論理プログラムの並列実行について

梅山伸二
（電子技術総合研究所）

1 はじめに

Prolog等に代表される論理プログラム言語は、そのセマンティクスの明確さ等によって第5世代計算機研究の核言語としても採用され、現在大きな注目を集めつつある。また、現在までのところは、さまざまな効率上の制約によって逐次実行が行なわれているが、論理プログラム言語は本来一階述語論理式（のサブセットであるホーン式）に基づいた言語であるから、その実行順序には何の制限も無く、本来は完全な並列実行が可能である。将来、ハードウエア技術の進歩により、高性能なマルチプロセッサ・システムの構築が可能となるであろうことを考えると、その並列処理方式を検討することは重要なことであろう。

論理プログラム言語の並列処理を考えた場合、その方式には

(1) OR 並列
(2) AND並列
(3)ストリーム並列
(4)サーチ並列

の4種が考えられることがConery[1]によって示されている。ここでは、そのうちでも、逐次実行におけるバックトラック処理に替わる、OR 並列処理について考察し、その1つのモデルを与える。また、そのモデルに基づいてシミュレーションを行ない、ここで与えたモデルの有効性を示す。

2 論理プログラムの並列処理モデル

2.1 OR 並列処理

論理プログラム言語のOR並列処理を考えた場合、その動作形態から考えて、節を1つのプロセスと見る見方は自然であろう。（図1） つまり、あるプロセス（図1ではB）は親からの質問を受けると、その質問に対して答えるために子プロセス（C）を生成し、その子プロセスに対して質問を発する。子プロセスから解答が送られて来れば、プロセスBは、その解答に基づいて、さらに次のリテラル（g(*y)）に対応する子プロセス（D）を生成し質問を送る。全ての負リテラル（ゴール）がこのようにして解決できたならば、その結果を解答として、最初に質問を受けた親プロセス（A）へ返すことになる。

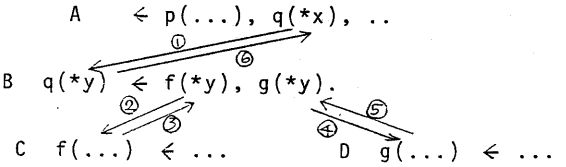


図1 論理プログラムの処理

論理プログラムのOR 並列処理は、このように多数のプロセス間のメッセージ交換によって実現することが可能であるが、このようなプロセス内部についても並列処理可能な部分は多い。例えば、図1のリテラルf(*y)に対して複数の解答が子プロセスから送られて来た場合（図2）それらの解答（θ_1, θ_2）を次のリテラルg(*y)に適用し、それらに基づいてg(*y)を解決することになるが、このθ_1, θ_2のg(*y)への適用は並列実行可能である。

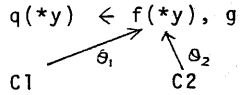


図2 OR並列処理

そこで1つのプロセスの動作を、更に単純な機能を持つ要素に分解し、それらの要素を並列に動作させることを考える。この場合、情報はリテラルを単位として

要素間を流れ（トークンと呼ぶ）、全体はデータフロー的に動作するものとする。こうすることにより、より高度な並列化が計れると同時に、単純な機能要素に分解することにより、その実現が単純，明快なものとなり、見通しが良くなる。

2.2 並列処理モデル

さて、このようにリテラルを1つのトークンと考えた場合、プロセスの動作を実現するためにはどのような機能が必要であろうか。

図3のようなプログラムの実行を見てみる。

[gp(aiko,*g)] という質問トークンは [gp(*x,*y)] と統一化（unification）され①、その結果 [gp(aiko,*g)]，[p(aiko,*z)]，[p(*z,*g)] という3個のトークンが生成される②。[p(aiko,*z)] は更に [p(aiko,kazuo)]，[p(aiko,yooko)] と統一化され③、*z←kazuo，*z←yooko という情報が [p(*z,*g)] へ送られることになる④。その結果、[p(kazuo,*g)]，[p(yooko,*g)] の2個のトークンが生成され⑤、前者は [p(kazuo,emiko)] と統一化されて⑥、*g←emiko という解答を得る。

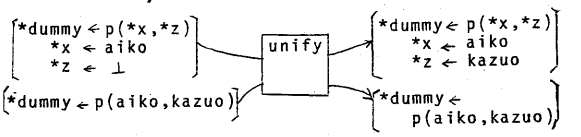
この例から判るように、プロセスの機能を実現するには、

(1) トークン同志を統一化する機構(①,③,⑥)。
(2) 統一化の結果を他のトークンに伝達する機構(④)。
(3) トークンを保存しておき、必要に応じて複製する機構(②,⑤)。

の3種の機構が必要である。

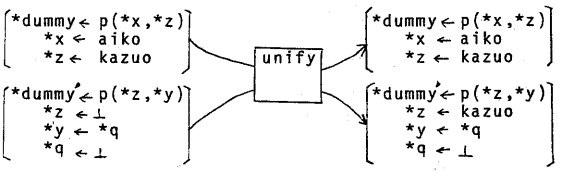
ところで、このうち(1)(2)の機構は、トークンを入力リテラルとその変数への代入例の集合で表現することにより、代入例の集合間の統一化として、共に実現できる[2]。

例えば、図3の③の場合、ダミーの変数 *dummy を用いて



とすることができる。ここで"⊥"は、値がまだ未定義であることを示している。

同様に図3の④の場合は、



とすることができる。

そこで我々のモデルにおいては、統一器（Unification Unit；UUと略記）と呼ぶ機能要素を、プロセスを実現するための基本要素として用いる。統一器は代入例集合間の統一化を行なう要素である。また、補助的な機能要素として、複製器（Copy Unit；CUと略記），マージ器

```
C1      ← gp(aiko,*q).
C2    gp(*x,*y)  ← p(*x,*z), p(*z,*y).
C3    p(aiko,kazuo).
C4    p(aiko,yooko).
C5    p(kazuo,emiko).
```

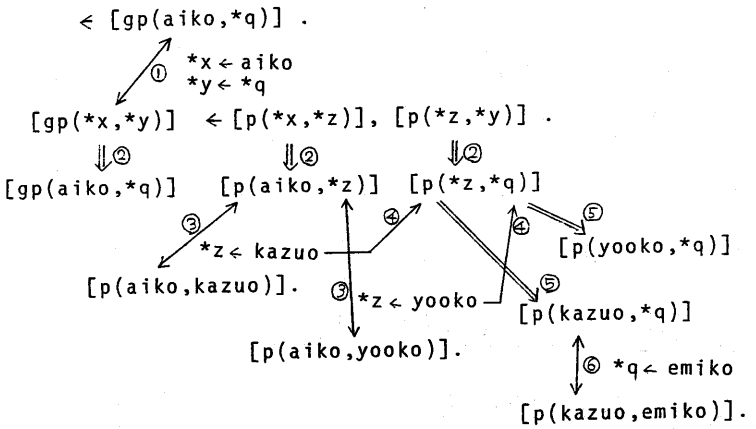


図3 論理プログラムの実行

(Merge Unit；MUと略記) と呼ぶ機能要素を用いる。以下それぞれの機能について簡単に説明する。

(1) 統一器

統一器は、本モデルにおいて中心的役割を持つ要素である。前掲の(1)(2)(3)の機能を実現しており、図4.(a)に示す図式で表現される。

図で判るように、入力に2本，出力に2本のポートを持ち、入力の2本をそれぞれトリガポート (trigger port) ストアポート (store port) と呼ぶ。またトリガポートに送られて来るトークンをトリガトークン (trigger token)，ストアポートに送られて来るトークンをストアトークン (store token) と呼ぶ。

統一器は入力トークンの種類によって動作が異なる。

I. ストアトークンの場合

そのトークンをそのまま統一器中に保存する。

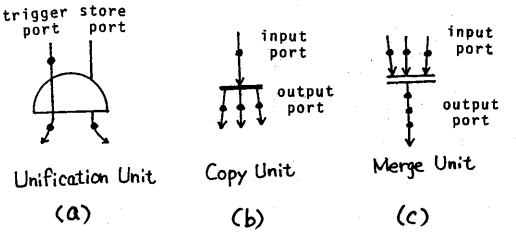


図4 機能要素

II. トリガトークンの場合

後述するように、トークンにはそれぞれ"タグ"が付けられている。統一器にトリガトークンが送られて来た場合には、統一器中に保存されているトークンのうち、タグのdestination, context (後述) が、トリガトークンのそれと一致するトークンが複製され、両者間の統一化が行なわれる。成功すれば両トークンは出力ポートより出力され、逆に統一化に失敗した場合には、この2個のトークンは消去される。

(2) 複製器

複製器は図4.(b)の図式で示される。入力ポートへ送られて来たトークンを出力ポートの数だけ複製し、出力する。

(3) マージ器

マージ器は図4.(c)の図式で示される。入力ポートへ送られて来たトークンは、送られて来た順に出力ポートへそのまま出力される。

以上の3種の機能要素を用いると、節 $p \leftarrow g_1, g_2, \cdots, g_N$ に対応するプロセスは図5のようなグラフとして表現される。ここで⊕⊖は入力節 (プログラム) のリテラルを表わすトークンであり、プログラムの実行の前に、ストアトークンとしてそれぞれの統一器中に保存されるものとする。

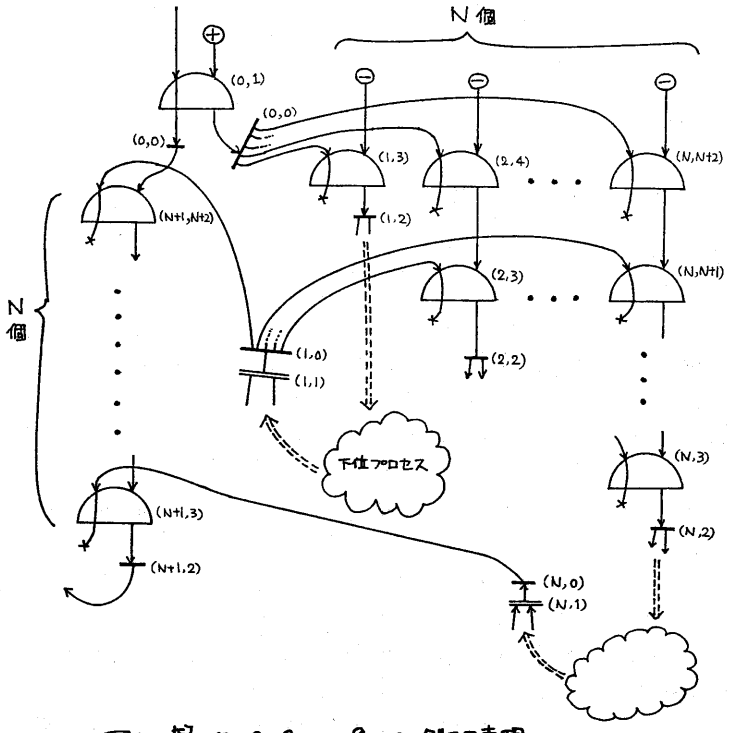


図5 節 $p \leftarrow g_1, g_2, \cdots, g_N$ のグラフ表現

2.3 例

前の例(図3)を使って動作を説明する。入力節C2,C3,C4,C5を図5のように表現、接続すると図6のようになる。時間を追って見ていくと、

① 質問トークンt1はUU1をトリガし、t2,t3の2個のトークンを生じる。

② t2はUU2にそのまま保存されるが、t3は複製器でコピーされ、UU4,UU5をトリガし、t4,t5を生成する。

③ t5はそのまま保存されるが、t4は質問トークンとして子プロセスへ送られる。

④ t4への解答としてt6,t7が子プロセスから送られて来るが、これらはマージ器でマージされ、複製器でコピーされ、UU2,UU6をトリガし、それぞれt8,t9,t10,t11を生じる。

⑤ t8,t10はUU3にストアされ、質問トークンとして子プロセスへ送られたt9,t11への解答を待つ。t11への解答は得られない(統一化できない)が、t9への解答はt12としてUU3へ送られて来てt8をトリガし、最終的な解答t13が得られる。*)

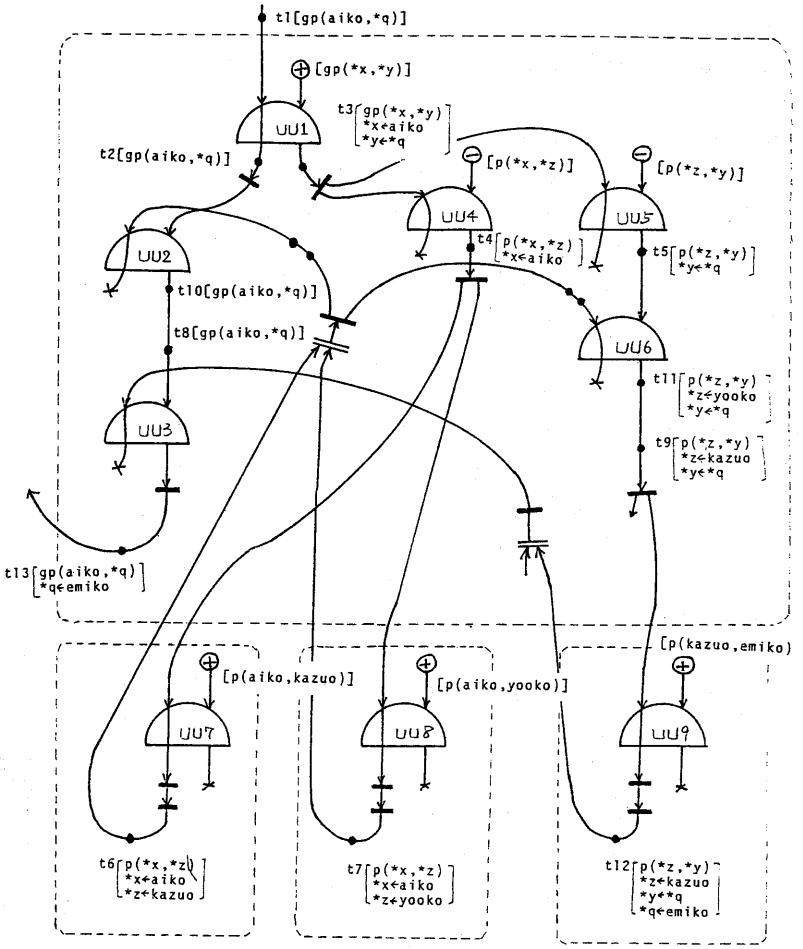


図6　図3の例の実行

3 トークンのタグ

前節のようにしてプロセスの動作を3種の機能要素およびそれらの間のトークンの流れとして実現できた。

しかし例えば、多数の親プロセスから質問トークンを受けた場合、あるいは子プロセスから複数の解答(OR並列で動作しているため)を受けた場合、多数の証明が同じグラフ上で同時に進行するわけであり、それらにかかわるトークン間の交通整理を適切に行なわなければならない。つまり統一器において、トリガトークンに対する適切な相手トークンを選択するための情報が必要である。また、

---

*) 統一器中に保存されているt2,t4,t5等のトークンは最終的にgarbageとなる。このgarbageの回収については4章に示す。

実際にこのようなグラフを結線するわけではないから、トークンの行き先情報もトークン中に持たねばならない。

そこで、それぞれのトークンに対して次のようなタグが付加される。タグはそれぞれのトークンについてユニークである。

token-tag = token-number<destination, context, return-unit, trigger-flag>

(1) token-number

それぞれのトークンに付けられるIDナンバー。

(2) destination

トークンの行き先であり、どのグラフ(ある節を表現する)内のどの要素にトークンが送られるかを示す。例えばappend(d1,d2)は、appendという節の(d1,d2)という2つ組で示された要素に送られることを示す。この2つ組は、図5のそれぞれの要素の右肩に付されている。

(3) context

グラフ上では複数の証明が同時に進行しているわけであり、contextはトークンがそのうちのどの証明にかかわっているかを示す。例えば[ct1,ct2]というように2つ組で示され、ct1,ct2はtoken-numberである。第1項(ct1)は、このトークンが親からのct1という質問トークンに対する証明にかかわっていることを示し、第2項(ct2)は、簡単に言えば子からのct2という解答(子からの複数解答のうちの1個)を用いた証明にかかわっていることを示す。

(4) return-unit

質問トークンを送る際には、それへの解答をどこへ返してほしいかをreturn-unitへ記しておく。return-unitは例えば、{append(d1,d2),[ct1,ct2]}のようにdestinationとcontextの2つ組である。

(5) trigger-flag

トークンがトリガトークンであるかストアトークンであるかを示す。

4 不要トークンの回収

2章の最後の脚注で触れたように、統一器に保存されているトークンは、それをトリガするトークン、つまり下位プロセスからの解答が尽きた際にはgarbageとなる。そこでこれらの不要トークンを回収する機構が必要であり、ここで失敗トークン(fail-token)、消去トークン(delete-token)の2種の制御用トークンを導入し、またマージ器の機能を拡張する。

プロセスのグラフ表現の1部を抜き出すと図7のようになる。質問トークンt1は下位プロセスへ送られ、それへの解答が次々とUU2,UU3,UU4中のトークン(t2,t3,t4)をトリガする。下位からの解答が尽きた場合には、これらのトークンはgarbageとなる。そこで下位プロセスはt1への解答を送り尽くすと、失敗トークン(t5,t6,t7)を送って来るものとする。MUは全ての下位プロセスから失敗トークンが送られて来た場合、消去トークンt8を生成し、CU2を介してUU2,UU3,UU4へ送る。この消去トークンにより、t2,t3,t4の不要トークンは回収される。

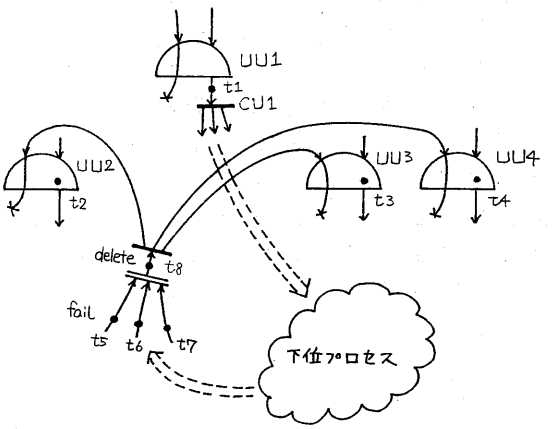


図7 不要トークンの回収

次に、質問に対する全ての解答を送り尽くした際に、失敗トークンを親プロセスへ送る機構が必要である(図8)。図8で、親プロセスから質問トークンXが送られて来てUU1で統一化が行なわれると、

(成功か否かにかかわらず) 失敗トークンが作られUU2へ送られる。この失敗トークンのct1(タグのcontextの第1項)はXであり、それが一致するトークンがUUi(2≤i≤N+1)に無ければ、失敗トークンは順々に下の統一器へ送られる。失敗トークンが一番下のUUN+1から出力された際には、UU2,…, UUN+1にはXをct1とするトークンは存在しない。つまりXに対する解答は尽くされたことが判る。

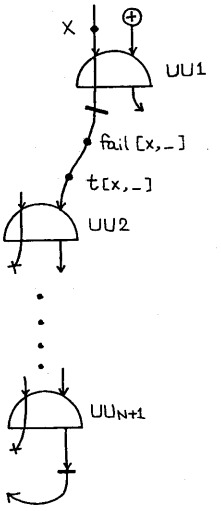


図8 fail-tokenの生成

5 シミュレーション

前章までに述べたモデルが、問題の持つ並列性をどれだけ有効に引き出しているかを見るために、簡単なシミュレーションを行なってみた。

シミュレートしたプログラムは図9に示すような4QUEENS問題である。使用されている述語のうちで、"add", "subt", "not_eq"はそれぞれ加算、減算、2引数の不一致を調べるための組込み述語である。また関数"c"は"cons"のつもりであり、リストを表現するために用いている。

```
←queens(c(1,c(2,c(3,c(4,nil)))),nil,*q).

queens(nil,*y,*y).
queens(c(*x1,*x2),*y,*z) ←select(*u,c(*x1,*x2),*v),safe(*u,*y,1),
                           queens(*v,c(*u,*y),*z).

select(*x,c(*x,*y),*y).
select(*u,c(*x,*y),c(*x,*v)) ←select(*u,*y,*v).

safe(*u,nil,*w).
safe(*u,c(*p,*q),*n) ←nodiag(*u,*p,*n),add(*n,1,*m),safe(*u,*q,*m).

nodiag(*u,*p,*n) ←add(*p,*n,*t1),subt(*p,*n,*t2),
                  not_eq(*u,*t1),not_eq(*u,*t2).
```
図9 4QUEENS問題

図10はシミュレートした全体の構成を示している。各ユニットはネットワークを介して接続され、ネットワークからの入り口には入力バッファを設ける。ユニットから出力されたトークンは、そのdestination, contextをキーとしてハッシュされ、適切なユニットへネットワークを介して送られる。CU,MU数はそれぞれ3個と固定し、UU数を変化させてマルチプロセッサ化の効果を見てみる。

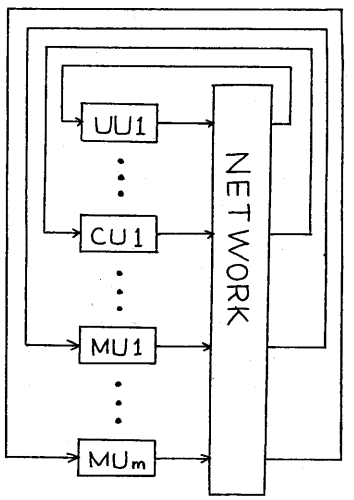


図10
全体の構成

シミュレーション結果を図11に示す。図11はUU数と処理時間の関係を見たものであり、両者の関係を明確にするために1/処理時間についてもプロットしてある。4QUEENSでは全体で1670個のトークンが生成され、(対称な)2つの解が得られる。図11を見て判るように、シミュレートした問題がかなり規模の小さな問題にもかかわらず、十分に並列性を引き出していると考えられる。

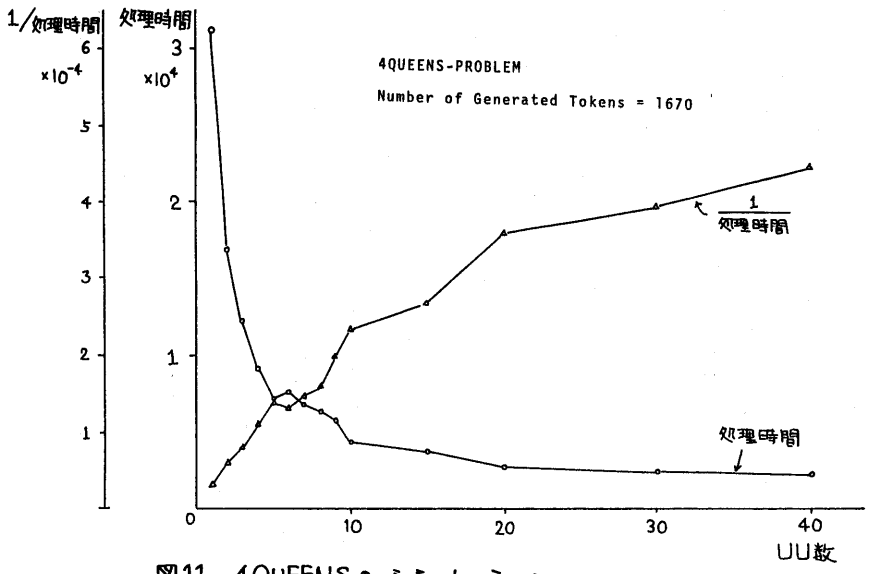


図11 4QUEENSのシミュレーション

それらは直ちにg以降のリテラルに適用され、gθ1, gθ2, … の処理が行なわれた。一方blocked executionでは、pへの解答のうち1個(θ1)がgへ適用されると、その他の解答(θ2, θ3…)はgへ適用することをブロックされる。そしてgθ1の処理のみが先行し、gθ1の処理が全て終了して(つまりgθ1への解答が全てr以降のリテラルに適用され終わって)初めて次の解答θ2のgへの適用が行なわれる。

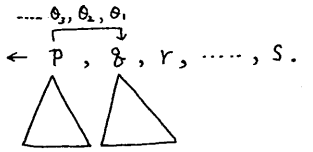


図12 部分解の次のリテラルへの適用

こうすることにより、全体としてはdepth-firstに近い制御が行なわれ、特に1個の答で十分な場合には、無駄なトークン生成の抑制に非常に効果的である。もちろんこの制御法によれば、無限ループに入る節が存在する場合、解が得られないことがあり、注意する必要がある。*)

blocked executionは前章までのモデルを次のように変更することにより実現で

6 実行の制御

今まで述べてきたモデルにおいては並列処理は野ばなしに行なわれ、処理が進むにつれてどんどん証明木の展開が行なわれる。このため証明木末端の処理に忙殺されて有効な処理が行なわれなくなる恐れがある。特に答として全ての答ではなく、唯1個の答を得れば十分であるような場合、例えば4QUEENSの1個の答のみを得たいような場合には、証明木を全て同時に探索することは無駄である。また無限ループに入るような節が存在した場合も問題であろう。

このような理由から証明木展開の制御が必要となる。この方法としては、トークンに優先度を付け、最終的な解答に近いトークンほど高い優先度を与えるような方法も考えられるが、ここでは証明木展開をもっと直接的に制限する方法を報告する。この方法を仮に"blocked execution"と呼ぶ。

図12のような状況を考える。pを解くとpへは複数の解答(θ1, θ2, …)が送られて来るが、今までの方法によれば

---

*) blocked executionは[1]のlazy evaluationと基本的には良く似ているが、最終的な親への解答の取り扱いが異なっている。lazy evaluationでは親への解答もブロックされるが、blocked executionでは親への解答はブロックされず、得られしだい直ちに親へ送られる。

きる。図13に示すように複製器の入口にそれぞれトークンキューを設ける。今、あるトークンt1がQ1より出力されたとすると、Q1はt1によりブロックされる。つまりt1と同じcontextを持つトークンはこのブロックが解除されるまでキューを出力されない。t1はCU1を通ってUU2をトリガし、質問トークンt2が下位プロセスへ送られる。下位プロセスは質問t2への解答をMU2へ送って来るが、最終的には全ての解答を送り終え、MU2からQ2へ消去トークンt3が送られる。このt3がQ2を出力されると同時に制御線（図の点線）を通じて解除トークン（unblock-token）がQ1へと送られ、Q1のt1によるブロックを解除する。

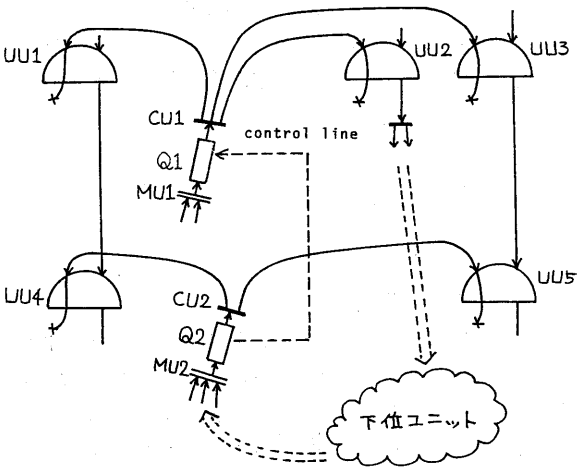


図13 blocked execution

5QUEENSの問題について、前章までのモデルと、blocked executionの両者でシミュレーションを行なった結果を表1に示す。5QUEENSでは全部で10個の解が得られるが、要した処理時間と共に、得られた解を順番に並べてある。従来のモデルは"eager execution"として示してある。

blocked executionは並列度が落ちるため、全ての解を求めるためにはより多くの処理時間が必要であるが、第1解は従来のモデルと比較して1/7程度の時間で求められており、blocked executionの効果を示している。

	Eager Execution	Blocked Execution
c(4,c(2,c(5,c(3,c(1,nil)))))	16945	2253
c(3,c(5,c(2,c(4,c(1,nil)))))	17251	4286
c(5,c(2,c(4,c(1,c(3,nil)))))	17281	8076
c(2,c(5,c(3,c(1,c(4,nil)))))	17521	11562
c(4,c(1,c(3,c(5,c(2,nil)))))	17561	13698
c(1,c(4,c(2,c(5,c(3,nil)))))	17791	16467
c(3,c(1,c(4,c(2,c(5,nil)))))	17824	18653
c(5,c(3,c(1,c(4,c(2,nil)))))	17931	22458
c(2,c(4,c(1,c(3,c(5,nil)))))	17941	26427
c(1,c(3,c(5,c(2,c(4,nil)))))	18081	28296

表1. 5QUEENS問題の処理時間

7 さいごに

論理プログラム言語のOR並列処理のための新しいモデルを提案し、またそのシミュレーションを行って有効性を確認した。今後このモデルに基づいて当研究室の論理型分散処理システムDIALOG-S [3]の開発を行なっていきたいと考えているが、細部については問題も多く、今活発に研究が行なわれているデータフローマシン等の成果も吸収して、改善してゆきたいと考えている。

参考文献

[1] J.S.Conery, et.al, "Parallel Interpretation of Logic Programs", Proc of the 1981 Conference on Functional Programming Languages and Computer Architecture.

[2] 梅山他 "分散処理システムDIALOG-Sの基本計算機構", 情処学会第23回全国大会, 1981.

[3] 田村他 "並行論理型プログラムの分散処理型アーキテクチャによる実現", 情処学会記号処理研究会 11-1, 1980.