Vol. 31 No. 6 情報処理学会論文誌 June 1990

並列 Lisp 言語 QLisp による OPS5 の並列処理の記述†

奥乃 博††* アヌープ・グプタ†††

プロダクション・システム（あるいはルールベース・システム）は広くエキスパート・システム構築用ツールとして使用されている．しかし，広範な応用あるいは実時間応用にはプロダクション・システムの高速化が不可欠であり，並列処理による高速化が研究されている．従来の研究では低レベル言語を使用しているため，他のシステムとの融合が難しかった．また，OPS5 はエキスパート・システム構築用ツールであり，その並列度はユーザ・プログラムに大きく依存している．本論文では，OPS5 を共有メモリ型マルチプロセッサ上で高級言語を用いて並列処理する上での様々な問題点について議論する．本論文の主要な主張点は（1）並列 Lisp 言語 QLisp による並列処理の記述，（2）実行時に並列度を決定する制御方式，（3）プロダクション・システムの競合解消と行動段階での新しい並列処理アルゴリズムの提案，である．とくに実行時並列度制御方式は，個々のタスクの処理時間にばらつきのある記号処理では，資源の有効利用には不可欠である．また，行動段階での並列処理は競合解消で選択される可能性のあるルールを先行実行するという概念に基づいている．高級言語を用いることにより多様な並列性は容易に記述することができた．

1. はじめに

プロダクション・システムは大規模なエキスパート・システムの構築に広く使用されてきている[12),17)]．しかし，プロダクション・システムの実行は遅く，このために実時間応用領域に適用するのが問題となっている．本論文では，プロダクション・システム一般を論ずるのではなく，特定のプロダクション・システム，OPS5，に注目する．というのは，OPS5 はエキスパート・システムを構築するのに広く使用されているし，また，その動作特性もよく解析されているからである．本論文では OPS5 を高速化する方法として共有メモリ型マルチプロセッサ上での並列処理を取り上げる．共有メモリ型マルチプロセッサをターゲットとしたのは，

1. ワーキング・メモリが変更された時に影響を受けるプロダクションの数が高々 30 と小さいこと．
2. 影響を受けるプロダクションの処理時間のばらつきが大きいこと．
3. 実行サイクル当たりのワーキング・メモリに対する変更が 2～3 と非常に小さいこと．

という観測による[4)]．

OPS5 の並列処理は，いくつかのグループで研究されてきている[4),5),10),14),16)]．これらのアプローチは，(1)OPS5 の実行に適した専用並列計算機の設計，(2)OPS5 コンパイラと実行時環境の並列計算機への移植，という 2 段階から構成されるのが一般的である．第 2 段階では，ハードウェアとオペレーティングシステムとに依存した機能を使用して OPS5 の並列処理プログラムを書くことが中心となる．いずれの研究でも，照合段階が全体の大部分の実行時間を占めることから照合の並列処理にだけ注目しており，実装にはハードウェアに近いレベルの言語を使用している．

本論文でのアプローチは，(1)並列 Lisp 言語 QLisp による並列処理の記述，(2)実行時に並列度を決定する制御方式，(3)プロダクション・システムの競合解消と行動段階での新しい並列処理アルゴリズムの提案，という 3 つの点で従来のとは異なっている．まず第 1 に，並列処理の設計とコーディングに高級並列言語，すなわち，並列 Lisp を使用している．並列 Lisp を用いる利点としては高い移植性，デバッグ時間の短縮，高い可読性，保守のしやすさを挙げることができる．また，プロダクション・システムを埋め込んだような AI システム―多くの場合 Lisp で書かれている―との整合性が容易になるという利点もある．並列 Lisp を用いる最大の欠点は，よりハードウェアレベルに近い言語（例えば，C）と比べて効率が低下することである．

並列処理によって Lisp プログラムの高速化を狙う並列 Lisp としては，例えば，Multilisp[7)~9)] や QLisp[3)] など，数多く提案されている．QLisp は Common

† Parallel Implementation of OPS5 in QLisp by HIROSHI G. OKUNO (NTT Software Laboratories) and ANOOP GUPTA (Department of Computer Science, Stanford University).
†† NTT ソフトウェア研究所
††† スタンフォード大学コンピュータ科学科コンピュータシステム研究所
* この研究の一部はスタンフォード大学コンピュータ科学科知識システム研究所に客員研究員として滞在した時に行われた．

Lisp[15] を並列処理の観点から再吟味した言語であり，様々な並列構文を提供している．他の並列(Lisp)言語にはない QLisp の主な機能は，(1)並列処理単位の実行時の制御のための基本構文と，(2)共有データあるいは大域データへのアクセスの排他制御用基本構文(ロックというデータ構造ではない)，の2点である．本研究の1つの目的は，文献 5) で将来の課題として掲げられている，高級並列言語での OPS5 の実装の相対的なメリットを探ることである．また，OPS5/QLisp は QLisp プロジェクトにとって最初の大きな(『現実的な』)応用プログラムであり，QLisp の記述力を実証するのにも貢献できる．

第2に，並列処理単位を実行時に決定していることである．一般に，並列処理においては，並列処理単位の大きさは設計時に決められ，実行時には固定されている．しかし，数値計算の場合と異なり，記号処理では部分タスクの実行時間が不均一であるため，並列処理単位の大きさを固定してしまうとプロセッサ利用率が下がり，高速化が達成できない[6]．例えば，各サイクルで変更されるワーキング・メモリから影響を受けるプロダクションの数はサイクルによって2桁程度と大きく異なるので，計算機資源を最大限活用するように並列度を指定することが重要な課題である．この並列度決定問題に対してあらかじめ各タスクの処理時間を見積って実行を始める前に(例えば，コンパイル時に)タスクの最適な分解を指定する方法が考えられる．しかし，ガーベッジ・コレクタのような実行時システムが介在する Lisp システムの場合には，タスクの処理時間を予測することが困難であり，上記の方法は現実的ではない．本論文では，並列処理単位の大きさを実行時に決定することによって，並列度決定問題を解決している．この実装には，実行時にプロセス生成の制御[13]が行える並列処理言語 QLisp を使用している．実行時並列度制御方式は，さらに，システムごとに最良の性能を引き出すことを可能にするので，より移植性を高めることにもなる．

本論文での OPS5 の並列化技法は，CMU のプロダクション・システム・マシン (PSM) プロジェクトで開発された並列照合アルゴリズム[5]に基づいている．PSM プロジェクトでは，粗い並列処理単位(『ルール・レベル』)から細かい並列処理単位(『ノード間』)へと並列度を上げることによってどの程度の速度向上が得られるかを研究している．本論文では，この様々な並列アルゴリズムを，実行時に並列度を制御する観点から再構成する．

第3に，並列照合だけではなく，競合解消およびプロダクション・ルールの実行部 (RHS) の並列アルゴリズムも提案する．照合が全実行時間の 90% を占めることから，従来は競合解消や RHS の実行は並列処理の対象とはなっていなかった．しかし，照合のみの並列処理では最大10倍の速度向上しか期待できない．なお，RHS の実行部の並列処理は，『先行実行』* という新しい考えかたに基づいている．

本論文の構成は次のようになっている．第2章で OPS5，Rete 照合アルゴリズム，および QLisp という本研究の背景について説明する．第3章では，QLisp を使用した OPS5 の並列アルゴリズムの提案とその実装について述べる．第4章で，考察と結論を述べる．

2. 背　景

2.1 OPS5 プロダクションシステム言語

OPS5 プロダクション・システム[1]は『プロダクション (production)』と呼ばれる *if-then* ルールの集合，および『ワーキング・メモリ (WM)』と呼ばれる assertion のデータベースから構成される．ワーキング・メモリ中の assertion は『ワーキング・メモリ要素 (WME)』と呼ばれる．プロダクションはルールの *if* 部に対応する素条件の連言 (conjunction) と，*then* 部に対応する行動の集合から構成される．プロダクションの *if* 部と *then* 部は，各々，『左辺式 (*LHS*)』と『右辺式 (*RHS*)』と呼ばれる．プロダクションで可能な行動はワーキング・メモリの追加，削除，および変更，あるいは，入出力である．**図1**に p1 と p2 という2つの簡単なプロダクションを示す．

プロダクション・システム・インタプリタは次に示す『認識・行動』サイクルを繰り返すことによって，

```
(p p1 (C1 ↑color ⟨x⟩ ↑size 12)
      (C2 ↑price 38 ↑color ⟨x⟩)
      (C3 ↑color ⟨x⟩)
      ⟶
      (remove 2) )
(p p2 (C2 ↑price 38 ↑color ⟨y⟩)
      (C4 ↑color ⟨y⟩)
      ⟶
      (modify 1 ↑price 50) )
```

図1 プロダクションの例
Fig. 1 Example of productions.

* 並列計算は，必要な計算を並列に実行する『必須計算 (mandatory computations)』と将来無駄になるかもしれない計算を並列に実行する『先行実行 (speculative computations)』に分類することができる[9]．

プロダクション・システム・プログラムを実行する．

- **照合**：すべてのプロダクションの LHS がワーキング・メモリの内容とマッチするかどうかを調べる．この照合の結果，照合に成功したすべてのプロダクションの『具体化 (instantiation)』を『競合集合 (conflict set)』に登録する．
- **競合解消**：競合集合から次に実行すべきプロダクション具体化に定められた戦略に従って１つ選択する．
- **行動**：競合解消で選択されたプロダクション具体化の RHS を実行する．

各ワーキング・メモリ要素は，要素の『クラス (class)』と呼ぶシンボルと『属性・値対』から構成されるリストである．ただし，属性・値対はなくてもよいし，いくつあってもよい．属性は“↑”の付いたシンボルである．値は定数シンボルか数である．LHS の素条件は，クラス名と１つ以上の項から構成される．各項は，“↑”の付いた属性と演算子と値から構成される．演算子が省略された場合には“＝”が指定されたことになる．値は定数か変数である．変数は，１対の“<”と“>”とで囲まれた名前で表現され，任意の値とマッチする．ルールの LHS 中の同じ変数が現れているところはすべて同じ値とマッチしなければならない．素条件はマッチの対象とするワーキング・メモリ要素のすべての属性・値対を含んでなくてもよい．素条件の前に“－”が付いておれば，否定素条件と呼ばれ，そのマッチが成功するのは否定を除いた素条件にマッチするワーキング・メモリ要素がない時だけである．

2.2 Rete 照合アルゴリズム

様々な OPS5 プログラムを解析することによって，『時間的冗長性』と『構造的類似性』という２つの重要な特徴が明らかになっている[2]．時間的冗長性とは，ワーキング・メモリはルールの発火によってごく一部だけが変更され，その他の大部分は変更されないという事実を指す．構造的類似性とは，すべてのプロダクションは全く異なっているのではなく，異なったプロダクションの条件式間で類似性が少なからずあるという事実を指す．Rete 照合アルゴリズムは，この２つの特徴を活用してインタプリタの照合段階での高速化を達成している．Rete 照合アルゴリズムは，プロダクションの LHS からコンパイルされたデータフローネットワークを使用して，照合を行う．図１で示した２つのプロダクションに対するネットワークを**図２**に示す．この図では，ノード間の線が情報の流れる道を示している．

ノード間を流れるデータは『トークン』と呼ばれる．トークンは，『タグ』とワーキング・メモリ要素の順序付きリストで構成される．タグは“＋”か“－”であり，前者は『追加』，後者は『削除』を意味する．

Rete アルゴリズムで使用されるデータフローネットワークには４種類のノードがある．

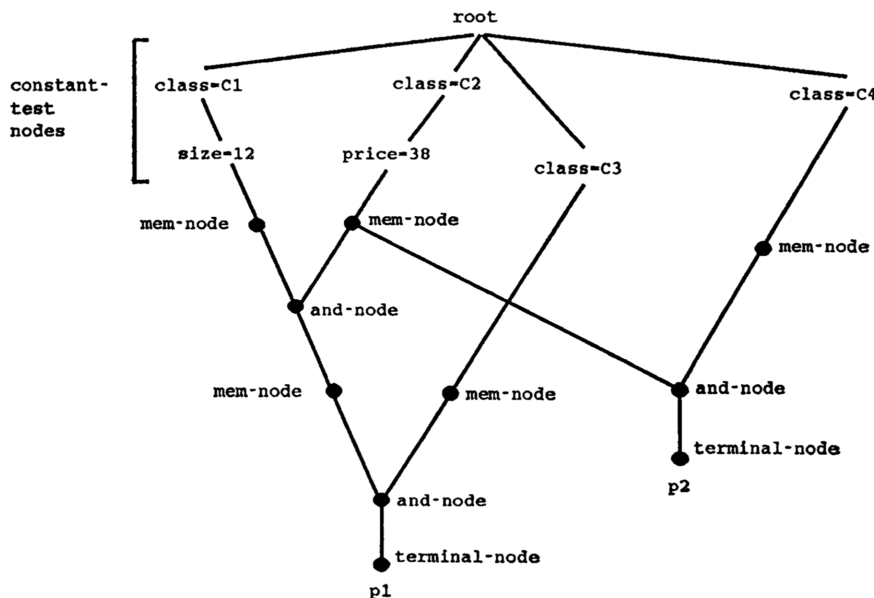

図 2 Rete ネットワーク
Fig. 2 Rete network.

1. **定数テスト・ノード**：素条件中の定数値を持つ属性が満足されるかを調べるのに使用される．
2. **メモリ・ノード**：以前のサイクルにおける照合段階での結果を状態として格納しておくノード．状態を保存することによって同じテストを繰り返し行うことを避ける．メモリ・ノードの状態は関連するプロダクションの LHS の部分とマッチするトークンのリストである．
3. **2元入力ノード**：プロダクションの LHS 中の素条件の連言テストを行うのに使用される．2元入力ノードには，『and ノード』と『not ノード』の2種類がある．2元入力ノードの入力はいずれもメモリ・ノードである．入ってきたトークンは他の入力に接続されたメモリ・ノードと比較される．無矛盾な変数の値を持つようなトークンの対は，2元入力ノードの後続ノードに送られる．
4. **終端ノード**：終端ノードに到達したトークンはタグに従って対応するプロダクション具体化が競合集合に追加されるか，あるいは，削除される．

Rete ネットワークを並列処理する上で注意しなければならないのは，トークンが入ってきたのとは反対側のメモリ・ノードは2元入力ノードの処理中は変更があってはいけないという性質である．本稿ではこの性質を『2元入力ノード処理の逐次性』と呼ぶ．具体例で説明する．今，トークン t_1 の処理中にもう一方のメモリ・ノードに別のトークン t_2 が登録され，かつ，t_1 と t_2 がマッチして新しいトークン t_3 が作成されるとしよう．もし，2元入力ノード処理の逐次性が破られると，次にトークン t_2 に対して再び2元入力ノードが処理され，同じトークン t_3 がもう1つ生成されることになる．OPS5 では重複トークンがあると，競合解消での意味が異なるので，正しい答えを保証できなくなる．

2.3 並列 Lisp 言語—QLisp

QLisp は Common Lisp を並列処理に拡張したプログラミング言語[3]であり，Alliant FX/8 共有メモリ型マルチプロセッサ上に処理系が開発中である．QLisp では他の並列処理言語とは異なり，プロセスの制御やロック機能等の並列処理の基本演算が言語機能に組み込まれている．生成されたプロセスは，システム・キューに登録され，スケジューラがプロセスをキューから取り出してプロセッサに渡し実行を行う．QLisp の基本機能を以下にまとめる．

QLET "(qlet 〈述語〉〈変数リスト〉・〈本体〉)" フォームは変数リストを並列に評価する構文である．この〈述語〉は **qlet 述語**と呼ばれ，プロセスを生成するかどうかを決定する．すなわち，qlet 述語が eager 以外の非 NIL であれば，〈変数リスト〉の値の計算を並列に行い，すべての変数に値が代入されたら〈本体〉の実行を再開する．qlet 述語が eager であれば，〈変数リスト〉の実行と〈本体〉の実行を並列に行い，もし，〈本体〉でまだ値の求まっていない変数を参照した場合には〈本体〉の実行は中断される．qlet 述語が NIL であれば，qlet は let と同じである．

QLAMBDA "(qlambda 〈述語〉〈Lambda リスト〉・〈本体〉)" フォームは，lambda がクロージャを作成するように，〈述語〉が非 NIL であれば『プロセス・クロージャ』を作成する．NIL の時は，qlambda は lambda と同じである．プロセス・クロージャはプロセス間の変数の共有だけでなく，共有変数へのアクセスの制御を行う．すなわち，同一プロセス・クロージャは何時の時点でも1つしか実行されず，排他制御が自動的に行われる．プロセスをスポーンするだけであれば qlambda の変形である spawn を使用する．

QWAIT "(qwait 〈フォーム〉)" フォームは，指定されたフォームの実行中に生成されたプロセスがすべて終了するまで，本体の実行が中断する．プロセスの同期を取る方法には，この qwait のほかに qlet がある．本処理系では，こまかい同期が取りにくいのでこの関数は使用しない．

3. OPS5 の並列処理

本章では，OPS5 インタプリタが繰り返す『照合・競合解消・実行』というサイクルの各段階での並列化について述べる．

3.1 照合段階での並列処理

3.1.1 Rete ネットワークのノード実行

定数テスト・ノードの実行（『定数テスト』実行と呼ぶ）は単純なテストだけでよいので実行のコストは極めて安く，並列処理に伴うオーバヘッドのほうが並列処理の効果よりも大きいので，並列処理は行わない．したがって，後続ノードが定数テスト・ノードであれば，同じプロセス内で逐次的に処理をする．さもなければ，新たなプロセスを生成して後続ノードの処理を行う．このような評価戦略のコードを**図 3** に示す．

```
(defun match (token root-node)
  (spawn (ctest-match token (successor root-node))) )
(defun ctest-match (token node-list)
  (dolist (node node-list)
    (if (ctest-node? node)
        (if (do-ctest token node)
            (ctest-match token (successor node)) )
        (spawn (eval-node token node)) )))
(defun eval-node (token node)
  (if (funcall (body node)
               token (arguments node) )
      (eval-node-list token (successor node)) ))
(defun eval-node-list (token node-list)
  (if node-list
      (let ((node (pop node-list)))
        (qlet (if (check-system-status-by-strategy)
                  t
                  'eager)
              ((foo (eval-node token node))
               (bar (eval-node-list token node-list)
              ))))))
```

図 3 Rete ネットワークのノードを並列実行する QLisp コード
Fig. 3 QLisp code to evaluate Rete network in parallel.

2番目の種類のノード実行は『2元入力ノード・グループ』の実行である．本論文では，2元入力ノードとそれに付随するメモリ・ノードを一まとまりのものとして取り扱い，それらを『2元入力ノード・グループ』と呼ぶ．というのは，これらのノードは，相互に関連しているだけでなく，前述した Rete アルゴリズムの2元入力ノード処理の逐次性により，2元入力ノードの処理中はトークンが入ってきた入力と反対側の入力は変化してはいけないからである．これらのノード・グループの処理は軽くはないので，別のプロセスを生成して実行する．2元入力ノード・グループの実行においては，その実行順序に注意しなければならない．これは，あるトークンがメモリ・ノードに付加され，次に，同じトークンが削除されるという『共役トークン (conjugate token)』[6] のためである．共役トークンに対しては最終的にはメモリ・ノードは全く変化しない．しかし，並列処理では，スケジューラが付加要求より前に，削除要求を取り上げるということが生じ，このような処理を正しく扱わないとメモリ・ノードに余分なトークンが残ることになる．このために，各メモリ・ノードは『過剰削除リスト』[5] を持っており，まだ到着していないトークンを削除しようとした時には，そこに削除要求を登録しておく．

最後に，終端ノードでは，トークン（プロダクション具体化）を競合集合へ付加したり，競合集合から削除したりする．ここでも，同様に，共役トークンが出現しうる．また，競合集合は大域的データであるから，複数のプロセスが同時に変更しないように，終端ノードの直前にロック・ノードを挿入し，Rete ネットワークを並列処理用に変更する．

3.1.2 Rete ネットワークの並列処理

本論文で使用する並列アルゴリズムは，Rete ネットワークの2元入力ノード（異なっていてもよいし，同じノードでもよい）を並列処理するノード間並列 (intranode parallelism)[5),13)] を基本にしている．複数のトークンが同じ2元入力ノードに到着すると，そのノードの処理が複数のプロセスで並列に処理される．このような並列処理で注意しなければならないのは，2元入力ノードの入力となるメモリ・ノードに対するアクセスの排他制御である．メモリ・ノードのロックは(1)複数のプロセスが同時にメモリ・ノードを変更してはならないだけではなく，(2)2元入力ノード処理の逐次性，を満足しなければならない．本論文では，Gupta[5] が提案した手法を使用し，上記のような場合にも最大の並列度が得られるようにしている．すなわち，トークンは Rete ネットワークのメモリ・ノードではなく，大域的なハッシュ表に格納する．トークンはハッシュ表のバケツに格納し（鍵が競合しても再ハッシュしない．），鍵は2元入力ノードとテストしている値とから計算する．バケツへのアクセスの排他制御を行うためのロックの実装は何種類か可能であるが，本論文では QLisp の記述性を調べるために陽にロックを使用するのではなく qlambda を使用している．**図 4** にハッシュ表の構造を示す．2元入力ノード・グループの片側から入ってきたトークンを処理する時には，ハッシュ表からバケツを引き，それがロックされていない場合に限り，まずそのトークンをバケツに格納し，反対側に格納されているトークンと比較を行う．また，それぞれのメモリ・ノードは共役トークンのために過剰削除リストも持っている．ロックされている時には，ロックが空くまでプロセスの実行は中断される．

ロック	左側メモリ・ノード		右側メモリ・ノード	
	トークン・リスト	過剰削除リスト	トークン・リスト	過剰削除リスト
バ ケ ツ				

図 4 メモリ・ノード用ハッシュ表
Fig. 4 Hash table for memory nodes.

複数のトークンが同一の2元入力ノードに与えられてもそれらのハッシュ鍵が同じになる確率は極めて小さいので，ハッシュ表へのアクセスがボトルネックにならず，その結果，同一2元入力ノードに対する処理を同時に複数個実行することができる．

上述したようなトークン管理方法では，逐次処理用の Rete ネットワークを並列処理用に特別の変更を加える必要はない．

3.1.3 実行時並列度制御

よく知られているように，並列処理単位が小さければ並列度が大きくなり，逆に，並列処理単位が大きくなれば並列度は小さくなる．Lisp のような関数型言語ではタスクを小さな部分タスクに分解し，並列実行させることは容易であるが*，プロセス生成，プロセス切り替え，メモリ競合といったオーバヘッドにより，並列処理の効果が無に帰すことがありうる．

例えば，再帰的に定義されたフィボナッチ関数 *fib*(n) で，部分計算 *fib*(i)，$i<n$ がすべて別プロセスで実行されるとしよう．この時生成されるプロセスの数は，n が 10，20，25 の時，それぞれ 176，21890，242784 となる．QLisp シミュレータの結果によると，プロセスの数が多くなると並列処理の効果が低下する．そこで，qlet によって，再帰呼び出しの深さがある数（例えば，10）以上の時には，新たなプロセスを生成しないようにすると並列処理の効果がプロセッサの台数に比例する．

したがって，並列処理の効果が最大となるようにタスクを最適分解することが並列処理の成功の鍵である．本論文では，以下に述べる理由によりタスクを階層的に分解し，上位のレベルを優先させながら実行時に並列処理単位を決定する方法を採用している．

2元入力ノード・グループと終端ノードの実行が並列処理の対象となるが，システムの負荷によって，新たなプロセスを生成して並列度を高めるかどうかの判定を行っている．その戦略をまとめると次のようになる．ただし，トークンが伝播しない時には，なにもせず終了する．（図3参照）

- **ctest 実行終了時の判定**：ctest 実行は，後続の2元入力ノード・グループあるいは終端ノードを実行するための別のプロセスを起動して終了する．これは，spawn を用いて実装している．
- **2元入力ノード・グループ実行終了時の判定**：後続ノードが終端ノードの時には，終端ノードを処理する別のプロセスを起動して終了する．後続ノードが2元入力ノード・グループである時には，次のような戦略に従って後続ノードの処理を行う：『活性化されたプロセスの個数がプロセッサの個数を越えない時一つまり，システムの資源が緊迫していない時一には，新たなプロセスを起動する．資源が緊迫している時には，現プロセスが親からスポーンされたものである時に限って，別のプロセスを起動する．』この戦略の目的は，プロセスを作成する時にネットワークの上位にあるノード・グループをより優先することにある．

* もちろん，副作用がある場合はそれほど簡単ではない．

実行時並列度制御方式が持つ具体的な意味を，具体的なハッシュ関数を与えることによって検討してみよう．今，ハッシュ関数が2元入力ノードにのみ依存すると仮定する．「ctest 実行の直後のメモリ・ノードに対して新たなプロセスを生成し，それに後続するノードを同一のプロセスで実行する」という戦略による処理は，ルール単位あるいはルールの集合単位で並列処理する『プロダクション・レベル並列』[5] と同じである．また，「すべての2元入力ノードを独立のプロセスとして実行する」という戦略による処理は，各2元ノードが高々1個しか並列処理されない『ノード・レベル並列』[5] と同じである．つまり，プログラムを実行時並列度制御方式で記述しておき，システムの負荷を考慮しない固定した戦略として qlet 述語を与えると，『プロダクション・レベル並列』あるいは『ノード・レベル並列』を実現することができる．このように，実行時並列度処理方式を使用すると，上記の2つの Rete ネットワークを使用する OPS5 の並列アルゴリズムを統一的に表現することができる．

競合解消段階の説明に入る前に，照合段階終了の検出方法について述べておく．メインプログラムの流れは，（1）行動段階ですべての行動が完了すると，（2）次に照合段階が終了するのを待ち，（3）競合解消段階で最優先要素を求め，（4）最優先要素の行動を実行する，というサイクルを繰り返す．なお，照合段階は陽に起動されるのではなく，行動によってワーキング・メモリが変更されると照合が開始される．

照合段階では生成されたプロセスはすべて大域変数に登録され，この変数を通じてメインプログラムは照合段階が終了したことを検出する．すなわち，各プロセスの実行が完了すれば，照合段階が終了したことになる．プロセスの完了は，そのプロセスの状態を見ることで検査している*．なお，トークンが競合集合に

登録される場合にはそのプロセスは競合集合への登録プロセスを生成して実行を完了する．また，登録プロセスはこの大域変数には登録されない．

3.2 競合解消での並列処理

競合解消段階では，競合集合中に含まれるプロダクション具体化の中から次に実行すべきものを1つ選択する．この選択の方法は競合解消戦略と呼ばれ，OPS5ではLEX（辞書式-Lexical）とMEA（手段効用解析-Means-Ends Analysis）の2種類が提供されている．しかし，これら2つの戦略はプロダクション具体化を順序付ける方法が異なるだけであり，競合集合は競合解消によって順序付けられたリストへと変換される．

競合解消を並列に行う時の要求条件としては：

- プロダクション具体化の共役対を正しく処理すること．すなわち，具体化の削除要求がその追加要求よりも先に到着した場合の対処をすること．
- 最も優先度の高い具体化が，残りのものが完全に順序付けられていない時でも，求まっていること．この結果，最も優先度の高い具体化が何時の時点でもRHSを評価するプロセスに利用可能となる．

この要求条件を満足する解として，本論文では競合集合を非同期的シストリック・キュー[11]で表現し，これをQLispで実現している．プロダクション具体化の追加・削除要求は，この優先順位付きキューの先頭に入力され，鍵の比較を行いながら，キューの所望の位置まで送られていく．削除要求は，対応する具体化があれば，それを削除して終了する．もしなければ―すなわち，共役トークン問題―所望の位置に特別のフラグを立て，対応する追加要求が来るのを待つ．本アルゴリズムの重要なポイントは，最優先でない要素がまだ順序付けが終了していなくても，最も優先度の高い具体化が常にキューの先頭に置かれているということである．前節で述べた終端ノードの処理において，プログクション具体化がこのシストリック・キューに与えられる．

関連するコードを**図5**に示す．大域変数 *conflict-set-lock* は，qlambdaによるプロセス・クロージャを保持しており，競合集合を保持する *conflict-set* という変数の排他的アクセス制御を行う．register-csが値を返すとロックが解放される．

```
(proclaim
  (special *conflict-set-lock* *conflict-set*))
(defun ops-init ()
  (setq *conflict-set-lock*
    (qlambda t (name data key flag)
      (register-cs name data key flag) )))
(defun insertcs (name data rating)
  (funcall *conflict-set-lock*
      name data
      (cons (sort-time-tag data) rating)
      t ))
(defun removecs (name data rating)
  (funcall *conflict-set-lock*
      name data
      (cons (sort-time-tag data) rating)
      nil ))
(defun register-cs (name data key flag)
  (if *conflict-set*
    (sort-conflict-set
       name data key flag *conflict-set* )
    (setq *conflict-set*
      (create-new-cs-element
           key nil name data flag ))))
```

図5 競合集合のロック
Fig. 5 Locking for conflict-set.

上述の方法で最大要素を計算する時間は $O(k)$ である．ただし，k はOPS5インタプリタのサイクル当たりの競合集合の変更回数である．多くの応用では，k はおおよそ5であるので，この時間は全く問題とはならない．しかし，順序付けを終了するのに要する時間ははるかに長く，競合集合中の全要素数を N とすると，$O[N\times k]$ である．この方法は，全要素を順序付けするには最適ではないが，最優先度要素を得るには，最善である．最優先度要素は，次節で述べるRHSの先行実行の対象となる．

競合解消段階の終了は，最優先度要素であるプロセス・クロージャに対するキューの長さが0で，かつ，そのプロセス・クロージャの状態が実行完了である時である．なお，プロセス・クロージャは排他制御を行うので，それ自身がキューを持っている．ここで，競合解消段階の終了は競合集合の完全なソートが終了したことを意味しないことに注意しておく．

3.3 RHSの先行実行

ルールベース・システムでは，通常，競合解消が完全に終了してから最優先度要素のRHSの実行を始める．しかし，並列処理系では，このような方法は余りにも逐次的である．本論文では，照合と競合解消が進行中でも，最優先度のプロダクション具体化を推測し（現在の処理系では，このような要素は競合集合の先

---

* QLispプロジェクトでは，現在，著者らの要請を受け，qwaitとは別にこのような小まわりの利く終了検査関数を検討中である．

頭にある.), その RHS の簡単化を始めるという RHS の先行実行を提案する. 簡単化では, ワーキング・メモリを変更する手前の処理まで, すなわち, ワーキング・メモリへの変更のためのトークン作成までを行う. もし, 最優先度の要素が変更され, 推測が間違ったとしても, 別のプロセスを生成して新しいプロダクション具体化の簡単化を行うだけであり, 既に行った簡単化を元に戻したり, 破棄したりする必要はない. というのは, 簡単化は具体化中に指定された局所的な値しか使用していないので, 時間に依存せず, いつの時点でも有効であるだけでなく, さらにワーキング・メモリに対して何ら悪影響を及ぼさないからである. さらに, 簡単化を行ったプロダクション具体化が後で最優先度を持ち, その簡単化が役立つ可能性もある.

OPS5/QLisp システムはさらに, 副作用なし関数呼び出しという新たな行動コマンド sfcall を提供している. これは, ユーザ定義の Lisp あるいは QLisp 関数を呼び出す. この時, システムは sfcall で呼び出される関数が一切副作用を及ぼさないと仮定しているので, いかなる時点でこのような関数を実行してもシステムに対して悪影響を及ぼすことはない.

先行実行の効果は OPS5 のプログラムに依存する. いくつかの大きなプログラムの測定によると, ワーキング・メモリ要素変更する以外の行動は大体 30% 程度である[5]. もし, 副作用のない外部関数が多数呼ばれている場合には, 先行実行の効果は大きくなると期待できる. OPS5 の並列処理研究においてベンチマークとして使用されるプログラムは Rete ネットワークの有効性をチェックするのが目的であるので, 何れも外部関数の使用は少ない. しかし, OPS5 プログラムの解説書 (例えば, 文献 1)) では,『速度向上の一手法として外部関数の使用を進めている』ので, 実際の応用プログラムでは先行実行の効果が高いと期待できる.

OPS5 インタプリタの簡単化アルゴリズムを示す.

1. 行動がワーキング・メモリ演算であるならば, すべての引数を計算し, トークンを作成する. すなわち,

   make : 追加トークンを作成し, もとの行動を置き換える.

   remove : 削除トークンを作成し, もとの行動を置き換える.

   modify : 削除トークンと追加トークンを作成し, もとの行動を2つのトークンで置き換える.

   ただし, 行動が入出力といった副作用を及ぼす関数を含んでいる場合には, これらの関数は実行しないで, 行動には指定されていない属性・値対を補って行動の完全な記述を作成し, もとの行動と置き換える.
2. 行動が副作用なし呼び出し sfcall であれば, 指定された関数を実行する.
3. 上記以外は, 行動に対して何らの変更もしない.

簡単化ルーティンが起動されるのは, 競合集合 (シストリック・キュー) の先頭にあるプロダクション具体化が変更された時であり, 簡単化された RHS はそのプロダクション具体化中に登録される.

照合が終了し, 競合集合の最優先度の要素が決まった時点で, 競合集合の先頭要素の RHS が実行される. もし, それがすでに簡単化されておれば, ワーキング・メモリに対して実際に変更を加え始め, それと同時に照合段階が始まる.

4. おわりに

本論文では, QLisp を用いた OPS5 の並列処理の詳細について論じた. 本研究のまとめと今後の研究の展開を以下に記す.

- 並列化に要した OPS5 処理系の逐次処理プログラムの変更は極めて少ない. このことから, 高級言語による並列化のアプローチは並列処理研究において強力で柔軟なツールを提供することが確認できた.
- 個々のタスク (ノードの処理) は, 平均すると 1 MIPS 計算機で 1 ms 程度に時間がかかると予想している. この予想からプロセス生成, ロック競合, スケジューリングといった並列処理に伴うオーバヘッドは 1 ms よりかなり小さくなければ本処理系は有効ではない.
- QLisp 処理系はサブセットしか実現していないので, 本処理系は現在 QLisp シミュレータの上でしか走らない. QLisp シミュレータ CSIM[18]* によるシミュレーションの結果, 以下のような並列処理による性能向上が確認されている. しかしシミュレータは使用しているハードウェアを正確には反映していないので, オーバヘッドの見積りは得

* CSIM では qlambda が提供されていないが, MultiLisp の future は提供されている. したがって, シミュレーションでは qlambda を future とロック関係の関数を使用して実現した. また, qlambda によって生成されるプロセス・クロージャに付随したキューはロックに付随したキューで代用している.

られていない．並列プロセッサ上での実行評価については稿を改めて報告する．

プログラム	性能向上		
	1台	4台	8台
ルービック・キューブパズル（70ルール）	1.0	3.2	6.2

- 本論文で提案した並列度を実行時に制御する方式は単に1つの処理系上で『プロセスの組み合わせ的爆発を防ぎ，プロセスの生成の判断を行う時点での資源を有効利用する』という意味での最適な実行環境を求めるだけではない．同じプログラムをオーバヘッドが異なる様々な処理系にコードを適合させることが極めて容易になる．また，同じ並列計算機上でも，ロードによって並列度を調整するのにも使用できる．
- OPS5をQLispで実装する別の利点として，OPS5をLispで作成された他のAIシステムに埋め込むことが容易であることが挙げられる．さらに，プロダクションのRHSに使用される複雑な関数をQLispで書くことによって，ユーザ・レベルでも並列処理を追求することができる．このような機能は他のOPS5の並列実装では不可能であった．
- 既存のOPS5システム高速化の究極の手段として，OPS5プログラムをインタプリタで走らすのではなく，QLispに直接コンパイルすることを計画している．

謝辞 スタンフォード大学のE. A. Feigenbaum教授は著者が同大学KSLに滞在する機会を与え，本研究を支援してくださり，また，J. McCarthy教授は著者が同大学滞在中にQLispを使用する便宜を図ってくださったので感謝します．また，本稿に貴重なコメントをくださったNTTソフトウェア研究所の平田圭二博士に感謝します．なお，本研究を進めるに当たってKSLに対するDARPA Contract F30602-85-C-0012, QLispプロジェクトに対するDARPA Conract N 00039-84-C-0211から研究資金援助を受けた計算機を一部使用した．

参考文献

1) Brownston, L. et al.: *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley (1985).
2) Forgy, C. L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artif. Intell.*, Vol. 19, No. 1, pp. 17-37 (1982).
3) Gabriel, R. P. and McCarthy, J.: Queue-based Multiprocessor Lisp, *Conf. Record of 1984 ACM Symp. on Lisp and FP*, pp. 25-44 (1984).
4) Gupta, A. et al.: Parallel Algorithms and Architectures for Production Systems, *Proc. of 13th Int. Symp. on Computer Architecture*, pp. 28-37 (1986).
5) Gupta, A.: *Parallelism in Prudoction Systems*, Morgan Kaufman Publishers, Inc. (1987).
6) Gupta, A. and Tucker, A.: Exploiting Variable Grain Parallelism at Runtime, *Proc. of the ACM/SIGPLAN PPERLS*, pp. 212-221 (1988).
7) Halstead, R.: MultiLisp, *Conf. Record of 1984 ACM Symp. on Lisp and Functional Programming* (1984).
8) Halstead, R.: Multilisp: A Language for Concurrent Symbolic Computation, *ACM TOPLAS*, Vol. 7, No. 4, pp. 501-538 (1985).
9) Halstead, R.: Parallel Symbolic Computing, *IEEE Comput.*, Vol. 19, No. 8, pp. 35-43 (1986).
10) Hillyer, B. K. and Shaw, D. E.: Execution of OPS5 Production Systems on a Massively Parallel Machine, *J. of Parallel and Distributed Computing*, Vol. 3, pp. 236-268 (1986).
11) Leiserson, C. E.: Systolic Priority Queues, *Conf. on Very Large Scale Integration Architecture, Design, Fabrication*, pp. 199-214 (1979).
12) McDermott, J.: R1: A Rule-Based Configurer of Computer Systems, *Artif. Intell.*, Vol. 19, No. 1, pp. 39-88 (1982).
13) Okuno, H. G. and Gupta, A.: Parallel Execution of OPS5 in QLISP, Technical Report STAN-CS-87-1166, CSD, Stanford University (1987). also numbered Report KSL-87-43.
14) Ramnarayan, R. et al.: PESA-1: A Parallel Architecture for OPS5 Production Systems, *HICSS-19*, pp. 201-205 (1986).
15) Steele, G. L.: *COMMON LISP: The Language*, Digital Press (1984).
16) Stolfo, S. J. and Miranker, D. P.: The DADO Production System Machine, *J. of Parallel and Distributed Computing*, Vol. 3, pp. 269-296 (1986).
17) Vesonder, G. T. et at.: ACE: An Expert System for Telephone Cable Maintenance, *IJCAI*, pp. 116-121 (1983).
18) Weening, J. S.: A Parallel Lisp Simulator, Technical Report STAN-CS-88-1206, CSD, Stanford University (1988).

(平成元年5月16日受付)
(平成2年4月17日採録)

奥乃 博（正会員）

1950 年生．1972 年東京大学教養学部基礎科学科卒業．同年電電公社武蔵野電気通信研究所入所．1986〜1988 年スタンフォード大学コンピュータ科学科知識システム研究所客員研究員．Lisp，データベース，論理型プログラミング，プログラミング環境の研究を経て，現在は AI の並列処理の研究に従事．現在，NTT ソフトウェア研究所ソフトウェア開発環境研究グループ・リーダ．人工知能学会，日本認知科学会，ACM，AAAI 各会員．本学会欧文誌編集委員，人工知能学会編集委員．

アヌープ・グプタ

1958 年生．1980 年インド工科大学電気工学科卒業．1985 年カーネギー・メロン大学大学院コンピュータ科学科 Ph. D. 同年同大学同学科研究員．1987 年スタンフォード大学コンピュータ科学科助教授，現在に至る．プロダクション・システムの並列処理研究を経て，並列プロセッサおよびそのソフトウェアの研究に従事．現在，DASH マルチプロセッサ・プロジェクトを同学科コンピュータ・システム研究所にて主宰．著書「Parallelism in Production Systems」．IEEE，ACM，AAAI 各会員．