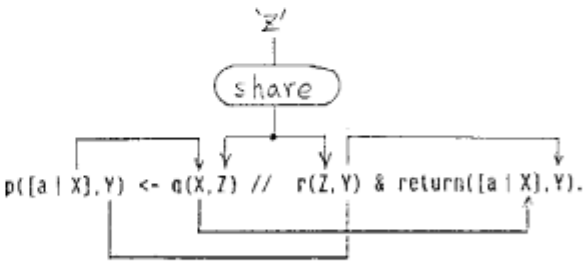

図７．　ＡＮＤ並列実行

この例においては q(X,Z) 及びr(Z,Y)の呼出しが並列に実行される．この場合，変数Ｚは２つの呼出しの間で共有されるので，share オペレータによって共有変数に更新される．２つの呼出しは，それぞれ，(Xi,Zi,Eqi) 及び(Zj,Yj,Erj) を要素とするストリームを返す．述語ｐに対する定義節側では，この２つのストリームの全ての組合わせを求め，それらの要素から得られた２つの結合環境Eqi 及びErj の間の無矛盾性検査を行えばよい．この無矛盾性検査で得られた最終結合環境によって，変数Ｘ及びＹに対するインスタンスとEpの内容を置換してreturnの処理へ渡す．

４．２　Concurrent Prolog における統一化処理

Concurrent Prolog における節頭部及びガード部の処理は４．１で述べた純Prologにおける統一化の処理とほぼ同様である．但し，share オペレータの代わりにcreate-global-variable オペレータが使用される．このオペレータはその入力オペランドが単純変数であるか，又は，単純変数を含む構造データである場合，そのすべての単純変数に対してメモリセルを割当て，単純変数をセルのポインタ（アドレス）で表現される大域変数に更新する．

純Prologと大きく異なるのは，ガード枠に対する処理，及び，読出し専用タグの処理である．

（１）　読出し専用タグの処理

ゴールリテラルの引数として現れた変数に読出し専用タグが付与されている場合，set-read-only-tag オペレータが実行される．このオペレータは入力オペランドが大域変

数である場合のみ，それを読出し専用変数に更新する（なおこの読出し専用変数はそれまでの大域変数が指していたメモリセルを指すポインタとして表現される），大域変数以外の場合は，入力オペランドをそのまま出力する．

なお，set-read-only-tag オペレータの入力オペランドが単純変数である場合も同様に読出し専用変数に更新してもよいが，この場合は他に並行して走行するプロセスが変数にインスタンスを結合するという保証が得られないために後に実行される統一化処理においてデッドロックに陥る恐れがある．従って，このような場合は単純変数をそのまま出力することにする．

読出し専用変数を含むゴールリテラルが与えられた場合の統一化処理を以下に示す．

節の頭部リテラルの引数が変数の場合はゴールリテラルの引数がそのまま次の処理に渡される（即ち，ゴールリテラルの引数が読出し専用変数であれば，読出し専用タグはそのまま継承される）．

これに対して節の頭部リテラルの引数が変数以外の場合は，まずゴールリテラルの引数が読出し専用変数であるか否かが調べられる．ゴールリテラルの引数が読出し専用変数であれば，その読出し専用変数で示されるメモリセルに対して読出し指令を送出する．もしこのメモリセルに変数以外のインスタンスが結合されていなければ，この読出し指令は待ち状態となる．これに対して，すでにこのメモリセルに変数以外のインスタンスが結合されていれば，その読出し結果と節頭部との統一化処理が起動される．即ち，読出し専用タグは大域変数に対する読出し指令であると解釈できる．この指令は，読出し専用変数と変数以外の項との統一化が行われようとしたときに，起動される．

（２） ガード制御機構

前述のようにガード棒の機能は２つ存在する．１つは排他的制御機能であり，もう１つは節の頭部及びガード部の処理で得られた大域変数に関する結合環境を他プロセスに対して公開する機能である．この排他的制御は非決定的である．即ち，最初にガード節を通過したＯＲプロセスのみが以降の処理を続行する．この非決定性は'don't care nondeterminism' と呼ばれる．

複数のガード節から成る定義の実行においては，ＯＲプロセス間の排他的制御を行なうために以下に示すようなセマフォオペレータを用意する．ゴールリテラルの呼出しが発生したとき図８に示すように，定義の先頭で，まずcreate-guardオペレータが実行される．このオペレータはＯＲプロセス間で共有される共有フラグ（セマフォ）を生成し初期化して，そのポインタをＯＲプロセス群に渡す．各ガード節の実行を行うＯＲプロセスは，ゴールリテラルと頭部リテラルの統一化が成功し，且つ，節のガード部の呼出し処理が成功したとき，共有フラグに対してtest&setオペレータを起動する．test&setオペレータは共有フラグの内容を読出してその結果を返すと共にフラグをオンに設定する．従って，このオペレータの結果は最初にガード棒を通過したＯＲプロセスであるか否かを示す．もし最初にガード棒を通過したプロセスであれば，それまでの統一化処理で得られた大域変数に関する結合環境を呼出したゴールに対して公開する．この操作は後述するbindオペレータが行う．

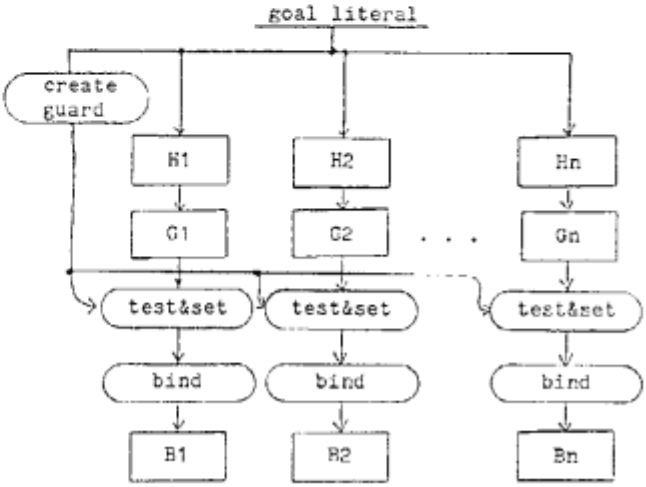

図８． ガード制御オペレータ

このとき実行中の他のプロセスの処理を継続させたままでもよいが，これらのプロセスの数が多いとき，又は，これらの処理が長い間継続されるような場合，資源の節約の面からこれらの無益なプロセスの実行を強制終了させるのが望ましい．このため，前述のtest&setオペレータはフラグのテスト結果がオフであるとき，自プロセスと兄弟関係にあるプロセス及びそれ等兄弟プロセスの全ての子孫プロセスを強制終了させるための指令を，それらプロセスを実行中の処理要素に対して送出する．各処理要素はプロセスの親子関係を管理するためのプロセス管理表を有しており，ガード棒を通過したプロセスはその親のプロセスを実行中の処理要素に対して，前記強制終了指令を送出する．親プロセスは，この指令を送出したプロセス以外の子プロセスに対し指令を伝搬させる．以下，同様の手順で，それらの子孫の全プロセスの実行が停止される．

このようなプロセスの強制終了を行うもう１つのメリットは再帰呼出しの最適化が行えることである．即ち，ガード節の本体部の実行は，その定義を呼出したゴールリテラルと同一の（即ち，親プロセスの）実行環境で行うことができる．Concurrent Prolog では，一般に再帰呼出しによって，メッセージパッシングや並行プロセスの記述を行っ

ている．従って，この最適化は処理効率に大きな効果をもたらすと考えられる．

図８におけるbindオペレータは，得られた結合環境の内容を調べ，それが'nil' でなければ大域変数に対する置換情報に応じて，結合されたインスタンスの内容を大域変数の指すメモリセルに書込む操作を試みる．

この書込みは以下の手順で行なわれる．まず結合環境から１つの大域変数とインスタンスのペアを取出し，大域変数の指すメモリセルへ，すでにその大域変数と結合されている（かもしれない）インスタンスとの統一化を行なうための指令を送出する．この指令送出の処理を結合環境に含まれる大域変数とインスタンスのペアの数だけ繰返す．

メモリセル側はこの指令を受取るとメモリセルをロックする．メモリセルのロック状態はP フラグ及びR フラグをいずれもオンにセットして表現する．メモリセルにインスタンスが結合されていなければ，指定されたインスタンスの内容を書込む（このとき，待ち状態のオペレーションがセルに存在すればそれらを起動する），そうでなければ，すでに書込まれているインスタンスとの統一化が試みられる．この統一化処理においてはいく通りかの場合がある．統一化が失敗したとき，及び，両インスタンスの内容が全く等しい場合は問題はなく，本定義を呼出したゴールは，それぞれ，失敗もしくは成功する．これに対して両インスタンスの統一化は成功するが，互いに異なる場合（即ち，いずれかに変数を含んでいる場合）の取扱いには，以下のような解釈が妥当であろう．

新たに書込もうとするインスタンスが旧インスタンスを包含していれば（即ち，新インスタンス内に変数が含まれていれば）問題はなく統一化は成功したと解釈してよいであろう．これに対して書込もうとするインスタンスの一部が旧インスタンスの対応する部分のサブセットである場合には，すでに旧インスタンスを他のプロセスが参照している恐れがあるためその取扱には問題がある．この様な場合正確には旧インスタンスを参照したプロセスの実行を停止し，新たなインスタンスを用いて再起動する必要があるが，このような処理は多大なオーバヘッドをもたらす恐れがあるため，本マシンにおいてはユーザに警告メッセージを出す等の手段を用意して処理を継続させることにする．

なお，大域変数と結合されているインスタンスが大域変数であるようなとき，両大域変数の指すメモリセルの間でリンクが張られる．このとき，少なくとも一方の大域変数が変数以外のインスタンスと結合されおり，他方の大域変数に読出し待ちの要求が存在すれば，読出し待ちの要求は起動される．両大域変数がすでに変数以外のインスタンスと結合されていれば，両者の統一化が試みられる．

V．　マシンアーキテクチャ

純Prolog及びConcurrent Prolog プログラムはマシンの機械語に相当するデータフローグラフに展開され，実行される．これによってこれらのプログラムにおけるＯＲ並列，ＡＮＤ並列，及び，統一化処理における引数間の並列実行が実現できる．

マシンは図９に示すように，PEM(Processing Element Module) 群，SHM(Structure Memory Module)群，及び，これらのモジュールを結ぶ３つの結合網から成る．PEM はデータフローグラフをロードし，実行するモジュールであり，さらに図１０に示すような機能ユニットから構成される．

図１０において，PQU(Packet Queue Unit)は結果パケット（トークン）の待合わせ機能を有している．ICU(Instruction Control Unit) はPQU からの結果パケット到着によって起動され，命令の全オペランドが揃ったか否かを検出する．ICU は結果パケットの内容を一時的に格納するためのオペランドメモリとこのメモリを連想検索するための機能を持っており，命令が２オペランド命令であれば，結果パケット内の命令宛先タグによってオペランドメモリを連想検索する．この連想検索が成功するか（即ち，命令の全オペランドが揃うか），又は命令が単一オペランド命令であれば，実行可能命令を生成してEIR(Executable Instruction Register) にラッチする．この連想検索が失敗した場合，結果パケットはオペランドメモリに格納される．

実行可能命令はAPU (Atomic Processing Unit)のいずれかに転送され実行される．このとき，APU は命令で指定された宛先に応じて結果パケットを生成しPQU に転送すると共に，構造データ操作の必要があれば構造データ制御コマンドをSMCR(Structure Memory Command Register) にラッチする．このコマンドは図９に示したPE-SH ネットワークを経由してSHM へ送られる．

SHM は構造データの格納，管理，及び，操作を行うモジュールであり，そのメモリ管理には参照数法を使用している．SHM はコマンドを受信すると指定された処理を行った後，必要に応じて結果パケットを生成してSH-PE ネットワークを経由してPEM 側に転送する．

Inter-PEネットワークは主として手続き呼出しにおける引数や結果の受渡しの際にPEM 間で結果パケットを交信するのに使用され，Inter-SHネットワークはガーベッジコレクション制御等のために使用される．

VI．　シミュレーション結果

シミュレータは図１０に示した機能ユニット群，SHM モジュール，各ネットワークの全てのノード，及び，動的に生成された個々の結果パケットをそれぞれ独立なプロセスとして実現しており，シミュレーション事象が発生する度にこれらのプロセスを起動・停止させることによって，マ

シンの時間的経過の様子を観測している．

各ユニットの処理時間はパラメタとしてもえている．それらの典型的な設定値（単位はマシンサイクル数）は文献［14］で示した．これらのパラメタは適当なハードウェア構成を仮定し，必要に応じてそのマイクロフローチャートを検討して決定した．その典型的な設定値の一部を示すと，ICU における単一オペランド命令の処理時間，２オペランド命令の場合の先着オペランドの処理時間，及び，２オペランド命令の場合の後着オペランドの処理時間は，それぞれ，２，３，及び，５マシンサイクルであり，APU におけるコピー命令の実行時間，及び，結果パケットを送出する場合のパケット構築時間は，それぞれ，３，及び，２マシンサイクルである．結果パケットを送出する場合，さらにPQU にパケットを転送するための時間を２マシンサイクルと仮定している．従って，例えば，APU においてコピー命令が実行され，その宛先が４つ指定されているような場合，APU の全処理時間はPQU への結果パケット転送の競合がなければ，３＋（２＋２）＊４＝１９マシンサイクルとなる．SHCRやネットワークノードに関しては，それらへパケットをラッチする時間を２マシンサイクルとし，パケット転送が完了するまでの時間を８マシンサイクルとしている．

PEM 内のAPU 台数は任意個に設定可能であるがAPU 台数を２以上にしても性能の改善はそれ程期待できないことから２台に設定した［15］．

評価対象としたプログラムは以下の通りである．

- summation （１から１００までの数値の和）
- four-queens 及びfive-queens
- reverse （１０個の要素から成るリストの反転）

このうち，summation プログラムはdevide&conquer法によってＡＮＤ並列を実現している．このプログラムは決定的であり，解は唯一つしか存在しないので純関数とみなすことができる．従って，ストリーム制御プリミティブを使用しないで実行することができ，ストリーム版及び非ストリーム版の双方についてシミュレートした．

モジュール台数（PEM 及びSHM の台数）を１から１６まで変化して，これらのプログラムを実行し全解探索したときの結果を図１１に示す．同図におけるLIPS(Logical Inferences Per Second) はマシンのサイクル時間を２５０ナノ秒と仮定した場合の単位時間における推論性能を示す．reverse プログラムのように並列性のほとんどないプログラムにおいては性能向上はみられない．これに対して，four-queens 及びfive-queens プログラムにおいては台数の増加に伴ってある程度の性能向上が見られる．summation プログラムにおいてはさらに台数効果が得られるが，非ストリーム版においてはストリーム版の２ないしは３倍の性能が得られた．従って，試作中のコンパイラにおいては，このようなプログラムに対してオプション指定を行い，ストリーム制御プリミティブを省略したコードを生成可能にしている．

VI. おわりに

データフロー方式並列推論マシンにおける純Prologおよびconcurrent Prolog に対する実行制御方式を明らかにし，これらの言語をサポートするためのプリミティブを説明すると共にマシンアーキテクチャの概要を述べた．これによって，本マシン上で，目的の異なる２つの論理型言語をサポートできる見通しが得られた．著者等は現在このマシンの細部の検討を進めている．これに伴って純Prologに対するシミュレータを開発し，簡単な純Prologプログラムを並列実行した場合のシミュレーション結果を示した．これによれば，並列度の高いプログラムにおいてはかなりの性能が得られることが判明した．現在，純Prologプログラムをデータフローグラフに変換するためのコンパイラの開発を進めており，今後評価プログラムの範囲を広げていき詳細な検討を進めていく予定である．また上記シミュレータ上でConcurrent Prolog のシミュレートするための機能も付加し，その評価も行なう予定である．

最後に，本研究の機会を与えていただいた渕研究所長，及び，日頃ご指導いただく村上第一研究室長や尾内主任研究室員を始めとするＩＣＯＴ研究員各位に深謝する．

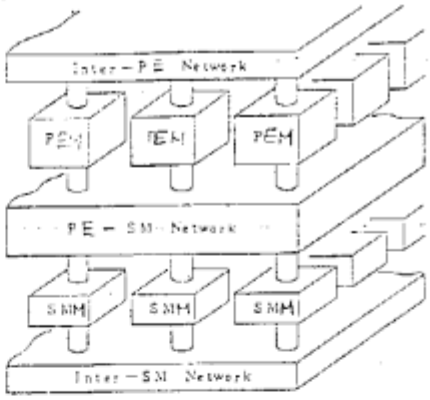

図９．マシンの構成

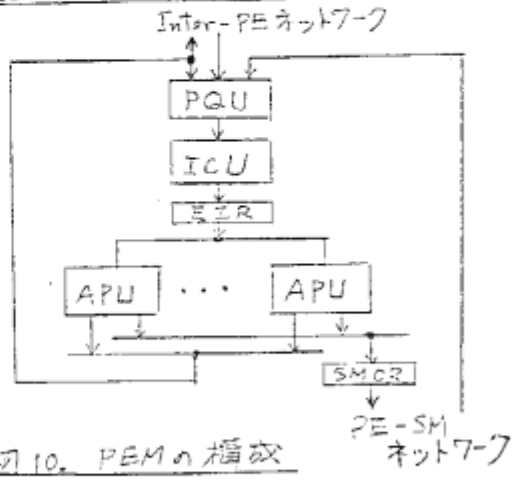

図10．PEMの構成

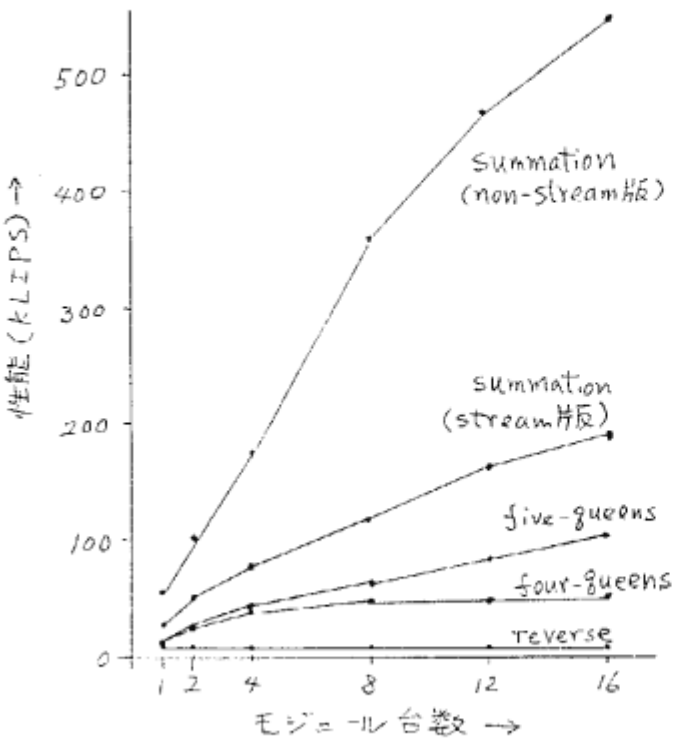

図11. 並列台数効果

<参考文献>

［１］雨宮，長谷川，"データフローマシンと関数型言語"，信学技報AL81-84 及びPRL81-63，1981.

［２］Arvind,K.P.Gostelow and W.F.Plouffe, "An Asynchronous Programming Language and Computing Machine", TR-114a, Dept. of ICS, University of California , Irvine, Dec. 1973.

［３］Arvind and R.A.Iannucci, "A Critique of Multiprocessing von Neumann S tyle", Proc. of 10th Int'l Symp. on Computer Architecture, June 1983.

［４］Clark,K. and S.Gregory, "PARLOG: A Parallel Logic Programming Language", Research Report DOC 83/5,Imperial College of Science and Technology, May, 1983.

［５］Cohen,J., "Garbage Collection of Linked Data Structures", Computing Surveys, Vol.13, No.3, Sep. 1981.

［６］Conery,J.S. and D.Kibler, "Parallel Interpretation of Logic Programming", Proc. of Conf. on Functional Programming Language and Computer Architecture, ACH, Oct. 1981.

［７］Dijkstra,E.W. "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J., 1976, (Chapt. 17) .

［８］後藤，相田，丸山，淵原，田中，元岡，"高並列推論エンジンPIE について"，Proc. of Logic Programming Conference '83, ICOT , １９８３.

［９］Guard,J.R. and I.Watson, "Data Driven System for High Speed Parallel Computing", Computer Design , 1980.

［１０］伊藤，尾内，益田，清水，"データフロー方式のPrologマシン"， Proc. of Logic Programming Conference '83, ICOT, 1983.

［１１］伊藤，益田，清水，"データフロー方式Prologマシンにおける非決定的制御機構"，情報処理学会第２７回全国大会，1983.

［１２］Ito,N., K.Masuda and H.Shimizu, "Parallel Prolog Machine Based on the Data Flow Model", ICOT TR-035, 1983.

［１３］伊藤，来住，"データフローマシン上での Concurrent Prolog 機能の実現"，情報処理学会第２８回全国大会，1984.

［１４］伊藤，久野，"並列Prologマシンのシミュレーションによる評価"，情報処理学会第２８回全国大会，1984.

［１５］益田，伊藤，清水，"データフロー方式Prologマシンのシミュレーションによる評価"，情報処理学会第２７回全国大会，1983.

［１６］松本，田中，平川，三吉，安川，向井，横井，"Prlog に埋め込まれたボトムアップパーザ：ＢＵＰ"，Proc. of Logic Programming Conference '83, ICOT, 1983.

［１７］尾内，麻生，"並列推論マシンにおけるGuard と入力annotationの制御機構"，情報処理学会第２７回全国大会，1983.

［１８］Pereira,C.N. and Warren,D.H.D ., "Definite Clause Grammer for Language Analysis - A Survey of the Formalism and a Comparison with Augumented Translation Networks", Artificial Intelligence 13, 1980.

［１９］Shapiro.E.Y., "A Subset of Concurrent Prolog and its Interpreter", TR-003, ICOT, 1983.

［２０］竹内，E.Y.Shapiro，"論理型言語によるコンカレント・プログラミングについて"，情報処理学会ソフトウェア基礎論４－４，1983.

［２１］梅山，田村，"論理プログラムのためのＯＲ並列実行モデル"， Proc. of Logic Programming Conference '83, ICOT, 1983.

［２２］Yasuura,H., "On the Parallel Computational Complexity of Unification", TR-027, ICOT, Oct. 1983.