プログラミング言語 22－3
（１９８９．１０．５）

ストリームに基づいた並列意味処理の記述

西山博泰(*)，板野肯三(**)
(*) 筑波大学大学院工学研究科
(**)筑波大学電子・情報工学系

言語処理系における意味処理をストリームに基づいて記述する方式とその実現，および評価について述べる．ハードウェア技術の進歩と共に，並列処理を行なう計算機の構築が容易になってきている．従来の言語処理系のアルゴリズムは一部を除いて逐次処理を対象としたものであり，並列処理環境での実行は想定されていない．そこで，意味解析処理を並列に行なうモデルとしてストリームで結合されたプロセスの相互作用を考える．意味解析を行なうプロセスは解析木のノードに対して生成され，それらは双方向または一方向のストリームで接続される．ここでは，中間コード生成，記号表参照の記述例と共に並列意味解析器の記述法を与え，並列意味解析器プロセスを構成するプロセス，及びストリームの実現を議論する．本論文で提案した方式に基づいたプロトタイプ・システムにより，PL/0コンパイラの意味解析部を記述した結果，５台のプロセッサで各々のプロセッサに実行待ちが生じないことが確認された．

Stream Based Description of Parallel Semantic Analysis

Hiroyasu Nishiyama(*), Kozo Itano(**)

(*)The Doctoreal Program in Engineering,
Graduate School, University of Tsukuba
(**)Institute of Information Sciences and
Electronics, University of Tsukuba

Univ. of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, Japan

This paper presents a parallel algorithm for the semantic analysis of a language processor with its implementation and performance evaluation. Ordinary sequential algorithms for language processing can not be applied to the parallel processing environment. To cope with this problem, we proposed a parallel execution model of semantic analysis viewed as reciprocal actions of processes connected with streams. Each process created for the semantic analysis corresponds to a node of a parse tree, and communicates each other by single- or bi-directional streams. We also proposed a description method of parallel semantic analysis with examples as intermediate code generation and symbol table manipulation. We implemented a PL/0 compiler of which semantic analysis was described by the proposed approach. The performance evaluation revealed that negligible waiting time was detected for the parallelism with 5 parallel processors.

1．はじめに

ハードウェア技術の急速な進歩によってメモリやプロセッサ等の資源が豊富に利用可能になり，マルチプロセッサシステムや専用プロセッサの開発が可能となっている．しかしながら，これらを利用する並列処理アルゴリズムについては，数値計算等の特殊な領域以外では未だ研究の途上にあるというのが現状である[1]．殊に，言語処理系のような基本的な道具は，プログラムの効率的な開発という観点から見て重要であるにもかかわらず，一部を除いて並列処理技術への対応は行なわれていない．

言語処理系の代表的存在であるコンパイラでは，通常，字句解析，構文解析，意味解析，コード生成の４つのフェーズ，もしくは，最適化フェーズを加えた５つのフェーズ，と記号表から成るものが一般的である．これらのフェーズは比較的独立性が高く，容易にパイプライン処理による並列処理を行なうことができるが[2-8]，パイプライン処理による並列化だけではそれほどの効率の向上は望めない．特に，意味解析部について考えてみると，意味解析処理は各々の言語の特質に深く結び付いた処理であり，その複雑さも他のフェーズの処理とは比較にならない．このため意味解析部に関してはより汎用的な並列処理の手法を考える必要がある．

コンパイラの意味解析部やインタープリタ等の言語に対する意味処理の記述を行なう場合，その言語の文法の各非終端記号の生成規則に対して意味処理を行なう動作ルーチンと呼ばれる手続きを与える方法や，各非終端記号に属性と呼ばれる値とそれらの関係を表す関数を定義するという方法が用いられる．前者は，手書きのコンパイラや，YACC[10]のような生成系で用いられる方法であり，構文解析法の解析順序等に従って意味処理を行なう手続きが呼ばれ，それぞれの意味処理間の情報の受渡しは基本的に大域的な変数を用いた副作用を介して行なわれる．一方，後者は属性文法[11]に基づいたシステムで用いられる方法であり，属性の依存関係に従って解析の順序を決定し，それに従って各々の属性の値を決定することで意味解析処理が行なわれる．

意味処理を並列に行なう場合には，YACCのように大域的な変数を使用した副作用によって情報の交換を行なう方法では，実行順序に依存した制御を行なう必要があることから，プログラミングの複雑さが増加し，またシステムの保守を難しくする原因ともなる．一方，属性文法に基づくシステムでは，属性の依存関係の持つ非決定性により並列性を抽出することができるが，意味処理の記述を行なう際に中間コードや記号表のような比較的大きなデータも１つの属性として扱う必要があることから，これらのデータの処理に内在する並列性を自然に記述することは難しい．また，動作ルーチンによる方法では，並列性の抽出や並列動作するプログラムの記述における問題点が存在するのに対して，属性文法に基づく方法ではダイナミックな並列性の抽出やインタープリタ等を実現する場合に動的な意味の記述が難しいという問題点がある．

そこで，これらの問題点を解決するため，ここでは，並列プログラムの実行環境としてメッセージ通信型[9]のモデルと共有メモリ型のモデルを組み合わせたものを考え，意味処理をストリーム[12]によって結合された複数のプロセスの相互として考えるモデルを提案する．本方式では，属性文法と同様に文法の各非終端記号にストリームとその値を計算するプロセスを定義する．ただし，属性文法の場合と異なり，ストリームをデータ構造の基本とすることで動的な意味の記述や，並列性の抽出を行なうことが容易となっている．以下，２節では意味解析処理の概要について，３節では本方式での意味処理の記述方に関して，４節では実現の方式について，５節ではプロトタイプ・システムでの性能評価に関して議論を行なう．

2．意味の並列解析モデル

ここでは，我々が仮定するコンパイラの構成を示し，次に意味の並列解析のモデルを与える．

2．1 コンパイラの構成

ここではコンパイラの構成として字句解析，構文解析，意味解析，コード生成を行なうモジュールがそれぞれパイプライン的に結合され，同時に処理を行なうことを前提とする．構文解析部にはＬＲ構文解析を想定し，構文解析部は移動，還元といった構文解析動作に関する情報を意味解析部に対して出力するものとする．また，字句解析部から受け取った終端記号の属性値をこの際付加する場合もある．意味解析器ではこれらの情報を受け取り，意味解析プロセスの生成とその実行を行なう．同様に，コード生成部を除いた構成でインタープリタを構築することや，意味解析処理を行なうモジュールに最適化処理を行なうモジュールを付加することも可能である．

2．2 解析モデル

言語処理系の中心的な処理である意味処理を並列化するために，意味処理を動的に複数の”意味解析プロセス”に分解し，これをストリームで結合して並列に処理を行なうモデルを提案する．

各意味解析プロセスは，構文解析の結果得られる解析木の特定の節に対して生成し，意味解析プロセス間は双方向または一方向のストリームによって結合する．ただし，構文解析部と意味解析部は平行に動作することが可能であり，必ずしも実際の解析木を作る必要はない．プロセス間を結ぶストリームには一方向へ値を転送する入力ストリーム，出力ストリームと，この２つを組み合わせた入出力ストリームが存在する．双方

向ストリームはデータ転送の向きが逆の一方向ストリームを組み合わせたものと等価であり，記述性の向上のために設けてる．

各意味解析プロセスには，対応する出力ストリームが１つ存在する．プロセスの生成処理は，構文解析部で対応する文法の還元動作が行なわれると同時に行なわれ，生成されたプロセスは生成と同時に実行を開始する．プロセスは対応する解析木の親，兄弟，子の節に属したプロセスの計算する複数のストリームを入力として取り，これを用いて出力ストリームの値を計算する．プロセスはストリームへの出力する全データの計算が終了した時点でその実行を終了し消滅する．

この方式では，基本的には属性文法と同様に，あるプロセスの入力ストリームの値を決定することによって，そのプロセスが生成するストリームの値が決定されるため，実行順序に依存しない記述を行なうことができる．また，ストリームをプロセス間の通信の手段とすることで，部分的な値の読み出しを可能にし，これによって並列性の抽出が容易となっている．さらに，ストリームの要素の値が自身の既に決定されている要素の値に依存することを許すことを利用して動的意味の記述をある程度自然に行なう事が可能になる．

３．記述法

ここでは，並列意味処理の記述法を与え，典型的な意味処理のいくつかについて，本方式を用いた意味解析処理の記述を与えその処理方式に関して説明を行なう．

３．１ 記述形式

並列意味処理の記述は目的とする言語の構文規則と共に与え，生成系によって，構文解析部へのデータ，意味解析プロセス生成のためのデータと，各意味解析プロセスに共通した意味解析プロセス用のプログラム・データを生成する．このために用いる構文の概要を以下に示す．

```
＜意味処理記述＞ → ＜宣言＞ %%＜文法定義＞%%
＜文法定義＞ →   ＜文法＞ ＜意味規則＞
                { ＜文法＞ ＜意味規則＞ }
＜文法＞ → ＜非終端記号＞
             ':' { ＜(非)終端記号＞ }
           { '|' { ＜(非)終端記号＞ } }
＜意味規則＞ → '(' { ＜ストリーム定義＞ } ')'
＜ストリーム定義＞ →
           ＜ストリーム＞ '=' ＜プロセス＞
＜ストリーム定義＞ →
           ＜ストリーム＞ '=' ＜ストリーム＞
＜プロセス＞ → [seq] ＜プロセス名＞
             '(' ＜ストリーム＞
                 { ',' ＜ストリーム＞ } ')'
＜ストリーム＞ →
           ＜(非)終端記号＞'.'＜ストリーム名＞
```

ここで，[] は省略可能，{} は０回以上の繰り返しを表す．また，宣言の定義については省略した．

意味解析プロセスは，並列プロセスと逐次プロセスとに分けられる．上の記述では，seqで指定したものが逐次プロセス，それ以外が並列プロセスとなる．後に述べるように逐次プロセスの指定を行なうことによって，ストリームの参照が制限された形ではあるが，仮想的な複数のプロセスを実際には１つのプロセスとして実行することが可能になる．

ここで，プロセスの実行の実体として指定する関数は，現在Ｃ言語を用いて記述することとなっているが，将来的にはリストまたは，ストリームをデータ構造の基本とした，より記述力の高い言語をこのために開発することを考えている．

３．２ ストリームに基づいた中間コード生成

ここでは，コンパイラにおける中間コードの生成の並列意味解析器による記述例を示し，この記述の中で最も重要なストリーム操作 append_streamについて説明する．プロセスに対応する関数の定義は本来 C言語で与えるが，以下では簡単のため仮想的なプログラミング言語により行なう．以下，{}はストリーム定数，+はストリームの先頭への要素の追加，head，tailはそれぞれストリームの先頭の要素と残りを返す関数，end_of_streamはストリームの終りかどうかを確かめる関数を表している．

一般に，中間コードの生成を行なう部分では，節の部分木の中間コードを適当に結合し，それを節の中間コードとするような処理が多い．例えば，while文の中間コード生成を行なう記述は次のようになる．

```
while_stmt : WHILE cond DO stmt
        { while_stmt.code
            = append_stream( GOTO  cond,
                             LABEL body,
                             stmt.code,
                             LABEL cond,
                             cond.code,
                             IF_TRUE body);
        }
```

ここで append_streamは引数のストリームを順に結合したストリームを生成する関数である．このように，中間コードの生成部分を副作用を排除した形で記述する場合には，append_streamの行なう処理は本質的に重要であり，その実現はコンパイラ全体の性能に大きな影響を与える．この append_streamの実現には，引数を生成しているプロセスと参照をする側のプロセスと

の間で同期を行なうかどうかによって同期と非同期の２種類の方法を用いることができる．同期を行なう場合の処理はリストの結合処理と本質的に同様である．ただし，同期による実現での

```
append_stream( stream1, stream2)
```

の実行を考えてみると，ストリームを有限長のバッファで実現している場合には stream2を生成しているプロセスは stream1のバッファがいっぱいの間は実行がブロックされてしまう．このため，次のような非同期な append_streamの実現を考える．以下，stream.idはストリーム streamの一意な識別子を与え，データの組を与える表記である．

```
append_stream( stream1, stream2) =
        〈 stream1.id, stream2.id〉
        + append_stream2( stream1, stream2)

append_stream2( stream1, stream2) =
        if( end_of_stream( stream1))
                copy_stream( stream2)
        else if( end_of_stream( stream2))
                  copy_stream( stream1)
        else if( not empty_stream( stream1))
                〈 stream1.id, head( stream1)〉
                +append_stream2( tail( stream1),
                                 stream2)
        else if( not empty_stream( stream2))
                〈 stream2.id, head( stream2)〉
                +append_stream2( stream1,
                                 tail( stream2))
        else
                append_stream2( stream1,
                                stream2)

copy_stream( stream) =
        if( end_of_stream( stream))
                []
        else
                〈 stream.id, head( stream)〉
                +copy_stream( tail( stream))
```

まず，append_streamでは引数として受け取った２つのストリームの識別子を組にし，ストリームの最初の要素とする．次に，append_stream2を呼びだして stream1と stream2の要素にそれぞれのストリームの識別子を付加したストリームを作る．copy_streamは引数のストリームと同じ要素を持つストリームを作る関数である．このように，ストリームの前後関係を始めに渡してやり，各中間コードにストリームの一意な識別子を付属してやることで．受け取り側では，この情報をもとにストリームの要素を同期による実現の場合と同様な一意な順序に並べ直すことを可能にしている．３つ以上の引数を持つ場合の，append_streamの処理も同様に考えることができる．

非同期な実現では，同期による実現に比べて並列性の抽出を行なうことが容易となるが，一方で，ストリームが具現化されているかどうかを確かめる関数を導入することによりプロセスの入力ストリームの値が同一ならプロセスの出力ストリームの値も同一になるという関係は破壊されてしまう．また，同期による方法と比較すると余分な情報を受け渡す必要がある．

３．３ 記号表参照

記号表の実現では，各意味解析プロセスに記号表のコピーをストリームを通じて情報を必要とする全プロセスに分配するという方法も可能であるが，ここでは，本システムの記述力を示すために，記号表を管理するプロセスを設け，必要な意味解析プロセスが記号表参照の要求を管理プロセスに送り，その結果を管理プロセスが要求したプロセスへ送るという方式での実現例を考える．

```
Proc :  Dcl Stmt
        { Stmt.result = symtab( Dcl.names,
                                Stmt.demand);
        }
Dcl  :  Var
        { Dcl.names = { Var.name};
        }
Dcl⁰ :  Dcl¹ Var
        { Dcl⁰.names = Dcl¹.names + { Var.name};
        }

symtab( names, demand) =
        (dem.pid, search( names, dem.name))
                + symtab( names, tail( demand))
        WHERE
                dem is head( demand)
```

symtabは引数として，ブロックで定義された識別子とその情報を保持しているストリーム namesと，記号表参照の要求を受け付けるためのストリーム demandを持つ．記号表参照を行なうプロセスは，プロセスの一意識別子と参照のための情報を ストリーム demandを介して記号表管理プロセス symtabに渡す．symtabではストリーム demandによって渡された識別子をもとに，手続き searchによって記号表の検索を行ない，参照元のプロセスの一意識別子と検索した識別子に関する情報を組にしてストリームに出力する．ここで，demは head(demand)の省略記法として定義している．

4．実現

並列意味解析器の実行は，２つの部分に分けて行なわれる．制御プロセスと呼ばれるプロセスは，構文解析部から構文解析情報を受け取り，それに従って実際の意味処理を行なう意味解析プロセスを生成する．意味解析プロセスは制御プロセスと独立に実行を行ない，それらは以下に述べるように実現されている．

4．1 制御プロセス

制御プロセスは，構文解析部から入力テキストの構文解析を行なった結果を受け取り，実際の意味解析処理を行なうプロセスの生成を行なう．意味解析プロセスの生成は次のようなアルゴリズムで行なわれる．

```
Step1:
  構文解析動作を読み込む;
  if( 構文解析動作 == 還元) goto Step2;
  else goto Step1;

Step2:
  if( 対応するプロセス == 逐次プロセス)
                                goto Step3;
  else if( ( プロセス == ストリームの代入) &&
           ( 代入側のストリーム == 既出))
    {
      非代入側のストリーム名を
      代入側のストリームidと同一にする;
      goto Step1;
    }
  else
    {
      プロセスを生成;
      goto Step1;
    }

Step3:
  if( 次の還元動作に対応するプロセス
                        == 並列プロセス)
    {
      逐次プロセスを生成;
      goto Step1;
    }
  else if( 入力キューの長さ )= α)
    { /* ここで，αは一定の値 */
      逐次プロセスを生成;
      goto Step1;
    }
  else
    goto Step1;
```

ここで，Step2での処理は，

＜ストリーム定義＞ →
　　＜ストリーム＞ '=' ＜ストリーム＞

で定義されるストリーム間の代入で，代入の右辺に現れるストリームが，対応する解析木の親あるいは左側の兄弟の節に属するストリームである場合，不要なストリームのコピーを行なうプロセスの生成を行なわず，分散プロセス内の実行でのストリームの同一化で置き換える．これによって，不必要なコピーを行なうプロセスの除去を行なうことができる．

また，Step3はグレイン・サイズが大きくなる可能性のある逐次プロセスを複数の逐次プロセスに分割する処理を行なうもので，ここで入力キューは構文解析部から受け取った構文解析動作が格納されているキューを表している．入力キューの長さが一定以上の逐次プロセスに関しては，これを複数の逐次プロセスへ分割する．

4．2 意味解析プロセス

並列プロセスを実行する場合には，プロセス間通信のコストやプロセス切り変えのオーバーヘッド等の問題から，概念的には並列実行を行なうことの可能なプロセスについても，それを複数のプロセスではなく１つのプロセスとして実行する方が実行の効率が良い場合がある．このため，意味解析プロセスには，複数の仮想的なプロセスを１つにまとめて実行する逐次プロセスと，その他の独立に実行される並列プロセスとが存在する．

逐次プロセスの実行結果は，それを複数のプロセスで実行した場合と同様であるが，逐次プロセスとして実行を行なう場合には，仮想プロセス間の情報の受渡しはスタックを介して行なわれるため，プロセス間通信やプロセス切り変えに要する手間が不要になる．ただし，あるプロセスを逐次プロセスとして指定する場合には，仮想プロセス間のデータの受渡しにスタックを用いることから，ある節に属したストリーム定義の代入の右辺に現れるストリームが，親あるいは，左側の兄弟の節に属したストリームでなくてはならないという制限がある．

意味処理の記述を行なう際に，どのプロセスを逐次プロセスとするかについては，生成系によって自動的に判別されることが理想であるが，意味解析処理の記述の静的な情報から，実行時の負荷を予測するのは困難であるため，現在の実現では，利用者が明示的に逐次プロセスを指定するという方式を取っている．

4．3 ストリームの実現

意味解析プロセス間での情報の受渡しは，ストリームを介して行なうが，逐次プロセスと並列プロセスでのストリームの実現は異なる．ここでは，これらのストリームの実現について説明する．

（１）並列プロセスにおけるストリームの実現

並列プロセス間の情報の受渡しにはストリームが用いられる．意味解析プロセスが，ストリームへの書き込みおよび読み出しを行なう際に，プロセスの同期が行なわれる．ストリームは固定長のバッファから構成され，プロセスがストリームからの読み出しを行なう際には，そのプロセスは他のプロセスがストリームへデータを書き込むまで実行を中断する．一方，ストリームへの書き込みを行なうプロセスは，ストリームのバッファがいっぱいの時にはバッファに空きができるまで実行を中断し，その後，書き込みを行なう．

（２）逐次プロセスにおけるストリームの実現

逐次プロセスでは，ストリームはリストとして処理が行なわれる．逐次プロセス内ではデータの受渡しはストリームではなくスタックを用いて行なわれる．スタック中には，データを格納するフィールドの他に，データの存在を示すフラグとそのデータを計算しているプロセスの idを格納するためのフィールドが設けられており，プロセスがスタック中のデータを必要とした時点でそのデータの計算が終了していなければ，データを計算しているプロセスの実行を待ち，その後実行を再開する．

（３）インタフェース

逐次プロセスと並列プロセスで情報のやり取りを行なう場合には，ストリームとリストの間でデータの変換を行なう．プロセス間の同期はデータを生成する側のプロセスの終了を待ち，データの変換を行なうという手順をとる．

５．評価

本論文で提案した並列意味解析器における並列処理の可能性を確かめるため，PL/0[13]を対象言語とした簡単なコンパイラを並列意味解析器で記述したプロトタイプシステムの開発を行なった．ここで各意味解析プロセスの実現には SunOS4.0の Light Weight Process (LWP)[14]を用い，実験環境としては，Sun4/110を使用した．このプロトタイプシステムでは，構文解析までの処理と，制御プロセスでの処理は実現の都合から意味解析プロセスを実行する前の段階で行なっている．LWPによって仮想的なプロセッサのシミュレートを行うために，プロセッサ毎にスケジューリング用のキューを設け，一定時間おきにスケジューラによってスケジューリング・キューの切り替えを行う．ある仮想プロセッサに属したプロセスがストリームでの同期等でブロックしたり，プロセスが実行を終了した場合には，同一スケジューリング・キューに属している実行可能状態にあるプロセスが１つ選択され実行される．

このコンパイラに対して簡単なPL/0プログラムを入力とし，５台のプロセッサで実行した場合の各仮想プロセッサにおけるプロセスの実行状態の変化を図１に示す．このグラフで，横軸は仮想プロセッサのスケジューリングの単位を表し，縦軸はプロセスの数を表している．Executableは実行可能なプロセス，Waiting LWPは双方向ストリームの入力待ちのプロセスを示し，Waiting Monitor及び，Waiting CVは一方向ストリームでの通信待ちのプロセスを表している．このグラフで，実行開始後しばらくは各プロセッサとも，実行可能状態にあるプロセスが増加している．この部分では，各意味解析プロセスが実行を開始し，プロセスの初期化を行っているものと思われる．続いて，ほとんどのプロセスはストリームに対する通信待ちの状態に入る．以後各プロセッサでは，通信待ちのプロセスがほとんどを占めているが，実行可能なプロセスが各プロセッサに対してほぼ均等に割り振られ，各プロセッサに対して実行可能状態にあるプロセスが概ね存在している．この結果から，プロセッサ５台の時は実行可能なプロセスがうまく割り振られ，並列性の抽出がうまくいっていることが判る．

今回実現したPL/0コンパイラは，物理的な実行環境として共有メモリ型の密結合プロセッサによるシステムに相当するが，厳密な評価を行うには各プロセッサによるメモリ・アクセスの競合や，プロセスのスケジューリングによって生じるオーバーヘッドを考慮する必要がある．今後，このような点を考慮したより詳細なシミュレータの実現や，並列マシン上での実現を行う必要がある．また，今回の実験は５台のプロセッサからなるシステムを想定しているが，並列意味解析器によって生成されるプロセスの数とその実行時の性質はコンパイラの入力とされるプログラムに強く依存する．このため，特定の言語に対して効率のよいプロセッサ台数を知るためには，複数のプログラムに対して評価を行い平均的な値を得る必要がある．

６．おわりに

ストリームに基づいた並列意味解析のモデルとその実現の方式を提案した．ここで試作したプロトタイプシステムは単純な言語に適用しただけであるが，意味解析を動的に複数のプロセスに分解して実行することに成功した．現在，本稿で述べたシステムは開発途上であり，さらに検討を加える必要がある部分も存在する．例えば，プロセス管理の手間や手法，また，プロセスのグレイン・サイズをどのようにコントロールするかについては，システムの性能を決定する重要な要素であり，実際にマルチプロセッサ上での実現を行なう過程で充分考慮を払う必要がある．また，動的な意味の記述に関してもgoto文や手続き呼び出しを記述する際に，今回の枠組では不自然かつ煩雑な記述となる可能性があり，ストリームの要素としてのストリームの導入などによるシステムの拡張も今後の課題である．

参考文献

1)Kung,H.T., Why Systolic Architecture?, Computer, Vol.15, No.1, pp.37-46,(1982).

2)西山博泰, ウン・チョン・セン, 板野肯三：ハードウェア・コンパイラの設計, 情報処理学会第３６回全国大会, pp.851-852,(1988).

3)西山博泰, 板野肯三：コンパイラにおける意味処理の並列化, 第３０回プログラミングシンポジウム, pp.123-134,(1989).

4)西山博泰, 板野肯三：ハードウェア・コンパイラにおける並列意味解析器の構成, 情報処理学会第３８回全国大会, pp.909-910,(1989).

5)Aho,A.V., Sethi,R. and Ullman,J.D.: Compilers - Principles, Techiniques, and Tools, Addison-Wesley,(1986).

6)板野肯三, 佐藤豊, 山形朝義：パイプライン型字句解析プロセッサの設計と実現, 情報処理学会論文誌, Vol.28,No.1,pp.82-90,(1987).

7)Itano,K., Sato,Y., Hirai,H. and Yamagata,T.: An Incremental Pattern Matching Algorithm for the Pipelined Lexical Scanner, Information Processing Letters, Vol.27, No.5, pp.253-258, (1988).

8)ウン・チョン・セン, 西山博泰, 板野肯三：ハードウェアＬＲパーサの構成, 情報処理学会第３６回全国大会, pp.853-854,(1988).

9)Hoare,C.A.R.: Commuicating Sequential Processes, Comm. ACM, Vol.21, No.8, pp.547-557, (1974).

10)Johnson,S.C.: Yacc - yet another compiler compiler, Computing Science Techinical Report 32, AT&T Bell Laboratories, Murry Hill, N.J., (1975).

11)Knuth,D.E.: Semantics of context-free languages, Mathematical Systems Theory 2:2, pp.127-145,(1968).

12)Henderson,P.: Functional Programming - Application and Implementation, Prentice Hall, (1980).

13)Wirth, N.: Algorithms + Data Structures = Programs, Prenitice Hall,(1976).

14)Sun Micro Systems: SunOS Reference Manual, Sun Micro Systems,(1988).

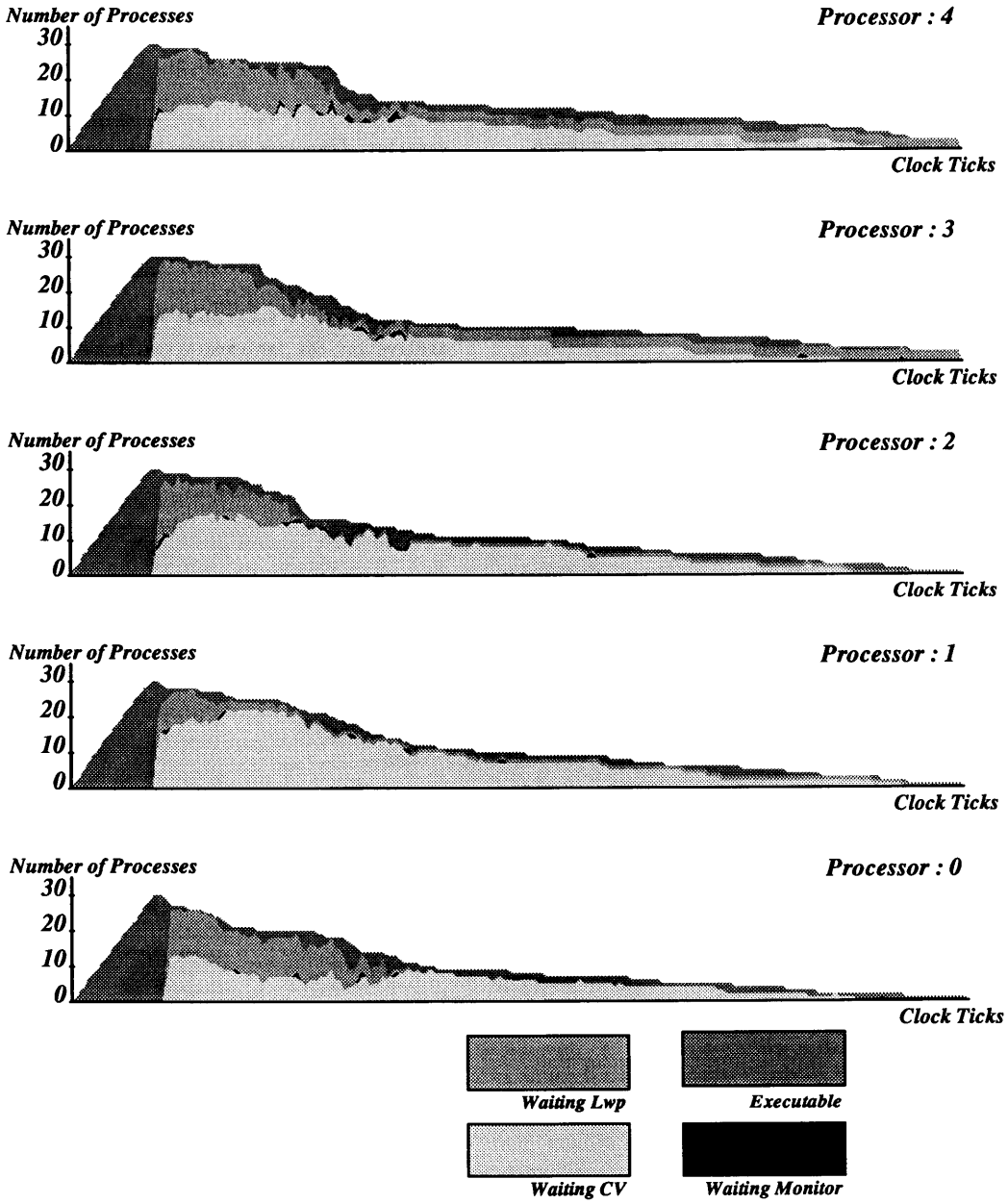

図１．並列意味解析器の実行の様子