マルチPSIにおける並列構文解析プログラムPAXの実現および評価

寿崎かすみ, 佐藤裕幸, 杉村領一, 赤坂宏二, 瀧和男
新世代コンピュータ技術開発機構 *
山崎重一朗
富士通株式会社
弘田直人
情報科学研究所

1 はじめに

ICOTでは、並列推論マシンハードウェアの研究開発とともに、この上で稼働する並列アプリケーションプログラムのための並列向きアルゴリズムおよびプログラミングパラダイムの研究を行っている。プログラムを実際に並列に実行するための負荷分散方式の研究も重要である。

このような目的で、並列アプリケーションプログラムの試作を行っている。今回発表するPAXもその1つである。

PAXは自然言語の汎用構文解析プログラムで、形態素解析部、構文解析部、構文解析結果の表示出力部から構成される。形態素解析部および表示出力部はESPで記述されフロントエンドのPSI上で動作する。KL1で記述されてマルチPSI上で動作するのは構文解析部のみである。構文解析プログラムは、DCGよりトランスレータを用いて生成される。今後、構文解析部をPAX(狭義)と呼ぶことにする。

PAXの基本となる実行モデル[1]は既に提案されており、逐次プログラムとして実現したものをSAXと呼んでいる。同じくICOTで開発中の談話理解システムDUALS[2]は、構文解析プログラムとしてSAXを使用している。

今回PAXをマルチPSI上に実現するための負荷分散方式を考案し、これに対する計測評価を行った。

* 東京都港区三田1-4-28 三田国際ビルディング21階 03-456-3693, susaki@icot.junet

2 PAX

2.1 アルゴリズム

構文解析は左隅構文解析法といわれるアルゴリズムに基づいて行う。これは、上昇型幅優先のアルゴリズムである。文法はDCGにより記述する。上昇型の解析により終端記号である単語から非終端記号を得ると、次にそれが右辺の左隅の要素となる文法が存在するかどうかを調べる。また、その非終端記号が右辺の途中に現れる文法規則に基づく解析の途中でないかどうか調べる。このためには、どの文法規則のどの要素まで解析が進んでいるかを把握している必要がある。そこで、文法規則の右辺に現れる要素それぞれに識別子を与える。

ある非終端記号を右辺の左隅にもつ文法規則が存在する場合は、文法規則の右辺を順に試みて、その文法規則が適用可能かどうか調べる。これは、その非終端記号以前に行われている解析処理の内容に無関係の処理である。これをタイプ1の処理という。右辺の2番目以降に現れる非終端記号は、タイプ1で選択された文法規則が適用可能かどうか確かめる処理をする。具体的にいうと、解析の進み具合を示す識別子が自分の左隣まで進んだことを示すものかどうかを調べる。この処理は、それ以前に行われた処理の内容に依存して行われるものでタイプ2の処理という。構文解析処理は全解探索である。とくに、自然言語の文法は曖昧であることが多いので結果が一意であるとは限らない。

全解探索を行うために適用可能な候補を一括して集合として扱うという方法を用いることにする。処理の各段階で可能な候補をすべて選択し、集合として保

持する。処理の過程で適切でないと分かった候補は、その時点で捨てるという方法である。この他に、候補ごとに解析処理をおこない、行き詰まったらバックトラッキング機能を用いて別の文法規則を試すという方法もある。しかし、この方法は、同じ処理を繰り返すことになり効率が悪い。またKL1はバックトラッキング機能を持たないので、この方法は使用できない。

このようなアルゴリズムをもとにした構文解析プログラムをKL1により記述する。このプログラムでは終端記号(単語)および非終端記号つまり解析木の各ノードをプロセスとして表す。単語に対応するプロセスは、隣り合うもの同士ストリームでつなぐ。各プロセスは左のプロセスとの間のストリームの出力端、右のプロセスとのストリームの入力端を握っている。解析処理は左のノードから右のノードへ、下のノードから上のノードへ解析の途中経過を知らせることによりすすむ。途中経過はプロセス間にはられたストリームにメッセージを送ることにより伝える。

KL1において集合はリストとして表し、ストリームとして扱うのが普通である。そこで、PAXでは解の集合を、集合を構成するデータの流れるストリームとして表す。このストリームは識別子と組み合わせメッセージとして扱う。

ストリームをメッセージとして扱っているためストリームの中にネストしてストリームが存在するという状態になる。このような使い方を「レイヤードストリーム法」[3]と名付けた。

レイヤードストリーム法を用いることの利点は次の2点である。

1. データの受け取り手がデータの集合ができあがるのを待つ必要がないこと。解を成すプロセスは、解の候補を1つ生成するごとにストリームに流すことができる。解を受け取るプロセスは、1つずつ送られてくるデータを順に処理することができる。このような方式を用いることにより解を生成するプロセスと解を受け取るプロセスが並列に実行できる。

2. 解を受け取るプロセスは、そのプロセスに対して送られてくるデータがなくなってストリームが閉じられるまで生き続ける。このため同一のプロセスが重複して生成されることがなく重複計算がない。

今までに述べたアルゴリズムを簡単な例を用いて説明する。

```
sentence --> np, vp.
np --> adj, noun.
np --> prep, noun.
vp --> verb, adj.
vp --> verb, adv.

adj  --> [failing].
adj  --> [hard].
prep --> [failing].
adv  --> [hard].
verb --> [looked].
noun --> [student].
```

この文法を用いて、"Failing student looked hard." という文を解析する。解析の様子を図1に示す。'failing' という単語に対して 'adj' と 'prep' の2通りの可能性が存在する。そこで、'failing' に対応するプロセスは 'adj'、'prep' の2つの非終端記号に対応するプロセスを生成し自分は終了する。'failing' は握っていた2本のストリームを親プロセスにわたす。このときとなりのプロセスへのストリームは入力端を2つにわけて 'prep'、'adj' のそれぞれがメッセージを送れるようにする。'student' は 'noun' を生成し、隣からデータが送られてくるのを待つ。'adj'、'prep' は自分の存在を隣に伝えるメッセージと自分の入力ストリームを組み合わせて送る。'noun' は、2つのメッセージを受け取ると、そのそれぞれに対して 'np' プロセスを生成する。'np' は 'noun' が 'adj'、'prep' からメッセージとして受け取った入力ストリームを自分の入力ストリームとする。出力ストリームは 'noun' の出力ストリームである。

'looked' から生成された 'verb' は、自分の存在をとなりに知らせる。そのとき、自分の入力ストリームも一緒にわたす。'noun' が 'np' にしたのと同じように 'hard' から生成された 'adj'、'adv' も 'verb' から受け取ったストリームを 'vp' にわたす。このようにして、解析木の1段上のレイヤーに現れるノード同士この場合 'np' と 'vp' が1つのストリームの入力端と出力端を握ることができる。

プロセスをこのように定義することにより、文法規則ごとに独立に実行ができる。1つの文法規則の右辺で、左隣の要素と2つめ以降の要素が、1つのスト

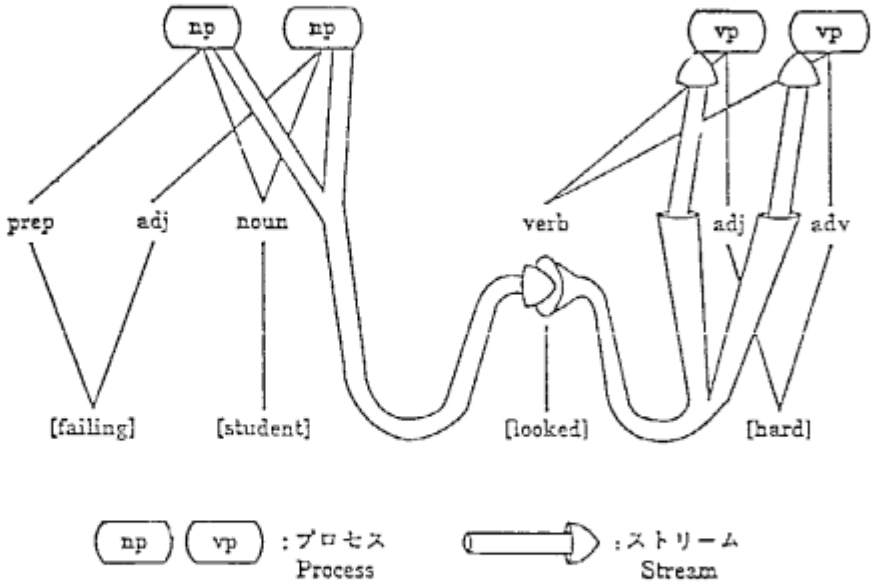

図 1: 解析木とレイヤードストリームによる通信

リームに対して作用するメッセージの生成者と生成されたメッセージの消費者群として並列に実行できる。候補が複数存在する場合は、つぎのプロセスへのストリームの入力端を分けて持たせることにしたので、これらのプロセスも独立して実行できる。このようにして、このプログラムから高い並列性が引き出せる。

3 PAX のマルチ PSI 上の実現

3.1 マルチ PSI

マルチ PSI システム[4] はマルチ PSI ハードウェア、KL1 言語処理系[5] およびオペレーティングシステム PIMOS[6] よりなる。

マルチ PSI ハードウェアは PSI-II の CPU を要素プロセッサとし、それらをメッセージ交換自動ルーティング機能を持つ専用のネットワーク[7] で二次元メッシュ状に接続した疎結合のマルチプロセッサである。このマルチ PSI では、ストリーム AND 並列論理型言語 KL1 が動作する。ストリーム AND 並列では、並列に実行されるプロセスがデータを共有しあい、あるプロセスの出力を別のプロセスが使うというデータ依存関係の連鎖としてプロセス構造が構成される。KL1 の処理系はマイクロコードにより提供されている。ネットワークを介した通信もこのマイクロプログラムにより実現される。オペレーティングシステム PIMOS は KL1 で記述され PIMOS 自身マルチ PSI 上で並列に動作する。PIMOS は実行管理、資源管理、入出力管理などオペレーティングシステムの基本機能を提供している。PAX をはじめとするアプリケーションプログラムは PIMOS 上で稼働する。マルチ PSI システムは I/O プロセッサとして PSI-II を使用しており、ユーザプログラムは PIMOS を通じてこの I/O 機能を使用することができる。

PAX は PIMOS 上で動作し、PIMOS を通じて PSI-II の I/O 機能を使用している。

3.2 負荷分散

プログラムのある部分とある部分を実際に並列に実行するためには、並列に実行したい部分を異なるプロセッサ上で実行する必要がある。高い並列性を引き出すことは多くのプロセッサが 1 つのプログラムの実行を分担しあって行うことを意味する。プログラムを効率良く実行するためには高い並列性を引きだし、なるべく多くのプロセッサに仕事を分担することが必要である。仕事を分担するときに、各プロセッサの受け持った仕事の量 (負荷) が均等になるように注意することも必要である。これを '負荷分散' と呼ぶ。

現在のマルチ PSI システムでは、プログラム作成時に次のような記法を用いてプログラマが分散のしかた

を指定している。

```
goal@processor(PE)
```

'goal' には、分散させるKL1 のgoalを記述し、PEでプロセッサ番号を指定する。

負荷分散を考えるときに注意すべきことがいくつかある。

- 実際のマルチプロセッサではプロセッサ台数が決まっている。
- ゴールを分散させるとゴール木のメンテナンスおよび異なるプロセッサ上で実行されているゴール間のデータ送受信のための処理が必要となる。これは、全体の実行効率を考えるとオーバーヘッドである。このオーバーヘッドは通信しあうプロセッサの距離が長くなるにしたがって大きくなる。

実際のプログラムに対する負荷分散方式は、これらの点を考慮してプログラムの特性に合わせて決定しなくてはならない。

3.3 PAXのための負荷分散方式

PAXのアルゴリズムでは次の2点に対し並列性を引き出すことを考えていた。

1. 異なる文法規則は並列に実行する。1つの文法規則の右辺に現れる各要素に 対応するプロセスも並列に実行できる。
2. 1つの終端記号あるいは非終端記号に対して複数の可能性が考えられる ときは、それらを並列に試みることができる。

上で述べた2つの並列性はそれぞれ次のような特徴を持つ。

1. 解析木の各レイヤーにあらわれる各ノードを並列に実行することになる。 したがって、並列度は実行開始時が一番大きく最大で単語の 数+αである。解析が進むにつれて並列度が下がる。 正しい解析木に対する通信は、隣あったプロセス間でのみおこり1対1で あるが、誤った解析木に対する無駄な通信もおこる。
2. 並列に試される各候補は、文法が曖昧でなければただ1つだけ成功する。 各候補は成功するとその結果をストリーム経由で返す。 失敗した場合はなにも送らず ストリームを閉じる。並列に試している各候補からの出力は 1つに集められ、このストリームを流れるメッセージを読む プロセス(タイプ2節)に渡される。このため、 このメッセージを待っているプロセスとの間に 多対1の通信がおこる。また、このストリームを読むプロセスも複数 存在する場合、つまりこのストリームを受け取る側にも複数の候補が考え られる場合、通信は多対1対多となり、通信量が大きくなる。

この1、2で示した並列性を実際に異なるプロセッサに分散して、マルチPSI上に実現すると、1だけでは並列度が低いが2も実現すると通信量が大きくなりすぎるという問題がある。通信量の増大は、オーバーヘッドの増大を意味する。 PAXは、メッセージを生成しそれをフィルターすることで処理を行う。このため、通信のオーバーヘッドが全体の効率に大きく影響する。

そこで、通信量を極力抑えた負荷分散方式を考えることにした。

2.において多対1対多の通信が発生する理由をもう一度見直す。多対1の通信が起こるのは、複数候補が存在する場合、それらが同一のレイヤーのストリームの入力端をわけて握り、それぞれがメッセージを流すからである。ストリームは1つにまとめられ、そのストリームを読むプロセスにわたされる。あるプロセスに対して、そのプロセスからのストリームを読むプロセスが複数存在する場合、つまり複数の候補が存在するときはそれぞれにそのストリームの出力端を分け与える。つまり 1対多の通信となる。多対1も1対多もあるレイヤーのストリームの入力端あるいは出力端を複数のプロセスが握っているために起こっていることである。このストリームのレイヤーごとに、プロセスを近いプロセッサに割り当てることにすれば通信のオーバーヘッドを抑えることができるはずである。このような方法に対する最初の試みとして今回はストリームのレイヤーごとにプロセッサを割り当てることにする。ただし、最初に各単語の間にはられたストリームそれぞれを1つのレイヤーということにする。ストリームのレイヤーごとにプロセッサを割り当てる

というのは、プロセスはすべて自分が読み込むメッセージの流れるストリームに割り当てられたプロセッサに移動して実行されるということである。ただし、タイプ1節はストリームを流れるメッセージを読まないので、自分が生成されたプロセッサで実行される。

この方法を用いると、ストリームの送りてもまた、自分が読み込むストリームのレイヤーに割り当てられたプロセッサ上で実行されているので、通信はそれぞれ多対1となる。

PAXの負荷分散についてまとめるとつぎのようになる。

まず、各単語のプロセスをその文中に現れた順に続き番号のプロセッサに分散する。このとき各単語間にストリームをはりストリームの先頭にそのストリームに割り当てたプロセッサの番号を入れる。単語数およびストリームの数がプロセッサの台数より多いときには始めに戻る。単語のプロセスは対応する非終端記号のプロセスを生成する。生成されるプロセスには、タイプ1とタイプ2の2種類が存在する。タイプ1節はストリーム中のデータを読まないので自分が生成されたプロセッサでそのまま実行される。タイプ2節ははじめてデータを読み込んだ時にストリームの先頭で指定されたプロセッサつまりそのストリームに割り当てられたプロセッサに移動する。このようにして、始めに生成される単語のプロセスおよびタイプ1節以外は、各ストリームに割り当てられたプロセッサで実行される。始めの単語に対応したプロセスおよびタイプ1のプロセスはすぐに終了することを考えるとこれは、ストリームごとの負荷分散であるといえる。

このような分散を行うと読み込むストリームが異なるプロセスは並列に実行できる。

前の例を使って説明すると、まず各単語を異なるプロセッサに分散する。つぎに、'prep'、'adj'は'failing'と同じプロセッサで実行される。'noun'は'failing'と'student'の間に張られたストリームに割り当てられたプロセッサ(現在の割り付けでは'failing'と同じプロセッサ)に移動する。同様に'hard'から生成された'adj'、'adv'は'looked'と同じプロセッサに移動する。2つの'vp'は、'student'と'looked'の間に張られたストリームを読むので、'student'と同じプロセッサに移動することになる。

この方法では通信量を抑えることのみを考えている。各プロセッサの負荷バランスについては考慮していない。また、ストリームの数しか並列度がでていないので、処理対象の文が短い (文を構成している単語の数が少ない)と始めから仕事の与えられないプロセッサが存在することになる。

文番号	単語	解析木
1	4	1
2	5	1
3	7	2
4	10	1
5	11	2
6	18	1
7	21	5
8	19	1
9	20	6
10	86	460

表 1: 例文

4 評価

実現したPAXプログラムに対して台数効果および各プロセッサの受け持った仕事量の2つの観点から評価を行った。またこれとは別に、並列推論マシンは大容量データの処理を目的としているという観点から、複数文を同時に解析した場合の台数効果についても評価を行った。

評価には英語の文10個をサンプルとして使用した。10個の文を構成している単語の数および結果として出力される解析木の数を表1に示す。

計測には、PIMOSの提供しているタイマを使用した。タイマ機能は、フロントエンドにメッセージを送るという方法により実現されている。また、フロントエンドのタイマは、50msの誤差を含む。このため、計測結果にはかなりの誤差が含まれる。

4.1 台数効果

英語の文10個をサンプルとして、解析処理に使用するプロセッサの台数を変えて処理時間を測定した。プロセッサの台数による処理性能の向上率を図2で示す。

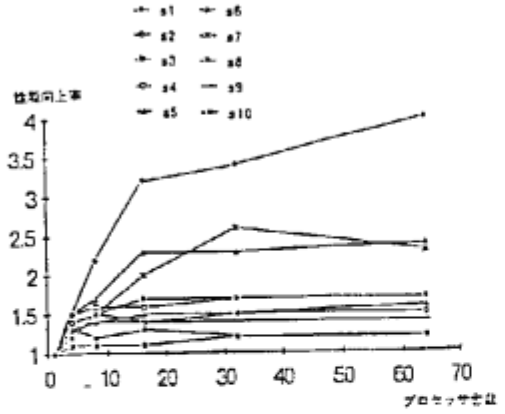

図 2: 性能向上率 (1)

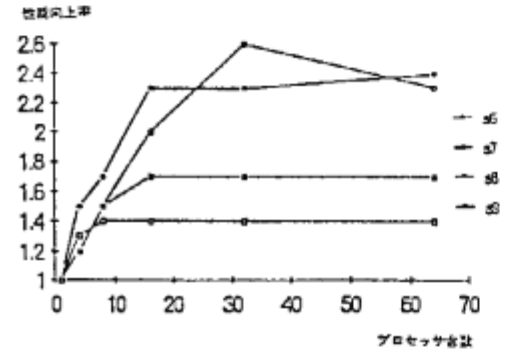

図 3: 性能向上率 (2)

グラフから処理に使用するプロセッサの台数が増えるとそれに比例して処理性能が上がっていることがわかる。しかし、S10 以外は文が短く解析処理の並列度が低いので上昇率はすぐ頭打ちとなる。また、S10 もプロセッサ台数 16 台を境に性能の向上率が下がっている。

S6 から S9 について拡大したグラフが図4である。

S6、S7、S8 および S9 は文の長さはほぼ同じであるが、性能の向上率に差があることがわかる。

4.2 プロセッサごとの仕事量

プロセッサあたりの仕事量を次のように定義した。

(仕事量) = (ストリームを流れたデータの個数) × (ストリームを参照しているプロセスの数)

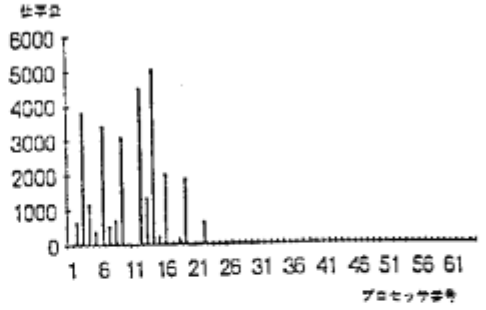

図 4: S7 に対する各プロセッサの仕事量

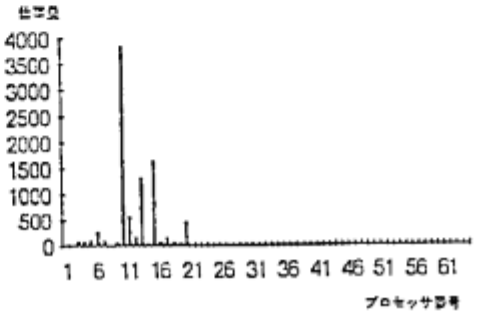

図 5: S9 に対する各プロセッサの仕事量分布

この定義に基づき、各ストリームに対し、そのストリームを流れたデータの個数及びそのストリームを流れるメッセージを読み込んでいるプロセスの数を測定した。

S7、S9、S10 に対する各プロセッサの仕事量をグラフにしたものが図4から図6である。

S7 と S9 を較べると、S7 のほうが負荷が均一である。S9 については総仕事量 9,457 のうち 3,857、約 4 割が 1 つのプロセッサに集中している。S10 に対するバランスも必ずしもよくない。

仕事量とは別に、データの個数とプロセスの数にも相当のばらつきがあることが図7からわかる。

4.3 複数の文に対する台数効果

複数文を同時に処理した場合の台数効果を図8に示す。S10 を 10 個同時に解析した場合 64 プロセッサで 16 倍という高い台数効果がでている。しかし、ここでもプロセッサ台数 16 台を境に性能向上率が下がって

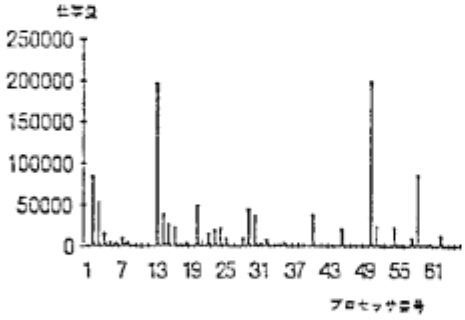

図 6: S10 に対する各プロセッサの仕事量分布

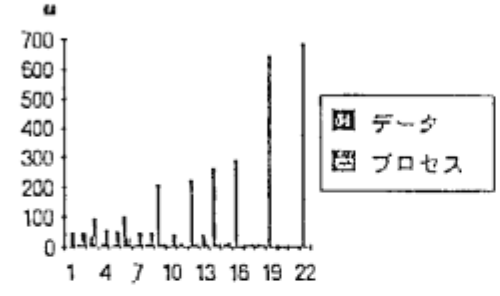

図 7: データおよびプロセスの分布 (S7 の場合)

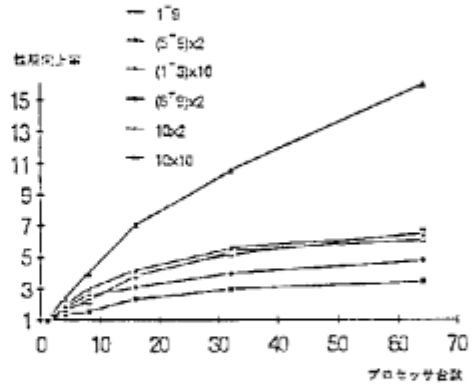

図 8: 複数の文に対する性能向上率

いる。

5 考察

1 文ずつの比較で、S7 は S9 より高い性能向上率を示した。また、S7 のほうが負荷バランスが良いこともわかっている。このことから、特定のプロセッサへの負荷の集中が、台数効果を抑えていることが予想される。

負荷が集中しているところは、文法カテゴリの切れ目である。たとえば、前の例の

```
sentence --> np, vp.
```

で 'np' と 'vp' のきれめである。現在の負荷分散方式だと、このようなところにプロセスが集まるからである。しかし、PAX プログラムは上昇型のアルゴリズムを用いているためこの位置を静的に検出し負荷分散として実現することは難しい。

1 つの文を処理している場合より複数の文を同時に処理している場合のほうが台数効果があらわれているのは、負荷バランスがよくなっていること、負荷の総量が大きくなって各プロセッサの仕事量が全体としてふえたことによると考えられる。

しかし、複数の文に対する処理でも、プロセッサ代数 16 台を境に性能の向上率が小さくなっており、このあたりを境に、処理能力の増大を通信オーバーヘッドの増大が打ち消すようになっていることが予測でき

る。

6 おわりに

1986年に提案された並列構文解析のアルゴリズムをPAXとして並列マシン上に実現した。今回報告したのは、その測定結果の速報である。

実現にあたり問題となったのは、レイヤードストリームと名付けたデータ構造である。レイヤードストリーム法は、PAXで実現した構文解析のアルゴリズムとともに生まれ汎用的な全解探索の手法として研究されている。

今回は、このデータ構造に対しプロセッサ間の通信を極力抑えるという観点からの分散を考えた。しかしこのままでは、負荷のバランスが悪く台数効果が出ないことが測定の結果明らかになった。また、並列度が低く抑えられてしまうという問題もある。

今回の分散方式に対して今後次の2つの観点からの見直しが必要である。

- 通信を抑えつつ並列度を上げる。現在のアルゴリズムでは、問題の大きさに対してストリームの数が少なく並列度が抑えられている。そこで、ストリームの数を増やしたプログラムとすることを考える。これには、1つのストリームを複数のプロセスが握っているとき、そのそれぞれを異なるストリームとして扱うことにするなどの方法が考えられる。

- 負荷のバランスを良くする。現在の負荷分散では、プロセスの数が多く仕事量が増えているところと、データの数が多く仕事量が増えているところがある。このうちプロセスの数が多いところについては、プロセスを分散させて負荷の集中を避けることが可能である。

また、通信が多いことを前提として、通信のコストが全体の効率に影響しないようなプログラミングテクニックおよび問題のクラスにかんしてもの研究も行う必要がある。

並列アルゴリズムについては、レイヤードストリームの利点を残しかつ通信の増えない全解探索アルゴリズムの研究も必要である。

プロセッサ台数16台あたりに、通信オーバーヘッドと処理能力増大の分岐点が本当に存在するのかどうか、存在する場合、それはこのアルゴリズムあるいは負荷分散方式に依存しているのかどうかも今後の研究課題である。

謝辞

日頃ご指導いただくICOT第4研究室の内田俊一室長に感謝します。また、ここで述べた研究にご協力いただいた皆様に感謝します。

参考文献

[1] Y. Matsumoto. A parallel parsing system for natural language analysis. In *Lecture Notes in Computer Science 225*. Springe-Verlag, 1986.

[2] 木村和広他. 談話理解実験システムduals第2版の設計と実装. In 日本ソフトウェア科学会 第3回全国大会論文集, 1986.

[3] A. Okumura and Y. Matsumoto. Parallel programming with layered stream. In *Proceedings of 1987 Symposium on Logic Programming*, 1987.

[4] K. Taki. The parallel software research and development tool : Multi-PSI system. In *Programming of Future Generation Computers*, North-Holland, 1986. Elsevier Science Publishers B.V.

[5] K. Nakajima et al. Distributed Implementation of KL1 on the Multi-PSI/V2. TR 439, ICOT, 1988.

[6] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, Japan, 1988.

[7] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama, and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, Japan, 1988.

マルチ PSI/V2 におけるコード形式について

宮崎芳枝[1], 中越靖行[1], 堂前慶之[1], 稲村雄[2], 木村康則[2], 中島克人[2]
1: (株) 富士通ソーシアルサイエンスラボラトリ
2: (財) 新世代コンピュータ技術開発機構

1 概要

第五世代コンピュータ・プロジェクトの一環としてマルチ PSI/V2 システムを開発している。マルチ PSI/V2 上には並列論理型言語 KL1 の分散処理系が実装されている。本稿ではこの KL1 分散処理系における、プログラムコードの形式ついて述べる。

現在のところコード生成はマルチ PSI/V2 のフロントエンド・プロセッサである PSI-II 上のクロスシステムでおこない、フロントエンド・プロセッサからマルチ PSI 本体内の要素プロセッサの 1 つにロードされる。その後コードは実行時に必要に応じて各要素プロセッサに分散される。

2 モジュール形式

2.1 モジュール

KL1 プログラムは述語の集まりであるモジュールという単位にまとめて記述されている。KL1 プログラムはモジュール単位にコンパイルされ、それらがリンクされて、ロードされる。デマンドロードやロードしたコードが不要になった時のガーベジコレクション (以下 GC) を容易にするため、モジュールは KL1 の通常のデータ型の一つとして扱われる。

2.2 GC 領域とコード領域

述語を呼び出すために張られたポインタは GC 時には張り直さなければならない。ポインタにはモジュール内の述語同士に張られたものとモジュール外の述語に張られたものがある。

モジュール内の述語呼出しに関してはポインタを相対アドレスで表現することにしておけば、他のアトミックなデータと同様に扱うことができて GC 時のメンテナンスは不要になる。GC 時にはモジュール単位にコードが移動するためモジュール外の述語呼出しのポインタと構造体定数 (3 章) はメンテナンスの対象となる。このような絶対アドレスポインタがモジュール中に占める比率は小さい。もし、絶対アドレスポインタをモジュール内の一箇所に集めることが出来れば、GC 時にはモジュール全体をメンテナンスの対象とする必要は無くなり、効率が良い。そこで、モジュールを絶対アドレスポインタを含む領域 (GC 領域) とそれ以外の領域 (コード領域) の 2 つに分けた。具体的には、GC 領域にはモジュール外述語呼出し用の述語記述子 (2.3 章) と構造体定数への絶対アドレスポインタが格納される。コード領域には命令語、モジュール内相対アドレス、整数・アトムなどのアトミック・データのみが入っている。

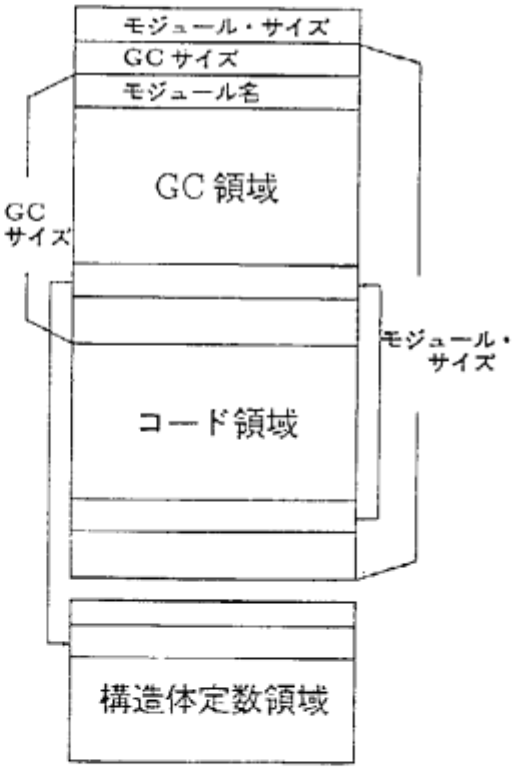

図 1: モジュールの形式と構造体定数領域

Code representation of Multi-PSI/V2
Yoshie MIYAZAKI [1], Yasuyuki NAKAGOSHI [1], Yoshiyuki DOUMAE [1],
Yu INAMURA [2], Yasunori KIMURA [2], Katsuto NAKAJIMA [2]
1: Fujitsu Social Science Laboratory Ltd. 2: Institute for New Generation Computer Technology

2.3 モジュール外呼び出し

モジュール外述語の呼出しは、モジュールへのポインタ、述語名アトム、引数個数の3つの情報を用いて行なわれる。呼び出される側のモジュールのコード領域の先頭には述語エントリ表と呼ばれるハッシュ表が置かれる。述語名アトムと引数個数をエンコードした値をキーとしてこの表を検索することにより、呼び出すべき述語のコードアドレスを求めることができる。

呼出し側モジュールには各呼出し述語に対して4語の述語記述子がGC領域中に設けられ、以下のような情報が格納される。

- モジュールフィールド
  呼出し先モジュールへのポインタを格納する。
- 述語名フィールド
  呼出し先モジュールの述語エントリ表を検索するためのキー、即ち、呼出し述語の述語名アトムと引数個数をエンコードした値を格納する。
- モジュール名フィールド
  呼出し先モジュールのモジュール名アトムを格納する。これはデバッグ用である。

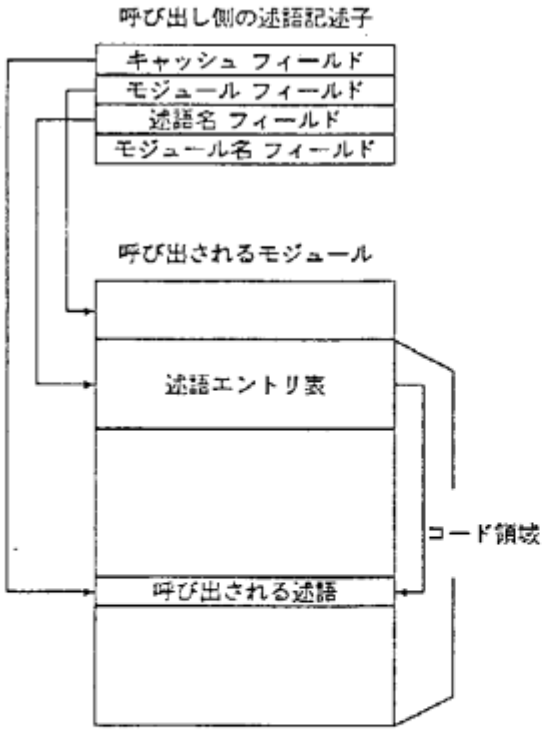

図 2: モジュール外呼び出し

- キャッシュフィールド
  モジュール外呼出し述語のコードアドレスを直接格納する。一度呼出した述語のアドレスをこのフィールドにキャッシュしておくことにより、2回目以降の呼出しを高速化することができる。GC時にはこのフィールドはクリアされる。

3 構造体定数の処理

3.1 構造体定数

構造体定数とは、プログラム中に陽に記述される構造体の中で、その要素として変数を含まないものの総称である。例えば、ベクタ{a, b}や8ビットストリング ascii#"program"、あるいはネストした構造体 [a,{b, c}, d]などである。WAMをベースにした処理系では構造体は実行時に生成されるが、変数を含まない構造体定数の場合はコンパイル時に予め生成しておくことができるため、実行時間および生成のための命令コードの両方の節約が可能である。定数表を検索するようなプログラムでは特に効果が大きい。

3.2 構造体定数の生成

コンパイラはソースプログラムに構造体定数を見つけると、構造体をモジュールの外部に生成し、その構造体への絶対アドレスポインタをモジュールのGC領域に置く(図1参照)。そしてコード中のその構造体を使用する箇所には、

```
put_structured_constant    Ai, Lab
```

なる命令を配置する。この命令は構造体定数へのポインタを引数レジスタAiにロードするためのもので、Labは GC領域中の構造体へのポインタの格納箇所を示す相対ポインタである。

謝辞

日頃御指導頂いている内田室長をはじめとするICOT第4研究室の方々に感謝します。

参考文献

[1] Y.Kimura and T.Chikayama: An Abstract KL1 Machine and Its Instruction Set, Proc. of SLP'87

[2] K.Nakajima et al.: Distributed Implementation of KL1 on the Multi-PSI/V2, ICOT TR-439

[3] D.H.D.Warren: An Abstract Prolog Instruction Set, Technical Note 309, Artifical Intelligence Center, SRI, 1983