コンパイラの意味処理の疎粒度並列処理の実現と評価

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

コンパイラの意味処理を並列に行なうプロセスを解析木に疎に割り当てる"疎粒度"並列意味処理のモデルを提案し，このモデルに基づいて実際に意味処理を実現し，シミュレーションによって性能の解析を行なった．この実現では，ブロックレベルの構文単位でプロセスを割り当てることにより，プロセスの粒度を大きくし，プロセス切替の頻度を減少させるアプローチをとった．この結果，以前に実現していた細粒度方式[7]でプロセスの管理に費やされていた時間が大幅に減少し，本方式では1.9～4.5倍の性能が得られた．

The Implementation and Evaluation of a Sparse-Grained Parallel Semantic Analysis of a Compiler

H.Nishiyama* and K.Itano**
(*) The Doctoral Program of Engineering,
Graduate School, University of Tsukuba
(**)Institute of Information Sciences and
Electronics, University of Tsukuba

We propose a "sparse-grained" parallel semantic analysis model to sparsely allocate each semantic process of a compiler to the relevant set of parse tree nodes. Based on this model, we actually implemented a parallel semantic analyzer and made a simulation to analyze the execution performance.The implementation adopts the approach to obtain large process granuality and reduce the process switch frequency by allocating the semantic processes in the syntactical unit of the block level. As a result, 1.9 to 4.5 times speedup was achieved by reducing the time consumed during the process management in the previous implementation on fine-grain model[7].

1. はじめに

現在我々は，非数値処理を対象としたアーキテクチャ及びアルゴリズム研究の一環としてコンパイラの並列処理を対象として研究を行なっている．

コンパイル処理を並列に行なうアプローチとしては，makeなどを利用してコンパイラ自身を同時に複数動作させるという方法[1]も存在するが，プログラミングのサイクルにおける待ち時間を減少させるには，コンパイラを高速化することが有効である．

コンパイラの構造を考えると，字句解析処理や構文解析処理などについては表を用いた効率的な実現方法が開発されている事から，専用のハードウェア[2][3]を用いることで十分な速度で実行を行う事が可能である．これに対して，最適化や中間コード生成を含んだ意味処理については，属性文法や構文主導型翻訳法による定式化が行われているが，これらは表を用いた単純な実現には向かないため，ハードウェア化によって処理の高速化を図るのではなく，意味処理を並列に行うことで意味処理の高速化を行なうという方針をとっている．

意味処理を並列に行う手法としては，ソースプログラムあるいは解析木を適当な単位に分割し，それぞれに対して逐次的な意味処理の手法を適用する方法や，意味処理を属性文法を用いて記述し，属性間の依存関係に着目し属性の評価を並列に行う方式などが研究されている[4][5]．

これらに対して，我々の採用しているアプローチでは解析木の各ノードに対して意味処理を行うプロセスを定義し，これらのプロセスが解析木に沿って張られたストリームを介して意味処理を行なう方式をとっている[6]．

これまでに，対象言語の式及び，ステートメントのレベルで意味処理を行なうプロセスを生成する，細粒度の意味処理モデルの実現を行ないその性能評価を行なった．これによって，細粒度のプロセスによる並列意味処理では，プロセスの実行管理に多くの時間が費やされているという結果が得られた[7]．

この結果を踏まえて，細粒度プロセスを対象言語のブロックレベルの比較的まとまった単位のプロセスに置き換えた方式の実現と評価を行なった．なお，ここでは解析木のノードに対してほぼ1対1に意味処理を行うプロセスを生成する従来の方式を細粒度方式，解析木に対して意味処理を行うプロセスを疎に割り当てる方式を疎粒度方式と呼んで区別している（図1-1）．

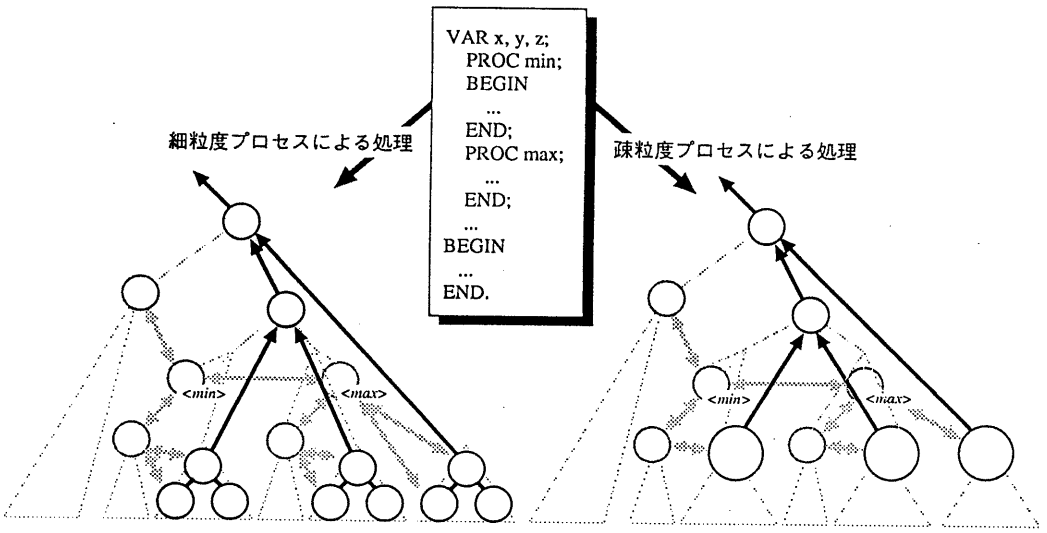


図1-1 解析木へのプロセスの割り当て

以下2節では意味解析処理を行うプロセスとその実行環境を提供する仮想マシンについて述べ，3節では細粒度の実行方式と疎粒度の実行方式についての実行結果とその考察を行う．最後にまとめを行い今後の課題を述べる．

2. 意味処理の並列実行

並列意味処理器は，実行時に動的に生成され実際の意味処理を行なう意味処理プロセスと，これら意味処理プロセスの実行環境である仮想マシン[8]によって実行される．

以下では，この意味処理プロセスと仮想マシンについて説明する．

2.1 意味処理プロセス

意味解析器では制御プロセスと呼ばれる特殊なプロセスが，構文解析器から構文解析動作に対応したデータの列を受けとり，意味処理を行なうプロセスの生成を行う．生成されたプロセスは，解析木の枝に沿って定義されたストリームを介して通信を行いながら意味処理を行い，中間コードを出力する（図2-1）．

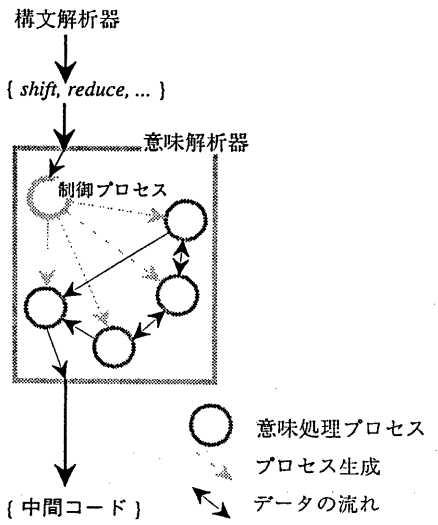


図2-1. 意味解析器の構成

このような処理方式を採用したのは，副作用をプロセス内に局所化することにより，意味処理の並列実行のための記述を行ないやすくし，また，構文解析器との間に解析木を作成しないで，1パスで行なうことを可能にするためである．1パスで意味処理を行なう場合，属性文法などでは，解析木上で右側の木の属性を左側の解析木で利用することを許さない場合が多い，これに対して，我々の用いている方式では，ストリームを介して通信を行なうことにより，右側の解析木に属したプロセスから左側の解析木に属したプロセスの情報の移動を可能にしている．

意味解析プロセスの定義は，構文木を構成する各ノードに対して行なう．例えば，式の値を計算するプロセスの定義は次のようになる．

```
Exp  : Exp '+' Exp
            { Exp[ 0].val = add( Exp[ 1].val, Exp[ 2].val); }
     | ...
       ...
     ;
```

ここで，Exp[i].valはそれぞれi番目の記号Expに対応したストリームvalを表し，addは文法に対応して生成されるプロセスを表している．生成されたプロセスは，等号の左辺及び，引数ののストリームをパラメータとする手続きとして実行される．

意味処理を行なうプロセスの生成は，実行時に構文解析部から対応する還元動作を受けとることにより行なわれる．また，意味処理記述時にプロセスに対して指定を行う事で，解析木上で連続したプロセスを1つのプロセスとして実行することを可能にしている．ただし，この場合には，1つのプロセスとして実行されるプロセス群中のストリームの参照関係にループがあってはならないなど，プロセスの記述が部分的に制限される．このプロセス群を1つのプロセスとして実行する機構は，意味処理を行うプロセスの粒度を意味処理の記述時にある程度コントロールすることを可能にしている．ただし，意味処理プロセスの生成は，

文法の定義に付加したプロセスの記述に従って，解析木に対応した入力に応じて動的に行なわれることから，静的なスケジューリングや粒度の細かな調節が難しいという側面がある．

意味解析器は，構文解析器でLR解析法を用いて構文解析を行うことを仮定しており，移動や還元といった解析動作とそれらに付随した情報が構文解析器から意味解析器へと受け渡される．制御プロセス中には構文解析器中の状態スタックと同期して動作するスタックが設けられており，プロセス間をストリームで結合する際に使用される．

意味解析処理を行うプロセスは構文木の枝に沿って定義されたストリームを介して通信を行ない，意味処理を行う．ただし，ストリームを介して受渡し可能なデータは1ワードの大きさに収まるものに制限し，それ以上の大きさのデータについては，データの格納された共有メモリ上のアドレスをストリームを介して受け渡す方式をとる．このため，中間コードや記号表に格納されたデータ等は共有メモリに確保した領域上に格納し，そのポインタをストリームを介して受け渡すようにする．

2.2 意味処理並列実行用仮想マシン

意味処理部を並列に実行するために必要な"実行環境"を抽象化し，意味処理並列実行用仮想マシンとして定義した．ここで採用している意味処理のモデルを直接実行するようなハードウェアを設計することも不可能ではないが，当面は，特殊なハードウェアのメカニズムは用いないで，ごく一般的なハードウェアの構成を前提としてシステムの動作特性を知ることを目標としたため，特殊な機能に関してはソフトウェアで実現することにした．

この仮想マシンは，通常の命令実行環境の他に，

・プロセス生成／消滅
・メモリ管理
・プロセス間通信

を行なうプリミティブを意味処理プロセスに提供する．

以下，これらのプリミティブについて簡単に説明する．

(1) プロセス生成

プロセス生成のためのプリミティブとしては，プロセスの生成をおこなうプリミティブ(create_process)と消去を行うプリミティブ(destoroy_process)を用意する．仮想マシンは，各意味解析プロセスに対して1対1に生成され，意味処理部ではスケジューリングやプロセッサ数などの実現の詳細に関しては考慮する必要がない．

(2) メモリ管理

メモリ管理のためのプリミティブとしては，局所／共有の各メモリの確保及び解放を行うプリミティブ(malloc,mfree,shmalloc,shmfree)を用意する．2.1節で述べたように，共有メモリはプロセス間で共有される比較的大きなデータをプロセス間で共有するために利用する．共有メモリ上での不可分命令を提供することで，共有メモリをプロセス間の通信や同期に利用することも可能であるが，プログラムの記述が煩雑になるため，意味処理の記述レベルでは，プロセス間の通信と同期を行うための機構としては次に述べるストリームを用意する．

(3) プロセス間通信

仮想マシンはプロセス間の通信機構としてストリームを提供する．ここで用いるストリームはCSP[9]におけるチャネル型のプロセス間のメッセージ送信機構を有限長バッファとして実現したもので，意味解析プロセスに対応した仮想マシン間のデータの送受信と同期処理を行う．仮想マシン間のストリームの結合は仮想マシンのレベルで自動的に行なわれ，意味処理を行なうプロセスのレベルでは関知する必要がない．ストリームに関連し

たプリミティブとしては，ストリームの生成(create_steream)，データがストリームであるかどうかの確認(is_stream)，ストリームへのデータの送信(send_stream)，ストリームからのデータの受信(recv_stream)，ストリーム中のデータの確認(poll_stream)，2つのストリームを用いた排他的な送受信(send_and_recv_stream)，を行うための6つのプリミティブを用意している．最後に示した排他的な送受信のプリミティブは，単一サーバ-複数クライアントの，クライアント・サーバ型の通信を可能とするために導入したもので，複数のプロセスが同一のストリームを利用して1つのプロセスに要求を送り，結果を行う事を可能とする．この操作はアトミックに行われ，複数仮想マシンで同一のストリームへの要求があった場合には排他制御が行われる．

現在，意味処理実行用の仮想マシンは，SPARCを用いたマルチプロセッサ上で実現されている．この実現では，実行時の効率を重視して，仮想マシンインタープリタなどを用いる方法をとらず，仮想マシンのプリミティブをライブラリの形で実現し，意味処理を行なうプログラムとオブジェクトのレベルで直接リンクする方式をとっている．

図2-2に仮想マシンと，実際のハードウェア，意味解析プロセスの関係を示す．

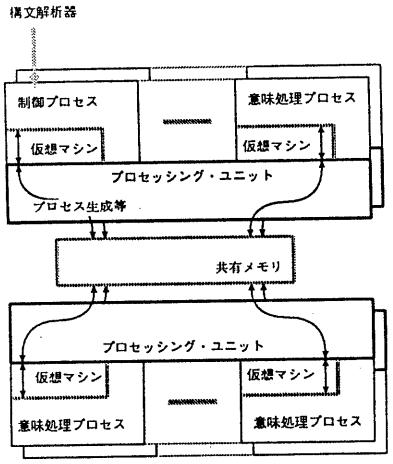


図2-2 仮想マシンの位置づけ

3. 疎粒度並列処理による意味処理

文献[7]では，対象言語の式のレベルの意味処理を含めて，並列に処理を行なった場合と，ステートメント・レベルで並列に意味処理を行なった場合について評価を行なった．この結果から，細粒度並列処理による意味処理の実行では，多数のプロセスを生成することにより十分な並列度が抽出されているが，現在の実現では，ハードウェアアーキテクチャ上での制約から，実際に実行を行なう際のプロセス管理に要するオーバーヘッドのた

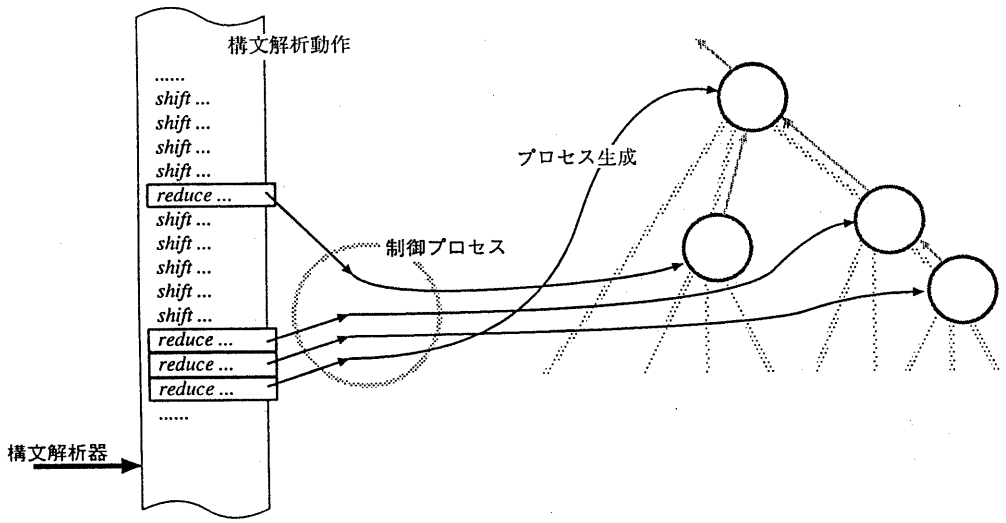


図3-1 細粒度方式の実現

め有意な性能向上が得られなかった．

このため，対象プログラムの解析木に対してプロセスの割り当てを比較的密に行なう従来の方法に対して，対象言語のブロックレベルの解析木に対してプロセスを割り当てて意味処理を行なうモデルの実現と評価を行なった．

3.1 実現方式

細粒度の実現では，意味解析プロセスの生成処理を次のように行っている（図3-1）．

(1)制御プロセスが構文解析器から還元動作を受け取った際，その還元動作に対応した意味解析プロセスが定義されており，かつ1つのプロセスに統合化するプロセスでなければ，新たにプロセスを生成しプロセス間をストリームで接続する．

(2)還元動作に対応するプロセスが1つのプロセスに統合すべきプロセスの場合，隣あったノードに定義されたプロセスをまとめ1つのプロセスとして実行する．

細粒度による処理方式の実現では，意味処理プロセス間を接続するストリーム及び，実行すべきプロセスの記録を行うためのデータ構造を保存するため，構文解析器中の解析状態を保存する状態スタックに同期して動作する意味解析スタックを制御プロセス内に設けている．疎粒度による処理方式では，同一のプロセスにまとめられるプロセスの数が多いため，上に述べたようなデータ構造を実行時に作成する方式をとらず，1つのプロセスにまとめられるプロセス群に対応した構文解析動作を新たに生成したプロセスに受け渡し，このプロセスで構文解析動作を解釈しながら実行することで意味処理を効率化を図っている．

この構文解析動作の区切りを表すために，1つのプロセスのまとめるプロセス群に対応した非終端記号列の前後にプロセスの統合の開始と終了を現すための特殊なマーカ用の非終端記号を導入した．例えば，

block → dcl compound_statement

のような文法に対して，dclとcompound_statementに対応した解析木に属したプロセス群をそれぞれ1つのプロセスにまとめるには，上の文法を次のように変形する．

block → SB dcl SE SB compound_statement SE

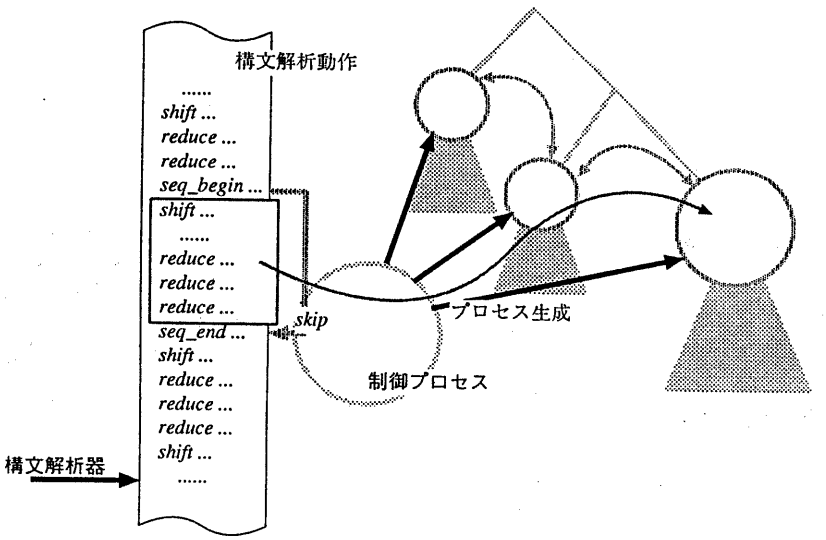


図3-2 疎粒度方式の実現

SB → ε
SE → ε

ここで，SBとSEはそれぞれプロセスの統合の開始と終了に対応した非終端記号である．意味解析器中の制御プロセスは，プロセスの統合の開始に対応したマーカの還元動作を構文解析器から受け取ると，新たにプロセスを生成して，対応した終了マーカの還元動作までの構文解析動作を読み飛ばし，その構文解析動作を新たに生成したプロセスに受け渡す．新たに生成されたプロセスでは受けとった構文解析動作を解釈しながら逐次的な意味解析処理を行なう．（図3-2）

次節で述べる実験に使用したPL/0[10]のコンパイラでは，ブロックのステートメント部，宣言部の定数宣言と変数宣言部をそれぞれ1つにまとめるように，計3箇所にマーカを挿入している．

3.2 実験環境

約250行のPL/0プログラムをサンプルプログラムとして，マルチプロセッサアーキテクチャのシミュレータ上で疎粒度並列処理方式と細粒度並列処理方式の実行を行った．

ここで用いたマルチプロセッサアーキテクチャは，意味処理を並列実行するために設計したもので，CPUとしてSPARCアーキテクチャ[11]を採用し，命令と局所データを格納するためのメモリを持った複数のPE(Processing Element)が，共有バスを介して共有メモリに接続されているというものである．共有メモリにはライトバック型のキャッシュメモリ[12]を持つ事を仮定している．

3.3 性能評価

ステートメント・レベルで並列処理を行った細粒度の処理の場合と，ブロック・レベルで並列に処理を行う疎粒度の場合それぞれについて，プロセッサ5台で実行した場合の，プロセス数，プロセス切り替えの間の平均的な実行クロック数，プロセッサ毎のプロセス切り替えの平均数を表3-1に，それぞれの方式での実行の様子を図3-3に示す．

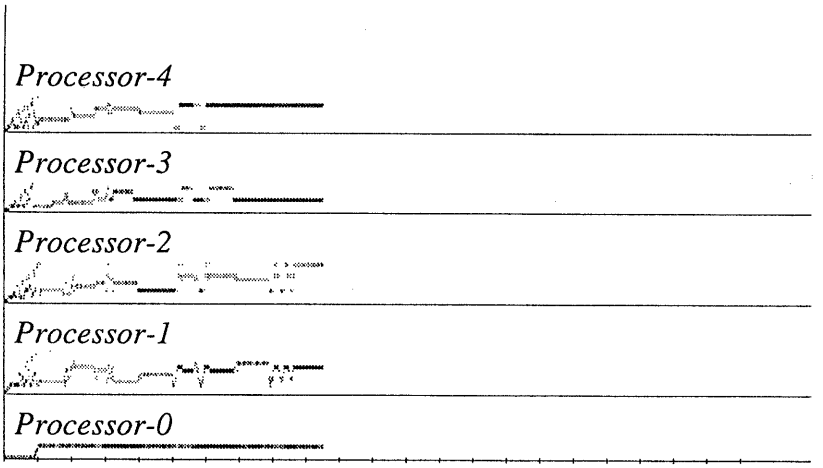
	細粒度並列	疎粒度並列
プロセス数	168	44
（区間あたりの）平均クロック数	3,805	6,101
プロセス切替数の平均	382	55

表3-1. プロセッサ5台での実行結果

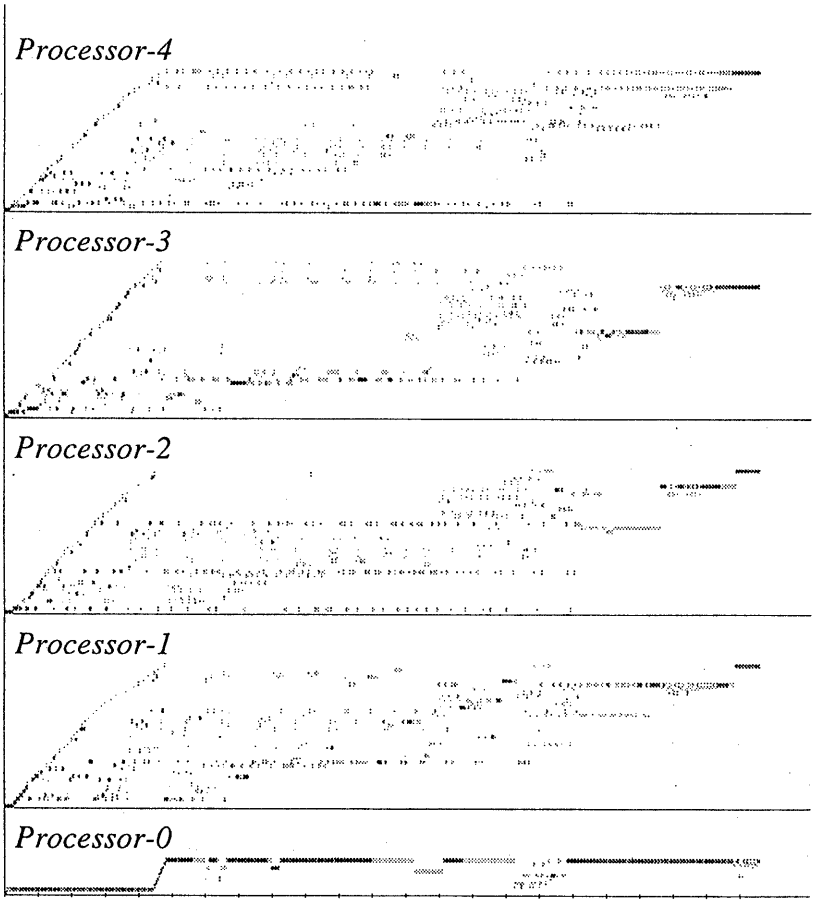
ステートメント・レベルでの並列処理に対してブロック・レベルの並列処理ではプロセス数が26%に減少し，プロセス切り替えの平均数は14%に減少している．プロセス切り替えの間の平均的な実行クロック数は細粒度方式に比べ疎粒度方式のクロック数が1.6倍となっている．プロセス数の現象に比べてプロセス切り替えの間の平均的なクロック数の増加が少ないのは仮想マシンの実行によるプロセス間の待ち合わせと，プロセスの統合処理を管理する為のデータ構造の操作が影響している．

図3-3の実行の詳細を表した図は，横軸が時間経過，縦軸は各プロセッサに割り当てられたプロセスを表している．また，データが細かいためはっきりと現れていない部分もあるが，色の薄い部分は意味解析プロセスの実行，濃い色の部分は仮想マシンでの実行を表している．細粒度方式では多数のプロセスが短い間隔で切り替えを行いながら実行を行っていることと，疎粒度方式では比較的時間の長いプロセスを切り替えながら実行を行っている様子がわかる．また，全体の処理時間は，細粒度方式が1,129,605クロックから475,287クロックとなり約2.4倍の速度向上となっている．

次に，サンプルプログラムを，プロセッサ台数を1から10まで変えて実行した場合の実行結果を細粒度，疎粒度それぞれの並列処理方式につい



(b)疎粒度方式の実行の様子



(a)細粒度方式の実行の様子

図3-3.意味処理の実行の様子

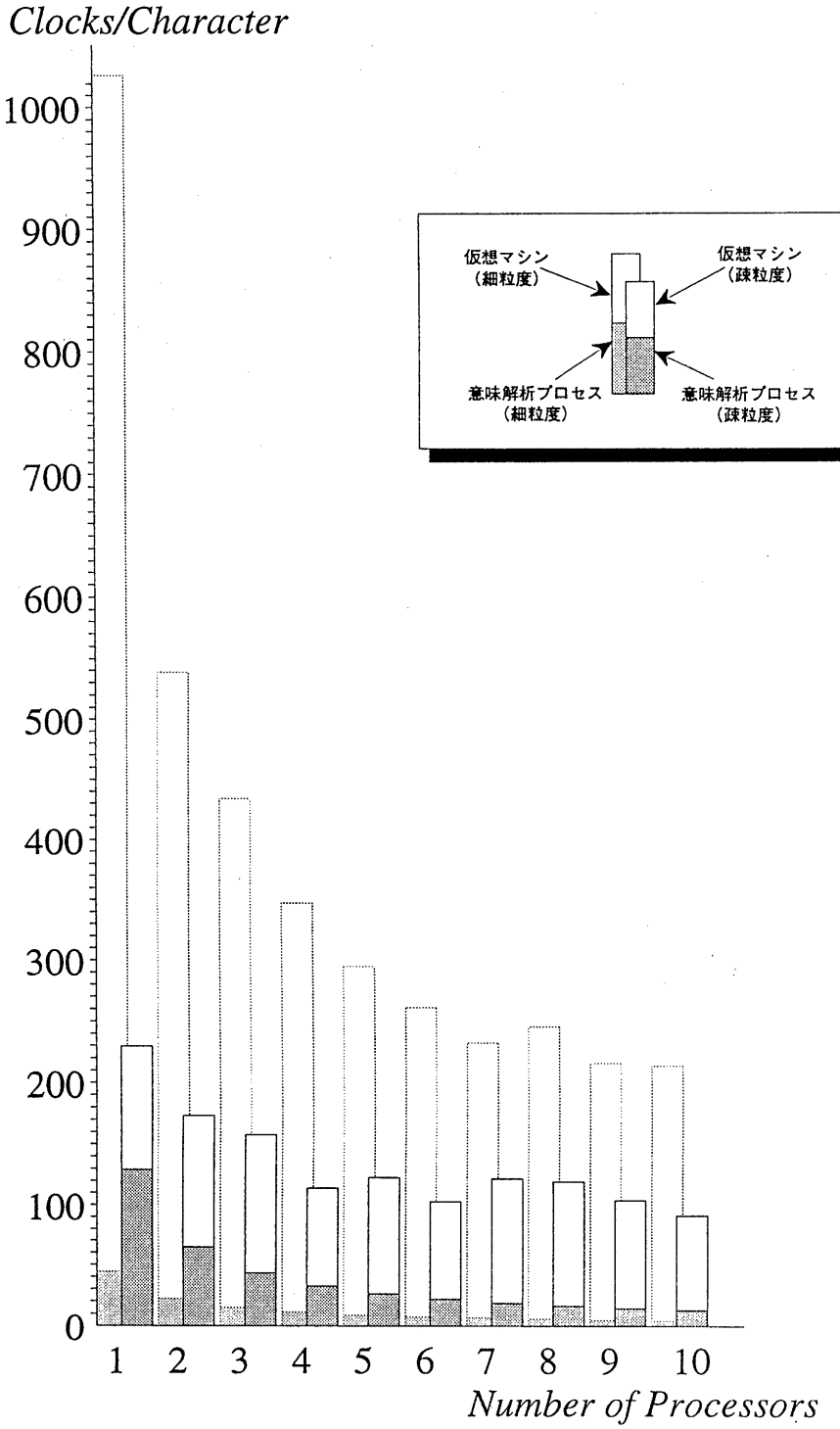


図3-4. 各方式の実行結果

て図3-4に示す．この図で，横軸はプロセッサ台数を表し，縦軸は入力プログラムの１文字あたりの意味処理に要したクロック数を示している．各データは，それぞれ仮想マシンと意味処理のクロック数に対応している．図から分かるように，意味処理の実行クロック数はプロセッサ台数に比例して減少しているが，プロセッサ数の増加により仮想マシンの実行に要している時間が増加し，実質的な速度向上が低く抑えられている．また，疎粒度の実現では，細粒度の実現に比べ，4.5倍から1.9倍の速度向上となっている．

4. おわりに

対象プログラムの解析木に対して，意味解析を行なうプロセスを疎に割り当てる方式の実現，および評価を行ない，粒度の小さなプロセスを密に割り当てる方式との比較を行なった．

現在の一般的なプロセッサアーキテクチャ上では細粒度プロセスを実行するのはオーバーヘッドが多きなため，疎粒度のアプローチの方が良い性能が得られている．ただし，現在の実現でも仮想マシンの実行に多くの実行時間が費やされており，仮想マシンレベルでアーキテクチャ面から粒度の小さなプロセスをサポートすることを考慮する必要があると思われる．

参考文献

1) Stallman, R.M. and McGrath, R.: GNU Make Manual, FSF (1991).
2) 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).
3) Itano, K., Nishiyama, H. and Chu, Y.: A Bottom-up Parsing Coprocessor for Compilation, Technical Report TR-2280, Department of ComputerScience, University of Maryland (1989).
4) Kuiper, M.F and Swierstra, S.D: Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism, LNCS-461, pp.61-75, Springer-Verlag (1990).
5) Schadri, V., Wortman, D.S, Junkin, M.D., Weber, S., Yu, C.P. and Small, I: Semantic analysis in a concurrent compiler, In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp.233-240 (1988).
6) 西山博泰, 板野肯三: ストリームに基づいた並列意味処理の記述, 情報処理学会論文誌, Vol.31, No.5, pp.731-739 (1990).
7) 西山博泰, 板野肯三: マルチプロセッサ・システムSMiS上での並列コンパイラCompasの実現と性能評価, 情報処理学会プログラミング-言語・基礎・実践-研究会 3-22, pp.195-203 (1991).
8) 西山博泰, 板野肯三: 並列コンパイラCompasの意味処理部の性能評価, 情報処理学会論文誌投稿中.
9) Hoare, C.A.R.: Communicating Sequential Processes, Comm. ACM, Vol.21, No.8, pp.547-557 (1974).
10) Wirth, N.: Algorithms+Data Structures= Programs, Prentice-Hall(1976).
11) Cypress Semiconductor: Sparc RISC User's Guide, ROSS Technology, Inc., Cypress Semiconductor Company (1990).
12) Archibald, J. and Bear, J.L.: Cache Coherence Protocols, ACM Trans. on Computer Systems, Vol.4, No.4, pp.273-298 (1986).