ICOT Technical Report: TR-077

TR-077

並列推論マシン PIM-R の アーキテクチャとソフトウェア・シミュレーション

尾内理紀夫，麻生盛敏，清水　肇
益田嘉直，松本　明

January, 1985

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

— 目次 —

1．はじめに

ICOTでは述語論理型言語をベースにした知識情報処理のためのソフトウェア、ハードウェアの研究開発を行なっている。述語論理の基本操作は推論であるから、そのハードウェアは推論マシンと呼ばれる。また、推論が基本操作であるから、このマシンは並列動作が基本となる。即ち、推論マシン（これを、我々は並列推論マシンと呼んでいる）は、逐次動作を基本とするノイマン型マシンではなく、並列動作を基本とするinnovativeノイマン型マシンとなる。並列推論方式[Moto 84],[Ito 84]、あるいは述語論理型言語[Shap 83],[Clar 84],[Pere 84] として、現在いくつかのものが提案されている。ここでは、ICOTが核言語第１版のベース言語として位置づけているPrologとConcurrent Prolog [Shap 83] をマシンの対象言語として選択した。また、並列推論方式としてはreduction 概念に基づきPrologをOR並列に、Concurrent Prolog をAND 並列に処理する方式を採用した。

次にreduction ベースのマシンを考えるに至った動機について簡単に述べる。例えば、式 7+3のreduction を考えた時、この式は加算に関するruleを用いて自らをmodifyし、10となる。そして式10はこれ以上reduction(modify) できない（既約）ので解となる。即ち、reduction はself-modification とみなせる[Turn 79] 。一方、PrologまたはConcurrent Prologプログラムの実行過程は、親 clause(goalとclause) からのresolvent の生成過程であり、resolvent として空節が導出されれば、それが解となる。これは、goalがclause群という一種のruleを用いて、自らをmodifyする過程とみなせる。このように、PrologあるいはConcurrent Prolog プログラムの実行過程とreduction との間に親和性の良さを見出すことができる。これがreduction 概念に基づく並列推論マシン(Parallel Inference Machine based on the Reduction concept : PIM-R) の研究開発の動機である。

PIM-R においては、プロセス内にgoalが複数あったとき、（それら複数goal全体を親プロセスと呼ぶ）は、そのうちのreducible なgoal(どれがreducible かは各種オペレータにより指示される）のみをreduceし、生成された子プロセス（子プロセスが複数あるときは、その一つ一つ）から親プロセスへポインタが張られる。このポインタにより子から親へ解が返される。即ちPIM-R でのProlog、Concurrent Prolog の処理過程はプロセス木の伸長、縮小の過程であり、処理が終了した時、木は論理的に消滅（物理的に消滅させるためには、ガーベッジコレクタが走らねばならない）する。

PIM-R はstructure-copy方式を採用しているが、copyおよび、それに伴なう処理量と、Network 通過packet数の低減のため、only-reducible-goal copy、独特のprocess 構成法、reverse compactionを採用している。アーキテクチャとしては、Concurrent Prolog のための分散化共有メモリの導入、効率的packet分配のためのNetwork Nodeの導入、大きい構造体データ中のGround Instance 格納のためのメモリの導入をその特徴としている。

本論文では、PIM-R におけるPrologとConcurrent Prolog の並列実行方式、PIM-R アーキテクチャそしてPIM-R ソフトウェア・シミュレーション結果について述べる。

2．PrologとConcurrent Prolog の並列実行方式

2.1 Prologの並列実行方式

Prologプログラムの並列実行には、引数間並列、AND 並列、OR並列がある。

Prologプログラムの解析により[Onai 84b]、goalの平均引数個数は 2～ 3であるので、引数間unification の処理を並列化したとしても、引数間のconsistency check を考えると得策ではないと考える。

また、Prologでは、複数の解が生成される可能性があるので、AND 並列実行は、AND関係にあるgoal間で共有される引数に関するconsistency check が複雑になり、並列化の効果が低減されるので、PIM-R では採用しない。

一方、OR並列は、問題(ex. BUP etc.)によって高い並列度が期待できる。そこで、PIM-R ではProlog（ここで言うPrologプログラム内には、cut 、assert、retract は存在しない。）の並列実行方式としてOR並列を採用する。以後、本論文では、OR並列に実行されるPrologを単にPrologと呼ぶ。

それでは、PIM-R におけるPrologのOR並列、AND 逐次実行について述べる。

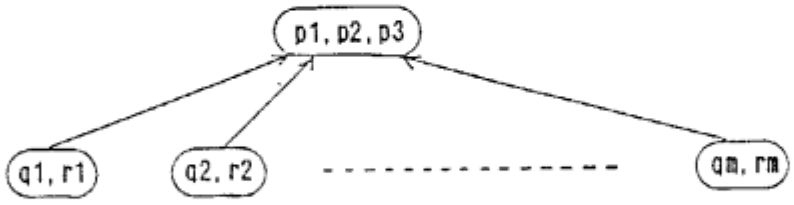
次のような、goal列とclause群があったとする。

goal列　　p1,p2,p3

```
p1:-q1,r1.
p1:-q2,r2.
    :
p1:-qm,rm.
```

このgoal列は、左から右へ逐次に実行される。即ち、この場合p1のみがreducible であるので、goal列p1,p2,p3全体ではなく、p1のみが選択、copyされて、Unification Unit(後述）に送られ、unification が実行される（only-reducible-goal copy）。

unification の結果、goal列p1,p2,p3を親プロセスとするm 個のresolvent （子プロセス）が生成される。



それぞれの子プロセスが求めた解を親プロセスへreturnするために、子プロセスから親プロセスへポインタが張られる。(親から子へのポインタはない)　一方、親プロセスは、子プロセス数を格納するcounter を保持する。この時、この mをOR fork 数と呼び、これら m個のプロセスを兄弟関係にあるプロセス（略して兄弟プロセス）と呼ぶことにする。それら m個の兄弟プロセスは、OR並列実行のため、できるかぎり異なる処理要素(Inference Module(後述)) へ分配される。

次に、例えば一つの子プロセスq1,r1 では、最も左側のq1がreducible となり、q1のみがcopyされて(Only-reducible-goal copy)、Unification Unitへ送られunification が実行される。

プロセスの状態としては、reducible(ready),run(unification実行中) ,wait(子プロセスからの解待ち),dead (解を親プロセスにreturnしてしまった、あるいは、failした)がある。

2.2 Concurrent Prolog [Shap 83] の並列実行方式

Concurrent Prolog のclauseは次のような形をしている。

```
h:-g | b.
```

g はguard partと呼ばれ、goal列である。

b はbody part と呼ばれ、これもやはりgoal列である。

| はguard あるいはguard bar あるいはcommit operator と呼ばれる。

goalが設定された時、対応するclause(たがいにOR関係にある) は並列に探索され、一番最初にGuard partのunification に成功したclauseが選択され、その時にguard 部での結合情報が公開され(commit operation)、そのBody部がresolvent となる。即ち、ある１つのgoalは、たった１つの解しか生成しない。そして、その他のalternative clauseのGuard 部が遅れて成功しても、そのclauseは消去される。この意味で、commit operatorは、cut symbolとして機能する。

goal列の実行に関しては、並列AND(,)と逐次AND(&)との二つのoperatorがある。

並列AND で連結されたgoalは、並列に実行される。

また、並列AND で連結されたgoalが変数を共有する時、それらgoalは、それぞれ、producerとconsumerの関係にあり、その変数はgoalをプロセスと考えると、プロセス間通信のために使用される。　つまり、並列AND というよりは、プロセス間の通信を論理変数を用いて実現しているといったほうがよいだろう。PIM-R ではこの通信のために使用される論理変数(これをチャネルと呼ぶ) のための分散化された共有メモリ(Hessage Board 後述) を設けている。

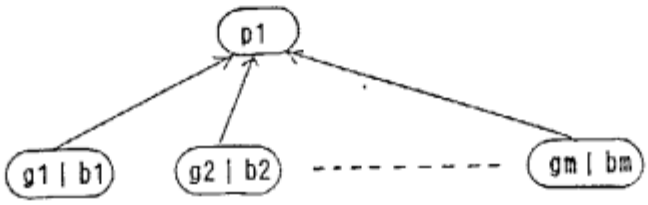
たとえばgoal列　　p1,p2　　(',' は並列AND オペレータ) があった時、p1とp2は、それぞれ別のプロセスとしてforkし（これをAND forkと呼ぶ）、並列にunification が実行される。consumerは変数に値が結合されるまで待たされることになる。なお、共有

変数に関して、read only annotationを付加することができる。read only annotation は "?"で表され、変数に付加して"X?"のように書く。共有変数にread only annotation を付加したプロセス(consumer プロセス) は、この変数をinstantiate できなくなり、他のプロセス(producer プロセス) がそれをinstantiate(メッセージを送出) するまで suspend される。

今、次のようなgoalとclause群があったとする。

```
goal        p1
clause群
P1:-g1 | b1.
P1:-g2 | b2.
     :
P1:-gm | bm.
```

p1のunification が実行されると、次のように、 m個の子プロセスが生成される。



ただし、Prologの場合と異なり、PIM-R では、これら m個の子プロセスは異なる処理要素に分配されるのではなく、1つの処理要素(IM)内に置かれ、まず、guard partの unification が実行される。これら兄弟プロセスは互いにguard partのunification を競い、guard partのunification に成功すると、親プロセス(この図では、p1) へ、自分が一番最初にguard partのunification に成功したプロセスかどうかを確かめに行く。このために、親プロセスにはC-tag (commit tag 後述) があり、最初にguard partの unification に成功したプロセスがこれをONにする。C-tag がすでにONになっていれば、この子プロセスは、自分が二番目以降の遅れてきたプロセスであることを知り、dead状態となる。PIM-R では、1つのプロセスがC-tag をONにした時、兄弟プロセスをkillしにはいかずに、遅れて成功した時に自殺する方法をとる。それは、guard partが深くネストし、遅れて成功する子プロセスがCPU 時間を多大に浪費することは (ないとは言わないが) まれであるので、親プロセスから子プロセスへのkill messageが、子プロセスからの遅れてきた成功報告と行い違いにならないように多少複雑な制御を入れて、いちいちkill処理することは得策ではないと判断したからである。

以上述べてきたように、兄弟プロセスは、commit処理のたびごとに、親プロセスのC-tag を見に行く。そこで、もし親プロセスと兄弟プロセスを異なる処理要素(IH)に割り付けると、C-tag のcheck そしてその結果報告のたびに、処理要素(IH)間のネットワークトラフィックが引き起される。Concurrent Prolog の各節には、commit operatorがあるので、このためのネットワークトラフィックは膨大なものとなり、異なる処理要素(IH)でこれら子プロセスをOR並列実行しても問題解決の時間は短くならない。いや、それどころではなく、かえって長くなることも予想される。

そこで、PIM-R では、Concurrent Prolog における兄弟プロセスは、まず同一処理要素(IH)内に格納して、そのguard 処理をすることにし、異なる処理要素(IH)には分配しない。一方、並列AND オペレータで結合されたgoalは異なるIHへ分配し並列実行する（AND 並列実行）

Concurrent Prolog プログラムの実行時のプロセスの状態には、reducible(ready), wait, run, suspend(consumer プロセスが論理変数に値が結合されるのを待っている), deadがある。

以上をまとめると、PIM-R では、PrologプログラムをOR並列に、Concurrent Prologプログラムを AND 並列（ただしliteral が並列AND operatorで結合された時のみ）に実行する。また、Concurrent Prolog においては、１つのプロセスがcommit処理に成功した時に、他の兄弟プロセスをkillしにはいかずに、他の兄弟プロセスが遅れて成功した時に自殺する方式を採用している。

3．PIM-R アーキテクチャ

PIM-R は図 3.1-1に示すように、Inference ModuleとStructure Memory Module という二種類のModuleとそれらを結ぶNetwork から構成される。

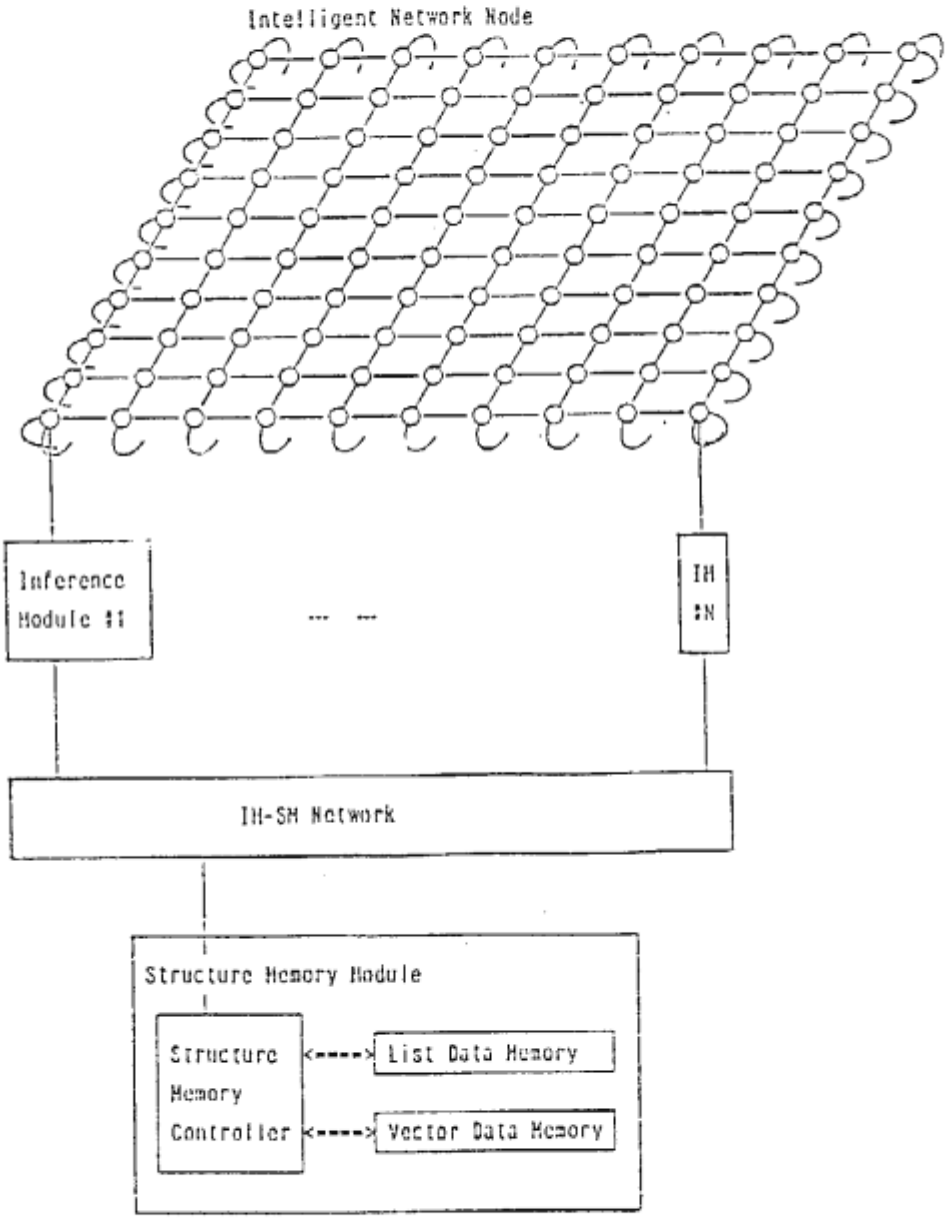


図 3.1-1 PIM-R全体構成図

3.1 Inference Module(図 3.1-2)

本Moduleには、Unification UnitとProcess Pool Unit という二種類のUnitがある。

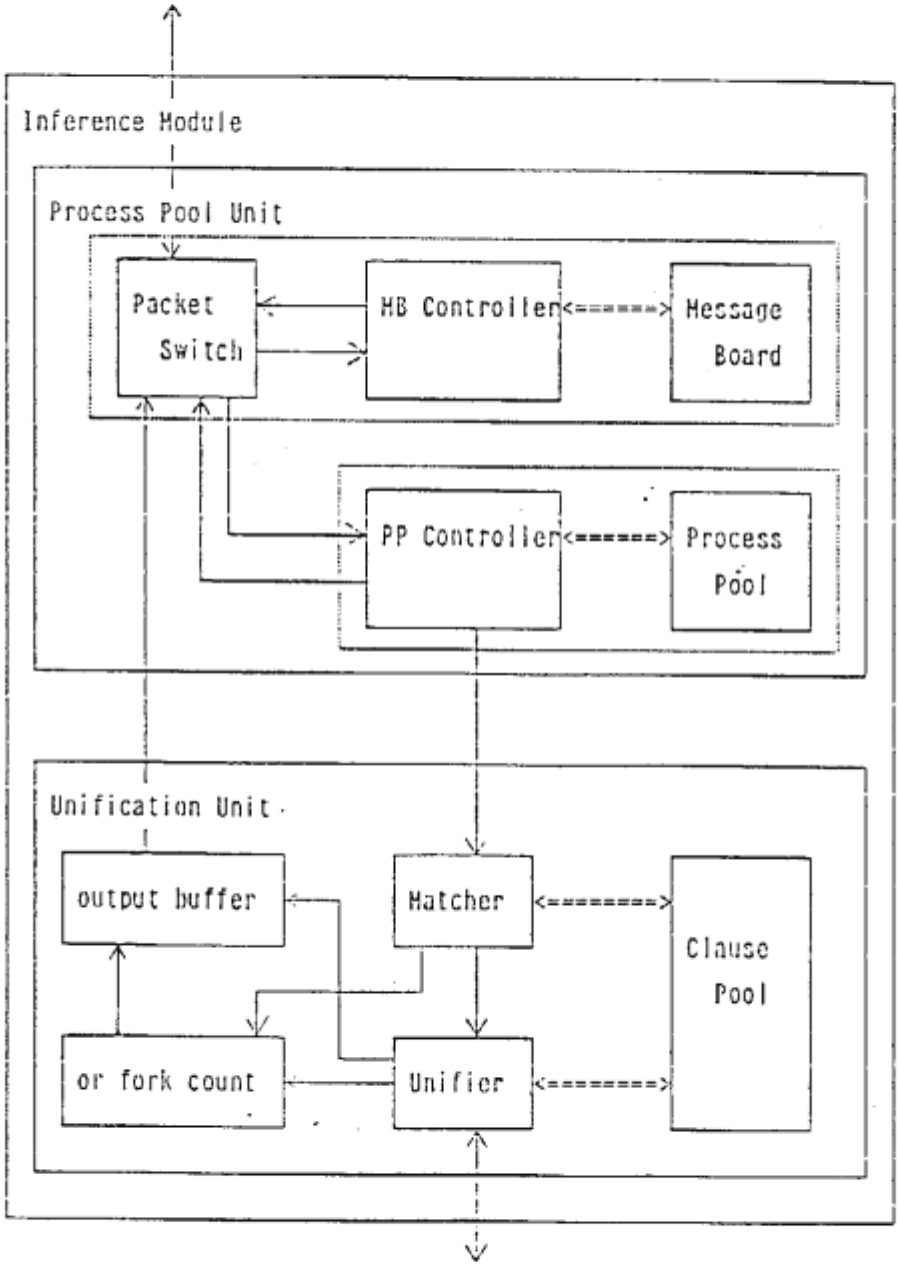


図 3.1-2 Inference Module構成図

3.1.1 Unification Unit

(1) Clause Pool

各Clause Pool は同一のclause群を格納している。PIM-R では、structure-copy方式を採用している。これは、個々のプロセスの独立性を高め、structure の共有に起因するNetwork traffic を低減するためであるが、一方、structure copy方式を採用することによって、copy overhead が増加する。

本節では、このオーバヘッド低減のための、clauseの内部形式について述べる。

付録１にデータタイプの一覧を示す。

まず、clauseの内部形式について述べる。Clause Pool は、Clause Definition group 管理部とClause Definition 部に分れる。

Clause Definition group 管理部は、OR関係にあるclause数、それぞれのclauseが格納されているClause Definition 部を指すpointer 、及び、Hatcher において unifiable なclauseの絞り込みを行うために必要なclauseのヘッドリテラルの第一引数のデータタイプ等の情報が格納されており、その構成は以下の通りである。

Int	N	注1)
Int	OR関係にあるclause数	
TYPE	Clause Definition 部へのpointer	注2)
	⋮	
TYPE	Clause Definition 部へのpointer	

注1) N : Clause Definition group 管理部とClause Definition 部の総word数

注2) TYPE : クローズヘッドの第一引数のデータタイプ

また、Clause Definition 部は、図 3.1-3に示すように、clauseの定義が格納されており、ヘッダー、変数エリア、リテラルヘッダー、リテラルエリア、ストラクチャエリアにわかれている。

ヘッダーは、clause長、ストラクチャエリア先頭番地、リテラルヘッダー先頭番地からなる。但し、ストラクチャエリア先頭番地が格納されている語を第 0番地とした相対番地で示される。何故ならば、reduction が成功しパケットとして通信されるような時は、ヘッダー内のストラクチャエリア先頭番地の前に、パケット長と親へのpointer が挿入されるが、このように規定しておけばストラクチャエリア先頭番地以下の番地を変更する必要がないからである。

変数エリアには、変数個数と各変数のバインド情報が格納される。

リテラルヘッダーには、リテラル数（逐次AND 関係にあるボディリテラル数+1）、ヘッドリテラル、逐次AND 関係にあるボディリテラル（ただし、commit operator は 1つのリテラルとみなす）を逆順に並べて格納する。

ここで、引数個数が 0であるリテラルはリテラルヘッダーに（データタイプのPoi,Sym として）格納されるが、引数個数が 1以上であるリテラルや、並列AND 関係にあるリテラルは、データタイプのLit,Paraをそれぞれに用いてリテラルエリアに逆順に格納する。

リテラルエリアに格納されるリテラル（Lit タイプのポインタによって指される）には、付録1 Ⅳ 7）に示すように、引数個数+1、Clause Definition group 管理部へのポインタ、各引数が格納される。並列AND 関係にあるリテラルについては、後で述べる。

このリテラルヘッダーや、リテラルエリアに、リテラルを逆順に格納する方式により、 3.1.2.1(2)Process Pool Controler の例にて説明する逆順コンパクションをやり易くしている。

構造体データ（リスト、ベクタ、チャネルに関する情報）は、ストラクチャエリアに格納される。ただし、構造体データがない場合はストラクチャエリアは存在しない。

#0	Int	clause長	ヘッダー
	Int	ストラクチャエリア先頭番地	
	Int	リテラルヘッダー先頭番地	
	Int	変数個数	変数エリア
		:	
		:	
	Int	リテラル数	リテラルヘッダー
	Type	ヘッドリテラル	
	Type	ボディリテラル N	
		:	
	Type	ボディリテラル 1	
		:	リテラルエリア
		:	ストラクチャエリア

注1)

注1) Type : Poi, Lit, Sym, Para のいずれか

図 3.1-3 Clause Definition 部の構成

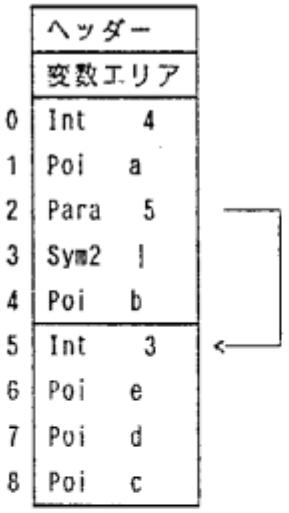
並列AND 関係、あるいは、逐次AND 関係を現わすために、並列AND 記述子と逐次AND 記述子が導入されているが、これについて説明する。

基本的には、guard 部と、commit operater"|" と、body部は逐次AND 関係にあるものとみなす。よって、guard 部やbody部に並列AND operator"," が現れたり、並列AND operatorで結合されたgoal内に、さらに逐次AND operator"&" が現れた時にのみ、並列AND 記述子と逐次AND 記述子が使用される。

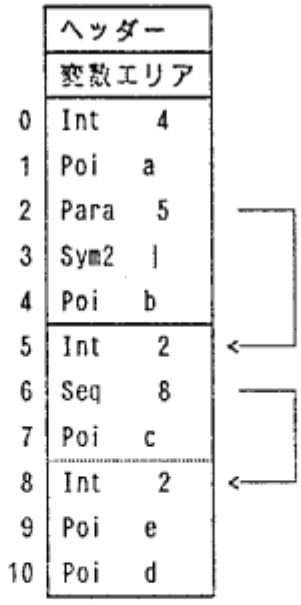
並列AND 記述子Paraは、(例 3.1.1-1) にみられるように、並列AND 関係にある(","で結ばれた) body goal 群を指示するものであり、その先に、並列AND 関係にあるリテラルの数と、各リテラルが(reverse compaction のため) 逆順に格納されている。この例においてもわかるように、commit operater"|" と b、c,d,e は逐次AND 関係にあるものとみなしている。

また、逐次AND 記述子Seq は、(例 3.1.1-2) にみられるように、並列AND 関係にあるリテラルの中に、逐次AND 関係にある("&" で結ばれた) リテラルがさらに現われた時、これらのgoal群を指示するもので、その先に、逐次AND 関係にあるリテラルの数と、各リテラルが(reverse compaction のため) 逆順に格納されている。

(例 3.1.1-1)　a:-b|c,d,e.



(例 3.1.1-2)　a:-b|c,(d&e).



(2) Matcher

Process Pool Unit から送られてきたgoalの、第一引数のデータタイプにより、unifiable なclauseの絞り込みを行なう。もし、このgoalがretry goal（いったんsuspend した後activateされて再びUUへ送られたgoal）でも組込み述語でもない場合は、絞り込まれたcandidate clauseの数を、or fork counter 内のor fork 数(return 数は0)にset する。そして、goalと、絞り込まれたcandidate clauseの格納場所とをUnifier へ送る。

(3) Unifier

組込み述語の実行や、Matcher から送られてきたgoalと、Clause Pool からコピーしたclauseとの間でunification を実行し、結果をoutput buffer へ送出する。このgoalが、retry goalでも組込み述語でもない場合は、or fork counter に対して次の処理を行なう。

a. unification が失敗した場合、or fork 数を-1する。

b. unification が成功またはsuspend した場合、return数を+1する。

(return数=or fork数となった時、その値をunification に成功したor fork 数として自IH内のProcess Poolに返す)

PrologをOR並列実行する場合、unit clause とのunification に成功した時は、自IH内のProcess Poolにある親プロセスに返す。一方、それ以外のor fork 数が 2 以上の時は、新たなresolvent を分配ストラテージに従って、自IH内Process Pool、あるいは、Network 経由で他IHへ分散させる。

Concurrent Prolog の場合、unification の結果は常にいったん自IH内Process Poolへ戻す。またsuspend した場合には、将来のretry 処理のために、与えられたgoal、unification しようとしたclauseの格納場所、suspend の理由となったgoal側のchannel 、それがgoalのどこに格納されているか、また、それはclause側のどういうタイプのデータとunify しようとしたのかといった情報を自IH内のProcess Poolに返し、後述するSPCB（Suspend Process Control Block ）が作成される。

次にunifier の処理を例を用いて説明する。

(例 3.1.1-3)

次のようなgoal,clause があったとする。

```
goal   :  a([1],[2|W])
clause :  a([A|X],Y) :- a(X,Y).
```

図 3.1-4に与えられたgoalとclauseがunifier のmemoryにset された状態を示す。即ち、Hatcher から送られてきたgoalはgoal memory に、Clause Pool からコピーされたclauseをclause memory に格納する。図 3.1-4～図 3.1-6において *は8bit目が 1であり、このことは、これがgoal側を指すpointer であることを示す。#W,#A..はそれぞれ、変数W,A,..が格納されている変数エリアのセルを示す。また 1語は32bit である。

unification の実行は、次の通りである。まず、goalの第一引数であるList *0と、clauseのヘッドリテラルの第一引数であるList 0 とをunify する。即ち、CAR 部同志、CDR 部同志のunify を行なう。CAR 部は、Int 1 と変数A (Var1)とのunify であるから変数エリア内の変数A のセルへInt 1 へ書き込む。同様に、CDR部同志のunify を行なう。次に、第二引数同志のunify を行なう。以上の結果を、図 3.1-5に示す。

次いで、変数Y のようにgoal memory 側のセルを指すpointer がある時（即ち、8bit目が 1であるpointer がある時）、clause memory を指すように調整する（構造体データの時はコピーし、変数がふえる時はいったん補助の変数エリアに確保する）。この結果を図 3.1-6のclause memory に示す。

最終結果を、goal memory に作成し、output buffer へ送出する。（図 3.1-6のgoal memory 参照）

この処理を通じてもわかるように、変数エリアやストラクチャエリアが増加する場合がある。この時でも、各番地は、リテラルエリア先頭番地やストラクチャエリア先頭番地からのdisplacementで表現されているので、ヘッダー部のリテラルエリア先頭番地、ストラクチャエリア先頭番地を変更するだけでよく、unification に伴う処理量を減少させている。

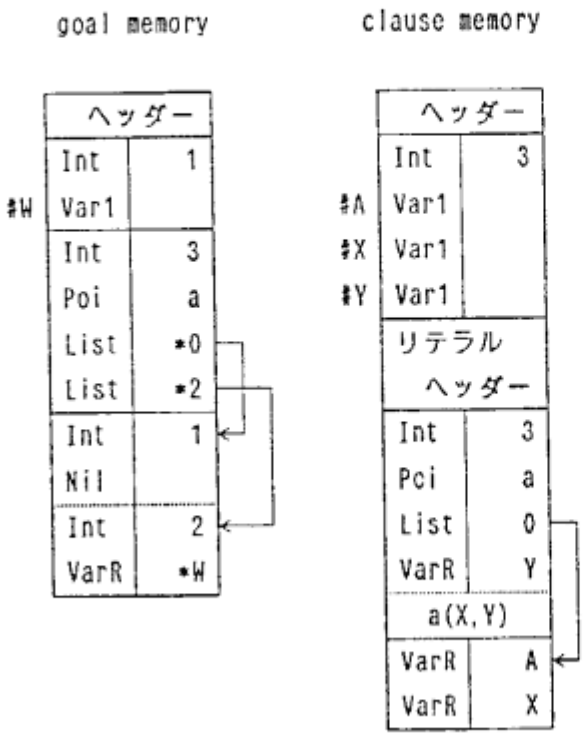

goal memory
clause memory
ヘッダー
Int 1
#W Var1
Int 3
Poi a
List *0
List *2
Int 1
Nil
Int 2
VarR *W
ヘッダー
Int 3
#A Var1
#X Var1
#Y Var1
リテラル
ヘッダー
Int 3
Poi a
List 0
VarR Y
a(X,Y)
VarR A
VarR X

図 3.1-4

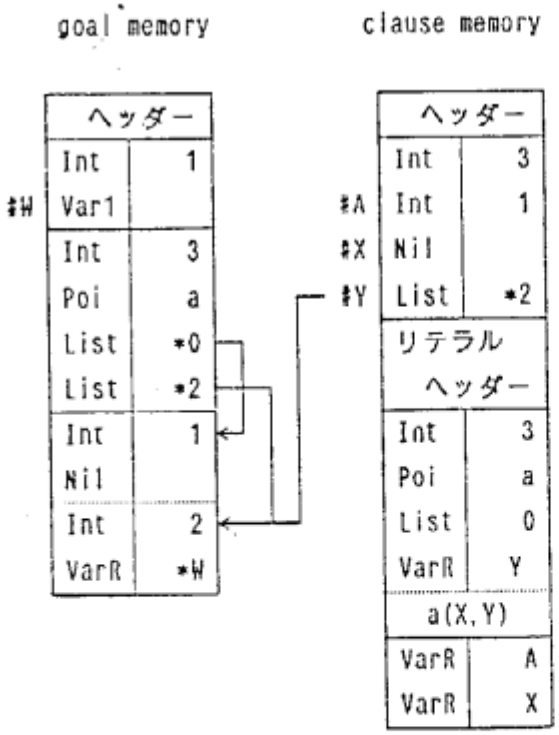

goal memory
clause memory
ヘッダー
Int 1
#W Var1
Int 3
Poi a
List *0
List *2
Int 1
Nil
Int 2
VarR *W
ヘッダー
Int 3
#A Int 1
#X Nil
#Y List *2
リテラル
ヘッダー
Int 3
Poi a
List 0
VarR Y
a(X,Y)
VarR A
VarR X

図 3.1-5

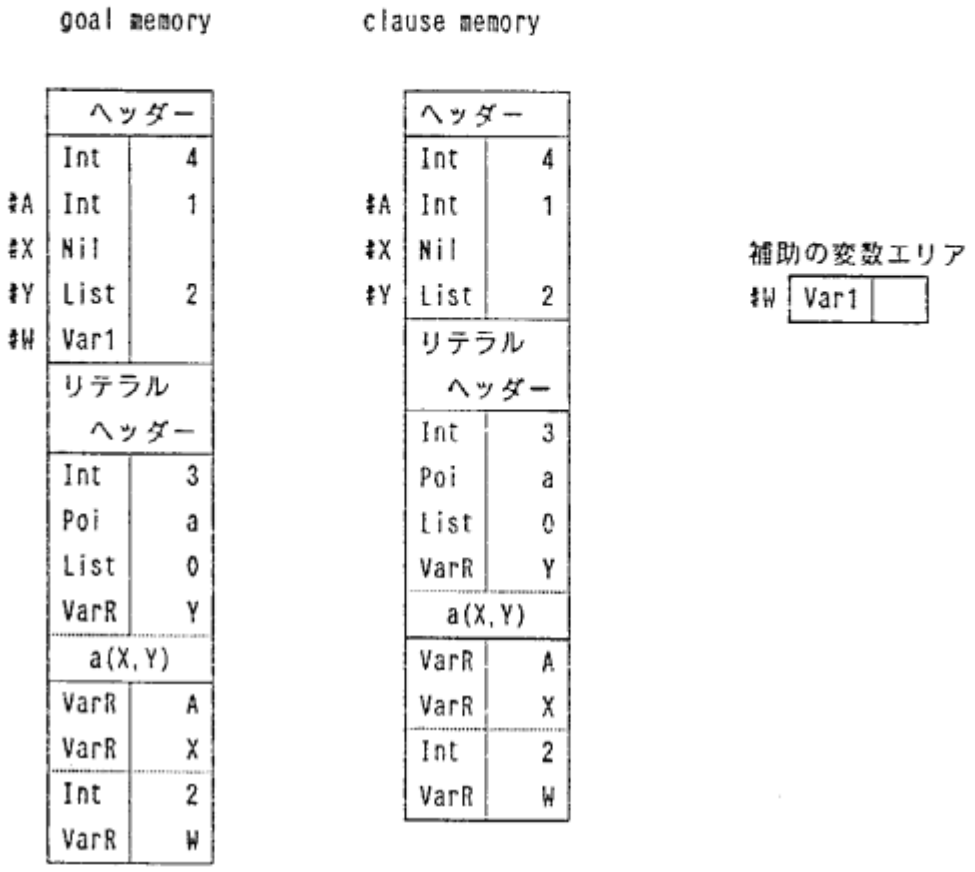

goal memory
clause memory
ヘッダー
Int 4
#A Int 1
#X Nil
#Y List 2
#W Var1
リテラル
ヘッダー
Int 3
Poi a
List 0
VarR Y
a(X,Y)
VarR A
VarR X
Int 2
VarR W
補助の変数エリア
#W Var1

図 3.1-6

3.1.2 Process Pool Unit

本Unitには、二種類のメモリ（Process PoolとMessage Board ）と、二種類のコントローラ（Process Pool Controller とMessage Board Controller）がある。

3.1.2.1 Process Pool とProcess Pool Controller

(1) Process Pool (PP)

PPはプロセスを格納するメモリであり、Cluase Pool と同様、1語32bit である。

PIM-R は、IM間の通信をなるべく少なくするようなプロセス構成法を採用している。まず1つのプロセスは、一つのgoal列の制御情報を格納するProcess Control Block (複数の場合あり、略してPCB ）、Process Control Block の数を管理するProcess Life Block(必ず一つ、略してPLB ）、そして、goal列のテンプレートを格納するProcess Template Block(Process Control Blockと対になるので同数、略してPTB ）から構成され、これらが論理的にひとまとまりとなって同一IMへ割り付けられている。簡単に言えば、goal列内のreducible goalがUUへ送られ、その解がPCB にreturnされると、PTB 内の該goal列はcopyされ、結合情報が代入され、新たなgoal列（PCB とPTB)が同一PLB 配下に生成される。次にこれらのBlock について述べる。（Block 内部形式を付録2に示す。）

①Process Life Block(PLB)

PLB は、プロセスの頂点に位置するBlock で、commit tag, 配下に生成されるPCB 数、それからのreturn数、等が格納される。 commit tagは、このプロセス内のPCB の内、一番最初にguard 部の実行に成功したPCB がONにする。

②Process Control Block(PCB)

PCB 内にはgoal列の状態(ready, run, wait, dead, suspend) 、reduction レベル、OR fork 数(OR 並列Prolog実行時) あるいはAND fork数(Concurrent Prolog実行時) 、return数(fork 先からのreturn) 、Ready Process Queue に連結する際のポインタ等が格納される。reduction レベルとは、プロセス木の根にあたるプロセスの深さを1とした時の各プロセスの深さである。goal列の状態がsuspend の時は、特にSuspend Process Control Block (SPCB)と呼ばれ、PCB との違いは、suspend の原因となったチャネルのPTB 内アドレスがPCB 内に格納されることと、このSPCB配下のPTB にはsuspend したgoalが格納されることである。

③Process Template Block(PTB)

PCB 配下にある時はclause長を除くClause Pool 内clauseと同一の内部形式である。SPCB配下にある時はsuspend したgoalとsuspend した相手clauseのClause Pool内アドレスとが格納される。

(2) Process Pool Controller (PPC)

(a) プロセス生成、更新、消滅処理

新たなプロセスは、新たなresolvent がUnification Unitから戻されてきた時に生成される。プロセスの更新とは、新たなPCB とPTB の対の生成である。旧PTB から新PTB が生成される際に 3.1.1で述べた逆順コンパクションが行なわれる。次に例を用いてこれを説明する。

(例 3.1.2-1) PP内でのプロセス更新処理と逆順コンパクション

まず、clause p(1,[X | Y]):- q(1,X) & r(X,Y).

のClause Pool 内clause definition 部を次の図 3.1-7に示す。ただし、'&' は逐次AND operatorとする。

・このようなclause definition 部はunification に成功するとProcess Poolへ送られ、プロセスのProcess Template Blockを構成する。これを図 3.1-8の左側に示す。

ここで、第２語目が 0番地であり、リテラルの格納位置は、リテラルヘッダーの先頭番地からのdisplacementで指される。また、構造体データの値の格納位置は、ストラクチャエリアの先頭番地からのdisplacementで指される。

リテラル数は、リテラルエリアの先頭に格納されている。この時点でリテラル数は、ボディリテラル数+1=3であり、リテラルヘッダーの先頭番地からの displacement 3にあるLit 12により、q(1,X)がreducible であることが示され、これをUnification Unitへ送る。

今、Unification Unitで、unit clause とunify し、q(1,2)がProcess Poolへもどってくると仮定する。

Prologの場合は、複数の解がreturnされる可能性があるので、まず、このProcess Template Block全体をコピーしてから結合情報X=2 を書き込む（この結果変数エリア、ストラクチャエリアが変化する場合がある）。

このままでは、Process Poolのメモリ使用効率が悪いので、不要となったリテラルq に相当する部分をリテラルエリアから抜き、ストラクチャエリアを前に詰め、リテラル数を-1する。この時、リテラルは逆順に格納されているためリテラルエリアの後から抜くだけでよく、構造体データの値の格納位置は、ストラクチャエリアの先頭番地からのdisplacementで指されるために、ポインタのはりかえは必要としない。また、リテラル数を-1することによって、次にreducible であるリテラル r(2,Y) が指される。即ち、リテラル数はrun,wait あるいはreducible なgoalリテラルを指している。

以上の操作後の状態を図 3.1-8の右側に示す。

ストラクチャエリアが変化する場合、即ち、新たに構造体データが追加される場合は、ストラクチャエリアの後につなげればよい。

変数エリアが変化する場合、即ち、新たな変数が追加される場合は、リテラル・ヘッダー先頭番地とストラクチャエリア先頭番地を変更する。

組込み述語や、Concurrent Prolog の場合は解は１つだけ（生き残れる兄弟プロセスは１つ）なので、解がreturnされてきた時、Process Template Block をコピーする必要はなく、この逆順コンパクションによりclause長が長くならない時は、直接overwrite してよい。

この方式により、コピー量、Process Poolの使用量を低減することができる。

Int	23				clause長	
Int	21		#0		ストラクチャエリア先頭番地	ヘッダー
Int	5		#1		リテラルヘッダー先頭番地	
Int	2		#2		変数個数	
Var1	X		#3			変数エリア
Var1	Y		#4			
Int	3		#5	#0	literal count	
Lit	4	p	#6	#1	ヘッドリテラル	リテラル
Lit	8	r	#7	#2	ボディリテラル 2	ヘッダー
Lit	12	q	#8	#3	ボディリテラル 1	
Int	3		#9	#4		
Poi	p		#10	#5	ヘッドリテラル	
Int	1		#11	#6		
List	0	[X | Y]	#12	#7		
Int	3		#13	#8		リテラル
Poi	r		#14	#9	ボディリテラル 2	エリア
VarR	3	X	#15	#10		
VarR	4	Y	#16	#11		
Int	3		#17	#12		
Poi	q		#18	#13	ボディリテラル 1	
Int	1		#19	#14		
VarR	3	X	#20	#15		
VarR	3		#21	#0	CAR Element	ストラクチャ
VarR	4		#22	#1	CDR Element	エリア

図 3.1-7 p(1,[X | Y]):- q(1,X) & r(X,Y).のclause definition 部

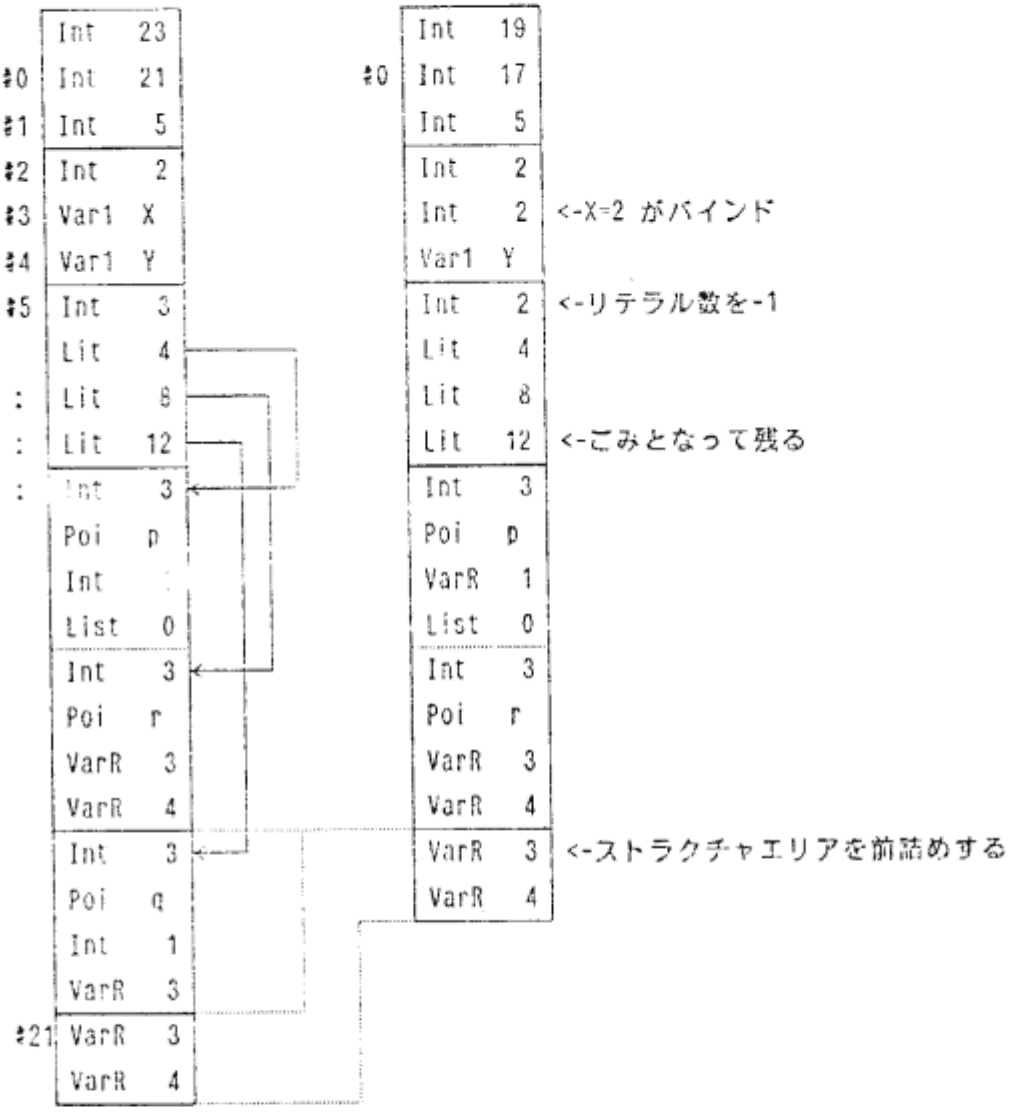


図 3.1-8 q(1,2)がreturnされた時のが逆順コンパクションの結果

前述したように、Process Life部には、このプロセス内のProcess Control 部の数に関する情報（PCB 数とreturn数）が格納されている。また、各Process Control 部には、fork count fieldがあり、このProcess Control 部の下にあるプロセス数に関する情報(OR fork数あるいは、AND fork数とreturn数）がセットされている。次にこれらのPCB 数、fork数、return数の設定、更新について述べる中でPPC による以下の各処理について説明する。

・fork処理（OR/AND fork 数の設定、更新とfork数down処理）

・子プロセス処理（子プロセスの成功、失敗、suspend 処理および親プロセスへの結果報告処理）

・commit処理

・deadプロセス処理

[PLB 内のPCB 数の設定と更新]

①Prologの実行はAND 逐次に実行されるので、最初PCB 数は１であり、そのPCBと対になっているPTB からreducible なgoalがcopyされ、Unification Unit(UU)へ送られる。もし、そのgoalがUUでunit clause とのunify に成功し、その結果がPCB に戻ってきた時、まだgoal（複数の場合もある）があれば、PTB をコピーし、結合情報を代入し、compactionして、新たなPCB とPTB を生成する時にその上のPLB 内のPCB 数を＋１する。 UUへ送られたgoalが新たな子プロセスを生成し、その子プロセスからの成功報告（その中には結合情報が含まれる）の時にはPCB 数は＋１されない。

②Concurrent Prolog 実行の際には、OR関係にあるものは、別々のプロセスではなく、一つのプロセス内の別々のPCB(勿論PTB は附随している）となる。よって、プロセスが最初に生成された時、PLB 内の生成されたPCB 数は、OR fork 数に等しくなっている。 そしてPCB の内の一つがcommitに成功した（guard 部が成功し、PLB 内のcommit tagをOFF からONにできた）時は、今度はbody部の実行に入るから、PLB 内の生成されたPCB 数を＋１する。ただし、Concurrent Prolog の場合は、commitできるPCB は一つなので、PTB をコピーせずに、PTB へ直接書き込み、PCB のみ新たに生成する。

[PLB 内のPCB return数の更新]

①PCB がdead（fork数＝return数）になったならば、PPC はdeadプロセス処理時にその上のPLB 内のPCB return数を＋１増加させる。

[PCB 内のfork数の設定と更新]

①Prolog実行の際、Unification UnitからOR fork 数（よって、unify は成功した）が戻ってきた時は、そのOR fork 数だけ、fork数を増加させる。

②Concurrent Prolog では、AND 並列関係にあるgoalが別々のプロセスとして生成される。よって、並列AND operatorで結合されたgoalが現れた時に、PCB 内のfork数にAND fork数をセットし、それらAND 関係にあるgoalを分配方式に従って、自IHあるいは他IHに分配する。

[PCB 内のreturn数の更新]

①UUからUnification 失敗報告がPCB に戻ってきた時と、commitに失敗した時（guard 部の処理に成功したが、PLB のcommit tagが既にONであった）は、fork数＝return数にする。

②UUにおいて、unit clause とのunify に成功し、その結果がPCB 戻ってきた時は、return数を＋１増加させる。

③自IH内にある子プロセスがdead（PLB の生成されたPCB 数＝return数）になったならば、PPC のdeadプロセス処理時にreturn数を＋１増加させる。

④他IHにある子プロセスがdead（PLB の生成されたPCB 数＝return数）になり、そのIH内PPC のdeadプロセス処理時に生成されたfork down packetがこのIHに到着した時、return数が＋１される。

Process Life Blockをプロセス内に導入することにより、あるプロセスの中で、新たなgoal列がいくつ生成、消滅しようとも、その生成、消滅のたびごとに、そのプロセスの親プロセスへ報告する必要はなくなる。よって、このようなプロセスの構成法を採用することにより、IH間network 通過packet数を低減することができる。また、メモリに余裕があれば、PPC でのdeadプロセス処理を休止することにより、fork down packetの生成を押え(Networkトラフィック低減）、PPC を他の処理にふりむけ、解を速く求めることもできる。次に例を用いて、PLB,PCB,PTB 間の関係と、PLB,PCB,PTB から構成されるプロセス間の関係について説明する。

(例 3.1.2-2) Prologの場合

プログラム　?-go.

```
go:-a,b.
a:-a1.          b:-b1.
a:-a2.          b:-b2.
a:-a3.
(a1,a2,a3,b1,b2 以下は省略)
```

このプログラムは、まずPTB がgo:-a,b.なるプロセスが生成され、このgoal 列 a,bの内のreducible goal a がUUへ送られreduction される (OR fork は 3)。その結果、３つの子プロセスがgo:-a,b のPCB 配下に生成される。そして、その内の一つから解 a:-a1が返されると、新たなPCB とPTB (go:-b) の対が生成され、今度はgoal b がreducible になる。このb がUnification Unitへ送られ、OR fork 関係にある二つのプロセスが生成されたところを図 3.1-9に示す。

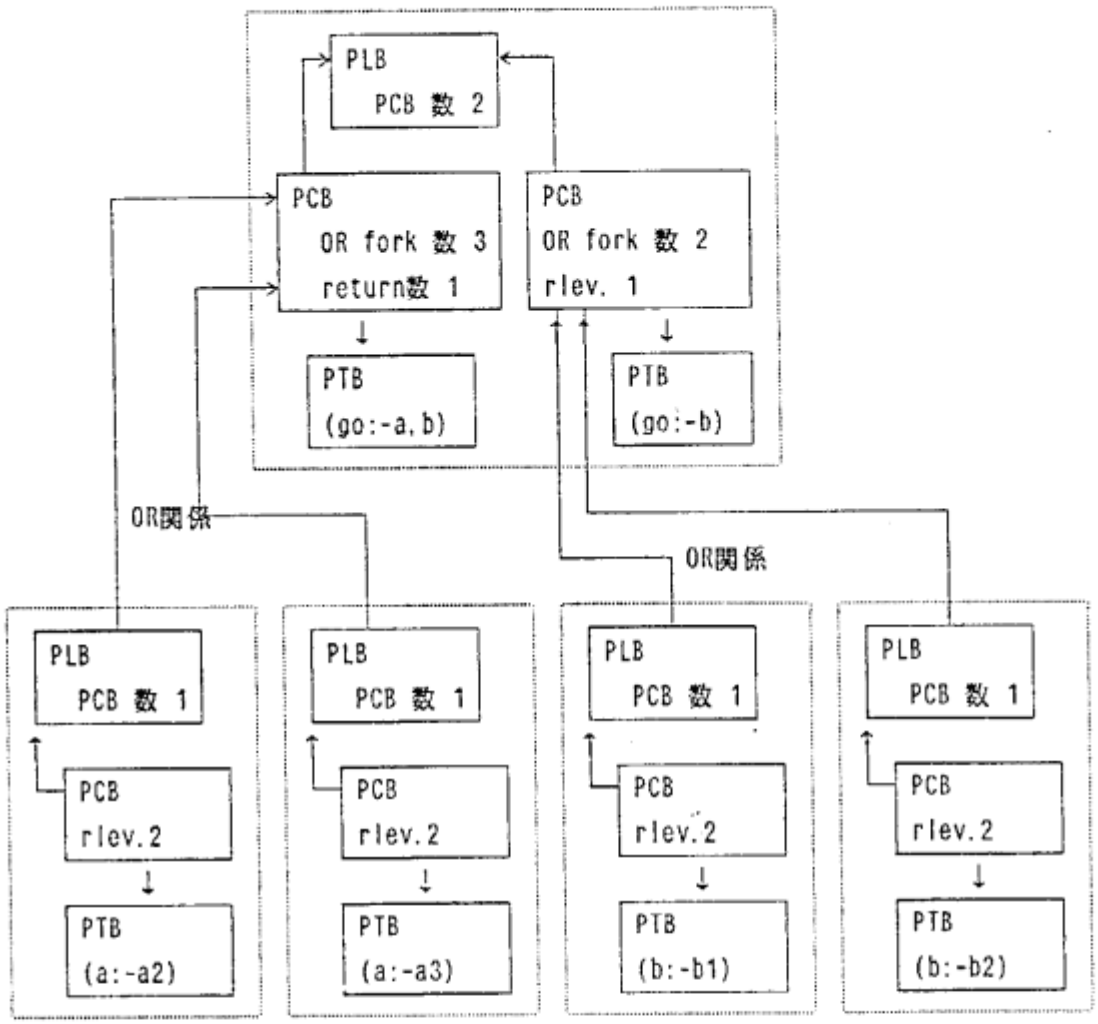


図 3.1-9

(return数が 0、or fork 数が未定義の時は省略。rlev. はreduction level の略。
点線で囲んだ部分が1つのプロセスであり、同一IH内Process Poolに格納される。)

(例 3.1.2-3) Concurrent Prolog の場合

プログラム ?-go.

```
go:-true | a,b.   (',' は並列AND オペレータ)
a:-ag1   | ab1.          b:-bg1   | bb1.
a:-ag2   | ab2.          b:-bg2   | bb2.
(ag1,ag2,ab1,ab2,bg1,bg2,bb1,bb2のclauseは省略)
```

このプログラムは、まず、goが実行され、trueの処理（これは成功）の後にcommitし、プロセスが更新される。そして、次にgoal aとb がAND forkされて別々のプロセスとなり、それぞれ並行に実行される。a とb のreduction 直後のプロセス間の関係を図 3.1-10 に示す。Concurrent Prolog ではOR関係にあるclauseは同一PLB 配下のPCB-PTB 対となり（即ち、同一IHに格納される。）、guard 部の実行を競う。

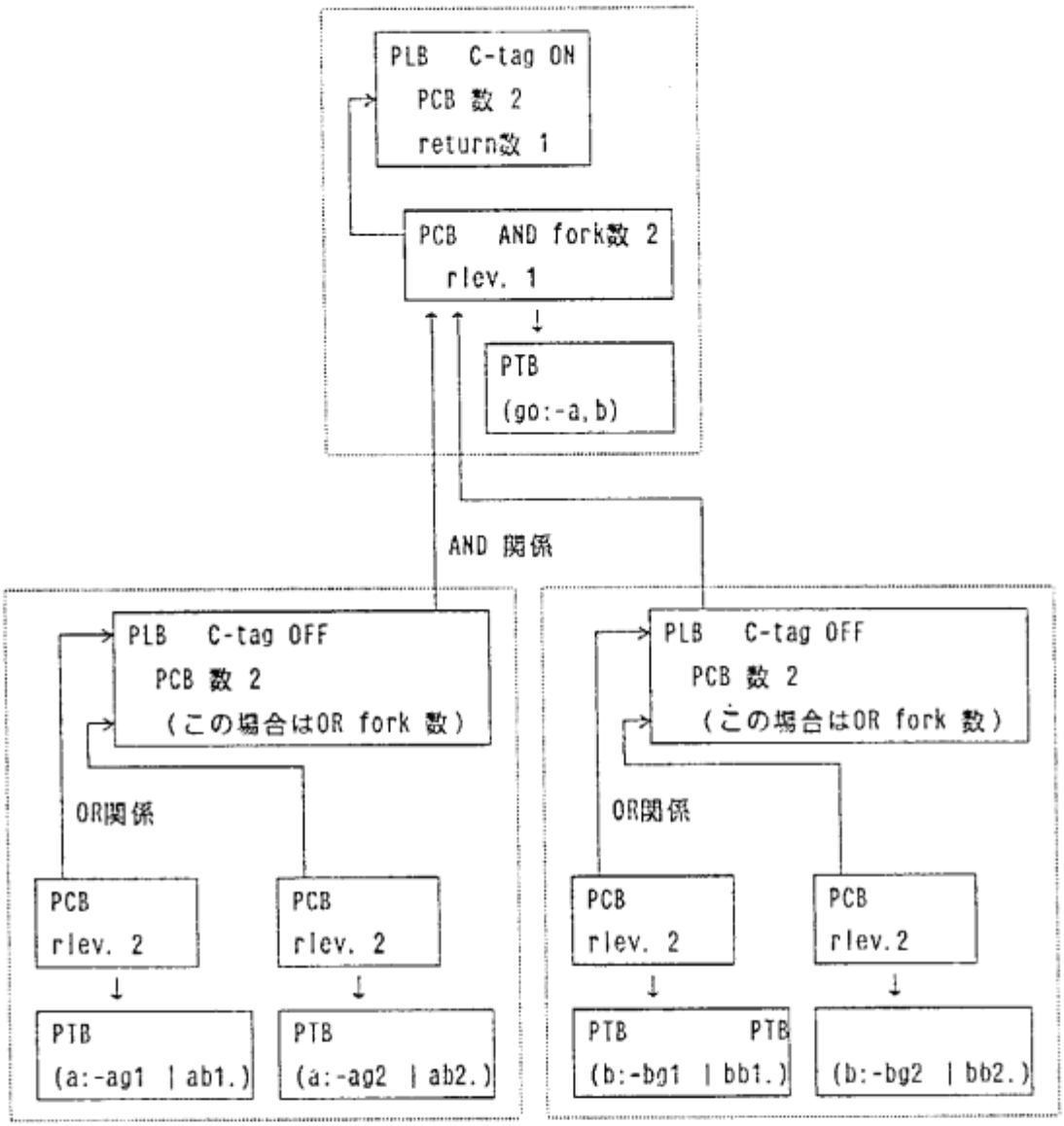


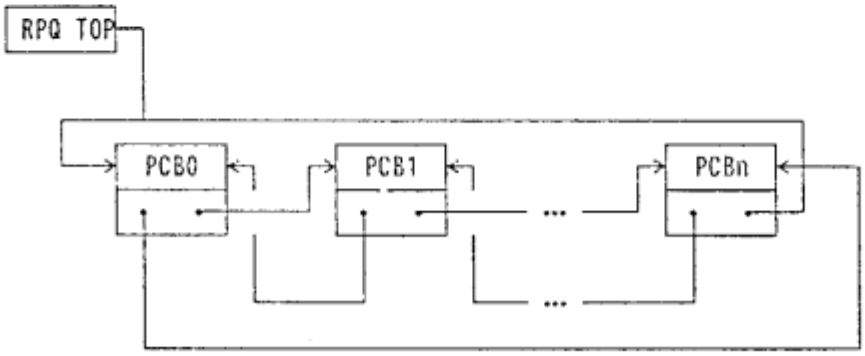
図 3.1-10

(return数が 0、or fork 数が未定義の時は省略。 rlev. はreduction level の略)

(b) reduction レベルを用いたPIM-R における局所的プログラム実行戦略

PIM-R では 100台あるいはそれ以上の台数のIMを結合することを考えており、そのようなシステムにおいては、General Scheduler が存在して、それが全体の実行戦略を制御することはむずかしい。そこで、PIM-R ではプログラムの実行制御を1台のIM内の局所的なものにとどめ、かつ、効果のある方式を導入している。

reducible なgoalを含むgoal列の制御情報を保持するPCB(Ready PCB と呼ぶ) は図のように、IM内Ready Process Queue(RPQ)に連結される。



特にプログラム実行戦略を指定しなければ、PPC はReady PCB をRPQ の最後尾に繋ぎ、また、先頭のReady PCB をUUへ転送する。 しかし、プログラムによっては、複数ある解のうち、一つだけを求めればよいという場合がある。そのようなプログラム実行戦略のために、Reduction Level を使用する。Reduction Level のより大きい (プロセス木の根からより遠い、つまり、より深い) Process Control Block をReady Process Queue のより先頭に置くことにより、IM内で疑似depth-first な Reduction 戦略をとることができる。もちろん、Reduction Level のより小さな (プロセス木の根により近い、つまり、より浅い) Process Control Block を、Ready Process Queue のより先頭に置くことによってIM内での疑似breadth-first なreduction 戦略をとることもできる。このようにReduction Level によりIM内での局所的なreduction 戦略を制御することができる。

3.1.2.2 Message Board (MB) とMessage Board Controller(MBC)

Concurrent Prolog におけるチャネル変数用の分散化共有メモリとして、Message Board を設けている。 Message Board は、Process Poolとはgarbage collection法が異なるので、別領域のメモリとし、Process Pool Controller の負担を軽くするために、Message Board Controllerにより各種処理を行なう。

(1)Message Board(MB)

チャネルとは、clause内の並列AND オペレータで区切られたリテラル間の共有変数である。

(例 1) go:-p1(X),p2(X),p3(X).

X はチャネルである。

(','は並列AND オペレータ)

(例２) go:-p1(X),c1(X?),c2(X?).

X はチャネルであり、X?を特にreadチャネルと呼ぶ。

(例３) 次の例では、X はチャネルではない。なぜならば、& は逐次AND オペレータだからである。

go:-p1(X)&c1(X?).

このように、PIM-R ではチャネルとして使用される変数（チャネルと呼ぶ）と、それ以外の変数（変数と呼ぶ）とを区別し、チャネルはMBに格納する。また、チャネルの性質は実行時に動的にinherit する。たとえば、clause側引数内変数は、goal側のチャネルとunify されるとチャネルとなる。

MBは、channel cell, 値cell,suspend process list から構成される。

channel cellの構成を次に示す。

write tag	suspend tag
値cellへのpointer	
suspend process 個数	
suspend process list先頭番地	

channel cellは一つのチャネル変数に対応し、各チャネルが保有するチャネルIDは、このchannel cellが存在するIH番号と、MB内のこのchannel cellの先頭番地から構成される。 値cellは、producerプロセスが送った値が格納される（値cellの内部形式はClause Pool に準ずる）。suspend process listはsuspend process へ

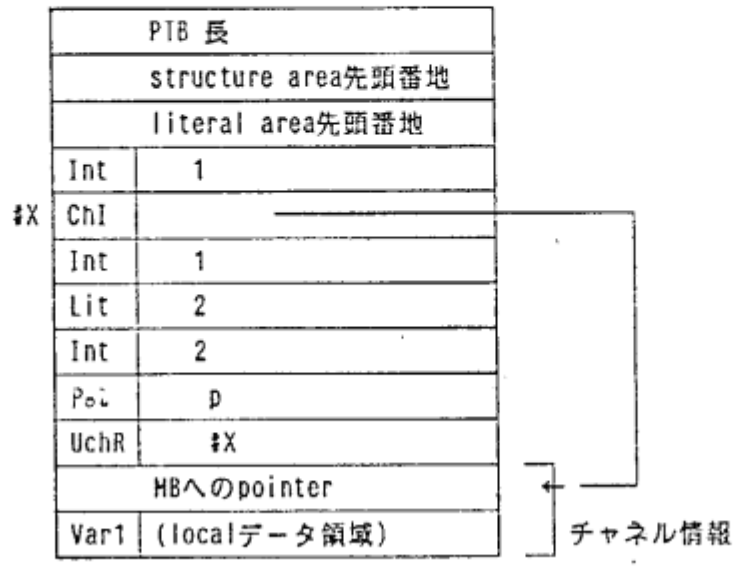
のpointer を格納する。suspend process 自体はPPU 内のProcess Pool に格納される。

(2)Message Board Controller

Concurrent Prolog において、consumerプロセスが、suspend すると、PPC は、suspend の原因となったチャネルセルを格納しているMBのMBC へチャネル値read要求を出す。MBC は、producerプロセスからメッセージが既に送られて来ているかチャネルセルをcheck する。もしメッセージが到着していれば、PPC へconsumerプロセスのactivate要求（そのメッセージ値を含む）を出し、到着していなければMBCはSuspend Process List(略してS.P.List) にconsumerプロセスのProcess Pool内番地を書込む。consumerプロセスがsuspend した時、該当MBが他IMにある場合はチャネル値read要求パケットをnetwork 経由でおくる。そして、メッセージが到着した時は、その他IM内MBC からconsumerプロセスへそのメッセージを含むactivate要求パケットが送られる。producerプロセスからのメッセージ（即ち、チャネル値）がPPC からMBC へ送られると、MBC はMB内の値セルにそのメッセージ（即ち、チャネル値）を書き込み、もしS.P.Listにsuspend 状態のconsumerプロセスが登録されていれば、それにこのメッセージを含むactivate要求を送る。suspend プロセスが存在するのが、他PPU 内Process Poolの場合、MBC はnetwork 経由のパケットとして、そのメッセージを含むactivate要求を送る。

チャネルは、PTB のstructure area内の二語の“チャネル情報”で表現される。

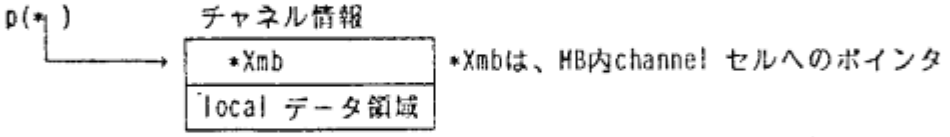
たとえば、goal列　p(X),c(X?)があった時AND forkしてIH内Process Poolに置かれたp(X)のPTB は次のようである。



(ChIは、チャネル情報へのポインタを格納するセルのデータタイプ)

第一語は、HB内該当チャネルセルあるいは親プロセスのチャネル情報へのポインタであり、第二語は、local データ領域である。guard が成功し、commitした時に、このlocal データがHB内該当チャネルセル、あるいは親プロセスのチャネル情報内local データ領域に書き込まれる。以下、例を用いてHBへのチャネルセルの確保について説明する。

(例 3.1.2-4)　goalは前述のp(X)であったとする。(よって、X はチャネルである)　ここで、簡単のためにPTB を次のように略記することにする。



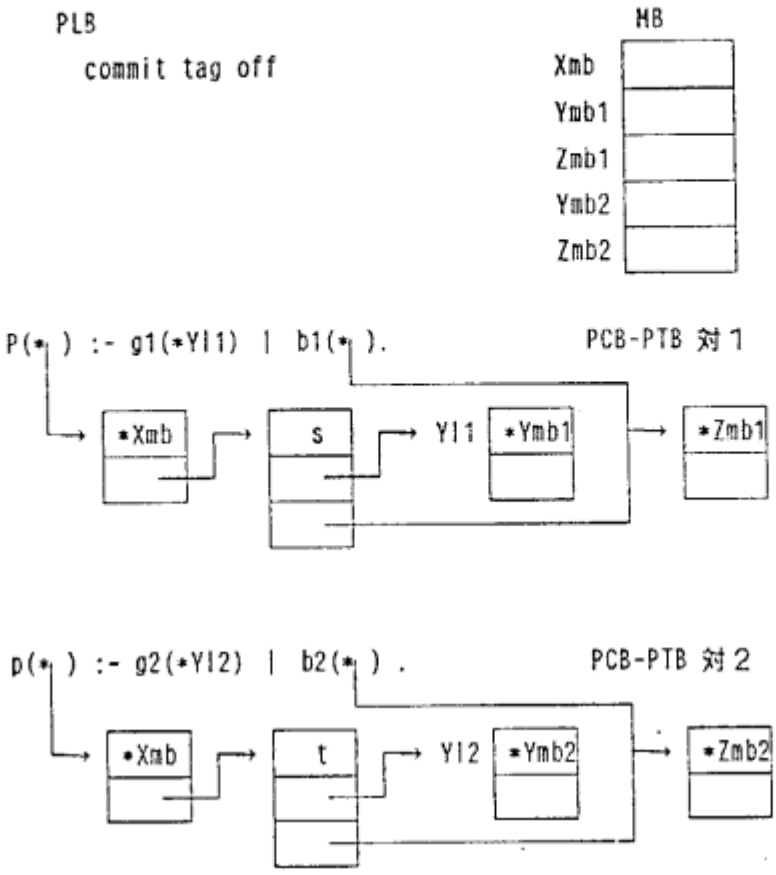
(以後これをさらに略して　p(**Xmb)と記述することがある)

OR関係にあるclause

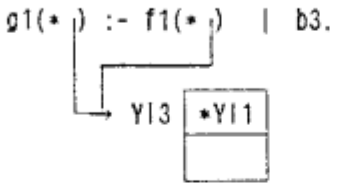
```
p(s(Y,Z)) :- g1(Y)  | b1(Z).
p(t(Y,Z)) :- g2(Y)  | b2(Z).      (まだY,Z はチャネルではない)
```

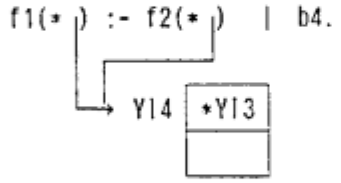
goal p(**Xmb)とのunify 時にチャネルの性質はinherit するので、各Y,Z は、チャネルとなり、OR clause ごとにHB内にセルが確保される。一方、g1 , g2のY に関しては，並行関係にあるgoalが存在しないので、このg1,g2 のY は、子プロセスレベルではチャネルとしては使用されない。よって、このg1, g2の Y のために、HB内に新たにセルは確保せず、head側のY とチャネル情報を共有する。（本並列環境では、reduction 時には引数はeager copyされる）

reduction 結果は、次のとおり。



g1 ,g2がそれぞれ成功すると、これらのPCB-PTB 対は、PLB 内のC-tag チェックを行う。そしてC-tag を先にONにしたPCB-PTB 対のみがcommitし、local データをMB内のXmb に書き込む。以下のようにguard g1がネストする場合は、f1,f2 には、やはり並行関係にあるgoalはないので、チャネル情報をheadチャネルと共有する。各local データの内容は、PCB-PTB 対が成功した時、結合情報として親PCB-PTB 対のlocal データ領域に返されていき、g1(*Y11)が成功し、commitする時に、local データがMB内のXmb ,Ymbに書き込まれる。ネストの先で、suspend が起り、PCB(SPCB)-PTB 対となった時は、チャネル情報を逆にたどり、MB内チャネルセルにアクセスして、値が既に書き込まれていれば、取り込み、まだならば、suspend プロセスリストに登録する。





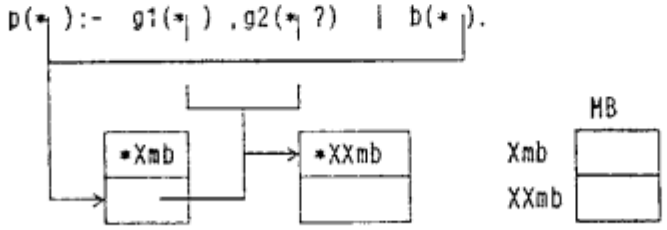
本例で、例えば、PCB-PTB 対１のguard が先に成功し、その時Yl1 のlocal データ値がY-val とすると、commitした後はHBへの書き込みが起り、次のようになる。

(例 3.1.2-5)

guard 内のあるチャネルに関し並行goalが存在し、各並行goal内チャネルのうちに、少なくとも一つreadチャネルでないチャネルがある時は、そのチャネルのために新たにHB内にセルを確保する。

```
p(**Xmb )                          reducible subgoal 側
p(X):- g1(X),g2(X?) | b(X).        clause側
```

reduction の結果、



となる。g1とg2は独立したプロセスとなり、HB内のセルXXmbを通じてメッセージをやりとりする。そして、g1,g2 が共に成功し、commitした時にXXmbの値をXmb に書き込む。

3.2 Structure Memory Module(SHM)

Structure Memory Module （以下SHM と呼ぶ）は、リストやベクタ等の構造体データのうち、ある一定の条件を満たす構造体データを格納し、unification に際しては必要な構造体データをIM内のUnification Unitに転送する。

図 3.1-1に示すように、SHM はIM-SM Network を介して多数台のIMと接続されるので、ある一定の条件を満たすSHM 内に格納される構造体データは多数台のIM群より共有される。このためSHM の構成に際しては、将来のVLSI化のことも考慮し、以下の点に特に留意しながら検討を進めている。

①メモリアクセス競合の集中化を避ける。
②メモリアクセスの頻度を減らす。
③ネットワークや接続バスの信号線を減らす。
④処理の実行中における不要セルの回収(Garbage Collection)をなるべく行わない。

上記①，②のメモリアクセス競合等の対策としては、

①SHM をBank分け等により複数のSHM に分散化することにより、SHM の共有化により生じるアクセス競合の集中化を避ける[Ito 83]。
②数台のIMに対してSHM を１台接続したものを単位とするモジュールを構成し、それを階層的に組合わせることにより全体システムを構成する[Moto 84] 。
③数台のIMに対して、同一の内容を持つSHM を１台ずつ接続して行き全体システムを構成する（図 3.2-1）。

等の方法が考えられるが、現段階では上記③の方法によりメモリアクセス競合の集中化の回避、及びメモリアクセスの頻度を減らす。この場合SHM の内容は同一であるため、各IM内のunification 時にSHM から構造体データを読出す時には各IMにそれぞれ接続されているIM-SM Network を介して構造体データは読出される。

また、リストやベクタ等の構造体データが未定義変数を含んでいて、他のプロセスからも共有されている場合、unification の結果その未定義変数がある値に結合されるとそのリストやベクタ等の構造体データ全体をコピーすることが必要となり、このコピー・オペレーションが頻繁に発生するとメモリスペースの節減、処理の高速化に対して問題となる。従って、基本的には未定義変数を含まないリストやベクタ等の構造体データをSHM 上の専用メモリに格納することにより部分的に構造体データを共有する方式を採用することで処理の高速化、資源の有効活用を図る。

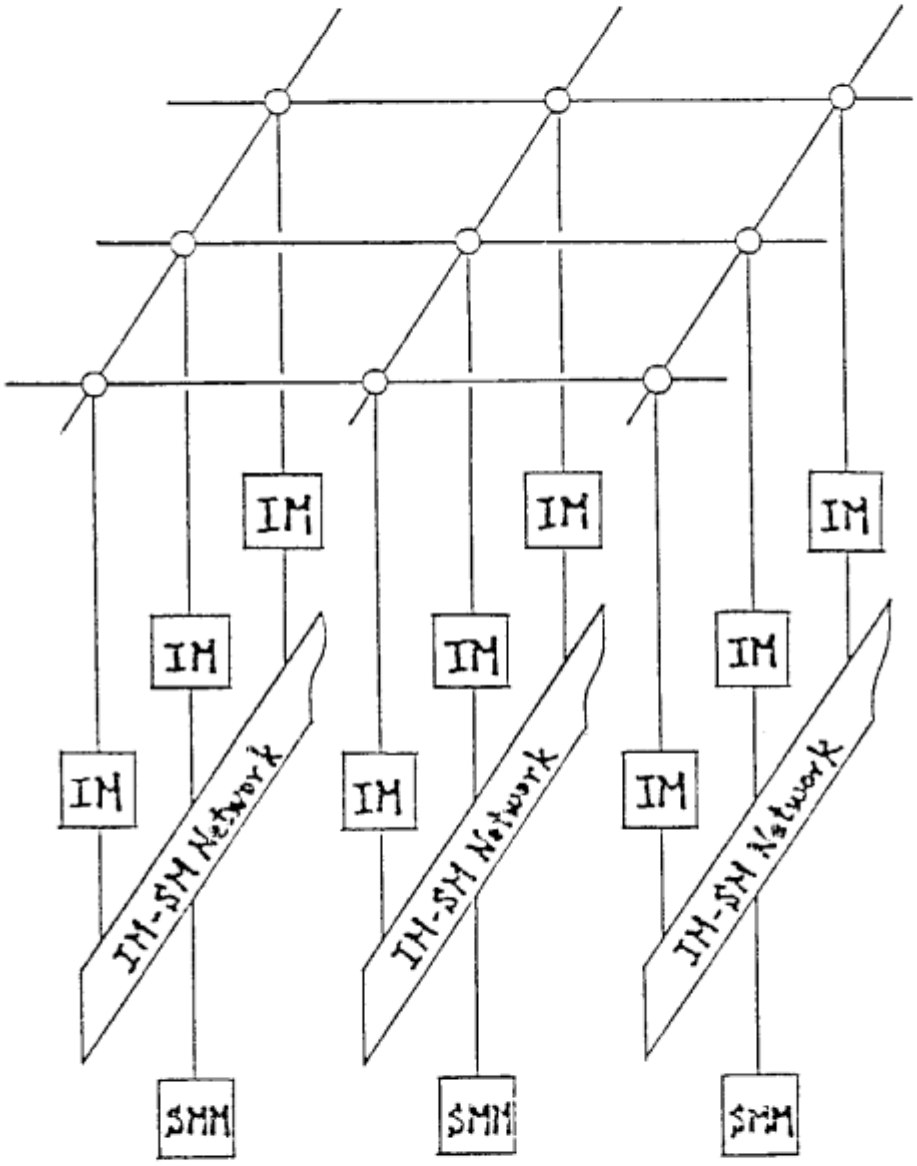


図 3.2-1 SMM を接続した全体システム構成図

(1) 構造体データの共有方式

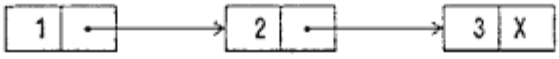
①共有化のレベル

リテラルレベルの共有、構造体データ全体レベルの共有、Ground Instance だけの共有等各種の共有化のレベルが考えられるが、共有度は低いが最も基本的なGround Instance だけの共有方式を採用する。即ち、現段階ではプログラムのCompile 時にclause中の構造体データのうち、SHM に格納すべき未定義変数を含まないGround Instance の部分の切り分けを行い、それをSHM に格納する。この方式では、読出しだけを行い、書き換えの生じないGround Instance の部分だけを共有するので、次々と発生するプロセス間の独立性を高く保つことが出来るという特徴がある[Hira 83] 。

なお、clause中の構造体データのうちlengthや構造等の条件指定を行うことにより、Compile 時にSHM に格納すべきGround Instance の切り分けを行い、それをSHM へ格納する訳であるが、現段階ではプログラマによってGround Instance の指示を行うことによりGround Instance の切り分けを行う。

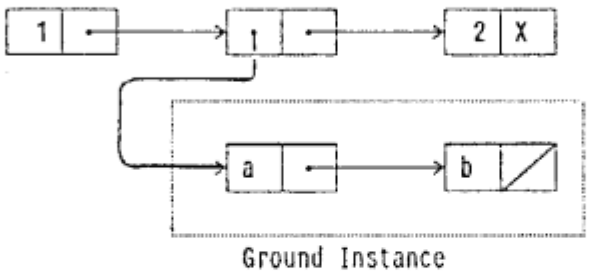
（例１） Ground Instance を切り分けられない例

[1,2,3 | X]



（例２） Ground Instance を切り分けられる例

[1,[a,b],2 | X]



ここで、（例１）の場合には先頭の２セルには未定義変数が含まれないが、最後のセルが未定義変数X を含むために全体が共有化の対象とはならない。しかしながら、（例２）の場合には[a,b] がGround Instance として切り分けられ、共有化の対象となり得る。

②共有化のメカニズム

clause中の構造体データのうち、指定されたGround Instance の部分がSHM に格納され、そこへのポインタが持ち運ばれることによりSHM 内の構造体データがIH群から参照される。この時、clause中のポインタによって参照されている構造体データはUnification による新しいプロセス生成時にそのポインタが伝播されて、新しく発生するプロセスやgoalからもこの構造体データが参照されることになり共有化される。(図 3.2-2, 図 3.2-3) その結果、新しく発生するプロセスやgoalも長い構造体データではなくポインタを持ち運ぶことになり、プロセスやgoalの大きさが小さくなり、サイズの大きいリストやベクタ等の構造体データを多数含むプログラムに対しては高速化が期待できる。

図 3.2-2ではclause中の構造体データのうちSHM に格納されているGround Instance の部分がunification 後新しく発生するプロセスにポインタが伝播されて行き、Ground Instanceの部分が共有化される状況を示す。同様に、図 3.2-3ではreducible goal中の構造体データのうちSHM に格納されているGround Instance の部分がunification 後新しく発生するプロセスにポインタが伝播されて行く状況を示す。

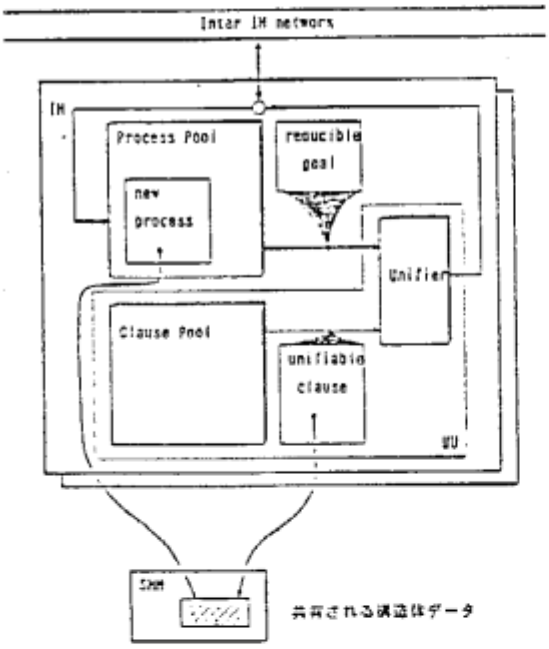


図 3.2-2 clause中の構造体データが新しく生成される process と共有される場合の概念図

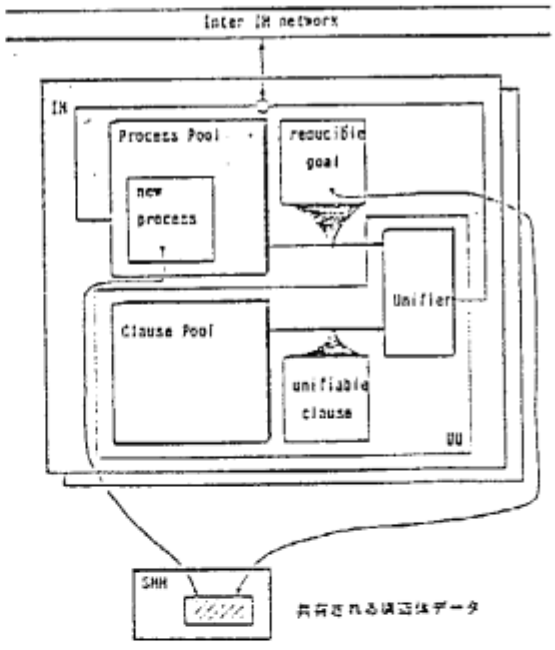


図 3.2-3 goal中の構造体データが新しく生成される process と共有される場合の概念図

(2) SHM の構成

①構造体データの格納形式

構造体データを格納する方法にはリスト形式と連続するメモリセルにデータを格納するレコード形式があるが、リストとベクタの格納メモリを分離することにより上記の両形式の混用で検討を進めている。即ち、リストに対してはリスト形式で、ベクタ等の構造体データに対しては一次元配列のレコード形式でそれぞれ専用のメモリに格納する。具体的には、リストに対してはList Data Memoryに格納し、またベクタ等の構造体データに対してはVector Address Table(VAT) を介してVector Data Memoryに格納する。

リスト形式のリストについては、メモリの使用効率は落ちるが、必要に応じてポインタを1段ずつたぐりながらリストデータを取り込むなどの実現が可能で、ポインタを利用して容易にデータを取り出せる利点がある。一方、レコード形式のベクタについては、アドレステーブル経由でデータを読出すことによるオーバーヘッドや高速な連続読出しの実現などの問題があるが、ベクタのサイズが大きくなるにつれてアドレステーブルを読出すオーバーヘッドの割合いは減少するし、またメモリの使用効率が良いなどの利点がある。

②構造体データの格納方法

構造体データのGround Instance の部分を格納する方法としては、基本的には同一の内容に対しても複数のコピーを持たせるコピー方式を採用するが、実行時においては子プロセスの生成の時にポインタが受け渡されて行くことになり、同一の構造体データが複数のポインタにより多重参照されることになり共有化される。これにより、参照しているポインタのアドレスが一致しているかどうかにより容易にunification の成功／失敗を判別することが一部可能となる。また、この方法ではメモリスペースは多量に必要とされるが、SHM に格納して行く時に同一の内容が存在するかどうかのチェックは不要であり、且つ同一データへのポインタによる参照が分散化でき多数台のIH群からのメモリアクセス競合の集中化を緩和できるという利点がある。

(例)　[1,2,f(a,[b,c])]

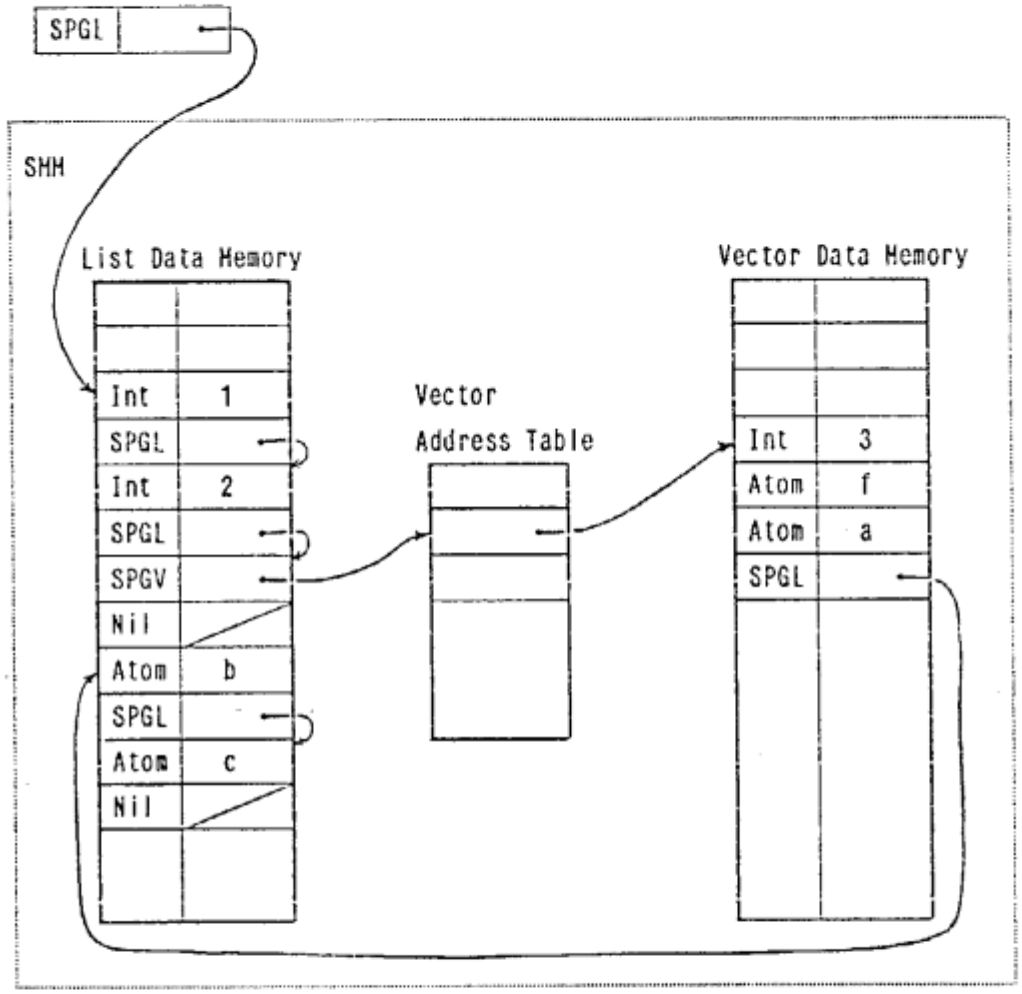


図 3.2-4　構造体データの格納例

(注) 格納例のデータタイプについては［付録１］を参照のこと。
SPGL:Structured Pointer Ground List の略。
SPGV:Structured Pointer Ground Vector の略。

③SHM の構成とデータ転送方式

図 3.2-5に示すようにSHM はStructure Memory Controller (SMC) 、Vector Address Table(VAT)、List Data Memory (LDM)、Vector Data Memory (VDM)から構成される。SMC はIM-SM Network を介してIMからの読出しや書込み等の処理要求が到着した時に起動され、処理を開始する。IM-SM Network は等距離ネットワークであり、現段階では共有バスによる実現を考えている。

Vector Address Table(VAT) にはVector Data Memoryに格納されるベクタに対応して、それらの先頭アドレスが保持される。このため、構造体データのGround Instance のポインタを持ち運ぶプロセス、goal、clause及びベクタのポインタを保持する場合のList Data Memoryからは実際にはVector Address TableのTable Address が参照されており、ベクタの内容を読出す際には必ずVector Address Tableを介してベクタの内容が読出されることになる。この方法ではベクタを読出す時には必ずVector Address Tableを読出すというオーバーヘッドはあるが、ベクタサイズが大きくなるにつれてVector Address Tableを読出すオーバーヘッドの占める割合いは減少するので、メモリの使用効率が上がるという点を考慮すると効果が期待できる。

これらSHM に格納された構造体データの読出しはBlock 単位に行う。即ち、リストの場合にはList Data Memory中のcar 部とcdr 部の２セル（List Blockと呼ぶ）がBlock 転送され、Unification Unit（以下UUと呼ぶ）内のバッファメモリに読出され、ベクタの場合にはVector Address Table 内のアドレスで指示されたVector Data Memory内のベクタの一まとまり（Vector Blockと呼ぶ）がBlock 転送されUU内のバッファメモリに読出される。この時、UU内のバッファメモリに読出されたベクタが、新たに別のベクタやリストのポインタを含んでいる様な構造体データであり、度重なるBlock 転送を必要とする場合などではUU内のバッファメモリは処理中の引数間のunification が終了するまで解放されない。

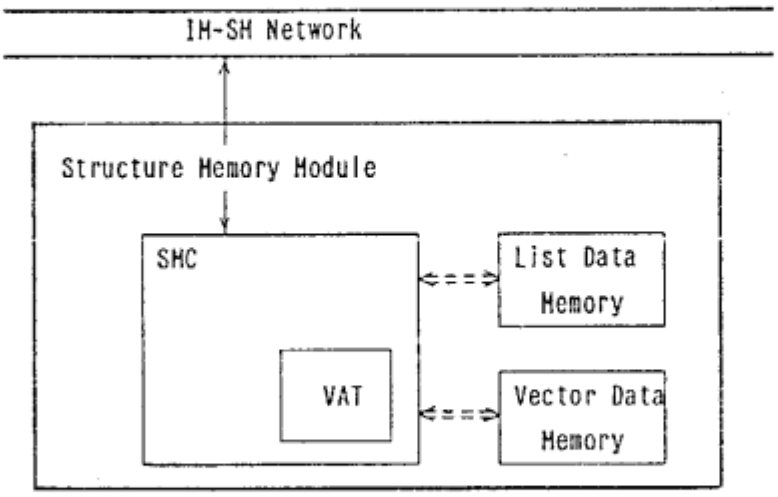


図 3.2-5
SHM の構成図

(3) 引数間のLazy Unification

SHM に格納されている構造体データの参照を必要とするunification は、SHM に対してリスト読出し要求又はベクタ読出し要求がIM内のUnification Unitより送られ、UU内のバッファメモリに構造体データが取り込まれてunification が実行される。この時、引数間のunification は並列には行わず、SHM を参照する引数がreducible goal又は選択されたclauseのどちらか一方に存在し、且つ他方が変数又はAtomでない場合には、その引数間のunification を後回しにし、その他のunification を優先して実行する様にし、それらが全て成功した場合に限りSHM を参照する引数間のunification を開始することにし、無用の引数間のunification を避ける。ここで、SHM に対する読出し要求は現段階ではSHM を参照しない引数間のunification が全て成功した時点で送られることにするが、将来的には読出し要求が先行して送られることにより高速化を図ることも考えられる。また、リテラル内の引数間で変数が共有されている場合でも、並列処理は行わないのでconsistency check は必要とされない。

(4) Garbage Collection

基本的には構造体データの共有化により発生するGarbage Collectionは最小限に押えることが望ましい。このため、現段階では静的に決まるclause中のGround Instance だけをCompile 時にinitial loadする方法を採用し、SHM への動的な書込みは行わないように限定している。従って、この場合にはSHM 中の構造体データは少なくともclauseからの参照があり、一つのQuery による解が全部得られるまではガーベッジとはならないので、現段階ではGarbage Collectionは行わない。

3.3 Network

IM間Network は、近傍PPU 内Packet Switch の入力バッファ長等の情報により子プロセスの各IMの分配を動的に制御できるIntelligent Network Nodeを格子状に配置した構成である。Intelligent Network Nodeは、例えば、Transputerのような通信機能（各10Mbits/sec の４本のチャネル）と、高速のメモリ(サイクルタイム50nsecの4kbytes のstatic RAM) を内臓したマイクロコンピュータを使用しても構成できる。

IM-SHM間Network(IM-SM Network)は等距離Network であり、現段階では共有バスによる実現を考えている。

4．ソフトウェア・シミュレーション

PIM-R における台数効果、アーキテクチャ等の検証のためにソフトウェア・シミュレータを開発し、テストプログラムを走行させ、各種データを収集した。なお、本シミュレータはDEC-10 Prolog/C-Prologで記述されており、その実行はDEC2060/VAX-11上で行なわれる。

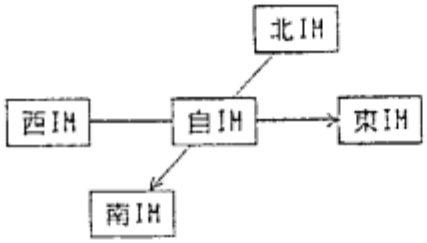
4.1 シミュレーション条件

①ネットワーク上での転送パケットの衝突はないものとする。

②各ユニットの入出力バッファは十分大きいとする。

③PPU/UUの処理時間比は通常のアセンブラ命令で記述した場合の、おおよそのステップ数よりその比を決めた。

④PrologではOR-fork 数が２以上の時に、Concurrent Prolog ではAND-fork時に、新しい子プロセスを各IH(Inference Module)に分散させるが、分配方式は自IH→東IH→南IH→自IH→…の繰り返しとする方式を採用した。（下図）



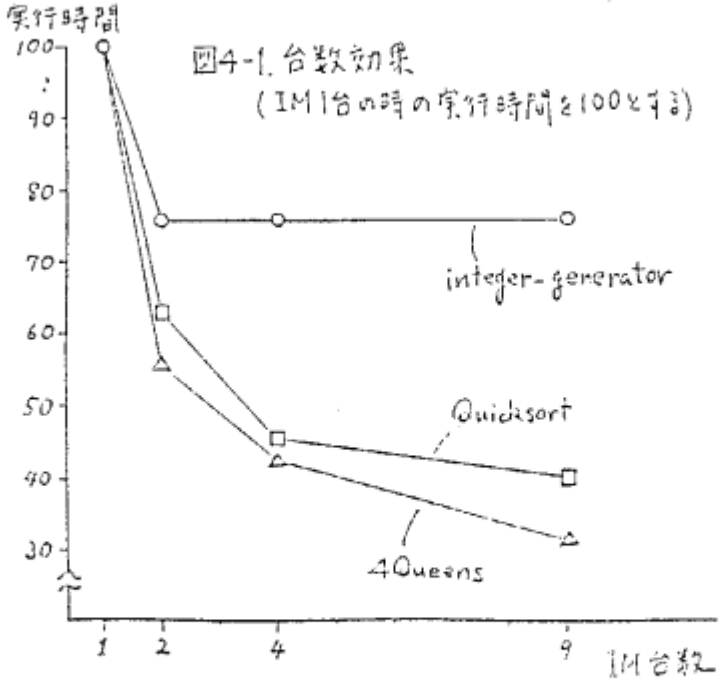


図4-1. 台数効果
（IH1台の時の実行時間を100とする）

4.2 シミュレーション結果

4.2.1 Prolog プログラム

4Queens プログラム（付録4-1)を走行させデータを収集した。

(1) 台数効果（図 4-1）

4Queens では、IH台数増加に伴ない処理性能向上がみられ、 8～ 9台頃から飽和状態に近づく。これは4Queens の動的な平均OR並列度 6.2[Onai 84b]とほぼ対応する。この結果は、PIM-R がPrologプログラムの持つ並列性を引き出していることを示している。

(2) ネットワーク

ネットワーク通過パケットの種類別個数を次表 4-1に示す。

表 4-1

	IH 2台	IH 4台
総パケット数	162	204
true return パケット数	40	44
OR fork パケット数	61	80
fork down パケット	61	80

fork down パケットは、子プロセスがdeadになったことを親プロセスに伝えるためのもので、これは解を求めることには直接は関係なく、garbage collection にかかわることである。そこで、Process Poolに余裕があれば、Process Pool Controller(PPC) におけるdeadプロセスへのマーク付け処理およびfork down パケットの生成、転送処理の優先度を下げ、解を求めるために必要な処理とパケット転送を優先させることができる。これにより、ネットワーク転送パケット個数を、IH 2台 ,IH 4台の場合で、それぞれ約40% 減少させることができる。また、これにより、PPC の処理が軽くなった結果、IH 4台の場合で、 1個目の解および 2 個目の解が求まるまでの処理時間が、それぞれ16%,18% 短縮される。

(3) 稼動率

IH 4台の場合のPPC の稼動率は、各々42%,55%,54%,80% であり、OR fork 時の子プロセスの分配方式を固定的にしために、それほどはバランスしていない。この時、Packet Switch の平均入力バッファ長は、 0.46, 0.69, 0.49, 4.5個であり、PPC 稼働率と相関を持っている。そこで、Intelligent Network Nodeにより Packet Switch の入力バッファ長の最も短かいIHに子プロセスを動的に分配する方式をとると、IH 4台の場合で１個目の解および２個目の解が求まるまでの処理時間をそれぞれ10%,14% 短縮することができる。この時 PPCの稼働率は各々69%, 72%,62%,65% とバランスする。

4.2.2 Concurrent Prologプログラム

integer-generator(付録4-2)とQuicksort プログラム（付録4-3)を走行させ各種データを収集した。

(1) 台数効果（図 4-1)

もともと並列度が低いinteger-generator(プログラムからもわかるように約２）に比べQuicksort では台数増加に伴ない性能向上がみられ、 5〜6 台頃から飽和状態に近づく。この例の場合の並列度は約 4であるから、これらの結果はPIM-RがConcurrent Prolog プログラムの持つ並列性を引き出していることを示している。

(2) ネットワーク

Quicksort 実行時のリダクション等の回数を表 4-2に示す。

表 4-2

リダクション回数	284
成功回数	196
失敗回数	19
サスペンド回数	69

またQuicksort 実行中のネットワーク通過パケットの種類別個数を次表 4-3に示す。

表 4-3

	IM 2台	IM 4台
総パケット数	141	211
true return パケット数	17	17
AND forkパケット数	21	26
fork down パケット数	21	26
MB関連パケット数	82	140

これらより、リダクションが 284回起り、そのうちの196 回が成功する間にIM 2台の時で141 個、IM 4台の時で211 個のパケット転送が起る。つまり、このままでも、リダクションが約 1.3〜2 回起る間に、1 個のパケット転送が起る。表4-3からもわかるように、Prologの場合と違ってMBに関連するパケットがIM 2台の時で58%,IM 4台の時で66% もあるから、Prologプログラムの場合のように、プロセスのdead処理とfork down パケットの生成、転送を止めてもネットワーク転送個数をIM 2台の時で15%,IM 4台の時で12% しか減少させることができない。このMBに関連するパケットは、fork down パケットと異なり、転送の優先度を下げる訳にいかない（下げたら解が求まらない）のでPrologの場合以上にパケット転

送の高速化が重要になってくる。

(3)Message Board Controller(MBC) 導入効果

MBへのチャネルセルの確保は、UUにおいてユニフィケーションが成功し、新しい子プロセスがPPU に帰ってきた際に行なわれる。MBへ格納されるチャネルの個数は38であり、表 4-2より、ユニフィケーション成功回数は 196回であるから、約 2.2回成功するたびに、一つのチャネルのためのセルをMBに確保しなければならない。よって、MB内にチャネルセルを確保する速度は、PIM-R の処理速度に影響を及ぼす。そこでMBC が導入され、MBに関する処理を担当させている。もし、MBC がなく、MBに関連する処理をPPC が担ったとすると、IM 1台の場合で14%, IM 4台の場合で11% ほど処理時間が増加する。

5．おわりに

reduction 概念に基づく並列推論マシンPIM-R のアーキテクチャ、その上でのPrologとConcurrent Prolog の並列実行方式、ソフトウェア・シミュレーションについて述べた。

PIM-R は、structure-copy方式を採用したために増加するcopy量とそれに関する処理量の低減およびNetwork を通過するpacket数の低減のため、only-reducible-goal copy方式、reverse compaction方式を採用し、プロセス内にProcess Life Blockを導入している。

アーキテクチャとしては、PIM-R の中で統一的にPrologとConcurrent Prolog を処理するために、並行プロセス間チャネル用分散化共有メモリ(Message Board) を導入した。

これにより、Back Communication、有限長バッファ通信をはじめとするConcurrent Prologの機能が実行できることを確認した。　また、効率的パケット分配のための Intelligent Network Nodeの導入、大きい構造体データ中のGround Instance 格納のための構造体メモリの導入をもアーキテクチャ上の特徴としている。

そして、これらの特徴をもつPIM-R のソフトウェア・シミュレータによるシミュレーションの結果、PIM-R は、PrologおよびConcurrent Prolog プログラムに内在する並列性を引き出せること、即ち、台数効果があること、Intelligent Network Nodeによる子プロセスの動的分配効果、Message Board Controllerの導入効果、Process Pool Controller におけるdeadプロセス処理休止とfork down packet生成、転送処理休止の効果、Concurrent Prolog実行時のpacket転送の高速化の必要性を確認した。

現在、我々はoccam で記述された詳細ソフトウェア・シミュレータおよびマイクロコンピュータ８台からなるシミュレーション専用装置を開発中である。今後は、これらソフトウェア・シミュレータとシミュレーション専用装置により、構造体データ共有化の効果を始めとする各種詳細なシミュレーションを行ない、PIM-R の有効性の確認と改良を行なう予定である。

最後に、御討論いただいた（株）日立製作所中央研究所　杉江、米山、坂部、岩崎各氏と、日頃御指導いただく村上第一研究室長に深謝する。

〈参考文献〉

[Clar 84] Clark, K.L. and Gregory, S., "PARLOG:Parallel Programming in Logic", Research Report DOC 84/4, Dept. of Computing, Imperial College London, 1984.

[Hira 83] 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”，信学技報EC83-38, 1983.12.

[Ito 83] 伊藤，尾内，益田，清水，“データフロー方式の並列PROLOGマシン”，Logic Programming Conference '83, Tokyo, 1983.3.

[Ito 84] Ito, N. and Masuda, K., "Parallel Inference Machine Based on the Data Flow Model", Proc. of the International Workshop on High Level Computer Architecture 84,pp.4.31-4.40, 1984.5.

[Moto 84] Moto-oka,T.,Tanaka,H. et al.,"The Architecture of a Parallel Inference Engine -PIE- ", Proc. of Int.Conf. on Fifth Generation Computer Systems 1984, ICOT, pp.479-488, 1984.

[Onai 83] 尾内，麻生，“並列推論マシンにおけるGuard と人力annotationの制御機構”，情報処理第27回全国大会 4p-5,1983.10.

[Onai 84a]尾内，麻生，“並列環境におけるConcurrent Prolog の実現法”，情報処理第29回全国大会 7B-5,1984.9.

[Onai 84b]尾内，清水，益田，麻生，“逐次型Prologプログラムの解析” ,Logic Programming Conference '84, Tokyo, 1984.3.

[Pere 84] Pereira, L.M. and Nasr, R., " DELTA-PROLOG: A Distributed Logic Programming Language", Proc. of Int. Conf. on Fifth Generation Computer Systems 1984, ICOT, pp.283-291, 1984.

[Shap 83] Shapiro, E.Y., "A subset of Concurrent Prolog and Its Interpreter", ICOT Technical Report TR-003, 1983.

[Take 83] 竹内彰一，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”，情報処理ソフトウェア基礎論 4-4, 1983.

[Turn 79] Turner, D.A."A New Implementation Technique for Applicatve Languages", Software-Practice and Experience , No.1, Vol.9, 1979.

[付録1] データタイプ一覧

Type Classification			Abbre-viation	Type Code			Hex Code of word	
Type	SubType1	SubType2		0,1,2 Type	3,4,5 SubType1	6,7 SubType2	Type Tag	Data
Variable	Void-variable		Void	0 0 0	0 0 0	0 0	00	000000
	Variable-1'st		Var1		0 0 1	0 0	04	000000
	Variable-ref		VarR		0 1 0	0 0	08	Pointer #0
Channel	Undef Channel	1st	UCh1	0 0 0	1 0 0	0 0	10	Address #0
		ref.	UChR			0 1	11	Pointer #0
	Read Channel	1st	RCh1		1 0 1	0 0	14	Address #0
		ref.	RChR			0 1	15	Pointer #0
	Write Channel	1st	WCh1		1 1 0	0 0	18	Address #0
		ref.	WChR			0 1	19	Pointer #0
Atomic	User Defined Atom		Atom	0 0 1	0 0 0	0 0	20	Identifier
	Nil		Nil		0 0 1	0 0	24	000000
	System Symbol	Type0	Sym0		0 1 0	0 0	28	Identifier
		Type1	Sym1			0 1	29	Identifier
		Type2	Sym2			1 0	2A	Identifier
	Integer		Int		1 0 0	0 0	30	Integer
	Real		Real		1 0 1	0 0	34	Real
Structure	List		List	0 1 0	0 0 0	0 0	40	Pointer #1
	String		Strg		0 0 1	Type	4*	Pointer #2
	Vector		Vect		0 1 0	0 0	48	Pointer #3
	Channel Inforomation		ChI		0 1 1	0 0	4C	Pointer #4
	And	Parallel	Para		1 0 0	0 0	50	Pointer #5
		Seq	Seq			0 1	51	Pointer #5
	Literal		Lit		1 0 1	0 0	54	Pointer #6
	Pointer		Poi		1 1 0	0 0	58	Address #1
Structured Pointer	Variable	void	SPVo	0 1 1	0 0 0	0 0	60	Address #2
		Var 1st	SPV1			0 1	61	Address #2
	Atomic		SPAt		0 1 0	0 0	68	Address #2
	Non Ground	List	SPNL		1 0 0	0 0	70	Address #2
		String	SPNS			0 1	71	Address #2
		Vector	SPNV			1 0	72	Address #2
	Ground	List	SPGL		1 1 0	0 0	78	Address #2
		String	SPGS			0 1	79	Address #2
		Vector	SPGV			1 0	7A	Address #2

Ⅰ Variable

リテラルに現れる（リテラルエリア、ストラクチャエリアに格納される）変数は、タイプVarR（Variable Referenceの略）のセルによって変数エリア内のセル（Void、Var1）を指す。後に unification等でその変数の値が定まった時、変数エリア内のセル(タイプVar1）にその値を書き込む。

Ⅱ Channel

リテラルに現れるChannel はタイプUChR, RChR, WChRのセルによって変数エリアに格納されているタイプChI のセルを指す。ここで、ChI はchannel information(後述するⅣ 5)) へのpointer である。RchRはRead Channel Referenceの、UChRはChannel Reference の略で、

p1(X), p2(X)

のような場合、X はchannel だが、どちらがproducorかconsumerか静的には分らない。そこで、undefineの意味でU をつけた。WChR(Write Channel Reference) は，現在は使用されていない。次の例において、リテラル pのChannel X はUChRによって、リテラル cのChannel X?はRChRによってChI を指す。

（例） go:-true | p(X), c(X?).

Ⅲ System Symbol

1) Sym0 : >,< 等変数のバインドを必要としないBuilt-in predicate
2) Sym1 : is,=等変数のバインドを必要とするBuilt-in predicate
3) Sym2 : guard, true, fail 等PPU が処理を行うBuilt-in predicate

Ⅳ Pointer

* Pointer の先頭の1bitはUnifier において 1の時goal側を、 0の時clause側をさす為に使用される。以下においてL はリテラルヘッダー先頭番地、S はストラクチャエリア先頭番地とする。

1) Pointer #0 : 変数エリアを指すPointer で、Pointer の先のTypeはVariable, Atomic, List, Vect, ChI 、および、Structured Pointerのいずれかである。

Pointer #0 :

Type	Data

2) Pointer #1 : Structure エリアへのPointer でPointer の先はListが格納されている。

S+Pointer #1 :

CAR Element
CDR Element

+1 :

3) Pointer #2 : Structure エリアへのPointer でPointer の先はStringが格納されている（Stringの詳細は未定）

4) Pointer #3 : Structure エリアへのPointer でPointer の先はVectorが格納されている。

S+Pointer #3 :

Int	N
1st Element	
N-th Element	

+1 :
: :
+N :

5) Pointer #4 : Structure エリアへのPointer でPointer の先はChannel information が格納されている。

S+Pointer #4 :

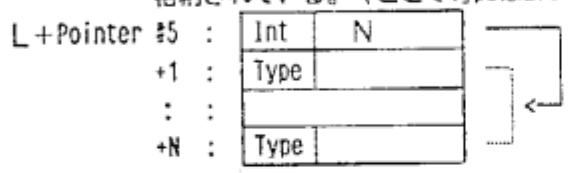
Type	Address #0
Local Data	

channel information

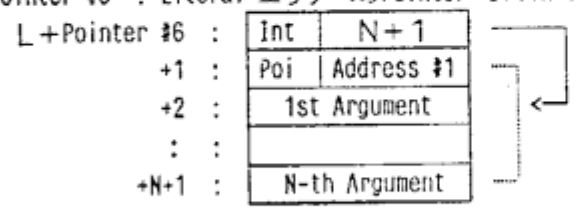
+1 :

（ここでTypeはUCh1, RCh1, WCh1のいずれか）

6) Pointer #5 : Literal エリアへのPointer でPointer の先は並列／逐次AND 関係にあるLiteral に関する情報が格納されている。（ここでTypeはLit ,Seq ,Para ,Sym ,Poiのいずれか）



7) Pointer #6 : Literal エリアへのPointer でPointer の先はLiteral が格納されている。



V Address

1) Address #0 : Message Board 又は、Process PoolのAddress

(2bit)	IH# (4bit)	Identifier(18bit)

00 : unfixed
10 : MB
11 : PP

2) Address #1 : Clause Pool の Address

00(2bit)	IH# (4bit)	Identifier(18bit)

3) Address #2 : Structure MemoryのAddress

Identifier(24bit)

[付録２] Process Pool内Block 内部形式

1) PCB(Process Control Block)

Int		注1)
Int		注2)
Poi	Previous Pointer	注3)
Poi	Next Pointer	注3)
Poi	PTB へのpointer	
Poi	PLB へのpointer	

注1)

0	8	9	10	11	12	13	14	15	16	31
Int									Process Status	

bit 8 ～10 : PCB(Block のタイプ 5)参照)

bit 11 : fail bit(並列and 関係にある子goalがfail時に1)

bit 12 : body実行中flag (兄弟プロセスとのcommit競争に勝った時に 1にset 。即ち、このPCB の下のPTB はbodyのみ)

bit 13 : reduction (1) / retry (0)

bit 14 : query (1) / その他(0)

Process Statusは、ready,run,dead

注2)

0	8	16	24	31	
Int	Reduction Level	Or Fork数	Return数		… ①
Int	Reduction Level	And Fork数	Return数		… ②

(①はOR Parallel Prologの時、②はConcurrent Prolog の時)

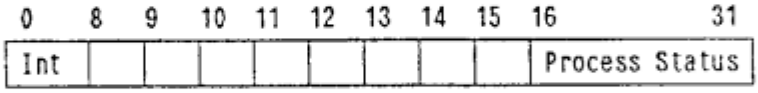
Fork数とReturn数が等しくなった時Statusはdead

注3) これらのポインタはReady 状態にある(reducibleなgoalを持つ) Process Control Block の管理を行うReady Process Queue(RPQ)に連結するために用いる。

2) PLB(Process Life Block)

Int		注1)
Int		注2)
Poi	親PCB へのPointer	

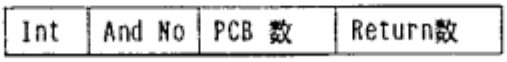
注1)



bit 8 ～10： PLB(Block のタイプ 5)参照)

bit 12 ：commit tag (一番最初にguard 実行に成功したPCB が、ここを1(ON) にする。もしすでにONになっていれば、注2)のPCB 数を1だけ減少させ、PCB のStatusはdeadとなる。)

注2)

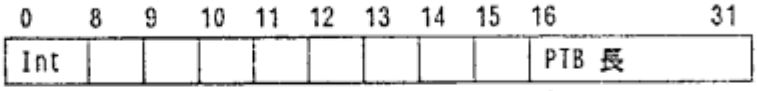


(And NoはConcurrent Prolog の時において複数goalが並列AND オペレータで結合された時、本goalがその何番目かを示す)

3) PTB(Process Template Block)

Int		注1)
		注2)

注1)



bit 8 ～10： Block のタイプ (5)参照)

注2) PCB につながっている時は、clause長を除くclauseの形式に同じで、SPCBにつながっている時は、

TYPE	suspend したCPへのpointer
goal長を除くgoalの形式に同じ	

ここでTYPEは、suspend をまねいたChannel 変数とunify しようとした引数のタイプ (activeされる時このchannel 変数が異なるタイプの値と結合されれば、UUに送るまでもなく、このgoalはfailとなる。)

4)　SPCB(Suspend Process Control Block)

Unification Unitに送ったが、suspend となって返ってきたgoalにたいしては、PCB ではなくこのSPCBが生成され、この下のPTB にはsuspend したgoalが格納される。

Int		注1)
ChI		注2)
Poi		注3)
Poi		注3)
Poi	PTB へのpointer	
Poi	PLB へのpointer	

注1)

0	8	9	10	11	12	13	14	15	16
Int									Process Status

bit 8 ～10：　SPCB(Block のタイプ　5)参照　)

Process Statusはsuspend

注2) suspendの原因となったchannel 変数のPTB 内address

注3)SPCBの際は使用されないが、SPCBがactivateされて、PCB となった時、これらはPCB の第三、第四語となる。

5)　Block のタイプ

	タイプ	8 9 10 bit
OR Parallel Prolog	PTB	0 0 0
	PLB	0 0 1
	PCB	0 1 0
Concurrent Prolog	PTB	1 0 0
	PLB	1 0 1
	PCB	1 1 0
	SPCB	1 1 1

［付録３］Concurrent Prolog 実行過程

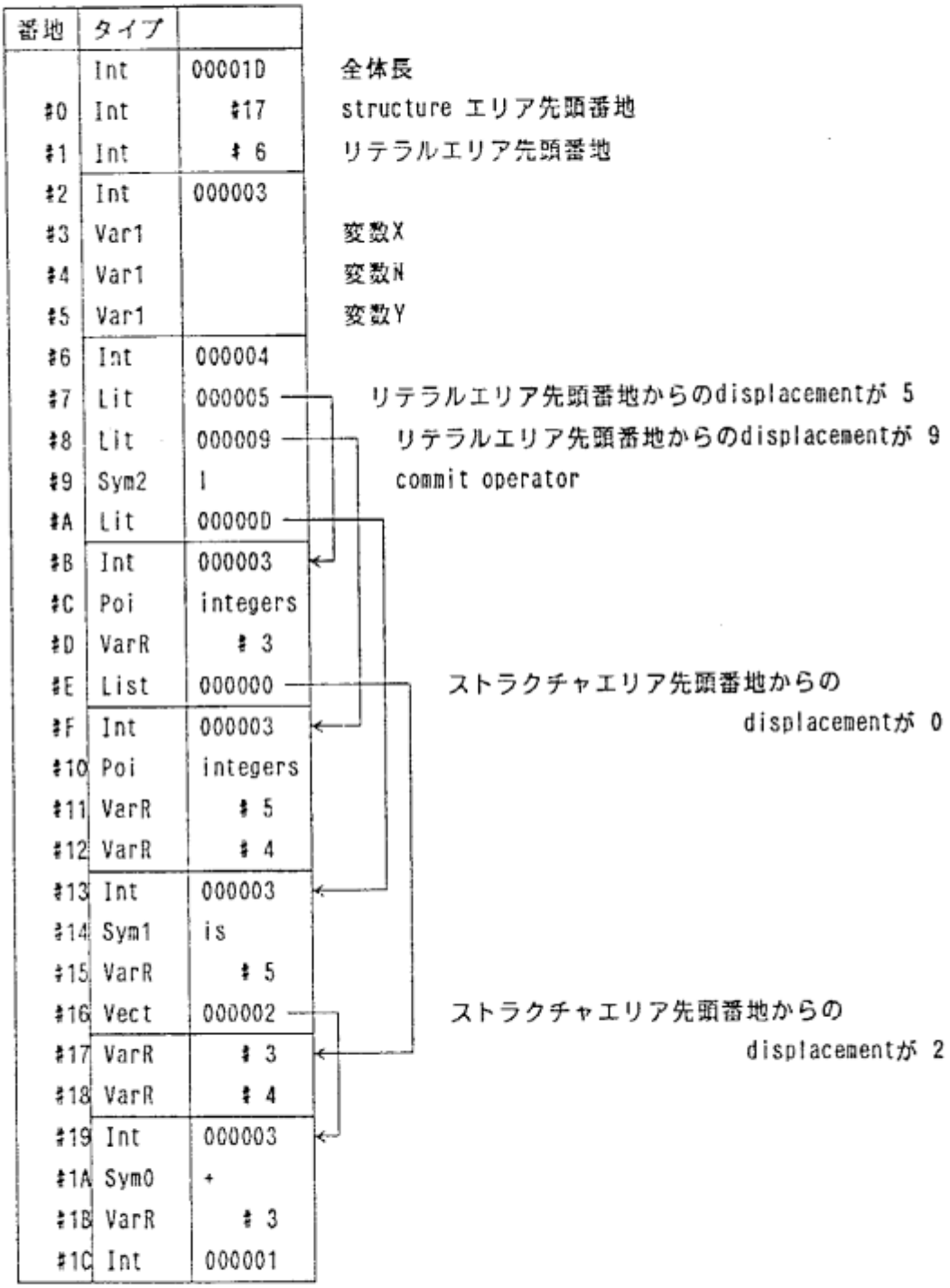
次のプログラムは、 0からはじまる整数を出力する。プロセスintegersは、 0から始まる整数を生成し、プロセスoutstream はそれを受けとり出力する。

```
goal     ?-integers(0,N) ,  outstream(N?).

clause-1  integers( X,[X | N]) :-  Y is X+1   |  integers(Y,N).
clause-2  outstream(  [X | N]) :-  write(X)   |  outstream(N?).
```

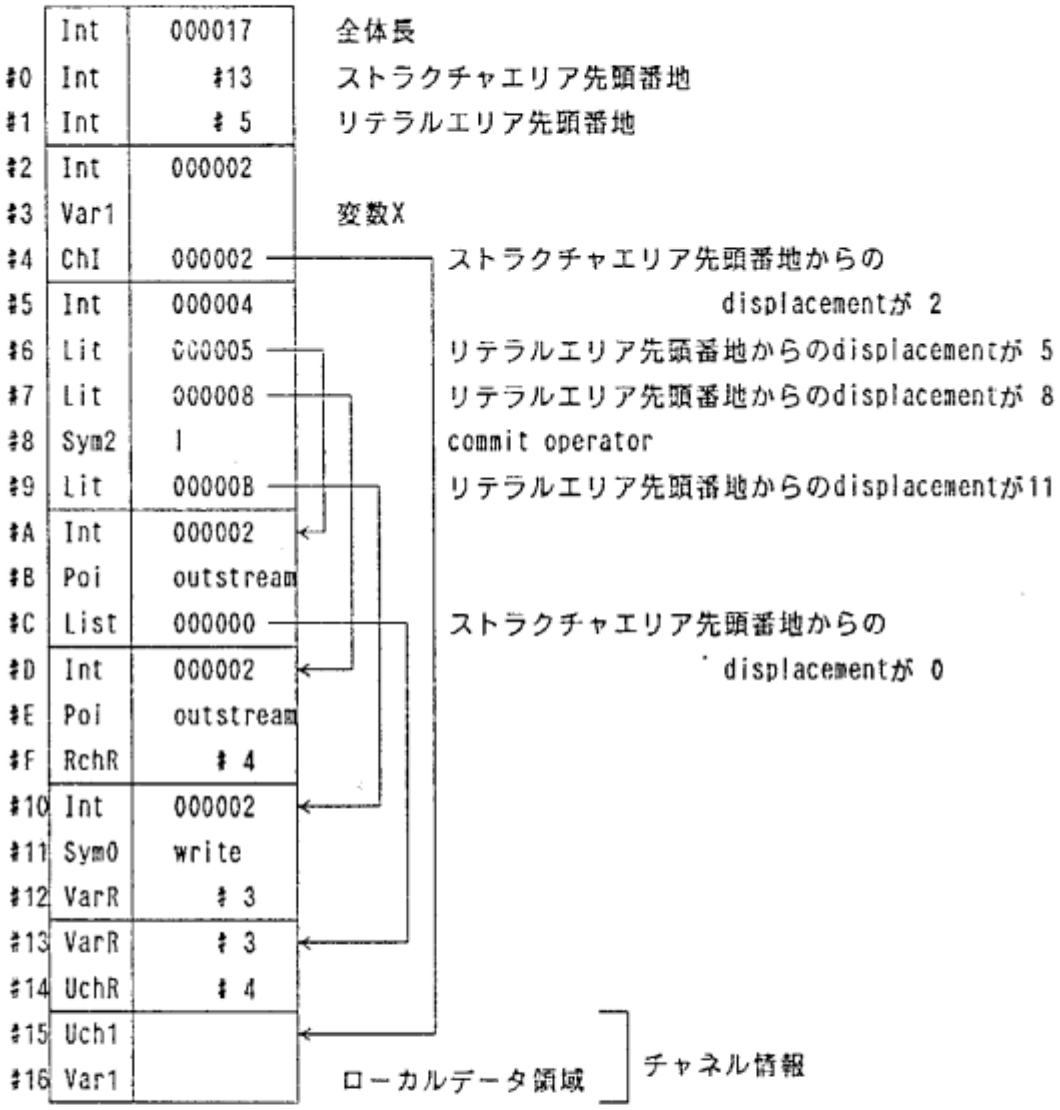
clause-1の各変数は、この時点ではチャネルではない。

このclause-1のClause Pool における内部形式を次に示す。

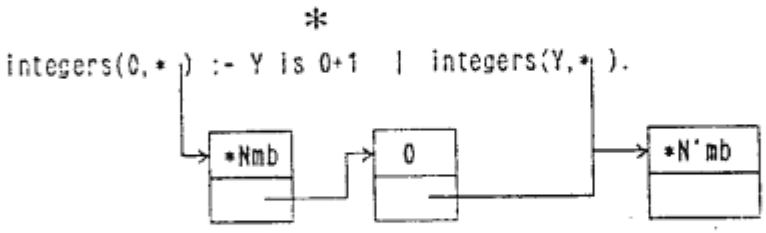


番地	タイプ		
	Int	00001D	全体長
#0	Int	#17	structure エリア先頭番地
#1	Int	# 6	リテラルエリア先頭番地
#2	Int	000003	
#3	Var1		変数X
#4	Var1		変数N
#5	Var1		変数Y
#6	Int	000004	
#7	Lit	000005	リテラルエリア先頭番地からのdisplacementが 5
#8	Lit	000009	リテラルエリア先頭番地からのdisplacementが 9
#9	Sym2	!	commit operator
#A	Lit	00000D	
#B	Int	000003	
#C	Poi	integers	
#D	VarR	# 3	
#E	List	000000	ストラクチャエリア先頭番地からの displacementが 0
#F	Int	000003	
#10	Poi	integers	
#11	VarR	# 5	
#12	VarR	# 4	
#13	Int	000003	
#14	Sym1	is	
#15	VarR	# 5	
#16	Vect	000002	ストラクチャエリア先頭番地からの displacementが 2
#17	VarR	# 3	
#18	VarR	# 4	
#19	Int	000003	
#1A	Sym0	+	
#1B	VarR	# 3	
#1C	Int	000001	

clause-2のClause Pool における内部形式を次に示す。

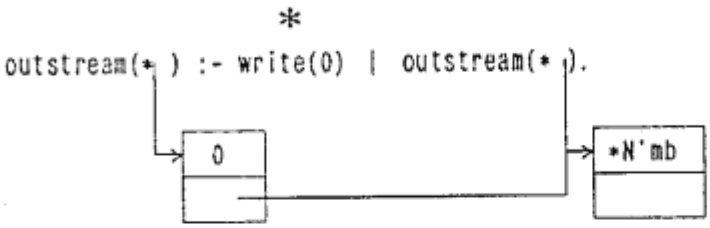


プロセスintegersはreduction の結果 ,次図－付１のようになる。なお、プロセスの内部形式は省略形で示す。＊はreducible state 、G はgrabage state 、sus は、suspend state である。



図－付１

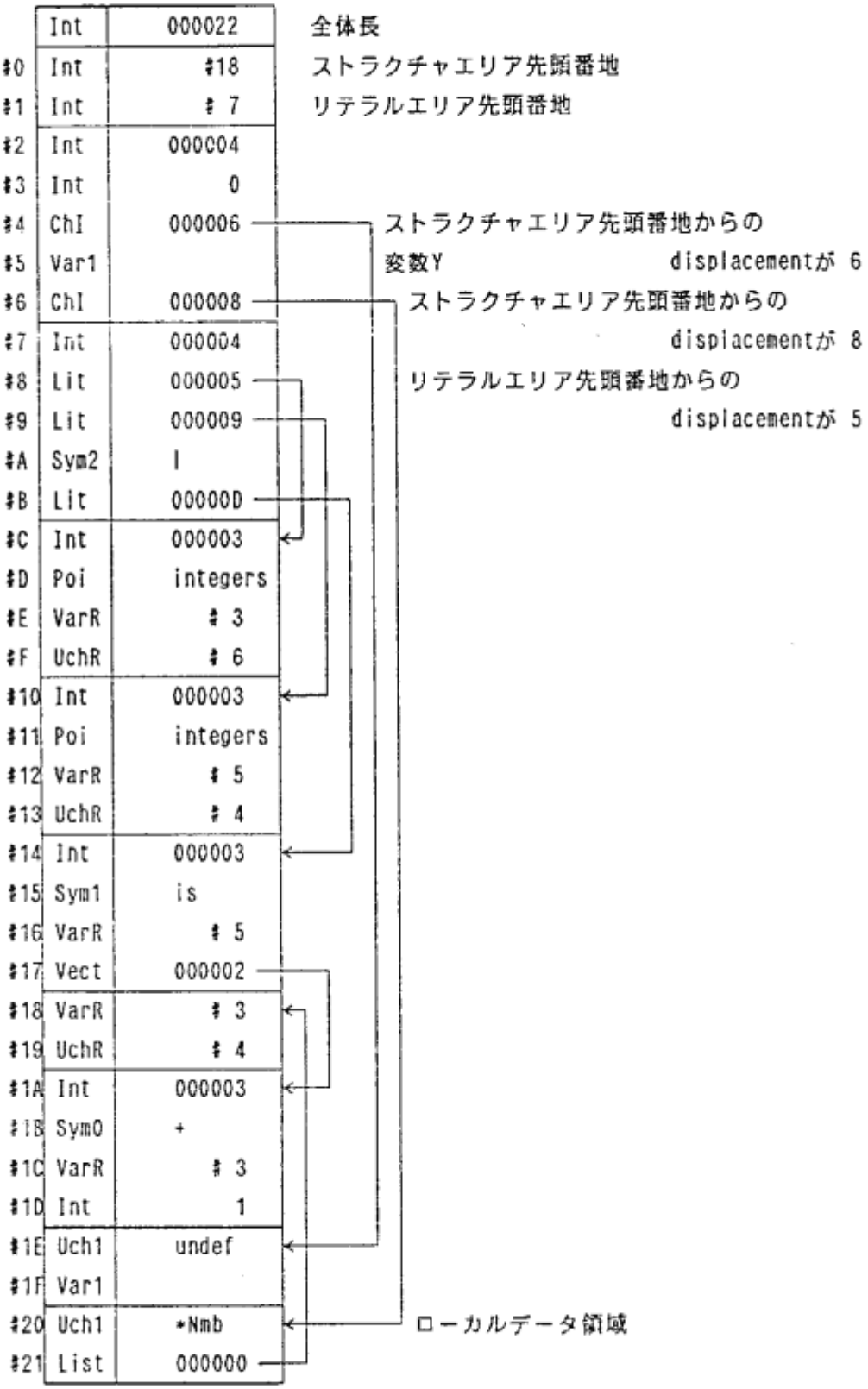
次のチャンネルのためのHB内チャネルセルはHB Controller により確保される。 N'mbはそのセルのアドレスである。プロセスoutstream は、Message Board(略してH.B.) にN?の値を見にいくが、まだ値が到着していないので、S.P.Listに繋ぎ込まれsuspend 状態になる。Y is 0+1 がまだ実行されていないので、N のローカルデータ[0 | N'] はHBへ書き込まれない。次にY is 0+1が実行され Y is 1となると、Guard 部は成功するのでC tag をONにし、N のローカルデータ[0 | N'mb] をHBのアドレスNmb のチャネルセルに（正確には値セル）に書き込む。そしてS.P.Listに繋がっているconsumerプロセスであるoutstreamにN が [0 | N'mb] であることを通知する。通知を受けたoutstream は、reducible となる。これのreduction の結果を次図－付２に示す。



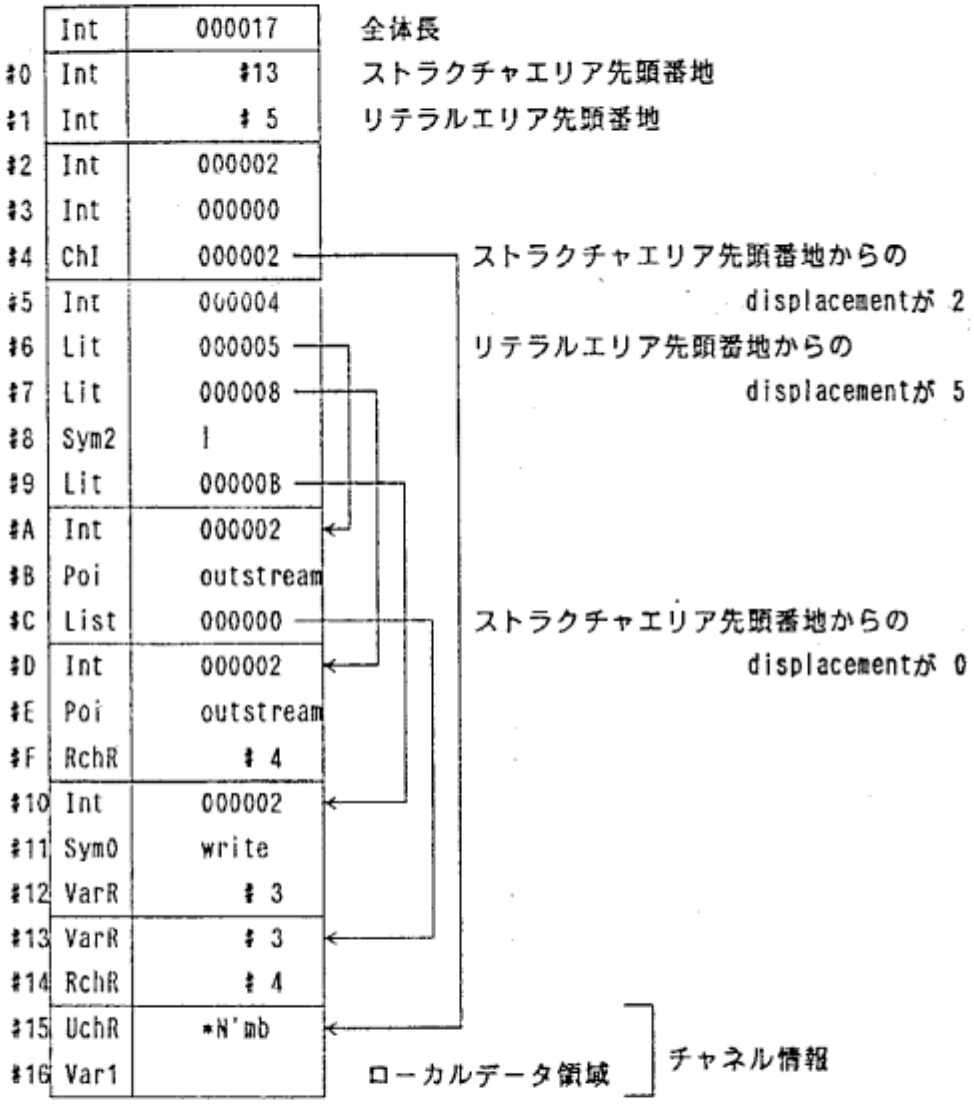
図－付２

次にwrite(0)が実行され、 0が出力されると、Guard bar に達し、今度は outstream(**N'mb) がreducible となる。よって、この例では次にintegers(1,**N'mb)と outstream(**N'mb) がreducible となる。

次に省略形でない図－付１の内部形式を示す。



次に、省略形でない図－付２を示す（一部省略を含む。たとえば、outstream は実際は Clause Pool 内の節outstream 格納位置へのポインタである）。



［付録４］シミュレーションを行なったプログラム

(付録4-1) 4Queens プログラム

```
go:-queens([1,2,3,4],[],X ).
queens([],Y,Y).
queens(X,Y,Z) :-
        select(U,X,V),safe(U,Y,1),queens(V,[U | Y],Z).
select(X,[X | Y],Y).
select(X1,[X | Y],[X | Z]):-select(X1,Y,Z).
safe(U,[],_) .
safe(U,[P | Q],N) :-
         nodiag(U,P,N),M is N+1, safe(U,Q,M).
nodiag(U,P,N):-
         T1 is P+N,T2 is P-N,T1 =\= U,T2 =\= U.
```

(付録4-2) integer-generator プログラム

```
go:-true | integers(0,X),outstream(X?).
integers(X, [X | N]):-Y is X+1 | integers(Y,N).
outstream([X | N]):-display(X) | outstream(N?).
```

(付録4-3) Quicksort プログラム

```
go:-true |
   quicksort([5,9,2,7,3,6,10,4,1,8],X),screen (X?).
screen([]):-display([]) | true.
screen([X | Y]):-display(X) | screen(Y?).
quicksort(X,Y):-true | qsort(X,Y).
qsort([X | Y],R):-true |
  partition (Y?,X,S,L),qsort(S? ,S1) ,
  qsort(L?,L1),lappend(S1?,[X | L1],R).
   (partition ,lappendは省略)
```