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†††

応答性とプログラムの高速性を両立させた高水準なプログラミング環境を構築するためには，高速なコンパイラが不可欠であり，特に，最適化処理を含んだ，意味処理を高速化する必要がある．そこで，意味解析処理を並列に行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提案する．意味解析を行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．これら意味解析プロセスの，ストリームを介した相互作用として，意味解析プロセスを記述する．また，プロセス管理の効率化のために，複数のプロセスを1つのプロセスにまとめる処理を行い，さらに，ストリーム間の無駄なコピーを行うプロセスの除去を行う．このモデルに基づいて，プロトタイプ・システムを作成し，PL/0 コンパイラの記述を行った．この PL/0 コンパイラを用いて，複数台のプロセッサといくつかのテスト・プログラムに対するシミュレーションを行った結果，マルチプロセッサ上で数倍程度の高速化が見込まれることが確認された．

1. はじめに

計算機の処理速度が向上しているとはいえ，コンパイラの応答性の悪さは，プログラミングに携わる人間の思考を阻害する一因ともなっている．これに対して，コンパイラの応答性を改善するために，インクリメンタル・コンパイル[1]や直接実行型の計算機[2]などが考えられている．このように，プログラム開発の局面において，コンパイル時間を短縮することは高度なユーザ・インタフェースを実現する上で不可欠である．

また，現在普及しつつある，RISC[3]や VLIW[4]，汎用並列処理マシンなどの高速計算機に対しては，最適化がプログラムの実行性能に多大な影響を及ぼすことを考え合わせると，プログラム開発時における計算機の応答性と実行性能を両立させるためには，人間と計算機との接点となっているコンパイラの処理に要する時間を，並列処理等の手法を用いることで短縮する意義は大きいと考えられる．

ここでは，コンパイラの応答性を改善することで，高度なユーザ・インタフェースを構築するために，コンパイラの実現に並列処理を用いることを考える．

コンパイラは通常，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互いに比較的独立性が高く，容易にパイプライン型の並列処理を行うことができる．また，字句解析や構文解析といった定型的な処理についてはハードウェア化によって高速化することが可能である．このような改良を行うことで，簡単なコンパイラでは処理時間を十倍程度短縮することができるが，意味解析処理がボトルネックになり，それ以上の効率の向上は望めない[5)~7)]．最適化処理を行うコンパイラでは，意味解析フェーズや最適化フェーズの，コンパイル処理全体に占める負荷がさらに大きくなると予想される．

コンパイラの意味解析部のように言語に対する意味処理の記述を行う場合，その言語の文法の各非終端記号の生成規則に対して，意味動作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[8]のような生成系で用いられている方法であり，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われる．一方，後者は属性文法[9),13)]に基づいたシステムで用いられる方法であり，属性の依存関係によ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われる．

意味処理を並列に行うことを考えた場合，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う方法では，実行順序に依存した制御を行う必要があることから，プログラミングの複雑さが

† Stream Based Description of Parallel Semantic Analysis by HIROYASU NISHIYAMA (The Doctoral Program of Engineering, Graduate School, University of Tsukuba) and KOZO ITANO (Institute of Information Sciences and Electronics, University of Tsukuba).
本研究は文部省科学研究費補助金（課題番号 01850072）により補助された．
†† 筑波大学大学院工学研究科
††† 筑波大学電子・情報工学系

増加し，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つの属性として扱われることから，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，意味動作を用いる方法では，並列性の抽出や並列動作するプログラム間での同期などの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を行えるよう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共有メモリ型のモデルを基本としてメッセージ通信型のモデルと組み合わせたものを考え，意味処理をストリームによって結合された複数のプロセスの相互作用として捉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，ストリームを意味処理間で受け渡されるデータ構造を基本とすることで，動的な意味の記述や，並列性の抽出を行うことが容易となっている．以下，2章では意味解析処理のモデルについて，3章では本方式での意味処理の記述法について，4章では実現の方式について，5章ではプロトタイプ・システムによる性能評価に関して説明する．

2. 意味の並列解析モデル

本章では，前提とする並列コンパイラの構成を示し，続いて並列意味解析のモデルを説明する．

2.1 コンパイラの構成

コンパイラの構成として字句解析，構文解析，意味解析，コード生成を行うモジュールがそれぞれパイプライン的に結合され，それぞれ同時に処理を行うものを前提とする．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と，トークンに付随する属性値を意味解析部に対して出力するものとする．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う．

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うモデルを提案する．

各意味解析プロセスは，文法と意味規則の定義によって指定された解析木の節に対して生成される．解析木の1つの節に対しては複数のプロセスが対応可能であり，生成されたプロセスは，1つの出力ストリームと複数の入力ストリームを持つ．入／出力ストリームは，意味解析プロセスに対応する解析木の節の親，兄弟，子の節に属したプロセスとのデータの受渡しに使用される．意味解析プロセスと対応する解析木の節の関係は3章で述べる記述形式に基づいて，文法の非終端記号に対する文法と生成規則の組として，意味解析器の記述時に与える．

これらプロセス間を結ぶストリームは，順序付けられたデータの並びを表すもので，具現化されていない並びを終端に持つことが可能である．ここでのストリームは，Concurrent Prolog[10] におけるリストや，CSP[11] でのチャネルに相当するものである．ここではストリームに対する操作として，次のようなものを考える．

1) ストリームの終端へ要素を追加する．
2) ストリームの先頭から要素を取り出す．
3) ストリームが終端かどうかを調べる．
4) ストリームを閉じる．

意味解析プロセスの生成処理は，構文解析部で各々のプロセスに対応したマーカ非終端記号[12]の還元動作が行われるのに対応して行われ，生成されたプロセスは，生成と同時に実行を開始する．プロセスの生成時に，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される．この際，単純なコピーを行うだけのプロセスを生成せず，値を生成するプロセスと受け取るプロセスの間を直接結ぶことで，処理の効率化を行う．生成された意味解析プロセスは，入力プロセスの値をもとに出力ストリームの値を計算し，出力ストリームの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．ここで，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．

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出力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に記述できるようになっている．さらに，ストリームの要素の値が，同じストリーム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，履歴に依存するような情報を扱えることから，動的意味の記述をある程度自然に行うことが可能になる．

ここで提案する並列意味解析系で解析可能なクラスは，ストリームの要素を1つだけとし，ストリームを属性文法での属性と対応させると，非循環な属性文法[13]のカバーするクラスに対応する．これに加えて，ストリームが複数の要素を持つことが可能であり，同一のストリームの要素間で依存関係を持った記述が可能なことから，実際には，ストリームの要素の依存関係にループがなく，ストリームの要素が，同一のストリーム上で自身以降に現れる要素に依存しないようなものとなっている．

3. 記　述　法

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プログラム・データを生成する．以下では，この記述形式を示し，典型的な意味処理のいくつかについて，その記述を与え，処理方式を説明する．

3.1 記述形式

並列意味処理の記述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〉→〈非終端記号〉
          ':'{〈(非)終端記号〉}
             [〈意味規則〉]
          {'|'{〈(非)終端記号〉}
             [〈意味規則〉]
          }
〈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．

効率化のためには，いくつかのプロセスを実行時に1つのプロセスとして実行したほうがよい場合もある．このために，意味解析プロセスに，並列プロセスと逐次プロセスとを用意している．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うことによって，プロセスのストリーム参照に右依存がない場合，つまり，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の下で，論理的には独立な複数のプロセスを，実際には1つのプロセス（複合逐次プロセス）として実行することを可能にしている．逐次プロセスと並列プロセスに関しては，4章の実現の部分で詳しく述べる．

3.2 ストリーム操作

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プロセスの定義を与えるための仮想的なプログラミング言語を用いる．[] で囲まれたデータはストリーム定数，+ はストリームの先頭へ要素を追加したストリームを生成する関数，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 はストリームの終端が具現化されているかどうか，つまり，ストリームが終りかどうかを確かめる関数，append_stream は引数のストリームを結合したストリームを結果とする関数である．

3.3 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す．一般に，中間コードの生成を行う部分では，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を行う．例えば，while 文の中間コード生成を行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']) ;
    }
```

ここでは，条件式に対応したノード cond の中間コードをストリーム cond. code から，実行文に対応したノード stmt の中間コードをストリーム stmt. code から受け取り，条件分岐等の命令と共にこれらを結合し，ストリーム while_stmt. code へ出力している．

属性文法でもコード生成に対してほぼ同様な記述を与えるが，属性文法では，一般に stmt. code や cond. code に対する while_stmt. code の依存関係で評価の順序を決定するため stmt. code と cond. code の実行は同時に行うことはできるが，while_stmt. code の評価はこれらの評価が終了した後で行われることとなる．これに対して，ストリームを用いた手法では，stmt. code, cond. code の評価と同時に while_stmt. code の評価を行うことが可能である．ただし，ストリームを有限長のバッファで実現する場合には，ストリームの結合を行う append_stream の実現が性能面で重要となる．この実現に関しては，4.3 節で説明する．

3.4 記号表参照

記号表の実現では，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，記号表参照を必要とする意味解析プロセスが，記号表参照の要求を記号表管理プロセスに送り，結果を要求したプロセスへ送り返すという方式での実現例を示す．

```
Block : '{' Dcl Stmt '}'
      {Stmt. symtab=symtab (Block. symtab,
                            DCl. names,
                            Stmt. symtab) ;
      }
Dcl : Var
      {Dcl. names=[Var. name] ;
      }
Dcl : Dcl Var
      {Dcl[0]. names=Dcl[1]. names
                   +[Var. name] ;
      }
Stmt : Block
      {Block. symtab=Stmt. symtab ;
       Stmt. symtab=Stmt. symtab ;
      }
      | .........

symtab(outer_symtab, names, inner_symtab)=
      ⟨demand. pid,
        search(outer_symtab,
               names, demand. name)⟩
      +symtab(outer_symtab,
               names,
               tail(inner_symtab))
      WHERE
            demand is head(inner_symtab)
```

ここで symtab は引数として，外側のブロックの記号表参照を行うためのストリーム outer_symtab, 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, ステートメント部からの記号表参照の要求を受けるためのストリーム inner_symtab を持つ．記号表参照を行うプロセスは，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 inner_symtab を介して記号表管理プロセス symtab に渡す．ここで，demand は head (inner_symtab) の省略記法として定義している．

この例では，プロセス symtab はストリーム inner_symtab によって渡された記号表の参照要求をもとに，関数 search によって記号表の検索を行い，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参照元のプロセスに返す．関数 search では，記号が局所的なブロックで定義されていない場合は，outer_symtab を介してより外側のブロックの記号表へ参照要求を送り，参照結果を受け取る．

このような記述を行うことで，一般的に局所的なブロックでは局所的な名前の参照が多いということから，記号表を全意味解析プロセスにコピーしたり，大域的な記号表管理プロセスを用意する場合に比べて，プロセス間の通信コストの軽減になる．また，この方法では，記号表を管理するプロセスの内部状態として記号表を持つことが可能であるので，エラー等を処理する場合に記号表を直接書き換えることが可能である等，属性文法による実現に比べ利点も多い．

4. 実　　現

並列意味解析器の実行は，実際の意味処理を行う意味解析プロセスと，構文解析部から構文解析情報を受け取り，それに従って意味解析プロセスを生成する制

御プロセスと呼ばれるプロセスの2つの部分に分けて行われる．以下では，意味解析プロセス，制御プロセスの双方についてその実現を示し，続いて，ストリームとその結合操作 append_stream の実現を説明する．

4.1 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，プロセス切り換えのオーバヘッド等の問題から，論理的には並列実行を行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ほうが，実行の効率がよい場合がある．例えば，入力ストリーム s1 と s2 を持つプロセスで，ストリーム s1 のすべての要素を読み込んだ後，ストリーム s2 の値を取り込む処理を行うとすると，ストリーム s2 の要素を生成しているプロセスは，ストリーム s1 の要素の生成が終了するのを待つことになる．このような場合，いくつかのプロセスを1つのプロセスとして実行することで，プロセス管理のオーバヘッドを除くことができる．このために，意味解析プロセスには，複数の論理的には独立なプロセスを1つにまとめて実行する逐次プロセスと，それぞれが独立に実行される並列プロセスとを用意している．

逐次プロセスを複数のプロセスとして実行した場合の実行結果は，それを1つのプロセスにまとめて実行した場合と同様であるが，1つのプロセスとして実行を行う場合には，逐次プロセス間の情報の受渡しはスタックを介して行われるため，プロセス間通信やプロセス切り換えに要するコストが不要になる．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，3.1 節で述べたようにプロセスのストリーム参照に右依存性があってはならないという制限がある．

意味処理の記述を行う際に，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とっている．

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言語でストリーム通信と並列処理用のライブラリを用いて記述している．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．

4.2 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った結果を受け取り，意味解析プロセスの生成を行う．

ここで，

〈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に右依存性がない場合，不要なストリームのコピーを行うプロセスの生成を行わず，制御プロセス内の実行で双方のストリームを同一のものとして処理する．これによって，不必要なコピーを行うプロセスの除去を行うことができる．

意味解析プロセスの生成は，対応するマーカ非終端記号の還元時に行われる．並列プロセスの場合はプロセスの生成が行われ，実行を開始する．逐次プロセスの場合は，逐次プロセスに対応する還元動作のうち隣接するものをまとめ，1つのプロセスとする．これにより，解析木上で隣接する節に対応した逐次プロセスを1つのプロセスにまとめることができる．この際，粒度が大きくなる可能性のある逐次プロセスについては，対応する逐次プロセスを適当な数ごとにまとめることで複数の複合逐次プロセスに分割する．

4.3 ストリームの実現

ストリームは固定長のバッファから構成され，ストリームからの読み出しを行うプロセスは，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，書き込みを行う．

仮想プロセスを構成する逐次プロセス間では，ストリームはすべての要素が評価された形のリストとされ，データの受渡しはスタックを用いて行われる．このスタック中には，データを格納するフィールドのほかに，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，逐次プロセスが他の逐次／並列プロセスからのデータを受け取るために用いられる．つまり，逐次プロセスは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を行うプロセスの実行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

実行の終了を待ち，結果を受け取った後，実行を再開する．

4.4 ストリームの結合操作

次に3.2節で述べたストリームの結合関数 append_stream の実現について説明する．

ストリームの結合関数 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．中間コードの生成部分を記述する場合には，append_stream の行う処理は重要であり，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と考えられる．この append_stream の実現には，引数のストリームの要素を生成している側のプロセスと，参照をする側のプロセスとの間で同期を行うかどうかによって同期と非同期の2種類の方法が考えられる．同期を行う場合の処理は Lisp 等におけるリストの結合処理と本質的に同様である．ここで，同期による実現での

append_stream(stream1, stream2)

の実行を考えてみると，今回のように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，〈 〉はデータの組を表すものとする．

```
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))
              〈stream.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 を呼び出して 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うことが容易となるが，一方で，end_of_stream のように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って，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．

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[14] 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った．ここで各意味解析プロセスの実現には SunOS 4.0 の Light Weight Process (LWP)[15] を用い，実験環境としては，Sun 4/110 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っている．LWP によって，単一プロセッサ上で仮想的な複数のプロセッサのシミュレーションを行うために，仮想的なプロセッサごとにスケジューリング用のキューを設けることとした．意味解析プロセスは，それぞれある仮想的なプロセッサに対するスケジューリング・キューに属すものとし，一定時間おきにスケジ

ューラによってこのスケジューリング・キューの切り換えを行うことで，マルチ・プロセッサ環境のシミュレーションを行う．ある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．プロセッサへのプロセスの割当は，今回の実現では，プロセスが生成される順に，プロセッサへ交互に割り当てるという方式をとっている．より効果的な処理を行うには，実行時の負荷によってプロセッサの割当を決定するという方式が望ましいと考えられる．

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は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．

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，縦軸は仮想プロセッサ数を表している．*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* および，*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．

同様にいくつかのプログラムについて，プロセッサ台数を変えてプロセッサ稼働率（各プロセッサで実行可能状態にあるプロセスの平均数）を求めたグラフを**図2**に示す．対象とするプログラムによってばらつきはあるが，プロセッサ数が十数台程度の場合に対してはプロセッサ稼働率は1以上となっているものが多く，細粒度の並列実行をサポートする並列処理環境の上で実行することで，プロセス間通信のオーバヘッドを考慮に入れても，数倍程度の速度向上を見込むことが可能であると予測される．

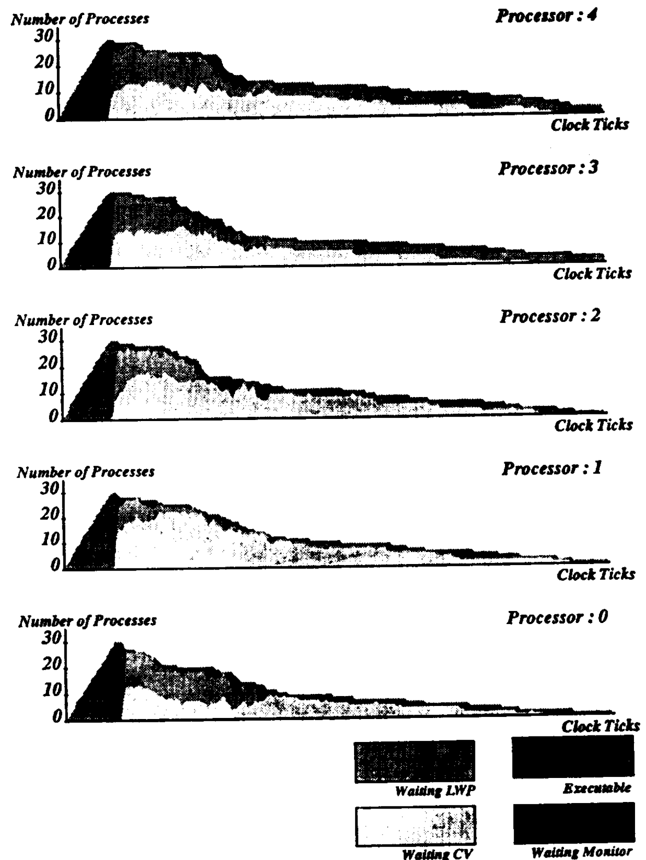

図1 プロセッサ5台での並列意味解析プロセスの実行状態
Fig. 1 Status of execution of processes in parallel semantic analyser with 5 processors.

さらに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，今回の実験は小規模の言語に対して，少数のプログラムとプロセッサの組合せに対する評価しか行っていないが，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とから，より大規模な言語の記述を行い，それに対して効率のよいプロセッサ台数を知るために，多数のプログラムに対して評価を行い平均的な値を得ることが重要であろう．

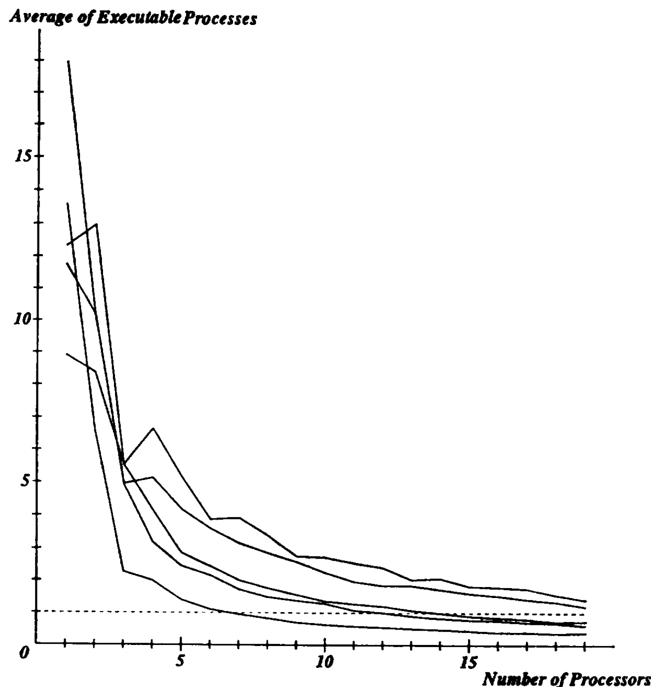

図 2 並列意味解析器のプロセッサ稼働率
Fig. 2 Processor execution rate of parallel semantic analyser.

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．

最近では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．また，細粒度の並列処理をサポートする機構の研究も行われており，本研究のような細粒度のプロセスを多数生成する並列処理を効率よく実行できる可能性がある．

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，プロセス管理のコストやその手法，また，プロセスの粒度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う過程で十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) Earley, J. and Caizergues, P.: A Method for Incrementally Compiling Languages with Nested Statement Structure, *CACM*, Vol. 15, No. 12, pp. 1040-1044 (1972).
2) Itano, K. and Sato, Y.: Architecture of the Universal Direct Execution Computer UDEC, *Proc. of the Hawaii International Conference on System Science*, pp. 206-213 (1987).
3) Katevenis, M. G. H.: *Reduced Instruction Set Architecture for VLSI*, MIT Press (1984).
4) Ellis, J. R.: *Bulldog—Compiler for VLIW Architectures*, MIT Press (1985).
5) 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，第 36 回情報処理学会全国大会論文集，pp. 851-852 (1988).
6) Itano, K., Sato, Y., Hirai, H. and Yamagata, T.: An Incremental Pattern Matching Algorithm for the Pipelined Lexical Scanner, *Inf. Process. Lett.*, Vol. 27, No. 5, pp. 253-258 (1988).
7) Itano, K., Nishiyama, H. and Chu, Y.: A Bottom-up Parsing Coprocessor for Compilation, Technical Report TR-2280, Department of Computer Science, University of Maryland (1989).
8) Johnson, S. C.: Yacc—Yet Another Compiler Compiler, Computing Science Technical Report 32, AT & T Bell Laboratories (1975).
9) Knuth, D. E.: Semantics of Context-free Languages, *Mathematical Systems Theory*, Vol. 2, No. 2, pp. 127-145 (1968).
10) Shapiro, E.: Concurrent Prolog—A Progress Report, *Computer*, Vol. 19, No. 8, pp. 44-58 (1986).
11) Hoare, C. A. R.: Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 547-557 (1974).
12) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers—Principles, Techniques, and Tools*, Addison-Wesley (1986).
13) Deransart, P., Jordan, M. and Lorho, B.: *Attribute Grammars*, LNCS 323, Springer-Verlag (1988).
14) Wirth, N.: *Algorithms + Data Structures = Programs*, Prentice-Hall (1976).
15) Sun Micro Systems: SunOS Reference Manual, Sun Micro Systems (1988).

(平成元年 6 月 19 日受付)
(平成 2 年 2 月 13 日採録)

西山 博泰（正会員）

1965年生．1989年筑波大学第三学群情報学類卒業．現在同大学院博士課程工学研究科に在学中．言語処理系，並列処理方式，プログラミング環境に興味を持つ．ACM 会員．

板野 肯三（正会員）

昭和23年生．昭和46年東京大学理学部物理学科卒業．昭和48年同大学大学院修士課程修了．昭和51年同博士課程単位取得後退学．理学博士．筑波大学計算機センタ準研究員，同大学電子・情報工学系助手，講師を経て，現在，同助教授．コンピュータアーキテクチャ，オペレーティングシステム，プログラミング言語処理系に興味を持つ．ソフトウェア科学会，IEEE, ACM 各会員．