ICOT Technical Report: TR-122

TR-122

リダクション方式並列推論マシン PIM-R の アーキテクチャ

尾内理紀夫，清水　肇，麻生盛敏
益田嘉直，松本　明

June, 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

リダクション方式並列推論マシンPIM-R のアーキテクチャ

尾内理紀夫　清水肇　麻生盛敏　益田嘉直　松本明
（財）新世代コンピュータ技術開発機構

ABSTRACT　This paper proposes a Reduction-based Parallel Inference Machine:PIM-R and describes the parallel execution mechanisms on PIM-R and software simulation. PIM-R uses the only reducible goal copy method, and the reverse compaction method to decrease the amount of copying and the number of packets passing through the network. PIM-R architecture features include the distributed shared memory for Concurrent Prolog, network nodes for efficient packet distribution, and the structure memory for reducing the copying overhead.

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],[Kumo 85]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。リダクション方式並列推論マシン(Reduction-Based Parallel Inference Machine:PIM-R)[Onai 85a],[Onai 85c] では、ICOTが核言語第１版(KL1(84)) 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 概念に基づくresolvent 生成過程の並列実行を基本動作とし，PrologをOR並列に、 Concurrent PrologをAND並列に処理する方式を，PIM-R では採用した。

PIM-R では、プロセス内にgoalが複数あった時（それら複数goal全体を親プロセスと呼ぶ）、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数低減のため，独特のプロセス構成法,only-reducible-goal copy,逆順コンパクションを採用している。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) によって高い並列度が期待できる[Onai 84b]。 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。

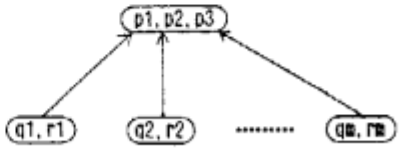
それでは、PIM-R におけるPrologのOR並列、AND 逐次実行について述べる。

次のような、goal列とclause群があったとする。

```
goal列　　p1,p2,p3
clause群　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(IM、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 bar あるいはcommit operator と呼ばれる。

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。

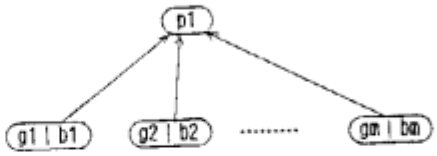
また、並列AND で連結されたgoalが変数を共有する時、それらgoalは、それぞれ、　producerとconsumerの関係にあり、その変数はgoalをプロセスと考えると、プロセス間通信のために使用される。　つまり、並列AND というよりは、プロセス間の通信を論理変数を用いて実現しているといったほうがよいであろう。PIM-R ではこの通信のために使用される論理変数(これをチャネルと呼ぶ)のための分散化された共有メモリ(Hessage Board 後述)を設けている。

たとえばgoal列　p1,p2　(","は並列AND operator)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)、並列に実行される(consumerは結果的には変数に値が結合されるまで待たされる)。なお、共有変数に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、この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、各guard partのunification がパイプライン実行される。　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　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(IM)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(IM)間のNetwork traffic 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Network traffic は膨大なものとなり、異なる処理要素(IM)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(IM)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(IM)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IM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。

3. PIM-R アーキテクチャ

PIM-R はFig.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。

3.1 Inference Module (IM Fig.2)

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

3.1.1 Unification Unit (UU)

(1) Clause Pool

各Clause Pool は同一clause群を格納している(1語32bit)。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り、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。)

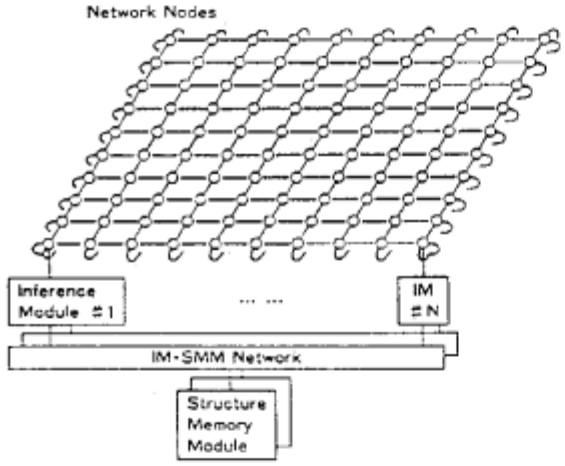


Fig.1 Conceptual configuration of PIM-R

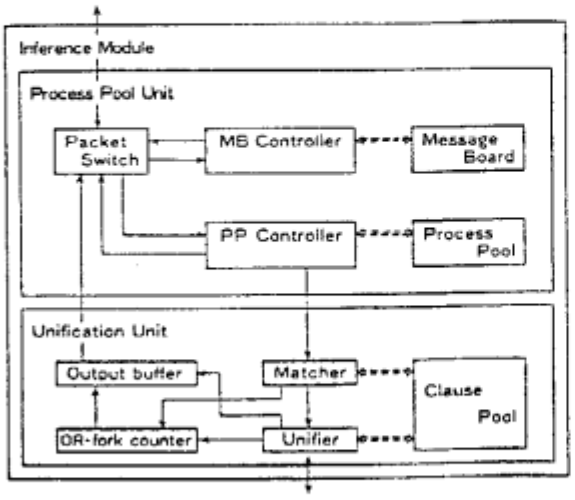


Fig.2 Inference Module configuration

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、Fig.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

#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 注1)
	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) Type : Pol, Lit, Sym, Para のいずれか

Fig.3 Configuration of the Clause Definition Block

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、整数値（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。

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。

(例1) a:-b|c,d,e

	ヘッダー	
	変数エリア	
0	Int	4
1	Poi	a
2	Para	5
3	Sym2	|
4	Poi	b
5	Int	3
6	Poi	e
7	Poi	d
8	Poi	c

並列AND 記述子Paraは（例 1）にみられるように、並列AND 関係にある（"," で結ばれた）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（逆順コンパクションのため）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。

(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 した後に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 内にor fork 数 （return 数は0)として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との間での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を+1する。（return数=OR fork数となった時は、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、結果を自IH内のProcess Poolにある親プロセスに返す。一方、unit clause 以外のclauseとのunificationに成功し，新たなresolvent が生成された時はそのresolventを分配ストラテジーに従って、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が自IH内のProcess Poolに返され、後述するSPCB（Suspend Process Control Block ）が作成される。

3.1.2 Process Pool Unit (PPU)

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IH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。

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(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)

i)プロセス生成処理

新たなresolvent がUnification Unitから戻されてきた時, PLB とPCB-PTB 対からなる新たなプロセスが生成される。

ii) プロセス更新処理

プロセスの更新とは、fork countおよびreturn countの更新と, 新たなPCB とPTB の対の生成である。 Prolog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。Concurrent 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IMあるいは他IMに分配する。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[Shim 85] が行なわれる。次に例を用いてこれを説明する。

(例 2) 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Fig.4 に示す。ただし、'&' は逐次AND operatorである。

このようなclause definition 部は例えば, goal p(1,[X|Y]) との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Fig.5 の左側に示す。

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	リテラル数	
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	X	#21	#0	CAR Element	ストラクチャエリア
VarR	4	Y	#22	#1	CDR Element	

Fig.4 Clause Definition Block of p(1,[X|Y]):-q(1,X) & r(X,Y)

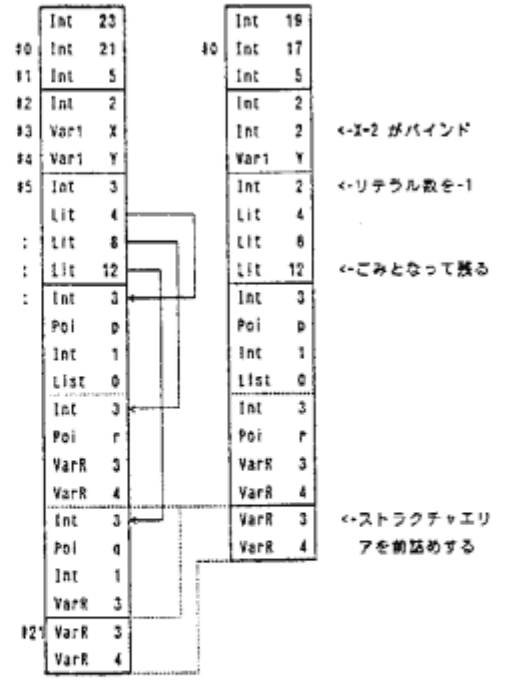


Fig.5 Reverse compaction processing

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る。そこで，このq(1,X)の部分のみがcopyされ，Unification Unitへ送られる。

今Unification Unitでunit clause とunify し、q(1,2)がProcess Poolへ戻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。すなわち、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Fig.5 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，解のreturn時には，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。

iii)プロセス消滅処理

PCB において，（fork数−return数）になったならば、PPC はdeadプロセス処理時に，そのPCB をdeadにし，その上のPLB 内のPCB return数を＋1増加させる。

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、IM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 traffic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）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 関係にあるこつのプロセスが生成されたところをFig.6 に示す。

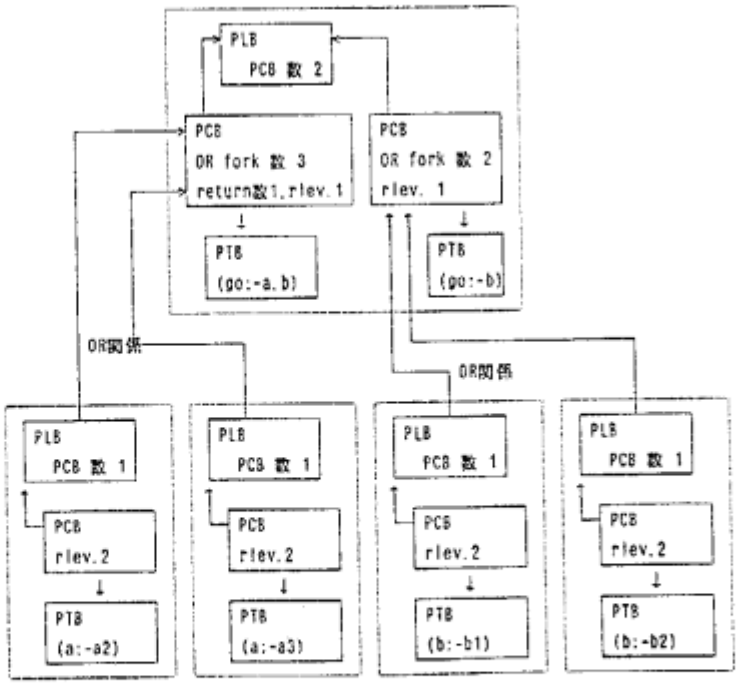


Fig.6 Relationship between processes

(3) 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難しい。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、Fig.7 のようにIM内Ready Process Queue(RPQ)に連結される。

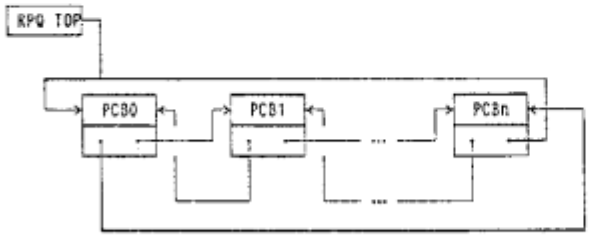


Fig.7 Ready Process Queue

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 レベルを使用する。Reduction レベルのより大きい (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 レベルの, 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 レベルにより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法が異なり、またProcess Poolとは独立にアクセス可能である。 なお, Process Pool Controllerの負担を軽くするために、Message Board Controller を導入し, MB関連の各種処理を行なう[Onai 83], [Onai 84a]。

(1)Message Board(MB)

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。

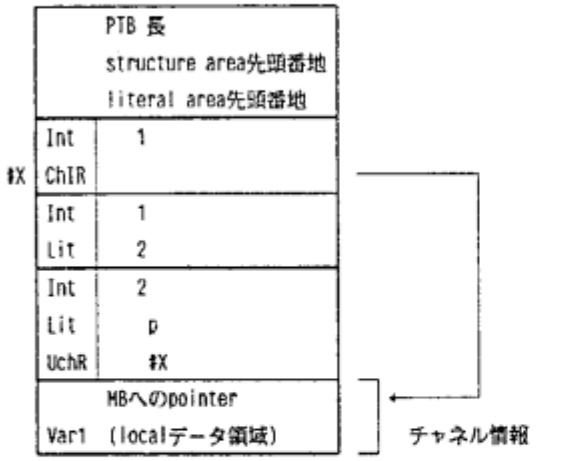
write tag	suspend tag
値cellへのpointer	
suspend プロセス個数	
Suspend Process List先頭番地	

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IM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 プロセスへのpointer を格納する。suspend プロセス自体はPPU 内のProcess Pool に格納される。

(2)Message Board Controller (MBC)

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channel cell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cell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IM内Process Poolに置かれたp(X)のPTB は次のようである。



(ChIR 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)

チャネル情報の第一語はMB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MB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[Onai 84a]。

3.2 Structure Memory Module (SMM)

関数型言語等をサポートするデータフローマシンにおいては、リストやベクタ等の構造体データの効率的な処理方式について長年各所で研究が進められて来ているが、論理型言語をサポートする並列推論マシンにおいては最近ようやく検討が開始されたばかりである[Hira 83],[Ito 83]。ここでは構造体データの効率的な処理を実現するものとして導入する構造体メモリ(Structure Memory) について述べる[Hasu 85] 。

Fig.1 に示すように、Structure Memory Module(以下SMM と呼ぶ) はIM-SMM Networkを介して多数台の Inference Module(IM) と接続されるので、ある一定の条件を満たすSMM 内に格納される構造体データは多数台のIMより共有される。このためSMM 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パ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、
①SMM をBank分け等により複数のSMM に分散化することにより、SMM 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SMM を１台接続したものを単位とするモジュールを構成し、それら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SMM を１台ずつ接続して行き全体システムを構成する。

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を回避する。この場合SMM の内容は同一であるため、各IM内のunification 時にSMM 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-SMM Networkを介して構造体データは読出される。また、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SMM 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。即ち、プログラムのcompile 時にclause中の構造体データのうち、SMM 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SMM 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、共有度は低いが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。

(1) 構造体データの共有化の方式

clause中の構造体データのうち、指定されたground instance の部分がSMM 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SMM 内の構造体データがIM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。

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。

この場合、clause中の構造体データのうちlengthや構造等の条

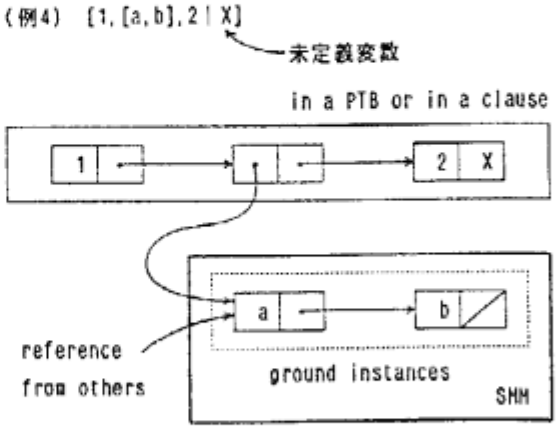


Fig.8 Example of sharing ground instances

件指定を行うことによりcompile 時にSMM へ格納すべきground instanceの切り分けを行い、それをSMM 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切り分けを行う。

(2) 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。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。

これらSMM 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(VAT) 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、多重なる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。

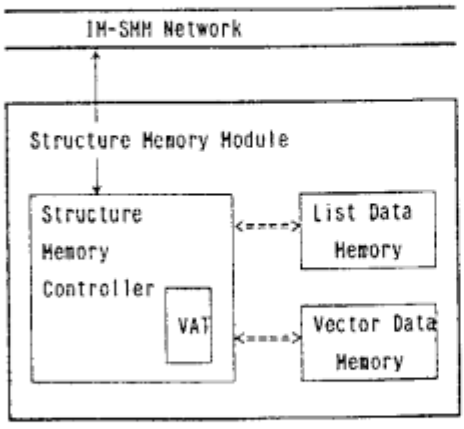


Fig.9 Structure Memory Module configuration

更に、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、SMM 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IM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。

(3) 引数間のLazy Unification

SMM 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、SMM 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、SMM 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SMM 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。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。

現段階では、静的に決まるclause中のground instance だけをcompile 時にSMM にinitial loadしSMM への動的な書込みは行わない。従って、この場合SMM 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

前述したように，PIM-R によるPrologあるいはConcurrent Prologの並列実行過程はPLS-PCBs-PTBs を一つのプロセスとするプロセス木の生成消滅の過程であり，プロセスがIMへの割り付け単位である。 そして，PrologあるいはConcurrent Prolog のPIM-R 上での実行に際しては，ruleとのunification 成功時に子プロセスが生成される。(fact(unit clause) とのunification に成功しても，結果は元のプロセスのPCB に返されるだけで，子プロセスは生成されない。) 一方，[Onai 84b]より，ruleを主体とするPrologプログラムの平均OR relation 数は，2～3である。 よって，平均的なプロセス木はcyclicなmesh構造の中にたたみ込むことが可能である。PrologあるいはConcurrent Prologなどの論理型プログラムでは，解が複数ある場合でもAND-OR木における解のある位置 (深さ) は普通異なるので，同時にforkした

子プロセスから解が同時に戻ってくることはほとんどない。よって，プロセス木の上方にあたるmesh型Network 内Nodeに子プロセスからの解のreturnが集中し，このNodeが過負荷になることはない。 また，PIM-R での実行中におけるNode間のアクセス距離に関しても，Concurrent Prolog プログラム実行時のMessage Board へのアクセスを除けば，後は，すべて隣同志のNode間のアクセスであり，mesh型Network で距離１である。 そこで，PIM-R ではIM間ネットワークとして，Network Nodeをmesh上に配置し，その下にIMを結合する構成を採用している。(Fig.1)

またIM間Network Node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への分配を動的に制御できる機能を持ち，Node間のチャネルは双方向である。

IM-SHM間Network(IM-SH 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。

このようなNetwork Nodeは、例えば、Transputer (各10Mbits/sec の４本のチャネルとサイクルタイム50nsecの 4kbytes の static RAMを持つ) のような通信機能と、高速のメモリを内蔵したマイクロコンピュータを使用しても構成できる。

4．ソフトウェアシミュレーション

4.1 基本動作確認ソフトウェアシミュレーション

台数効果、アーキテクチャの基本的検証等，PIM-R の基本動作確認のためにDEC-10 Prolog/C-Prologで記述された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[Onai 85b]。 なお、本シミュレータの実行はDEC 2060/VAX-11 上で行なわれる。

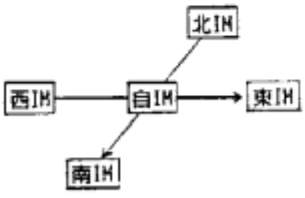
4.1.1 シミュレーション条件

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。UUにおけるunification と結果生成は 100μsec とした。4Queens におけるネットワーク通過パケット平均長は約20語(約600bits)なので，ネットワーク速度を10Mbpsとし平均ネットワーク遅延を60μsec とした。

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



4.1.2 シミュレーション結果

4.1.2.1 Prolog プログラム

4Queens プログラム (付録2-1)を走行させデータを収集した。

(1) 台数効果 (Fig.10)

4Queens では、IM台数増加に伴ない処理性能向上がみられ、7~8 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) 子プロセス消滅処理休止効果

ネットワーク通過パケットの種類別個数を表 1に示す。

表 1

	IM 2台	IM 4台
総パケット数	151	207
true return パケット数	31	47
OR fork パケット数	60	80
fork down パケット	60	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、IM 2台 , IM 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、IM 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 8%, 15% 短縮される。

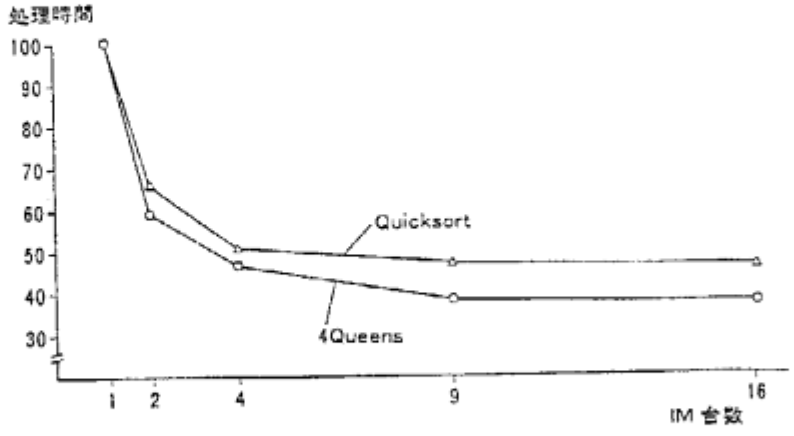


Fig.10 Effect of number of Inference Modules

(3) Reduction レベルを用いた実行戦略

より深いReduction レベルを持つReady PCB と子プロセスからのtrue return パケット処理とに高い優先度を与えることにより、局所的な疑似depth-first 実行が可能である（この時，同時に、fork down パケット処理へ低い優先度を与える。）。その結果、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をそれぞれ12%, 17% 短縮することができる。

(4) Network稼働率

各リンクの稼働率は，IH2 台の時で約10%, IH4 台の時で約6%である(IH2台の時，リンクは 2本，IH4 台の時，リンクは 8本である）。さらにIH台数が増加すれば，各リンクの稼働率は、さらに減少する。よって、シミュレーション条件①は満足されている。Packet Switch の最大入力バッファ長は、IH2 台の時で15, 4台の時で14である。Network 速度を10Mbpsから60Mbpsへ上げると、IH4 台の時で処理時間を約10% 減少できる。

(5) PPU 稼動率と動的なプロセス分配効果

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、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。

4.1.2.2 Concurrent Prologプログラム

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

(1) 台数効果 (Fig.10)

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) 子プロセス消滅処理休止効果

Quicksort 実行時のリダクション等の回数を表 2に示す。

表 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

表 3

	IH 2台	IH 4台
総パケット数	141	211
true return パケット数	17	19
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

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表 3に示す。

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IH 2台の時で141 個、IH 4台の時で211 個のパケット転送が起る。つまり、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。 そして，表 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IH 2台の時で58%, IH 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IH 2台の時で15%, IH 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88であり、表 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、IH 1台の場合で13%, IH 4台の場合で10% ほど処理時間が増加する。

(4) AND-OR並列実行効果

Concurrent Prolog プログラムをAND-OR並列に実行すると，親プロセスと異なるIHへ分散された子プロセスは，guard 部実行に成功するたびにNetwork 経由で親プロセスのC-tag をチェックしなければならない。 そして，子プロセスは親プロセスからのreturnパケットを待たねばならない。 これにより，Network traffic は増加する。その結果，Quicksort でIH4 台の場合，AND-OR実行すると，処理時間はAND 並列実行のみの場合に比べ13% 増加してしまう。このようにConcurrent Prolog に対しては，OR並列実行は適切ではない。

4.2 詳細ソフトウェアシミュレーション

PIM-R データ内部形式等のPIM-R 詳細構造を正確に反映したシミュレーション，PIM-R のIM16～64台以上のシミュレーションを主目的として，並行プロセス記述機能と通信記述機能を持つ言語Occam で記述された詳細ソフトウエアシミュレータを開発し，各種データを収集中である。本シミュレータの実行はVAX11 上で行なわれる。

4.2.1 シミュレーション条件

①PPC でのdeadプロセス処理の休止は行わない。

②組込み述語の解がreturnした時，逆順コンパクションによりPTB 語長が長くならない場合は，直接overwrite する。

③PPC はReady PCB をRPQ の最後尾に繋ぎ，先頭のReady PCB をUUへ転送する。

④IM 2台の場合の分配法則は、自分→隣→自分→…の繰り返しとする。

4.2.2 シミュレーション結果

(1) Clause Pool 共有化効果

IM 2台の時，5Queens(Prolog) と50要素Quicksort(Concurrent Prolog)実行時のPTB 等に関する収集データを次に示す。(リテラル長=リテラルヘッダー長+リテラルエリア長)

5Queens	平均語長(A)	平均リテラル長(L)	L/A(%)
PTB	43.2	21.2	49.1
OR fork	42.3	22.9	54.1
true return	34.4	9.9	28.8

Quicksort	平均語長(A)	平均リテラル長(L)	L/A(%)
PTB	37.7	11.0	29.3
And fork	36.2	4.5	12.4
true return	24.3	5.5	22.5

Clause Pool をPPC からもアクセス可能とすれば，リテラルヘッダー，リテラルエリアをPTB の中に置く必要はなくなり，PTB 語長をリテラル長分 (約30～50%)短縮でき，PPU におけるプロセス生成，更新，UUでのunification に伴うcopy 量を減少することができる。 更に，OR(AND) forkパケット，true return パケットもそれぞれ54%,29% (12%,23%) 短縮できその分，Network traffic が減少する。

(2) Structure Memory Module 導入効果

形態素解析プログラム(Prolog)，DCG プログラム(Prolog)を走行させSMM の導入効果に関する各種のデータを収集した。

(i) ストラクチャエリア減少効果

次の通り，PTB 語長を，SMM 導入により20～30% (A/B) ，Clause Pool 共有化の下でもSMM 導入により35～45% (C/D) 短縮で　　更に，OR fork パケット，true returnパケット

形態素解析	平均語長 (Clause Pool 非共有)			平均語長 (Clause Pool 共有)		
	SMM 使用:A	SMM 未使用:B	A/B(%)	SMM 使用:C	SMM 未使用:D	C/D(%)
PTB	44.2	64.2	68.8	23.6	43.7	54.1
OR fork	35.8	47.4	75.5	17.9	29.7	60.2
true return	32.3	52.6	61.4	24.0	44.3	54.2

DCG	平均語長 (Clause Pool 非共有)			平均語長 (Clause Pool 共有)		
	SMM 使用:A	SMM 未使用:B	A/B(%)	SMM 使用:C	SMM 未使用:D	C/D(%)
PTB	29.3	37.9	77.3	14.5	23.1	62.9
OR fork	28.3	34.6	81.8	13.2	19.7	66.8
true return	51.3	86.0	59.7	42.8	77.4	55.3

もSMM 導入により20～40% (A/B),Clause Pool 共有化の下でもSMM 導入により40～45% (C/D),短縮できる。なお，Clause Pool 共有化とSMM 導入の両者をあわせることによりPTB 語長を約60% (C/B) ，OR fork パケット,true returnパケットを約50～60% 短縮できる。

(ii) Lazy Unification の効果

形態素解析	lazy(L)	normal(N)	(N-L)/N (%)
SMM read要求回数	822	908	9.5
SMM read return 総長	2772	2956	6.2

形態素解析プログラムでは上表のようにLazy Unification の導入により約10% の効果が得られたが，DCG プログラムのようにLazy Unificationの効果がみられない場合もある。

(3) GHC 化効果とGHC 支援データタイプ

KL1(85) のベース言語であるGHC[Ueda 85]ではチャネルのためのローカル環境を各PTB が持つ必要はなくなる。IM2 台での50要素数Quicksort(Concurrent Prolog)実行時，PPU からUUへのgoal(ただし，組込み述語ではない) に関して，

平均goal語長	35.4
平均ストラクチャエリア語長	16.0
平均チャネル情報(HB へのポインタとローカル環境) 語長	8.8

であるから，GHC 化することにより，パケット長を25% 短縮できる(HB へのポインタは，変数エリアのタイプChlRの語に格納できる) 。ただし，GHC では，述語を呼び出した時，各節のguard を実行している間は，呼び出し元から観測できるようなbinding を生成できないので，そのようなbinding(unification)はbody側に移動する必要がある。例えば Concurrent Prolog のQuicksort(付録2-2)qsort の第２節は

GHC では

```
qsort([],X):-true | X=[].
```

となる。よって，新たな変数X のために変数エリアが 1語，X=[]の内部形式は 4語だからリテラルエリアが 4語増加する。

このように，GHC ではConcurrent Prolog に比べ，変数エリアとリテラルが増加するので，ローカル環境を持たなくてもよいというGHC の長所を生かすためにも，Clause Pool を PPC からもアクセス可能なようにする必要がある(このようにすれば，PTB 内にリテラルエリアを持つ必要はない) 。

また，三種類の新たなGHC 支援データタイプ，TopCh(最初の呼び出し元goal内チャネル) ，WritableCh(binding可能なチャネル) ，NestedCh(guard内から直接，間接に呼ばれているチャネル) を導入することにより，PIM-R における Concurrent Prolog 実行機構を変更することなくGHC を実現できると考えている。

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、逆順コンパクション方式を採用し、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、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，SMM導入効果，Lazy unification効果，Concurrent Prolog 実行時のpacket転送の高速化の必要性を確認した。

また，現在、マイクロコンピュータ８台からなるハードウェアシミュレータを開発し，データを収集中である[Sugi 85] 。今後は、ソフトウェアシミュレータとハードウェアシミュレータにより、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、御協力いただいた（株）日立製作所　杉江衛、米山貢、岩崎正明，坂部俊文，古住誠一各氏，御難儀いただいた関一博ICOT研究所長，村上国男　元第１研究室長，御指導，御討論いただいた東京大学　元岡達教授（プロジェクト推進委員長），田中英彦助教授（ワーキンググループ１主査）ならびに関係各位に深謝する。

〈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.

[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", International Workshop on High Level Computer Architecture 84,1984.

[Kumo 85] 久門他，“並列推論処理システム -改良型節単位処理方式-，” 情報処理第30回全国大会，1985

[Hasu 85] 益田他，“並列推論マシンPIM-R の構造体メモリの一構成法”，情報処理第30回全国大会，1985

[Hoto 84] Hoto-oka,T.,Tanaka,H. et al.,"The Architecture of a Parallel inference Enngine -PIE- ",Proc. of Int.Conf. on FGCS 1984,ICOT, 1984.

[Onai 83] 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”，情報処理第27回全国大会，1983.

[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，1984.

[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

[Onai 85a]尾内，麻生，清水，益田，松本，“並列推論マシンPIM-R のアーキテクチャ”，情報処理第30回全国大会，1985

[Onai 85b]尾内他，“並列推論マシンPIM-R のソフトウェアシミュレーション”，情報処理第30回全国大会，1985

[Onai 85c]Onai,R.,et al,"Architecture of a Reduction-Based Parallel Inference Machine:PIM-R"，New Generation Computing, Vol.3, No.2, 1985.

[Pere 84] Pereira, L.M. and Nasr, R., " DELTA-PROLOG: A Distributed Logic Programming Language", Proc. of Int. Conf. on FGCS 1984,ICOT, 1984.

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

[Shim 85] 清水他，“並列推論マシンPIM-R のプロセス内部表現”，情報処理第30回全国大会，1985

[Sugi 85] 杉江他，“並列推論マシンPIM-R のハードウェア・シミュレータ試作”，Logic Programming Conference '85,Tokyo.

[Ueda 85] Ueda,K. "Guarded Horn Clauses"，ICOT Technical Report TR-103,1985(to be published)

［付録 1］データタイプ一覧

Type Classification			Abbre-viation
Type	Sub Type1	Sub Type2	
Variable	Void-Variable		Void
	Variable-1st		Var1
	Variable-Reference		VarR
Channel	Undef Channel	1st	UCh1
		ref.	UChR
	Read Channel	1st	RCh1
		ref.	RChR
	Write Channel	1st	WCh1
		ref.	WChR
Atomic	User Defined Atom		Atom
	Nil		Nil
	System Symbol	Type0	Sym0
		Type1	Sym1
		Type2	Sym2
	Integer		Int
	Real		Real
Structured	List		List
	String		Strg
	Vector		Vect
	Channel Information Reference		ChIR
	And	Parallel	Para
		Sequential	Seq
	Literal		Lit
	Pointer		Poi
Structured in SMM	Variable	void	SPVo
		Var 1st	SPV1
	Atomic		SPAt
	Non Ground (not instantiated)	List	SPNL
		String	SPNS
		Vector	SPNV
	Ground (instantiated)	List	SPGL
		String	SPGS
		Vector	SPGV

注　釈

VarRはリテラル・エリアあるいはストラクチャ・エリアに格納される変数タイプである。VarRは変数エリア内のVoidあるいはVar1を参照する。
Var1は変数エリア内の変数タイプである。

UChR, RChR, WChRはリテラル・エリアあるいはストラクチャ・エリアに格納されるチャネルのタイプである。これらは変数エリア内のChIRを参照する。

ChIRはストラクチャ・エリア内のチャネル情報を参照する。

Undef Channel というのは、コンパイル時に読み出しチャネルか書き込みチャネルか決定できないチャネルという意味である。

Sym0とは変数へのバインドを必要としない，"<" や">" のような組込み述語である。

Sym1とは"is"や"=" のような変数へのバインドを必要とする組込み述語である。

Sym2とはPPU によって処理される"guard","true","fail"のような組込み述語である。

Lit タイプはリテラル・エリア内のゴールへのポインタを持っている。

Poi タイプはClause Pool あるいはProcess Poolを参照する。

"Structured in SMM" はSMM 内に格納される構造体データ・タイプである

［付録２］シミュレーションを行なったプログラム

(付録2-1)　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2-2)　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).
qsort([],[]):-true | true.
 (partition ,lappendは省略)
```