高並列推論マシンにおける基本処理単位及び構造記憶に関する考察

平田 圭二, 丸山 勉, 田中, 英彦, 元岡 達
(東京大学工学部)

Abstract

When we design a highly parallel inference machine which executes logic programs, there are two important problems that greatly influence the machine performance. One problem is how to decide the elementary unit of parallel execution. The less the size of the parallel execution unit, the less is the overhead of copying and transferring the unit, but the unit interaction control becomes more complicated. The other is how to share the structure data which compose the parallel execution units. By sharing structure data, units size decreases, but the access conflict and load concentration on the shared memory decrease the performance.

Until now, a few model of parallel inference machines are proposed. However, we think that the above two problems are not discussed enough in these models. In this paper, we discuss these two problems in detail.

1. はじめに

並列推論マシンにおいて、その基本処理単位の設定とその表現形式及びそれらを構成する構造データの共有・制御方式の決定は、並列推論マシンのアーキテクチャや性能を決定する極めて重要な要素である。現在までに幾つかの並列推論マシンが提案されている［1］［2］［3］［4］［5］。しかし一般には、並列推論マシンにおける基本処理単位の設定とその表現形式および構造データの共有・制御方式について、他の形式・方式を用いた場合との比較・検討が不十分であり、より詳細な検討が必要であると考える。本論文では、高並列推論エンジンPIEに於けるこれまでの研究成果［11］［12］［13］を基にこれらの問題についての検討・考察を行なう。

2. 並列推論マシンにおける共有

一般に並列推論マシンにおいて、その性能を決定する主な要因として次の2つがある。

- 基本処理単位の設定
- 基本処理単位を構成する構造データの共有・制御方式の決定

基本処理単位の設定に於いては、処理単位のレベルを高くすればするほど、処理単位の制御は簡単になるが、その場合、一般に処理単位長はレベルを高くするに従って大きくなり、コピー・転送等のオーバーヘッドが増大する。逆に処理単位のレベルを低くすれば処理単位長を小さく抑えることができるが、それらに対する制御はより複雑となり処理時間の増加を招くことになる。さらに、推論マシンにおいては論理型プログラミング言語の持つ非決定性のために、処理単位の設定のレベルに応じて、ある処理単位が複数の処理単位により共有される場合が生じる。例えば、ある親ゴール（基本処理単位）の単一化を行なう時、そのゴールに関するOR並列性によって生成された複数の子ゴールによりその親ゴール自体が共有され、子ゴールの実行により親ゴールの変数の値が書き換えられるような場合である（この時、親ゴール中で値が結合される変数のみコピーする方法や、親ゴール自体をコピーする方法が考えられる）。基本処理単位の設定の仕方によっては、このような共有によって生じる負荷の集中により処理速度の低下を起こす危険性がある。この共有は処理単位の設定のレベルに応じて生じるものであって、処理単位の設定と密接に関係するものである。もし処理単位自体がそれまでの単一化の環境を全て持っているならば、このような共有は生じない。

これとは別に、処理単位を構成する構造データの共有を考えることができる。これは処理単位中に現われる構造データをあるメモリに格納し、場合によってはOR並列性によって生じた複数の処理単位間で共有することによって処理単位長の短縮を図り、それによってコピー・転送等のオーバーヘッドを軽減し処理速度の向上を狙うものである。また共有の結果、メモリの使用量を減少させることができる。しかし、むやみに構造データの共有を行なったのでは構造データを処理単位から切り分ける手間もさることながら、その構造データを共有する複数の処理単位からその構造データを管理するメモリへのアクセスが集中し、処理速度の低下を招くことになりかねない。この共有メカニズムを並列推論マシンに効率良く実装しようするなら、それは構造記憶の導入という形で自然に反映される。

3. 基本並列処理単位の設定

並列推論マシンの基本処理単位の設定を行なう際、次の2点が重要である。

① 基本処理単位のレベル
② 基本処理単位の内部表現形式

①は基本処理単位のレベルをどの程度に設定するかという問題である。既に述べたようにレベルを低くすれば基本処理単位長は短くなり、コピー等のオーバーヘッドを小さくすることができるが、基本処理単位間での共有が生じ負荷の集中による処理速度の低下を起こす危険性がある。②は基本処理単位を並列推論マシン上で表現する時に、どのような表現方式を取るかというものであり、具体的にはリテラルの構造をコピー／共有するか、また構造データ自体をコピー／共有するかという問題である。一般に構造は共有する程、処理単位長を短くすることができるが、共有の度合によっては基本処理間の独立性が損われ、負荷の集中を起す可能性がある。

並列推論マシンにおいて実行され得る並列性には、

① 引数間並列性
② ＡＮＤ（リテラル間）並列性
③ ＯＲ（定義節間）並列性

および、②③の結果生じる

④ 基本処理単位間並列性

の４つが考えられる。ストリーム並列は④に含まれるものとする。以下、主に②～④の並列性を対象とした並列推論マシンにおける基本処理単位の表現形式を決定する問題について考察・検討を行なう。

３．１ 基本処理単位のレベルの設定

逐次型処理系［１４］［１５］において、論理型プログラミング言語の持つＯＲ並列性等による非決定性はバックトラック等によって実現されている。逐次型処理系に簡単な変更を加え並列推論マシンとしてＯＲ並列性を積極的に活かそうと思うならば、例えば、逐次型処理系におけるスタック等を基本処理単位と考え、ＯＲ並列性による処理の分岐が生じるごとにスタックを全てコピーし他のプロセッサに転送するようなモデルが考えられる。しかし、一般にはこの処理単位長はかなり大きくなるものと考えられ、スタック全体をコピーするというのは非現実的である。

これに対して、転送量をある程度以下に押えるために基本処理単位を、例えば１つの定義節に対する環境というレベルまで下げ、ＯＲ並列性の結果生じた複数の処理単位がそれ以前の環境（スタック）を共有するという方法が考えられる。この時、ある単一化の結果３つの子ゴール（処理単位）が生じたならば、それらの３つの子ゴールは親の環境（即ち、自分の親に当る処理単位）を共有することになる。この場合、３つの子ゴールは親ゴールに対してそれぞれ異なった値を結合する可能性があり、それらに対して親ゴールを基本的にはコピーしなくてはならない（変更された場所のみコピーすることも可能）。これにより一度にコピーしなくてはならない量を適当に減らすことができるが、親ゴールを管理するメモリへの子ゴールからのアクセスの集中、及びそれにより生じる負荷の集中が問題となる。

以上の例から明らかなように、処理単位のレベルをどの程度に設定するかは処理単位の独立性に関する問題となる。処理単位のレベルを低くすることで処理単位長を減少させることはできるが、処理単位間の制御の複雑化、及び負荷の集中という問題を引き起こす可能性があり、基本処理単位のレベルをどの程度に設定するかは極めて重要な問題である。

並列処理の単位としては、次の４つを考えることができる。

① 引数
② リテラル
③ 定義節
④ 初期ゴールからの環境

①～④の順に応じて処理単位は大きくなり、④では処理単位間の共有は生じない。①～③においては、数個の引数、数個のリテラル、または数個の定義節を１つの処理単位として捉えることも可能であり、②において数個のリテラルを処理単位と見なした時、その数が定義節のリテラルの数と等しければ③と同等になる。しかし、それらは共有に対する効率化の問題であり、基本的には上記の４つに分類することができる。

上記の①～④に対する処理例を図１～４に示す。これらは単に処理方式の一例であり、他にもいろいろな制御方式を考えることができるが、図１～４は①～④の各処理単位のレベルに応じた処理の特徴を表現している。図１における処理方式の特徴は、本来ＡＮＤ関係にあるリテラルの引数が並列に処理されるため、各引数の単一化が終了した後、無矛盾性検査が必要である点と（無矛盾性検査を避けるためには動的に引数間のＡＮＤ関係を判定し引数のクラスタリング等を行なう必要がある）、②～④とは異なり各引数の単一化をリテラルという枠に縛られずに実行できる点にある。各リテラルの引数の単一化は各引数の単一化が可能となり次第、他の引数とは無関係に次々と実行される。図２の処理方式の特徴は、各ゴールが変数に値を結合した場合に（図２ａ）、その値を親ゴールに書き込む必要がある点と、その時点で定義節中の次のリテラルをゴールとして生成する点にある（図２ｂ）。変数に結合された値を親ゴールに書き込むタイミングとしては幾つか考えられるが、ここではそのゴールの実行が全て終了した時点で親ゴールに値を書き込むものとした。図３の処理方式は図２の処理方式とほぼ同様であるが、処理単位自体が定義節に対応するため、図２の様にリテラルをゴールとして生成する必要がない点（図３ａ）や、親ゴールへの値の結合と子ゴールの生成方法という点（図３ｂ）で多少異なる。②の場合（図２）は③の場合（図３）より処理単位は小さいが、定義

```
?-app([1,2],[3],X),print(X).
app([H|A],B,[H|C]):-app(A,B,C).
```

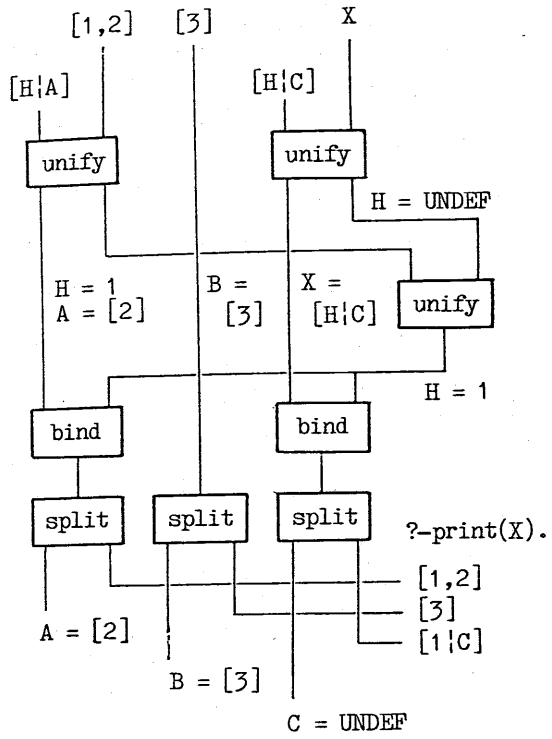


Fig. 1 An Example of a Data Flow Graph

```
?-f(X,Y),g(Y,Z).

f(X,a) :- p(X), q(X).
f(X,b) :- r(X), s(X).

p(a).   q(a).   r(a).   s(a).
p(b).   q(b).   r(b).   s(b).
```

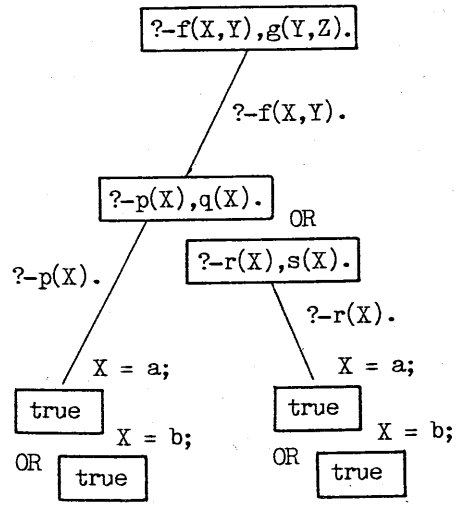


Fig. 2a An Example of Execution of Literal Level Units.

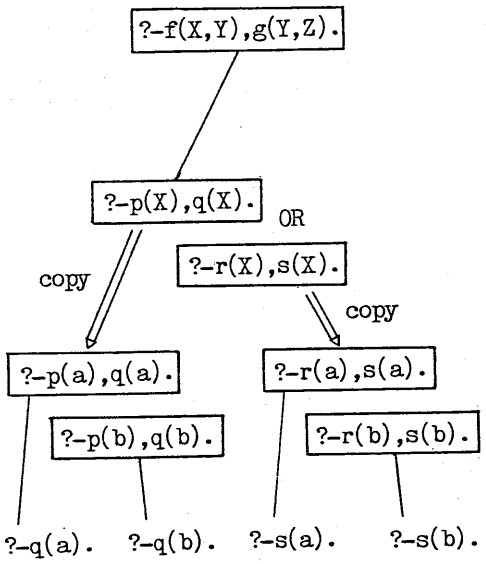


Fig. 2b An Example of the Execution of Literal Level Units

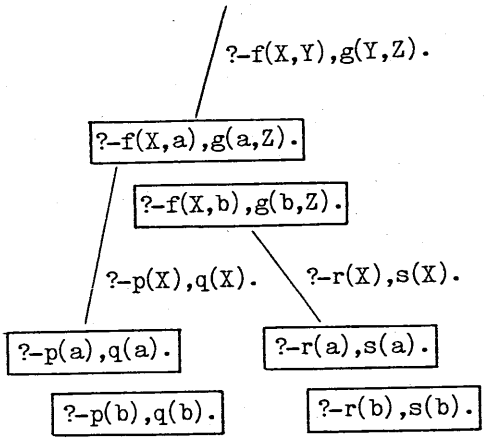


Fig. 3a An Example of the Execution of Clause Level Units

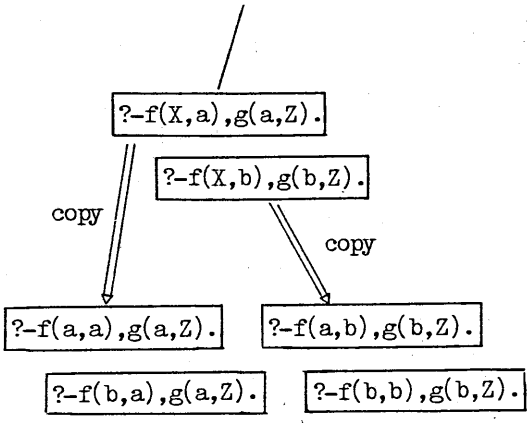


Fig. 3b An Example of the Execution of Clause Level Units

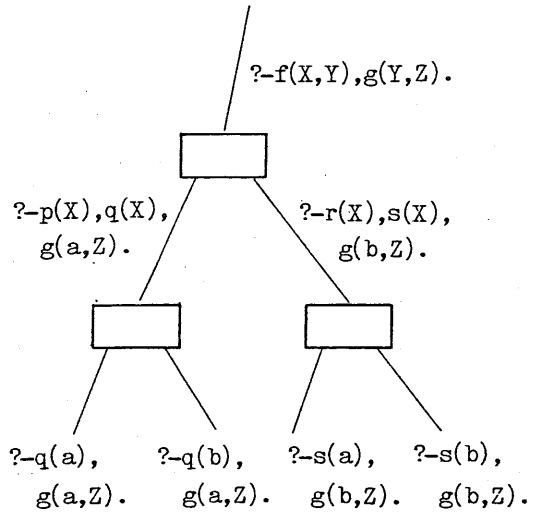


Fig 4. An Example of the Execution when the Unit has Environment from Initial Goal.

節から次のゴールを生成する分だけ処理の手間は大きい。④の場合（図４）では、各処理単位がそれ以降の処理を行なうために必要な全ての環境を持っているため親ゴールに値を書き込む必要はなく、②③とは大きく処理形態が異なる。

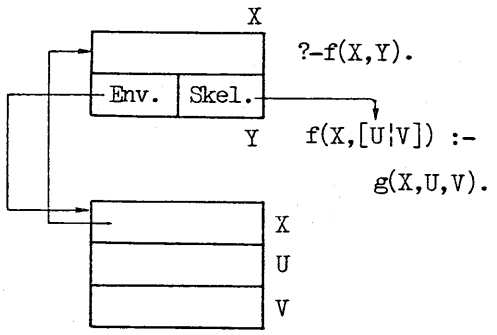
３．２　Skeleton の共有／コピー

一般に逐次型計算機における論理型プログラミング言語の処理系においては、リテラルの構造はコピーされず共有の対象となる。これは逐次型処理系においてリテラルの構造のコピーを行なっていたのでは、その量が定義節中の変数の数を大きく上まわり、処理速度が低下する為である。並列推論マシンとして専用ハードウェアを考えるならば、これらの定義節中のリテラルのコピーの手間は単一化とのパイプライン実行によって殆ど無視できる程度に押えることができる。この時、処理単位量の増加に伴なう転送のオーバーヘッド等はやや増加するが、リテラル構造はコピーしているので、単一化の際にリテラル構造への値の参照が不要、変数領域をリテラル中に埋め込むことにより変数領域の初期化が不要などの利点がある。

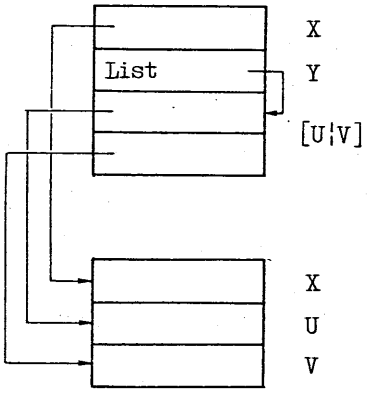
並列推論マシンの基本処理単位の内部表現形式について、次の４つの組み合わせを考えることができる。以下、構造体という用語は、単一化において共有／コピーされる構造データの skeleton 及び構造データ自体を指すものとする。

① リテラル、構造体供に共有
② リテラルのみ共有、構造体はコピー
③ リテラルはコピー、構造体は共有
④ リテラル、構造体供にコピー

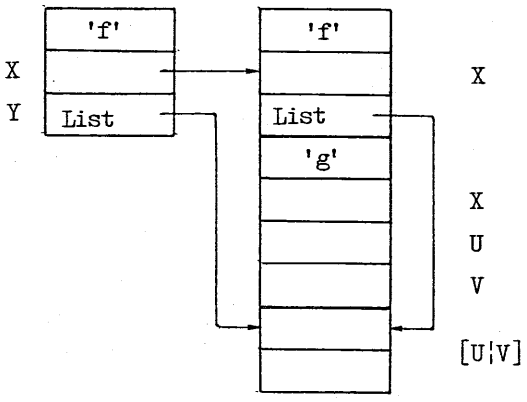
これらのうち、③はあまり現実的な方法であるとは考えられないので、ここでは議論の対象とはしない。また、処理単位が引数である場合には各リテラルは前処理においてデータフローグラフ等に展開されていると考えるのが自然であり、リテラルはコピーの対象とはならないので、①及び②に含まれるものとする。処理単位が定義節であるとき①，②，④の例を図５に示す。



(1) Literal Share / Structure Share



(2) Literal Share / Structure Copy



(3) Literal Copy / Structure Copy

Fig. 5 An Example of Internal Representation of the Unit (Clause Level)

①では変数に構造体が結合された時 molecule （skeletonへのポインタと環境へのポインタの対）として表現される。この方式では基本処理単位としてそれ以降の処理に必要である全ての環境を持っていない限り、単一化の際に親の処理単位への書き込みが起こることになる。この親の処理単位への書き込みを実現する方法として２通り考えられる。１つは、単一化の最中に常に親ゴールの環境を参照しつつ単一化を行ない、親ゴールの持つ変数に値が結合された時 molecule の環境へのポインタが親のゴールを指す場合には、親ゴールをコピー（または値が結合された変数のみコピー）するというものである。他の１つは、親ゴールへの参照の手間を避ける為に予め親の環境のうち必要な部分を自ゴール中にコピーしておき、自分の処理がすべて終了した後で、必要に応じて親ゴールの変数に子ゴールでが結合した値を書き込むというものである。このとき親ゴールはコピーされなくてはならない（値が結合された変数のみコピーすることもできる）。しかし、前者においては、親ゴールを管理するメモリへの負荷の集中が問題となり、後者においては構造体の skeleton の持つ変数番号の関係から最小限度必要なデータを親ゴールからコピーすることは難しくそれほど効率的に親ゴールの環境を自分の中に取り込むことはできない。最悪の場合は、それまでの環境を全て持ち運ぶことになってしまう。

②，④においても①と同じ処理方式が考えられるが、これらの場合には構造体のコピーを行なうので①と異なり、 skeleton 中の変数番号等の問題は生じず効率的に単一化に必要な最小限度のデータを自分の中に取り込むことができる。④は②より、リテラルの構造をコピーする分だけ処理単位長が長くなるが、単一化の際リテラルの構造を参照する手間が不必要であるため、単一化をやや高速に行なうことができると考えられる。しかし、定義節がコンパイルされている場合には殆ど差異はなく、②のほうが有利であると考えられる。

４．基本処理単位の内部表現形式

３章では並列処理単位を考える際の主な観点について述べた。これらを組み合わせることで、表１に示すような基本処理単位の内部表現形式を考えることができる。これらの内部表現形式の優劣を議論する時、評価すべき点として次のようなものを考えることができる。

（ａ）基本処理単位長

基本処理単位長は、その処理単位の独立性を損うことがない限りにおいて短い方が良い。

（ｂ）基本処理単位の独立性

基本処理単位長と密接な関係を持つが、処理単位間の論理的共有の度合は低い程良い。

（ｃ）単一化の高速性

単一化をより高速に行なうことができる処理単位の内部表現形式がよい。また決定的である定義節に対する処理も高速に行なえるものであることが望ましい。

以下表１に示した内部表現形式について、それぞれ（ａ）～（ｃ）の立場から簡単な検討を行なう。

４．１　引数間並列処理

表１の①，⑤に示した引数間並列処理については２つの立場を考えることができる。１つは、引数間並列性を最大限に活かして高並列な処理を行なおうとするものである。この方式に於いては処理単位のレベルが低いため他の処理方式とは異なり、リテラルの枠に捕らわれずに各リテラルの単一化が可能となり次第、次々とその引数の単一化を行なうことができるので、他の処理方式よりより高度な並列処理を実現することが期待できる。しかし、無矛盾性検査（または動的な引数のクラスタリング）や、論理型言語では引数の入出力が動的に決定等の問題があり、その制御はかなり複雑なものとなるため、そのオーバーヘッドが問題となる。

他の立場は、引数の並列処理を１つのリテラル内の引数に限定するものである。この場合、引数間並列処理を他の処理方式と組み合わせることができる。しかし、１つのリテラルの引数は平均数個程度であり、ＡＮＤ関係にある引数を並列に処理するためのオーバーヘッドを考えるならば、現在のところその効果については評価しがたい。

引数間並列処理を効率的に行なうためには、無矛盾性検査等を高速に行なうアルゴリズム、専用ハードウェア等の研究が必要であり、それらは今後の課題である。

table 1　Internal Representaion of the Elementary Parallel Execution Units

share/copy	level	argument	literal	definition	Envs.
structure share	literal share	①	RIT. ②	③	④
	literal copy				
structure copy	literal share	⑤	⑥	⑦	⑧
	literal copy		⑨	⑩	⑪

Envs. is environments from initial goal.

4．2　リテラル共有／構造体共有

表１の②，③は構造体の skeleton を共有しているために変数に構造体が結合された場合には、 molecule が作られその環境（処理単位）へのポインタが他の処理単位を指すことがある。既に述べた親ゴールの値を参照しつつ単一化を行ない親ゴールの変数に値を結合するという方法については、その親ゴールを管理するメモリへのアクセス競合が起る他、親ゴールをコピーする手間も大きく（値の結合された変数のみコピーする場合には、その値を見つける手間が大きい）、その処理がボトルネックとなり、あまり有効な方式であるとは考えられない。また、親ゴールの環境のうち必要なものだけを自分の環境にコピーするという方法も、共有される構造体の変数番号の付け換えが困難であるため（構造をコピーすることも考えられるが、これは表１の⑥，⑦とほぼ同様の方式になる）、結局親ゴールのもつ環境を全てコピーすることになりかねない。従って、⑥，⑦と比較した場合、①，②の方法はあまり効率的であるとは言えない。④のようにそれ以降の処理において、必要なデータを全て持ち運ぶならば上記のような問題は生じないが、逐次型処理系において実際に使用されるスタックを考えると非現実的である。以上述べたように構造体の skeleton を共有する方法は、高並列処理には、適してはいないと考えられる。但し、高々数個の並列処理を行なう場合に限れば有効な１つの方法と成り得ると考えられる。

```
?-f([U|V],X),g(U,X),h(V,X).    (a)

f(X,Y) :- p(X,Z),q(Z,Y).       (b)
```

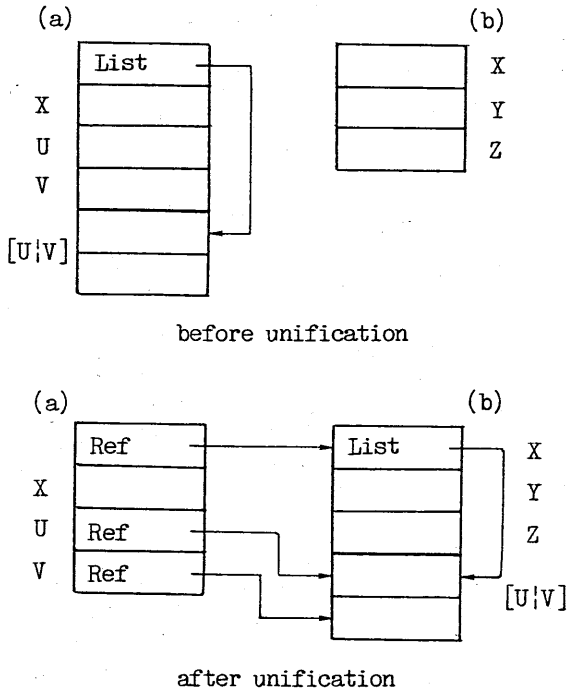


Fig. 6　An Example of Copy of Structure

4．3　構造体コピー

表１の⑥，⑦，⑨，⑩に対しても、②，③の場合と同様に２つの処理方式が考えられるが、親ゴールの値を直接書き換えるという方法は高並列処理には適してはいない。しかし、後者に対しては、これらの場合、親ゴール中のデータを自分自身の環境の中に効率的に取り込むことは容易であり有効な方法であると考えられる（図６）。

まず⑦と⑩の比較については、⑩のほうがリテラル構造のコピーを行なう分だけ処理単位長が一般に大きくなるが、単一化ではリテラル構造への参照がない分だけ多少高速な単一化を行なうことができる。しかし、定義節がコンパイルされている場合には単一化の速度は両者とも同程度になると考えられる。⑦の⑩に対する利点は、並列性の無い定義節との単一化に対して処理単位をそのまま逐事処理系におけるスタックの一部とみなすことによって、より高速な処理を行なうことができる点にある。しかし、両者の是非は今後、慎重に検討する必要がある。

⑥，⑨と⑦，⑩については、⑥，⑨の方が処理単位が小さい分だけコピー・転送等のオーバーヘッドは小さいが、その分リテラル間の分割及び変数に結合された値の引き渡し等が必要になり、制御の手間は大きくなる。これを実現する高速なアルゴリズムを見い出すことができれば、⑥，⑨の方式の方が有効であると思われるが、現時点ではどちらがより有効な表現形式であるかは微妙な問題である。⑥，⑨はリテラルの共有／コピーという違いはあるが、⑥においても単一化に必要な引数はいわゆるサブルーチンコールのような形をとることになるので、両者は実際にはほとんど変わらないことになる。⑥は⑦と異なりリテラル間の制御が必要となるので、決定的な処理を高速に行なうことは難しい（⑨についても同様である）。

⑧と⑪については、⑪のほうがリテラルの構造を持つため処理単位長が長いように思われるが、実際には、⑧では単一化を行なう時リテラルの構造を参照し、その結果、処理単位中にある変数の値を参照するので、変数番号の問題により不要となったデータをむやみに取りのぞくことができないため、それ程単位処理長を短くすることはできない。従って、どちらの処理単位長がより小さいかは、実行されるプログラムに依存することになると思われる。

⑦，⑩と⑧，⑪については、⑧，⑪の方が処理単位長は大きいが、これによるオーバーヘッドをパイプライン処理等によって隠すことができるならば、⑧，⑪の方が親ゴールに値を書き込む手間が不要な分だけ有利である。しかし、⑧，⑪では実行プログラムによって、処理単位が長くなりパイプライン処理等によってもそのオーバーヘッドを隠すことが不可能になる場合がある。

ＡＮＤ並列処理を行なうには基本的に処理単位をリテラルまで下げる必要があるが、ＡＮＤ並列に実行されるリテラル以外のデータも処理単位として持ち運ぶことに

すれば、いずれの表現形式においてもＡＮＤ並列処理を行なえる。しかし⑧，⑪では処理単位長が大きくなりすぎる可能性があり、処理単位全体のコピーは現実的ではない。これに対し、⑦，⑩では処理単位長はそれ程大きくならないと考えられ、ＡＮＤ並列処理に適した表現形式であると思われる。⑦，⑩と⑥，⑨の優劣については、⑥，⑨の方が処理単位長は小さくなるものの、ＡＮＤ関係にある処理単位の生成などの手間は大きいため（⑦，⑩では処理単位全体をコピーすればよい）、慎重に評価する必要がある。

４．４　逐次推論マシンから並列推論マシンへの拡張

以上の考察は最初から並列推論マシンを意図した場合のものであるが、これとは別に逐次推論マシンを複数台並べることにより、並列処理を行なう方式が考えられる。この場合は、高並列処理を目指しているマシンとは評価基準が基本的に異なり、複数ある逐次型処理系をいかに効率よく稼動させるかが第１のポイントとなる。この方式では、全ての処理系が稼動している限り他の処理系への処理負荷の積極的な分散を行なわない為、処理単位の転送回数は少なくその手間は多少大きくとも問題とはならない。このような観点から見れば、③，④，⑦及び⑧を逐次型処理系における処理途中のスタックとして転送する方法が考えられる。しかし③，⑦を転送したのでは、それらに関する一連の処理量がそれ程大きくない可能性があり、転送の回数がむやみに増えてしまうことが考えられる。従って、逐次推論マシンを結合したモデルにおていは④，⑧を処理単位とするのが適当でると考えられる。この時スタック全体を送る必要はなく、ある程度の処理単位となり得るものを転送すればよい。具体的には、転送量及びゴールの切り分けの手間を考え、できるだけ初期ゴールに近い alternatives を転送することが考えられる［５］。

以上、幾つかの基本処理単位の内部表現形式について議論した。どの内部表現形式が有効であるかは処理単位長に伴なうオーバーヘッドと制御の複雑さに伴なうオーバーヘッドとのトレードオフであり非常に難しい問題である。実際には、⑦，⑧の組み合わせ等のように、その時点におけるマシンの負荷状況等によって処理単位を動的に変えるのがもっとも有望な方法であると思われる。

５．構造データの共有とコピー

第２章でも触れたように、並列推論マシンにおいては並列処理モデル上の基本処理単位間で環境を共有／コピーするという選択肢とは別に、環境や構造体を表現する構造データを共有／コピーするという選択肢がある。論理的には環境や構造をコピーするような処理モデルでも、その実現法においては構造データをコピーせず、ある部分を共有して残しておくこともできる。これによりコピーの手間の節約ができる。その共有される部分を構造データの内未定義変数を含まない部分（Ground Instance）に限る場合について議論する。この共有メカニズムを効率良く実現する為には構造メモリを導入するのが得策である。本章以下では構造データの内部表現法と構造データを効率良く処理できるハードウェア、特に構造メモリについて議論する。

本章では構造データの内部表現方法について述べ、共有の対象となるＧＩの切り分けについて述べる。

５．１　構造データの内部表現法

論理型プログラミング言語で取り扱う構造データ中には、変数が存在する。変数に値の束縛が生じると、その変数を含む構造データは全く異なるものを表現することになる。今、変数を含むような構造データをポインタで直接指すことによって共有する場合を考える。もし変数に対する値の束縛によって生じる副作用を避けようとするならば、必然的に構造データ全体をコピーすることが必要になり、大きなオーバーヘッドである。しかし、構造データの読み出しについては、変数は未定義のものしか存在しないので、いわゆる「たぐり」という操作は不要になるという利点がある。これに対して、逐次推論マシンで行なわれているように、構造データを molecule（ skeleton へのポインタと環境へのポインタの対）で表現すれば、変数への束縛は環境つまり変数部のコピー及び書き換えだけで済み、上に述べた方法より仕事量はずっと少なくて済む。但しこの場合は構造データを読み出す場合に変数の本当の値を知る為にたぐるという操作が必要になる。

前にも述べたように並列推論マシンでは基本処理単位を細くし、処理に不要な部分を選択的に持ち運ばないことで、基本処理単位の転送時間の短縮、オーバーヘッド軽減による処理時間の短縮を達成できる。これと同様の議論は構造データの内部表現形式に対しても適用され得る。即ち未定義変数を含まなければ書き換えは生じないので、未定義変数を含まない構造データ（ＧＩ）を切り分けてしまい、コピーや書き換えの対象から外せば、処理の効率化が図れる。ＧＩを切り分け、それをポインタで参照することの結果としてＧＩの共有が生じる。

５．２　Ground Instance の切り分け

共有／コピーの対象となる構造データはＧＩ及び nonＧＩという異なる性質を持つ２種類の構造データから構成されており、各々に対して効率的な処理を考えねばならない。 nonＧＩを共有／コピーするというのは、その基本処理単位としての性質も考慮に入れて決定せねばならない。これに対してＧＩは書き換えが生じないので、

その転送オーバーヘッド、処理時間を考えると、共有する方が有利である。

論理型プログラミング言語の実行が進行するとともに新に変数が導入されつつも、徐々に値が束縛されて行き最終的にはＧＩが構造データの大半を占めるに到る場合が多い。ＧＩというのはＩストラクチャ［１０］のように単調増加的に生成されて行く。識別され切り分けられたＧＩの部分は、適当なメモリ中に格納され、転送の対象から外され、その結果として共有が生じる。したがってＧＩも切り分けられる以前はコピーの対象として取り扱われる。

６．Ground Instance と構造メモリ

論理型プログラミング言語を実行する並列処理マシンにおいて、最初から存在しているＧＩには初期ゴール、定義節中のＧＩがあり、さらに子ゴールを生み出して行く際に段階的に生成されるＧＩもある。ＧＩのメモリへの格納のタイミングは切り分け時でなくとも良い。また共有されているＧＩを読み出すのも単一化を行なっている最中でなくとも良い（あらかじめ適当な量だけメモリから読み出しておいても構わない）。構造データ（ nonＧＩ）どうしの単一化を進めて行く内、ゴールあるいは定義節のどちらか一方以上にメモリ中のＧＩへのポインタが出現した場合は、さらにそのポインタをたぐり、その先の値をとり出して来なければならないが、この単一化の際中にメモリ中のＧＩをたぐる操作をＬazy Ｆetchと呼ぶ。現在提案されている並列推論マシンを概観すると、構造データのレベルでこれらのＧＩの格納、取り出しを行なうために構造メモリ（Ｓtructure Ｍemory ）を設けているのはＰＩＥ［１］、ＰＩＭ／Ｒ［３］などである。一般に構造メモリという時は、構造データ全体を扱い、さらに高度な構造データ操作機構を組み込んだものを指すが、構造メモリの一番基本的な位置付けとしては、上述したように、ＧＩのみを対象とした構造メモリという立場があり；効率化の手法としては殆どの並列推論マシンに適用可能な考え方である。本章では、ＧＩ専用のＳＭについて、その内部構成及びＧＩの内部表現法について議論し、次章では構造データを処理する上で避けて通れないガーベジコレクションの問題について述べる。

６．１　アドレス変換テーブルと可変長セルメモリ

並列推論マシンは複数台のプロセッサがネットワークで接続され、独立に処理が遂行されているので、ＳＭ中のＧＩを参照するポインタはマシン中に散在している。構造データのノードは、リスト型やベクタ型から成るので、ノード長は不定となり、ガーベジコレクションによってメモリ再利用のための圧縮を行なわなくてはならない。その結果、ノードの存在場所は移動せざるを得ず、あるノードの移転先をそのノードを指すすべてのポインタに知らせるということを、移動したすべてのノードについて実行するのは非現実的である。何故なら並列推論マシン環境では、ある構造データの参照数はＬisp の処理系などとは比べものにならない位大きくなってしまうからである。そこで多少のオーバーヘッドは生じるが、ノードの実体を格納するメモリの他に、アドレス変換テーブル（Ａddress Ｔranslation Ｔable）を設ける必要がある。ノードの実体を格納するメモリを可変長セルメモリ（Ｖari-sized Ｃell Ｍemory ）と呼ぶ。ＡＴＴ上のノードは１度割り当てられたらゴミになるまで移動しない。それに対してＶＣＭはガーベジコレクションによってメモリ領域は圧縮される。しかしＶＣＭが圧縮されたことによる影響はＡＴＴまでで食い止められる。ＡＴＴとＶＣＭを用いた構造データ表現法は、ＳＭ内参照の方法によって大きく３つに分類できる。

① ＳＭの外部から現在参照のあるノードのみＡＴＴにエントリする（図７、Ｔop Ｅntry方式）。

ＳＭの外部から参照のあるノードは必ずＡＴＴにのせる必要があるという意味では最も基本的な方法であり、ＶＣＭ内での直接参照が生じる。ＡＴＴのエントリは常に外部参照のあるセルに限ると、今までＡＴＴにエントリのあるノードとそうでないノードを逐一管理する必要があり、かなり面倒である。

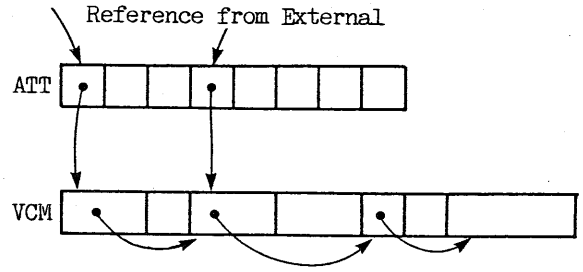


Fig. 7 Top Entry (TE) Method

② ＳＭの外部から過去参照があったノード及び現在参照があるノードをＡＴＴにのせる（図８、Ｐartial Ｅntry方式）。

ＴＥ方式では外部からの参照に伴ない、ＡＴＴのエントリを管理していたが、ＰＥ方式では１度ＡＴＴにエントリすると、そのままノードがゴミになるまでエントリされ続ける。ＡＴＴ更新の手間はＴＥ方式より軽いが、ＴＥ、ＰＥ方式ともにノードがＬＦされた時、ＶＣＭにしかエントリがなかったノードについては、その都度新にＡＴＴのエントリを生成する必要がある。

③ すべてのノードに対してＡＴＴのエントリを作る（図９、Ｆull Ｅntry方式）。

たぐりを行なう時は、毎回ＡＴＴを調べるオーバーヘッドを伴なうが、ノードの管理は３方式中、最も簡単である。

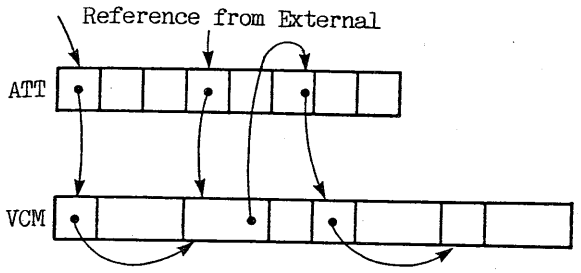


Fig. 8 Partial Entry (PE) Method

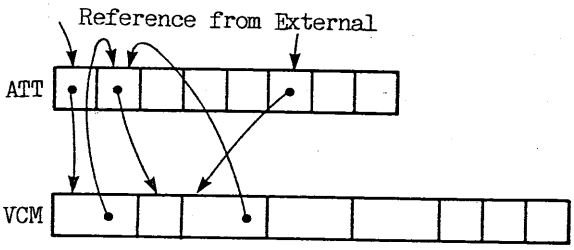


Fig. 9 Full Entry (FE) Method

6．2　Lazy Fetchに適したメモリ構成

ＬＦは単一化の最中に共有されてＳＭ中にあるＧＩの一部が必要となった時に、プロセッサがＳＭにアクセスすることを言う。単一化の際中であるから一般には高速な応答が要求される。したがって同じノードのフェッチをするのは無駄が多いので、プロセッサ側にはバッファ（ＬＦバッファ、ＬＦＢ）を設け、ＬＦしたノードをある一定期間プロセッサ側に保持しておくのが得策である。また定義節をコンパイルする時その形を見ることでＬＦ時に何段たぐれば十分であるかは予想がつくので、ＬＦ１回で一般には複数個のノードフェッチが行なわれる。以下の議論は我々が現在開発を進めている高並列推論エンジンＰＩＥの研究によって得られた知見を基にしている。ここで生じ得るＬＦのパターンについて分類を行なう。

（ａ）同一のたぐりを繰り返すＬＦパターン
同じＧＩを指す２つのポインタから同じ深さまでＬＦが生じる場合、或いは２回目のＬＦが１回目のＬＦとは全く違うＧＩに対して生じる場合（図１０）。このパターンの場合はＴＥ、ＰＥ方式がたぐりの時ＡＴＴを経由しない分だけ若干有利である。しかしＬＦした最後のノードをＡＴＴにエントリしなければならない。

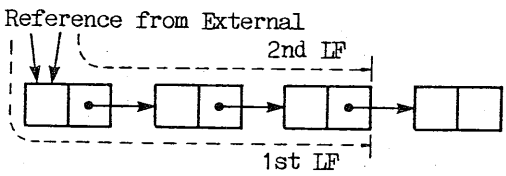


Fig. 10 Lazy Fetch Pattern (a)

（ｂ）途中まで同じ経路であるようなＬＦパターン
同じＧＩを指す２つのポインタから異なる深さまでＬＦするパターン、或いは途中から違う枝をＬＦするパターン（図１１）。ＦＥ方式では１回目のＬＦで取って来たすべてのノード情報がＬＦＢに残っているので、２回目のＬＦでは１回目とは違う部分のみＬＦを行なえば良い。これに対してＴＥ、ＰＥ方式では最悪の場合、先頭のノードしかＬＦＢに存在しないので、そこから再びＬＦを開始すると１回目と同じノードをＬＦすることになり不利である。

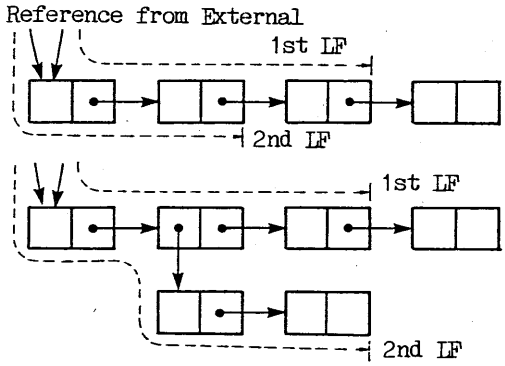


Fig. 11 Lazy Fetch Pattern (b)

（ｃ）たぐりの道筋は同じで、深さだけ異なるＬＦパターン
一方のポインタがもう一方のポインタの指すＧＩの途中からを指している場合（図１２）。２回目のＬＦは１回目のＬＦの途中から生じる、或いはその逆のパターンである。図１２で１回目のＬＦがＡからはじまるならば、フェッチされたノードがＬＦＢにあるため、ＰＥ、ＦＥ方式はＴＥ方式よりも有利である。しかし１回目のＬＦがＢから始まるならば、ＴＥ、ＰＥ、ＦＥ方式すべて１回目にフェッチしたノードを再度取りに行かねばならず効率は良くない。

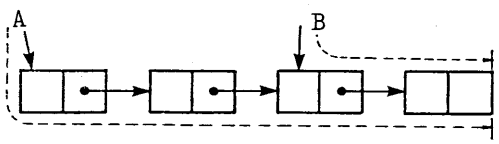


Fig. 12 Lazy Fetch Pattern (c)

以上の議論から次のことが言える。

- パターン（ｂ）が増える程ＦＥ方式が有利
- １度にＬＦする段数が増えると、ＡＴＴを経由する分だけＦＥ方式は不利

ここでＰＩＥについてのシミュレーション結果を参考にする。表２にパターン（ｂ）の生じた割り合いと、ＬＦが１段で済んだ割り合いを示す。シミュレーションで用いたテストプログラムにおいて、ＬＦパターン（ｃ）は生じなかった。従ってパターン（ａ）人間が書く自然なプログラムでは最も普通に生じるものと思われる。またＬＦは殆どの場合１段しか行なわれないので、ＬＦの立場からはＦＥ方式が有利であると言える。

table 2 Simulation Data of Lazy Fetch on PIE

program	Equiv2	Pentomino	8Queens
% of pattern (b)	28	23	0
% of 1 level IF	96	72	100

6．3 Ground Instance の格納

ＧＩ格納のコマンドやデータがパケットとしてＳＭに送られて来る時、構造データが相対番地ポインタで表現されていれば、ＴＥ、ＰＥ方式はブロックコピーができて有利であるが、ＴＥ方式はＡＴＴのエントリを常に管理していなくてはならない。ＦＥ方式はＡＴＴのエントリを作る手間が必要である。ＧＩを格納する時、プロセッサ側ではその格納番地を知る必要がある。しかしＧＩの格納番地をＳＭからプロセッサ側に返送するのは効率的でないので、あらかじめプロセッサ側にＳＭの空き番地を知らせるような機構を設けるのは有効である。

7．構造メモリのガーベジコレクション

前章まで比較的処理が簡単なＧＩに限った場合の構造メモリの構成について議論して来た。本章では、このような機能を持つＳＭにおけるガーベジコレクション（ＧＣ）方式について議論する。

並列推論マシンのＧＣに最も必要なことは、できるだけオーバーヘッドの少ないことである。例えば逐次型ＧＣであれば、処理の中断している時間はできる限り短い方が好ましく、並列型ＧＣであればガーベジコレクタによるメモリの読み書きの回数はできる限り少ない方が良い等である。並列推論マシンでは、各プロセッサ中で、ゴールが独立に処理されているので、一括逐次型ＧＣでは一斉に処理を中断させるのが非常に困難である。よって以下では並列型ＧＣに限って話を進めることとする。またＳＭを何台に分散して置くか、集中して置くかもＧＣ方式に大きく影響するが、適当なネットワークによって各プロセッサとＳＭは接続されているものとする。

7．1 並列推論マシンにおけるガーベジコレクション方式

一般にガーベジコレクション処理は２つのフェーズから成る［９］。

Ｍフェーズ　：　ゴミとそうでない記憶領域を識別するフェーズ

Ａフェーズ　：　ゴミを回収し、使用可能な領域を作り出すフェーズ

Ｍフェーズを実行する方法としては、次の２つの方法が代表的である。

・ＲＣ法　：　ノード毎に参照カウンタ（Reference Counter）を設け、いくつのノードから指されているかを数える。参照カウンタがゼロになった時、そのセルはゴミになったと判断される。

・Ｍ＆Ｔ法：　根ノードから構造データの印付け、辿り（Mark ＆Trace）を行ないながら到達可能な領域を生きている領域として行く。

Ａフェーズを実行する方法は次の２つに分類できる。

・ＦＬ法　：　ゴミと識別されたノードは自由リスト（Ｆree Ｌist ）につながれる。

・Ｃ法　　：　生きているノードを別のメモリ領域に詰め換える。詰め換えた後、ノードの相対位置関係から、さらに任意順、直線化、横すべりに分けられる。

ＧＣ方式は以上の方法を組み合わせ、並列処理環境に適した方式を選択せねばならない。　並列推論マシンでは、メモリの応答時間の増大がパフォーマンス低下の一因となっている。そこでＡフェーズについては、メモリを長時間ロックしてしまう横すべりによる圧縮は好ましくない。ＶＣＭについてはコピーによる詰め換えが良いであろう。これはメモリ空間を２つの部分空間に分けて、一方の部分空間のみで処理を進め、空き領域がなくなった時点で生きているノードをもう一方の部分空間に移動する方法である。この方法による詰め換えはＶＣＭ上の移動したセルを参照するポインタをＡＴＴに書き込む時だけのロックで済む。ＡＴＴについては、ＧＩの格納のための空きアドレス管理が必要であることからＦＬ法が適当であろう。これより並列処理マシンで比較的効率良くＳＭのＧＣを実行するためのアルゴリズムとしては次の３つの方式が有望である。

・ＭＴＣ法　：　印付けと辿りを行ないながら、ＶＣＭ同志でコピーする。

・ＲＣＦＬ法：　ＲＣ法で生きているセルの識別をし、ＦＬ法で回収する。

・ＲＣＣ法　：　ＲＣ法で生きているセルの識別をし、ＶＣＭ間でコピーを行なう。

以下さらに詳しく説明する。

（１）ＭＴＣ法

ＳＭ外部からの構造データへの参照を見い出し、何らかのネットワークでその参照情報を転送する。ＳＭは１つのＡＴＴと２つのＶＣＭを備えたメモリ構成をとり、Ｂakerのアルゴリズムのように［８］、根ノードから構造データを辿りながら、もう一方のＶＣＭに圧縮・コピーしていく。ＡＴＴ上のゴミは自由リストにつなぐ。

（２）ＲＣＦＬ法

ＳＭ外部で構造データの参照を操作する時に、参照カウンタ（ＲＣ）命令を発生させ、何らかのネットワークでＳＭへ転送する。ＳＭ中でＲＣがゼロになったノードについて、ＡＴＴ上のセルはＡＴＴの自由リストにつなぎ、ＶＣＭ上の領域は同じＶＣＭサイズの自由リストに

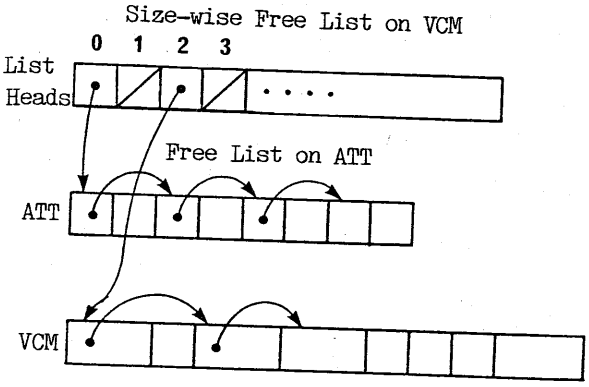


Fig. 13 RCFL Method

連結される（図１３）［７］。メモリ割り当ての高速化とＳＭの空きアドレス分配のために、このような自由リスト管理を行なっている。ＲＣ法は、循環リストを回収できないという本質的欠点を有するため、いつかは自由リストがなくなりノードが割り当てられなくなる。この時はＲＣがゼロでないノードのみもう一方のＶＣＭにコピーを行ない、ＶＣＭ上の領域を再編成する。

（３）ＲＣＣ法

ＳＭの外部から参照されているノードのみ参照カウンタで数え、ＳＭ内部からの参照は数えない。つまりＭＴＣ法では、根ノードを見つけるためにＳＭ外から根ノードをネットワークで転送していたが、ＲＣＣ法ではこれをＲＣ法で実現していることになる。

７．２　各ＧＣ方式の比較

以上の３方式を比較する際、問題になるのは次の２点である。

・プロセッサとＳＭ間のネットワークトラヒック及びネットワークトラヒック及びＧＣの為のプロセッサの負荷

・ＳＭ内部の負荷

まずネットワークトラヒック及びプロセッサの観点から考える。根ノードをプロセッサ側から送り出す為には、環境や構造体を格納しているメモリに対して、最悪の場合は全空間のスキャンを行なう必要がある。これに対してＲＣ法の場合は処理の途中で副次的にＲＣ命令を生成することができる。またＲＣ命令を生成しても即座に送出するのではなく、適当な期間蓄積し、最適化をかけると転送データ量を低く抑えることができる。ＲＣ命令に最適化をかけて転送すると、そのデータ量は根ノードを送る場合よりも少なくなると思われる。ＭフェーズをＲＣ法で実現するか、印付けと辿りで実現するかは、転送データ量とオーバーヘッドとのトレードオフで決まる。

次にＳＭ内部の負荷について考える。ＭＴＣ法のＭフェーズでは、構造データをたどるので、その仕事量は生きているノード数に比例する。この方法では、ＭフェーズとＡフェーズは混在しているが、ＶＣＭをコピーする時は新ＶＣＭのノード割り当て、ＡＴＴの書き換えという手間が必要である。すべてのノードが新ＶＣＭに移動し終えるとＡＴＴをスキャンし自由リストを作るが、この仕事量はＡＴＴの大きさに比例する。

ＲＣＦＬ法のＭフェーズにおける仕事量はゴミの生成速度に比例する。ゴミが生じなければ参照カウンタを操作するだけで構造データをたぐる必要はない。Ａフェーズはｍｔｃ法同様、Ｍフェーズと混在しているが、そこではゴミになったＶＣＭを解放しサイズ分けされた自由リストにつなぎ、ＡＴＴも別の自由リストに連結する。ＡＴＴのスキャンは不要であり、仕事量は生成されるゴミノード数に比例する。

ＲＣＣ法のＭフェーズは根ノードのみＲＣ法で、ＳＭ内はＭ＆Ｔ法で実現する。従って仕事の大部分はＭ＆Ｔに費され、仕事量は生きているセル数に比例する。Ａフェーズはｍｔｃ法のそれと同じである、またＭフェーズとＡフェーズを逐次的に実行するような方式も可能である。

８．おわりに

以上、本論文では高並列推論マシンの性能を決定する極めて重要な要素である並列基本処理単位の設定と、基本並列処理単位を構成する構造データの共有制御方式について検討を行なった。今後、これらの成果を基にシミュレーション等により詳細な評価検討を行なう予定である。

謝辞

いつも貴重な助言や議論をして頂くＳＩＧＩＥのメンバー諸氏及び坂本光弘氏、相田仁氏に感謝いたします。

＜参考文献＞

［１］Moto-oka，T.，Tanaka，H.，et al, “The Architecture of a Parallel Inference Engine －ＰＩＥ－”，ＦＧＣＳ´84，ＩＣＯＴ，pp479-488

［２］Ito, N.，and Masuda，K.，“Parallel Inference Machine Based on the Data Flow Model”，Proc．of the International Workshop on Highlevel Computer Architecture 84, pp4.31-4.40，1984

［３］尾内　他，“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”ＩＣＯＴ Technical Report ＴＲ-077, 1985

［４］Haridi，Ｓ.，and Ciepielewski，Ａ.，“An Or-parallel Token Machine”，Logic Programming Workshop 83, 1983

［５］久門　他，“並列推論処理システム－改良型節単位処理方式－”，情報処理学会第30回全国大会７Ｃ－８，1985

[6] Yuhara,M. ,et al, "A Unify Processor Pilot Machine for PIE", Proc. of the Logic Programming Conference '84, 7-2, Tokyo, March 1984
[7] Goldberg, A., and Robson, D., "Smalltalk-80: The Language and Its Implementation", Addison- Wesley, 1983
[8] Baker, H., "List Processing in Real Time on a Serial Computer", CACM pp280-294, April 1978
[9] Cohen, J., "Garbage Collection of Linked Data Structures", ACM Computing Surveys, Vol. 13, 3, Sept. 1981
[10] Arvind, and, Thomas, R. E., "I-structures : An efficient data type for function languages" Rep. LCS/TM-178, Lab. for Computer Science, MIT, June 1980
[11] 丸山 他, "高並列推論エンジンPIEの階層的構成とそのシミュレータ", 信学技報, EC 84-45, 1984
[12] 平田 他, "PIEにおける構造メモリの構成法", 第29回情報処理学会全国大会, 2B-3, 1984
[13] 平田 他, "A Note on Another Structure Sharing Method", 第30回情報処理学会全国大会, 3Q-6, 1985
[14] Warren, D. H. D., "Implementing Prolog -compiling logic programs. Vol. 1,2", D. A. I. Research Report, No. 39, 40, Univ. of Edinburgh, 1977
[15] Warren, D. H. D., "An Abstract Prolog Instruction Set", Technical Note 309, SRI AI Center, 1983