OR 並列化と動的負荷分散による並列構文解析システム PAX の改良

津邑 公暁

内容梗概

実用的な自然言語処理システムでは, システムの応答時間の短縮が重要な課題のひとつである。なかでも, 比較的アルゴリズムの確立している構文解析の並列化が, 近年活発に研究されてきた。本研究の対象である PAX は並列構文解析システムの一つである。PAX は, DCG で記述された文法規則を並列論理型言語 KL1 のプログラムに変換することによって構文解析が実現される。

PAX では, 並列型ボトムアップ解析にトップダウン的予測を併用したアルゴリズムが用いられている。この方法は, バックトラックや副作用が生じないため, 並列処理に向いている。PAX では最初プログラム起動時に, 各単語に対応するプロセスを 1 つのプロセッサに割り当てることによって, AND 並列性による並列処理を実現している。

しかし PAX の構文解析処理では非常に大量のプロセッサ間通信が生じるため, 複数のプロセッサを用いても処理速度があまり向上しなかった。そこでデータとプロセスの割り当てを逆転させることによって, プロセッサ間通信を減少させる改良法が提案されたが, この PAX も負荷分散はオリジナルの PAX と同じ方法を採用している。

つまり, これら 2 つの PAX で用いられている負荷分散方式は, AND 並列性にのみ基づくものである。よって利用可能なプロセッサ台数に関わらず単語数分の並列性しか得られない上, プロセッサ間にまたがるデータの依存性が高いことで, ゴールの中断回数およびプロセッサ間通信数が多くなっている。

そこで今回, この通信量を減少させた後者の PAX に対し, OR 並列性に基づく静的負荷分散と, 同じく OR 並列性に基づき, マスタスレーブ方式を導入した動的負荷分散の 2 種類の手法を取り入れることによって, より効率的な並列実行を目指した。具体的には上記の, 通信数減少が施される前後それぞれの PAX を AP1000 版 KLIC に移植した上で, OR 並列性による静的負荷分散と動的負荷分散の 2 種類の方式を実装, 評価した。

PAX アルゴリズムには 2 階層の OR 並列性が存在する。ひとつは処理の前段階までに構成された部分木, すなわち解の候補が複数個存在することから生じる並列性であり, もうひとつはある部分木から新しい部分木を構成する方法に複数の可能性が存在することから生じる並列性である。

静的負荷分散ではその前者の並列性による分岐数を予測することが困難であるため, 後者の並列性のみを用いた。また動的負荷分散では, ネックになると予想されるマスタの通信量を抑えるため, マスタは直接は仕事の実体を扱うことなく, スレーブ同士の間で仕事をやり取りさせる方式を用いた。

これら OR 並列性に基づく静的/動的負荷分散方式の効果を AP1000 版 KLIC 上で評価した結果, プロセッサ間依存の低下から, もとの AND 並列による PAX に較べてゴールの中断回数, メッセージ数ともに大きく減少した。これによって処理対象の文に含まれる単語数やプロセッサ間のデータ依存に縛られることのない高い並列性を引き出すことに成功し, 処理の高速化を実現できた。

文法規則約 500, 単語数約 200 からなる構文において, 単語数 20 からなる文を 20 台のプロセッサを用いて処理させた結果, OR 並列静的負荷分散版 PAX では, もとの通信が低減された PAX に対して, ゴール中断回数, メッセージ数ともに約 1/4 に減少した。また同じ 20 台において探索時間では 1.4 倍の速度向上が得られた。

An Improvement of the Parallel Parsing System: PAX by OR-parallelizing and Dynamic Load-Balancing

Tomoaki TSUMURA

Abstract

One of the most important problem in the natural language processing (NLP) to make it of practical use is to shorten the response time of system. Lately, parallel processing has been introduced into parsing system, which has relatively established algorithm among the research fields of NLP. The *PAX* performs parsing by translating Definite Clause Grammar (DCG) into KL1 program.

The basic algorithm used by the PAX is the *left-corner parsing*, in which phrases are constructed from bottom to top and from left to right. This algorithm is suitable for parallel processing, because it neither use backtrack nor cause side effect. On the PAX, the process corresponding to each word is assigned to one processor at the beginning of analysis, and the processes are conducted in parallel.

However, the PAX produces so many communications among the processors that it cannot use processors efficiently. There was a proposition of the improvement to decrease the communications in the PAX, but this *reformed PAX* adopts the same way on the parallelizing as the original PAX.

After all, the parallelizing form adopted by these PAX's are based only on the AND-parallelism. As a consequence, there is no way to use processors that outnumber words in sentence efficiently. What is worse, data on each processors are tightly interdependent.

Hence, we made parallel processing in the PAX algorithm more efficient, using static load-balancing and dynamic one by master-slave method. After converting two PAX's abovementioned to KLIC on AP1000, we implemented and evaluated the two above methods, static one and dynamic one based on OR-parallelism.

The OR-parallelism of the PAX consists of two layers, but we use only one since the other is inapplicable to static load-balancing. On the master-slave, in order to avoid excess of processing in the master-process, which is supposed to be bottleneck, it is not directly assigned to manage jobs. This task is assigned to be done among the slave processes.

We evaluate the performance of these methods with the DCG which have about 500 syntax rules and about 200 words. We confirm that the goal suspensions and the inter-PE communications fall off when using OR-parallelism. Because our algorithm is free from the number of words in sentence and from the interdependence among the processors. Therefore, we get effective use of processors and speedup, in the case the sentence, the target of parsing, has certain measure of length and up. To put it concretely, we obtain the following results.

Execution Speed:

For the sentence contains 20 words, we get about 1.4 times speed up.

The numbers of suspensions and Inter-PE communications:

Both the number of suspensions and the number of Inter-PE communications become about 1/4 of the *reformed PAX*.

1　はじめに

自然言語処理は多大な計算量を必要とし, 処理速度の向上が望まれる分野のひとつである。中でも比較的アルゴリズムの確立している構文解析の並列化が, 近年活発に研究されてきた。新世代コンピュータ開発機構 (ICOT) で研究された並列構文解析システム PAX(Parallel Analizer for syntaX and semantiCS) [1] もそのひとつで, 逐次型の解析システム SAX(Sequential Analizer for syntaX and semantiCS) [2] の並列版である。

PAX は並列論理型言語 KL1 [3] で記述されたプログラムである。PAX の構文解析アルゴリズムは, 並列性を持っている上にバックトラックおよび副作用を伴わないという特長を持つが, その通信量が非常に多いため, 疎結合並列計算機上では複数のプロセッサを用いても処理速度があまり向上しない。そこでこの PAX のプロセッサ間通信を減少させる改良が行われた [4]。以下これを通信低減版 PAX と呼び, 前者をオリジナル PAX と呼ぶ。

PAX のアルゴリズムには, 共通部分を持たない部分木を独立に構成する AND 並列性と, 解析途中の曖昧性をすべて保持する OR 並列性がある。オリジナル PAX と通信低減版 PAX は共に, 文の各単語に相当するプロセスに対し, 起動時にプロセッサを割り当てることで並列処理を実現している。即ち, AND 並列性のみを利用しており, OR 並列性には部分解の候補集合をプロセス間でやり取りすることで対応している。しかしこの方法では利用可能なプロセッサ数に関わらず, 最大でも単語数分の並列性しか出ない。またストリームから送られてくるデータを待つ必要のあるプロセッサや, 処理が早く終わって仕事のないプロセッサが存在するため, プロセッサの利用効率が悪くなってしまう。

そこで今回, 上記の問題を解決するため, 通信量の減少した通信低減版 PAX に対し OR 並列性による負荷分散とマスタスレーブ方式による動的負荷分散を導入し, AP1000 版の KLIC [5, 6] 上でその評価を行った。

以下, 次章で背景として並列構文解析と KL1 について説明し, 第 3章でオリジナル PAX と通信低減版 PAX による解析方法を示したのち, 第 4章で今回の改良方式を説明する。続いて第 5章で評価結果を述べ, 第 6章で結論を述べる。

2　背景

本章では, 並列構文解析と今回のターゲット PAX について簡単に触れたのち, PAX の記述言語である並列論理型言語 KL1 の概要を説明する。

2.1　並列構文解析

構文解析において非決定性を扱う方法は 2 種類存在する。一方は後戻り法と呼ばれるもので, 統語構造に関する仮説を一時に高々1 つしか保持せず, それと両立しない仮説を探索する際には以前の仮説を捨てる方法である。これに対し並列構文解析法 (並行型統語解析) は, 互いに必ずしも両立しない複数の仮説を一時に保持する方法である。並列構文解析法の代表的な例としては, CYK 法 [7] やチャート法 [8] などが挙げられる。

(1) s → np, vp.
(2) vp → a, n.
(3) np → prp, n.
(4) np → np, conj, np.
(5) vp → v, a.
(6) vp → v, av.
(7) vp → v, np.
(8) np → [they].
(9) a → [hard].
(10) av → [hard].
(11) v → [looked].

図 1: DCG による簡単な構文規則の例

図 1のような文法記述形式を, DCG (Definite Clause Grammars) [9] と呼ぶ。DCG は確定節をそのまま句構造規則として用いることによって記述した文法である。この DCG は Horn 節論理を基礎としており, Prolog プログラムによって補強される文脈自由文法と考えることができる。DCG は構文解析のプログラムとしてだけでなく, 自然言語の文法記述言語として高い能力を備えた記述形式である。

DCG によって記述された文法規則を, Prolog の節に一対一に変換すれば, 下降型の構文解析システムを得ることができる。しかし下降型構文解析では, 左再帰規則によって解析が無限ループに陥

る可能性があることに加え, 繰り返し計算を行うため効率上の問題がある。

この欠点を回避すべく, DCG で記述された文法規則を上昇型の構文解析プログラムに変換する方法 BUP(a Bottom-Up parser embedded in Prolog) [10, 11] が考案された。これと同様に, DCG で記述された文法規則を Prolog プログラムに変換することによって上昇型構文解析システムを実現するのが逐次型構文解析システム SAX(Sequential Analyzer for syntaX and semantiCS) であり, この並列版が ICOT で研究された PAX(Parallel Analyzer for syntaX and semantiCS) である。PAX は, DCG に識別子を付加するという手段を用いて上昇型のチャート法を並列論理型言語で実現したものとみなすことができる。

2.2 並列論理型言語 KL1

並列論理型言語 KL1 は, 第五世代計算機計画の核言語として開発され, シンタクスが単純であることや並列性を自然に記述できることなどの特徴を持つ。

PAX はこの KL1 を記述言語としており, 解析中に現れる曖昧性を保持するのに, 後述するストリームを層状にして用いている。

2.2.1 プログラムの構造と実行

KL1 は Flat GHC に基づく言語で, 以下の形をした節の集合で表される。

$$H\ :-\ G_1, ..., G_m\ \mid\ B_1, ..., B_n.$$

H,G,B は, それぞれクローズヘッド, ガードゴール, ボディゴールと呼ばれる。

クローズヘッドには引数を含む述語名を記し, 当節がその述語についての定義の一部であることを示す。他の変数と同様, 引数は各節についてローカルである。ガードゴールには当節が適用されるための条件を記す。要素すべての条件が成立したときガード部は成功したとみなされる。ボディゴールには当節が適用されたとき (ガード部が成功したとき) に起動すべきゴールを記す。

入力変数に対する単一化 (unification) ガードゴールは, ヘッド部に組み込むことができる。つまり以下の 2 つは同じ意味である。

- `p(In, Out) :- In = 0 | Out = 1.`
- `p(0, Out) :- Out = 1.`

述語は "述語名/引数の個数" で表記され, 初期ゴールは `main/0` で与えられる。

実行中のゴールのヘッドやガードで未具体化変数への参照が行われると, そのゴールは実行を中断 (suspend) し, 別のゴールが選択されて処理が継続される。中断されたゴールは, その変数が具体化されると実行を再開する。

```
main :- producer(1, X),
        consumer(X, 0)@node(1).

producer(Num, X) :- Num < 10 |
        X = [Num|NewX],
        NewNum := Num + 1,
        producer(NewNum, NewX).
producer(Num, X) :- Num >= 10 |
        X = [].

consumer([X|Rest], Y) :-
        NewY := Y + X,
        consumer(Rest, NewY).
consumer([], Y) :-
        print(Y).
```

図 2: KL1 プログラムの例

KL1 プログラムでは, 述語のボディゴールとして自身の述語を書く (自分自身を再帰的に呼び出す) ことによって, ある条件の満たされる間存在し続けるようなプロセスを実現することができる。例えば図 2のプログラムにおいて `producer/2` は, カウンタとなる第 1 引数が 10 未満である間存在し続け, それが 10 になった時点で消滅するようなプロセスである。

KL1 で負荷分散を行うには, 明示的にプログラム中にこれを指定しなければならない。`goal(`$Args$`)@node(`N_c`)` という形のボディゴールがノード N_p上に現れると, このゴールはノード N_c上に送られ (throw goal), そこで実行される。

図 2の例では`main`のボディゴール`consumer/2`が送信対象となっており, この`consumer/2`はノード 1 に送られたのち, 以下そこで実行される。すべてのノードについて実行可能なゴールがなくなると, プログラム全体の実行が終了する。

2.2.2　データ型

KL1 のデータ構造は大きく 3 つに分けられる。

論理変数 論理変数は参照ポインタで表現され, 特定の型を持たない。具体化によって対象へのポインタが格納される。論理変数は一度具体化されると, 別の値に再び具体化することはできない。

アトミックデータ型 アトミックデータは内部構造を持たず, 値そのものが意味を持つ。アトミックデータ型にはシンボリックアトムや整数がある。

構造データ型 構造データは複数のセルからなり, これらのセルには任意の型のデータ (またはデータへのポインタ) を格納することができる。構造データ型にはファンクタ, リスト, ベクタなどがある。

ここで, 構造データ型のひとつであるリストについて説明する。リストはデータ型 cons によって構成される。cons は car, cdr という 2 つの要素からなり, プログラムでは `[car|cdr]` と表記する。car, cdr はそれぞれ任意の型をとることができる。cons がリストを構成するのに使われる場合は, car がリストの 1 要素を表し, cdr はリスト内の残りを表す。空のリストはアトム `[]` で表される。

2.2.3　ストリーム

KL1 プログラムではリスト変数を用いてプロセス間の通信路を実現している。これをストリームと呼ぶ。

図 2は, 10 未満の自然数の総和を計算するプログラムである。このプログラムは, 1 から 9 を要素とするようなリストを作成する`producer/2`と, リストの要素を順次加算していく`consumer/2`の 2 つの述語からなる。この例では共有変数 `X` が, `producer/2` の出力を `consumer/2` の入力につなぐストリームとなっている。

`producer/2`の実行途中ではリスト変数の要素は一部しか決定していないが, このリストの第 1 要素さえ決定済であれば`consumer/2`のボディは実行可能である。つまり 2 つのプロセス`producer/2`, `consumer/2`がそれぞれ "リスト変数の car 部に対して処理を行い, その cdr 部を新しい共有変数とする" という操作を繰り返すことによって, この共有変数がストリームとして働く。この場合リストの各要素はプロセス間でやり取りされるメッセージである。ストリームは "`[]`"(nil atom) と同一化されることによって閉じられる。

ストリームを入出力とするようなフィルタ, マージャ等をプログラムで構成することにより, ストリームを介したプロセス間通信を実現することができ, より複雑なプログラムを構成することが可能である。

KL1 には組み込み述語として, 入力ストリーム数が動的に変化する場合にも対応したマージャ`merger/2`が用意されている。この組み込みマージャの入力を増やしたい場合は, その入力変数を複数要素のベクタで同一化すればよい。例えば`merger(In, Out)`というマージャに対して,

$$\texttt{In = }\{\texttt{NewIn1, NewIn2, ..., NewIn}n\}\texttt{,}$$

とすることによって入力 `In` の代わりに入力 `NewIn1`, `NewIn2`, ..., `NewIn`n が利用できるようになり, 結果としてマージャの入力数が $n-1$ だけ増加する。

3　オリジナル PAX と通信低減版 PAX

本章では, オリジナル PAX と通信低減版 PAX の解析法と, その負荷分散方式について述べる。

3.1　オリジナル PAX による解析法

PAX は, 逐次型構文解析システム SAX の並列版である。SAX アルゴリズムはボトムアップを基本とし, これにトップダウン的予測を取り入れた左隅解析法 [12] と呼ばれる方法によって解析を行う。

PAX はある “まとまった非終端記号や終端記号”(以下, カテゴリと呼ぶ) が構成されると, それをもとにしてより大きな構文木を構成しようとする。この際に, そのカテゴリを左隅の要素とするような新しい解析木を作ろうとする。つまり, このカテゴリを右辺の先頭要素として持つような文法規則を選び, その文法規則を完成させせようとする。このような文法規則の右辺の 2 番目以降の要素は, 新しいカテゴリが得られる度に埋め合わされ, 完全な解析木へと作り上げられていく。

このような処理を論理型言語で実現するために, 文法規則の右辺に位置を示す識別子を設ける。図 1 の DCG にこの識別子を導入したものを図 3に示す。ただし各非終端記号に付加した番号は説明のためのものである。

(1) s → np_1, **id1**, vp_1.
(2) np_2 → a_1, **id2**, n_1.
(3) np_3 → prp_1, **id3**, n_2.
(4) np_4 → np_5, **id4**, $conj_1$, **id5**, np_6.
(5) vp_3 → v_1, **id6**, a_2.
(6) vp_4 → v_2, **id7**, av_1.
(7) vp_5 → v_3, **id8**, np_7.
(8) np_6 → [they].
(9) a_3 → [hard].
(10) av_2 → [hard].
(11) v_4 → [looked].

図 3: DCG による簡単な構文規則の例 (識別子挿入済み)

図 3の例では **id1** から **id8** が識別子であり, 文法規則内の特定の位置と一対一に対応する。即ち, これらの識別子によって, 現在の解析状態を一意に表すことができる。たとえば id1 は文法規則 (1) の np_1まで解析が終了した状態に対応している。PAX では解析されたカテゴリに対応する KL1 のプロセスが必ず存在し, 割り当てられた識別子をそれらプロセスがストリームを介して受け渡しすることによって解析が進められてゆく。解析が進むと曖昧性に応じてそのストリームは多重化してゆくため, この方法は層状ストリーム法と呼ばれる [13, 14]。層状ストリームは一般に問題の部分解の集合を表すと考えられる。

各カテゴリは, 前からの情報を受け取る入力と, 次のプロセスへ情報を渡す出力の 2 引数を持つプロセスで表される。PAX でのカテゴリの完成は, そのカテゴリに対応するプロセスの起動に対応する。

例えば図 3の構文規則においてカテゴリ np が完成したとする。np を右辺の先頭に持つ文法は (1),(4) であるから, これらに対応する処理を行うため, プロセス np1 は id1, id4 の 2 つの識別子を出力する。

```
    np1(In, Out) :-
            Out = [id1(In),id4(In)].
```

また, 右辺の先頭以外に存在する np に対応する処理も必要である。図 3の規則において np が右辺の先頭以外に現れるのは (4),(7) であり, これらに対応する処理を KL1 の述語で表すと以下のようになる。

```
(a)    np2([id5(In1)|In], Out) :-
(b)            np(In1, Out1),
(c)            np2(In, Out2),
(d)            Out = {Out1,Out2}.
(e)    np2([id8(In1)|In], Out) :-
(f)            vp(In1, Out1),
(g)            np2(In, Out2),
(h)            Out = {Out1,Out2}.
(i)    np2([], Out) :- Out = [].
(j)    otherwise.
(k)    np2([_|In], Out) :-
(l)            np(In, Out).
```

ここで np_6に対応するのが (a) である。np_6は識別子 id5 を受け取ると文法 (4) が完成したとみなし, 左辺の np_4に対応するプロセスを (b) で起動する。そしてストリームの残りを (c) で再帰的に np2/2 に渡すことによって, 受け取る部分解候補のすべてに対して同様の処理を行う。そしてそれらのプロセスの出力は (d) でひとつのストリームにマージされる。同様に np_7は (e) に対応する。

受け取る部分解のすべてに対して処理が終了すると (i) にマッチし, 出力が閉じられる。受け取った部分解が自分と整合性の取れないものであったときには (k) によってその候補は捨てられ, 残りの部分解候補についての処理が (l) で続行される。

これら np1, np2 の述語の定義を np(名詞句) に対する Type1 述語, Type2 述語と呼ぶことにする。

np が完成するとプロセス np が起動されるが, 述語 np は np1 と np2 によって以下のように定義される。

```
np(In, Out) :-
        np1(In, Out1),
        np2(In, Out2),
        Out = {Out1,Out2}.
```

構文規則は, すべてのカテゴリについて上記のような KL1 プログラムに変換される。

Type1 の述語は識別子の集合を生成し, Type2 の述語はこのような識別子を取り込んで他の新たな識別子を出力したり他のカテゴリに対応する述語を起動したりする。このようにして出力された識別子の集合はすべてマージされ, 1 つのストリームにまとめられる。

例として “They looked hard.” という文を構文解析することを考える。この場合初期ゴールは以下のようになる。

```
main :- they([begin], X), merge(X, X1),
            looked(X1, Y), merge(Y, Y1),
            hard(Y1, Z), merge(Z, Z1),
            fin1(Z1).
```

ここで [begin] は文の先頭を示す識別子であり, 最もトップレベルのカテゴリ s はこれを受け取ると文末を表す識別子 [end] を生成する。fin1/1 がこの [end] を受け取ると文全体が s として認識されたことになり, 解析が成功する。

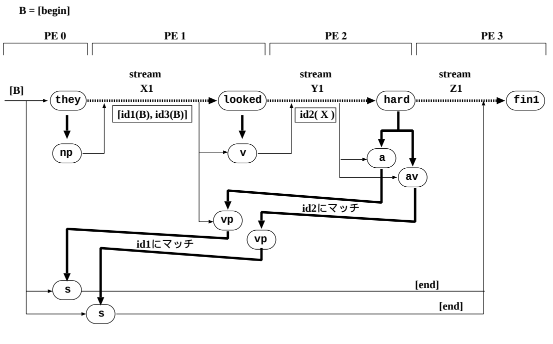


図 4: PAX による解析例

またプロセス間を結ぶマージャはこの時点では 1 入力 1 出力であるが, 解析が進むにつれ 2.2.3で述べた方法によってマージャへの入力数は変化する。

3.2 通信低減版 PAX による解析法

オリジナル PAX は, その通信量の多さゆえ並列化による効果があまり現れないという問題がある。そこでこの通信量を減少させる方法として考えられたのが通信低減版 PAX である。

オリジナル PAX では, 現在までの解析状況をデータ (識別子) で表し, 各カテゴリをプロセスとして表していた。これに対し通信低減版 PAX は, 解析状況にプロセスを対応させ, それらの間でカテゴリをデータとしてやり取りさせる。つまりオリジナル PAX のデータとプロセスを逆転させた方式である。

オリジナル PAX では, あるカテゴリが解析されると, それが文法規則中に複数箇所現れる場合, それぞれに対応する識別子を複数個データとして送る必要がある。これに対し通信低減版 PAX では 1 つのカテゴリをデータとして送るだけで済み, 通信量が減少する。

通信低減版 PAX は基本的に右隅解析法であり, あるカテゴリが解析されると左のプロセスに自分がどういうカテゴリであるかを知らせる。文法規則の右辺左隅のカテゴリに相当するプロセスは, 自分の右にどのようなカテゴリを許すかを知っており, そのカテゴリが送られてくると更にその右がどういうカテゴリであるかを調べることによって解析を進めていく。

図 3の構文規則において今カテゴリ np が完成したとする。np が右辺左隅に存在する文法は (1),(4) であり, それぞれ右に vp, conj を許す。よって np に対応する述語は以下のようになる。

```
(a)     np([vp(In1)|In],Out) :-
(b)             Out = [s(In1)|Out1],
(c)             np(In, Out1).
(d)     np([conj(In1)|In], Out) :-
(e)             np_conj(In1, Out1),
(f)             np(In, Out2),
(g)             Out = {Out1, Out2}.
(h)     np([], Out) :- Out = [].
(i)     otherwise.
(j)     np([_|In], Out) :-
```

```
(k)             np(In, Out).
```

（e）の np_conj は更にその右のカテゴリを調べるプロセスで, オリジナル PAX の識別子 id5 に対応する。このプロセスは右に np$_5$を許し, np$_3$に還元する。

```
(l)     np_conj([np(In1)|In], Out) :-
(m)             np(In1, Out3),
(n)             Out1 = [np(In1)|Out3],
(o)             np_conj(In, Out2),
(p)             Out = {Out1, Out2}.
(q)     np_conj([], Out) :- Out = [].
(r)     otherwise.
(s)     np_conj([_|In], Out) :-
(t)             np_conj(In, Out).
```

この際, 還元後の非終端記号 np は図 3の構文規則において右辺の左隅にも左隅以外にも現れるので,（m）でプロセス np を起動するとともに,（n）ではストリームを介して自分が np であることを次のプロセスに知らせている。

3.1で用いたものと同じ例文を解析する場合, 初期ゴールは以下のようになる。

```
main :- hard([begin], Z),
        merge(Z, Z1), looked(Z1, Y),
        merge(Y, Y1), they(Y1, X),
        merge(X, X1), fin2(X1).
```

通信低減版 PAX は右隅解析法であるから, オリジナル PAX の場合とデータの流れが逆になっている。またストリームを流れるデータはカテゴリであるため, `fin2/1` が `s([begin])` を受け取ったときに解析は成功する。

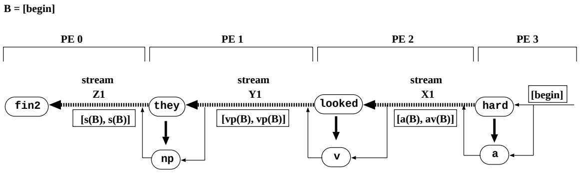


図 5: 通信低減版 PAX による解析例

3.3 PAX の負荷分散方式とその問題点

オリジナル PAX と通信低減版 PAX の負荷分散方式は, まず単語間を結ぶストリームにプロセッサを割り付け, 最初に起動する単語に相当するプロセスを, そのプロセスへの入力ストリームと同じプロセッサ上で起動する。以降各プロセスは, ストリームからデータを読み込もうとする際には, そのストリームに移動してからデータを読み込む (図 4)。

これは入力ストリームからデータを読んで出力ストリームにデータを送る働きをするプロセスが多い PAX において, 多対多の通信が起きるのを回避するために考え出された方法である。PAX ではストリームを流れるデータが非常に大きいため, データの移動よりもプロセスの移動の方がコストが小さい場合が多い。この方法を用いることによって同じストリームを読もうとするプロセスはすべて 1 つのプロセッサに集まるため, 通信は多対 1 となる。

しかしこの方法はあくまで AND 並列性のみを採用した負荷分散方式であるため, 最大でも単語数分の並列性しかなく, 利用可能なプロセッサ数が多いときにはプロセッサの利用効率が悪くなる。また, AND 並列性を採用しているためゴール間のデータ依存性が高くなり, ゴールの中断も頻繁に起こると考えられる。

4 改良方式の提案

本章では静的負荷分散を考えるときに問題となる, PAX が持つ OR 並列の 2 層性について触れたのち, 静的/動的負荷分散の方法について述べる。

4.1 OR 並列の 2 層性

PAX および通信低減版 PAX のアルゴリズムには以下の 2 種類の並列性がある。

AND 並列 独立した非終端記号を生成する過程における並列性。

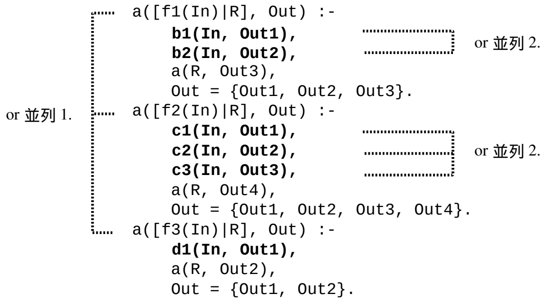
OR 並列 1 つの非終端記号を生成する際に, 複数の可能性が存在する場合, それらの可能性間における並列性。

この OR 並列性を更に詳しく見ると, 以下のような 2 層性が存在する。

OR 並列性 1. ストリームを通して受け取る部分解析木の候補が, 複数存在する場合。

OR 並列性 2. ある部分解析木の候補を受け取ったとき, 新しい部分解析木の生成に複数の可能性が存在する場合。

これを通信低減版 PAX の述語について示したのが図 6である。



```
a([f1(In)|R], Out) :-
    b1(In, Out1),
    b2(In, Out2),
    a(R, Out3),
    Out = {Out1, Out2, Out3}.
a([f2(In)|R], Out) :-
    c1(In, Out1),
    c2(In, Out2),
    c3(In, Out3),
    a(R, Out4),
    Out = {Out1, Out2, Out3, Out4}.
a([f3(In)|R], Out) :-
    d1(In, Out1),
    a(R, Out2),
    Out = {Out1, Out2}.
```

図 6: PAX プログラムにおける OR 並列の 2 層性

4.2 静的負荷分散

PAX プログラムの OR 並列性による静的負荷分散を考える。前述のように PAX アルゴリズムの OR 並列性は 2 層性を持っており, その両方の並列性を用いて負荷分散を行うのが理想である。しかし OR 並列性 1 に関しては, ストリームを通して受け取る部分解の個数が予測不可能である上, それら部分解のうちどれだけのものがそれ以降の解析に利用可能であるかも, 実行してみないと分からない。

例えば図 6における述語 `a` に対し, 入力ストリームに `f1`, `f2` という 2 つのデータ (前段階までにおける部分解の候補) が送られてくる場合を考える。結果的にはプロセス `a` は 5 つのプロセス `b1`, `b2`, `c1`, `c2`, `c3` を新たに起動することになるが, ストリームの第 1 要素である `f1` を受け取った時点では, とりあえず 2 つの新たなプロセス `b1`, `b2` を起動する, ということしか分からない (図 7)。このことから, OR 並列性 1 を用いて静的負荷分散を行うのは不可能であるので, 今回は OR 並列性 2 のみによる静的負荷分散を考えた。

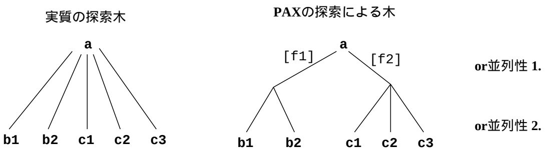


図 7: 解の探索木

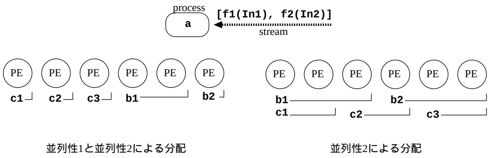


図 8: 静的負荷分散の例

具体的には, 各プロセスは 2 つの引数 `PEf`, `PEt` により自分とその子孫が利用可能なノードの範囲が PEfから $PEt-1$ であることを知らされる (ただし $PEf < PEt$)。最もトップレベルのゴールでは $PEf = 0$, $PEt = Num\_of\_Nodes$ である。

各ゴールは, PEfとノード番号が一致するプロセッサ上で実行され, 子のプロセスを起動するときに自らの利用可能範囲 (PEfから PEt) を子に分配する。それ以上分配できなくなる ($PEf+1 = PEt$) と, 以下の子孫プロセスはすべて同一ノード上で実行される。

例えば図 6の述語 `a` に対しては, `f1` を受け取ったとき起動する子プロセスの数は 2 であるので, `a` が PEfから PEt までを利用可能だとすると `b1` の利用可能範囲を PEfから $PEf + \frac{PEt-PEf}{2}$とし, `b2` の利用可能範囲を $PEf + \frac{PEt-PEf}{2}$から PEt とする。

この方法を用いるときの問題点としては, OR 並列性 1. を利用しないため, プロセッサの割り当てが図 8 右のように多重化してしまうことが挙げられる。また再負荷分散を行わないため, 割り当てられた仕事が短時間で終わってしまったプロセスは, その後全処理終了まで仕事を行わないことになってしまう。

4.3 マスタスレーブによる動的負荷分散

静的負荷分散は負荷分散のための処理が少なくて済むが, 再負荷分散を行わないため, 仕事の早く終わったプロセッサは実行終了時まで処理を行わないことになる。構文解析における解の探索問題では探索木が対称性のある形になることは稀であると考えられるため, この方式ではプロセッサの利用効率が悪くなることが予想される。これを解消するため, マスタスレーブ方式による動的負荷分散を考える。

4.3.1 基本仕様

マスタスレーブは, 仕事の分配を行う master と呼ばれるプロセスが 1 つ存在し, そのプロセスが仕事のない各 slave プロセスに適宜仕事を割り当ててゆく方式である。

この方式を用いるときの問題点としては, プロセッサ数が非常に大きくなった場合に master の仕事がネックになることが予想される。これを回避するため, master は仕事の管理を行うのではなく必要とするノードに idle なノードを紹介するのみで, 実際の仕事のやり取りは slave 同士の間で行われるよう工夫を施した。これによって master-slave 間の通信もそれほど大きくなることはなく, 今回実装を行った 64 セル構成の AP1000 という環境では, この点に関しては問題がないと思われる。また今回用いる方式では master は, ノードの紹介が終わったあとは slave 間通信の完了を待たず次のノード紹介を行える。よって slave 同士の間の通信はオーバラップして行え, 通信回数が増えても全体の実行時間に与える影響を小さく抑えることができる。この方法を実現するために行った具体的な実装方法について以下に述べる。

slave にあたる各プロセッサは master と共有ストリーム変数を持ち, idle 状態になるとその変数を介して自分のノード番号を master に送る。master は送られてきたノード番号をマージし, 仕事要求キュー (idle PE queue) に繋ぐ。一方プロセッサを要求する側は, ゴールの送信先プロセッサを指定する値を未具体化変数にしておき, その未具体化変数を master に送る。master はこれを同様にノード要求キュー (PE request queue) に繋ぐ。この時点では送信先が決まっていないため, このゴールは中断される。

master がこれら 2 つのキューの先頭要素から順次同一化していくことにより未具体化変数であったゴールの送信先があるノード番号で具体化され, ゴールが実際に送信される (図 9)。

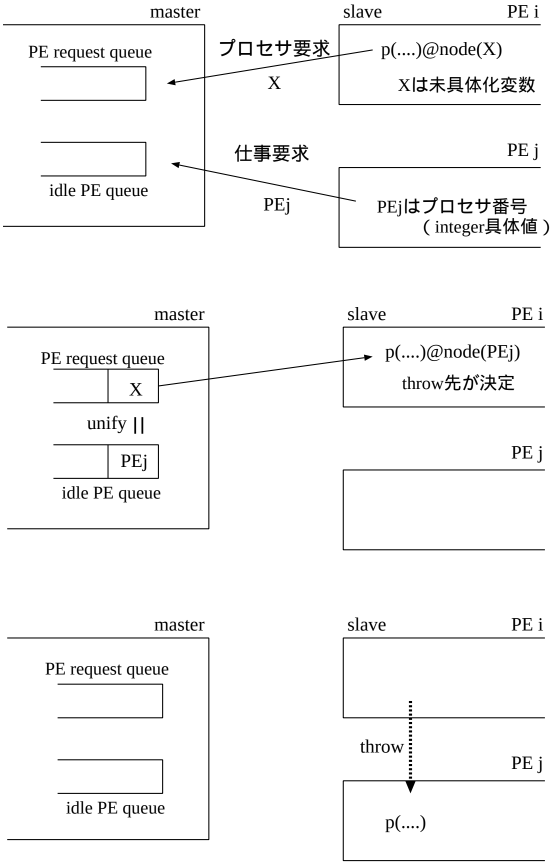


図 9: マスタスレーブによる負荷分散方式

4.3.2 スレーブプロセスの実現

プロセッサが idle 状態になったときに自分のノード番号を master に送るスレーブ側プロセス (idle であることを master に知らせるためのプロセス。以下, `tellidle` とする) の実現方法には以下の 2 つが考えられる。

1. 自らが担当する述語の実行が終了した時点で `tellidle` を起動する。
2. `tellidle` を, ノード内であらかじめ KL1 の `@priority` プラグマを用いて最低優先度で発

行しておく。

ここで, これらの方法の間では idle 状態の意味が異なっている。方法 1. では実行すべきゴールが存在しない状態を idle としているのに対し, 方法 2. では実行可能なゴールが存在しない状態を idle としている。

よって 1. の方法を採用した場合, 中断状態のプロセスしか存在しないようなノードは実際には仕事を行っていないにもかかわらず, idle とはみなされない。これは効率が悪い上に, AND/OR 両並列性を用いて負荷分散を行おうとした場合に AND 並列性の依存関係から deadlock に陥る危険性がある。

これに対し 2. の方法を採用した場合は, 中断状態も idle とみなされるため, 1. のようなプロセッサ利用効率の低下や deadlock の危険性は回避できる。しかしその反面, 実行可能ゴールの実行のためのデータ通信等が行われている状態も idle とみなされ, 余分に仕事を割り当てられてしまう危険性がある。

今回の研究では, AND/OR 並列と OR のみの並列に共に利用でき実行効率も良いと思われる 2. の方法を用いた。以下この方法によりマスタスレーブ方式を実現する手法について述べる。

4.3.3 idle PE queue 操作の実現

あるノードにおいて実行可能なゴールがなくなると, あらかじめ最低優先度で起動されていた `tellidle` が起動され master にノード番号が送られるが, そののち `tellidle` は新しい仕事 (ゴール) がそのノードに割り当てられるまで wait 状態に入る必要がある。これは, idle 信号を何度も送ってしまうことで余分な仕事を割り当てられることを避けるためである。そしてその wait はノードに新たな仕事が割り当てられた後に解除されねばならない。

これを実現する方法としては以下の 2 つが考えられる。

1. `tellidle` は idle 信号を master に送ったのち一旦消滅するが, その後要求に対し新しいゴールが到着したときに, そのゴール自身が `tellidle` を最低優先度で再起動する。
2. `tellidle` は送信完了の是非を示す未具体化変数を持ち, その変数が具体化されるまで (即ち送信完了が知らされるまで) wait 状態を保つ。その変数は要求に対しゴールが到着したときに具体化される。

1. の方法を用いる場合, 送信対象となっているゴールは送信先のノードで `tellidle` を起動するが, そのためにはその投げられる先のノードに対応する idle キューへのポインタを知っていなければならない。このため, idle であるノードが自分のノード番号を master に送るとき, 同時に自分の idle キューへのポインタも master に送っておく必要がある。

これに対し, 2. の方法を用いる場合は送信対象のゴールが直接 `tellidle` を起動するわけではないので idle queue へのポインタを知っている必要はないが, 送信が終了済か否かを監視する変数が必要である。

以上 2 つの方法について, ノード上で実際に起こるゴールの動きを示したのが図 10である。これを見ても分かるように, いずれの方法を用いてもマスタ, スレーブ間の通信が変数が 1 つ増えるだけで差がない。その変数がキューへのポインタであるかフラグであるかの違いだけである。今回の実装では方法 2. を用いた。

4.3.4 送信対象となるボディゴール

プロセスが複数の子プロセスを起動するとき, その一部を送信対象として残りは自分が受け持つことになる。この際どのプロセスを送信対象とするかが問題となる。

PAX におけるプロセスはストリーム入力の要素おのおのについて異なった処理を行うが, そのとき並列に起動される子プロセス間, 即ち OR 並列性 2. ではその時点までの部分解 (受け取ったストリーム要素) を当然共有することになる。これに対しストリームを通じて送られてくる各要素間, 即ち OR 並列性 1. の方が比較的独立性が高いと考えられる。したがって今回は, 自己再帰呼びのゴールを送信対象とすることにした。これはすなわち 4.1 で述べた OR 並列性 1. による並列化にほかならない。

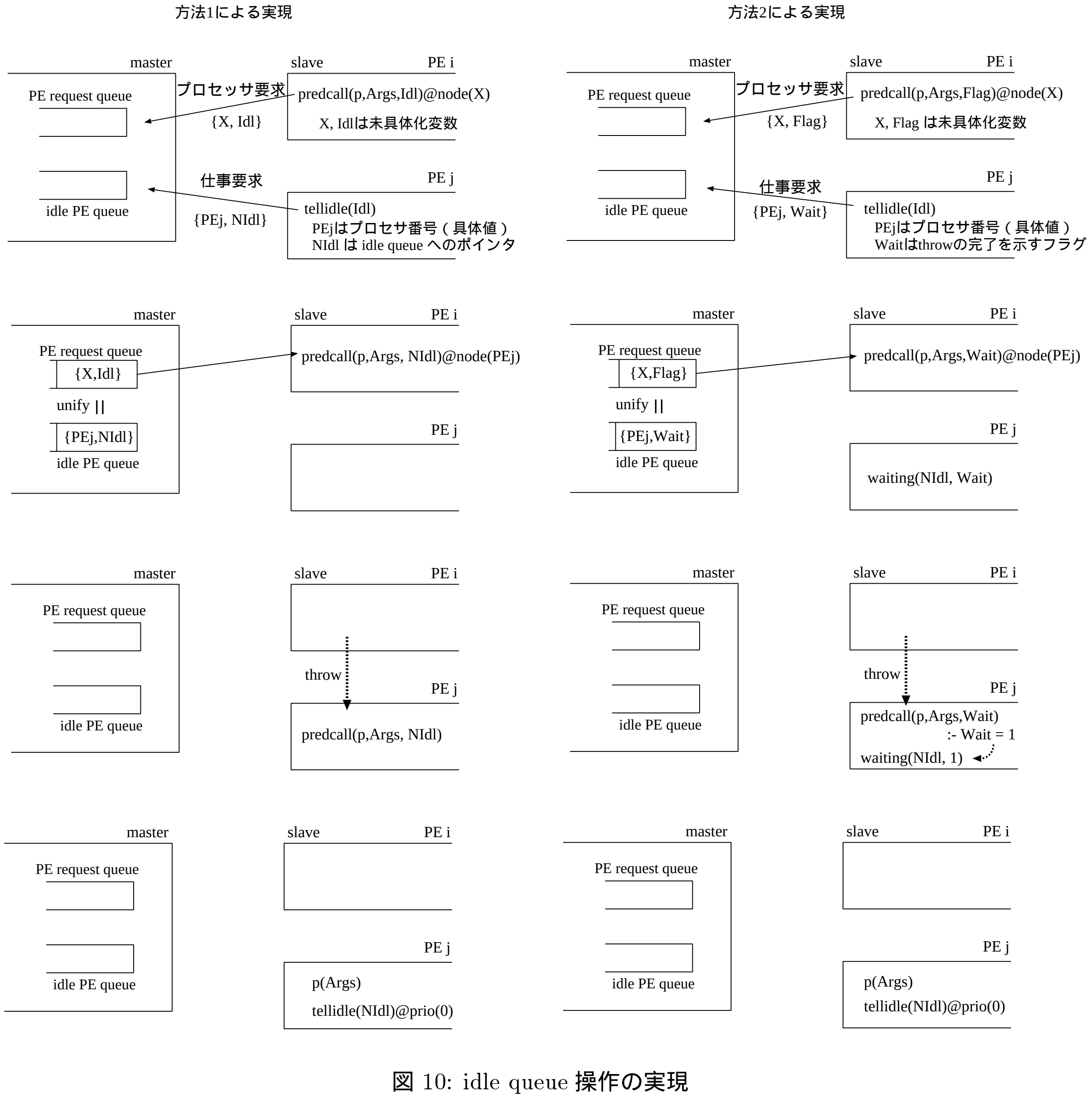


図 10: idle queue 操作の実現

またゴールを送信する前にその入力ストリームの次要素がnil atomか否かを調べることにした(図11)。入力の次要素がnil atomであるとき, 即ち入力ストリームが次で閉じられているときは, PAXのいかなる述語においても出力ストリームを閉じる仕事しか行われない。それだけの処理を行うためにゴールを送信すると, 送信されたゴールは即座に終了してしまい効率が悪い。

```
p([f1|In], Out, Req) :-
        Out = {Out1, ...},
        Req = {Req1, ...},
        q(...),
        ...,
        throwchk(p, In, Out1, Req1).
        %% 送信前にチェックを行う

throwchk(p, In, Out, Req) :- In = [] |
        Out = [], Req1 = [].
        %% In が nil なら送信せずに終了
otherwise.
        %% それ以外なら
throwchk(p, In, Out, Req) :-
        Req = [X|NewReq],
        p(In, Out, NewReq)@node(X).
        %% 送信先要求。決定次第送信
```

図11: ゴール送信前のチェック

5 性能評価

本章では4章で述べた静的/動的負荷分散を導入したPAXプログラムを, 当研究室で移植を行ったAP1000版KLIC [15] 上での実行により評価する。

KLICは, 従来並列推論マシンPIM上でしか動作しなかったKL1プログラムを様々な計算機上で動作させるために, (財)新世代コンピュータ開発機構(ICOT)が開発したKL1処理系である。KLICは, KL1プログラムをいったんCプログラムに変換し, オブジェクトコードの生成はターゲットマシンのCコンパイラが行うという方式をとる。このため, 通信関係などの機種依存部を除き移植性の高い処理系となっている。AP1000版KLICはPVM版に較べて通信オーバヘッドが少ないという特長を持つ。また専用マクロ`Cprobe()` [15] によりポーリングの高速化が図られている。

今回の性能評価で用いたのは, 文法数約500, 単語(終端記号)数約200からなる構文規則である。また, 解析対象として用いた文は表1に挙げた2つである。探索空間の広さを示すため, 補強項を用いない場合に得られる解の個数も示す。

表1: 性能評価に用いた文

	単語数	解の個数	補強項なし
(1)	11	1	36
(2)	20	3	約760

まず各PAXプログラムに文(1)を解析させ, その所要時間を測定した。表2に各PAXによる, 解探索に要した時間を示す。また, 解析の際に発生したゴールの中断とメッセージの数を表3に示す。ただしオリジナルPAXと通信低減版PAXの並列性が最も引き出される11台(即ち単語数に等しい台数)で比較を行った。今回使用した環境は64セル構成のAP1000であるが, うち1台はKLICのmasterとして用いられるため, 実際利用できるのは63セルまでである。

オリジナルPAXと通信低減版PAXは単語数以上のプロセッサは利用しないので, プロセッサ台数が単語数を越えると実行時間はほぼ一定となっている。

更に詳しく効果を調べるため, 通信低減版PAXとOR並列による静的負荷分散PAXについて文(2)を用いて探索時間を測定した。その結果を表4, 図12に示す。

オリジナルPAXと通信低減版PAXは,

- 再負荷分散を行わない。
- 利用可能なプロセッサ数がmであるとき, 起動時にn番目の単語を$n \bmod m$番目のプロセッサに割り当てるため, $n > m$である場合, 負荷の偏りが生じる。

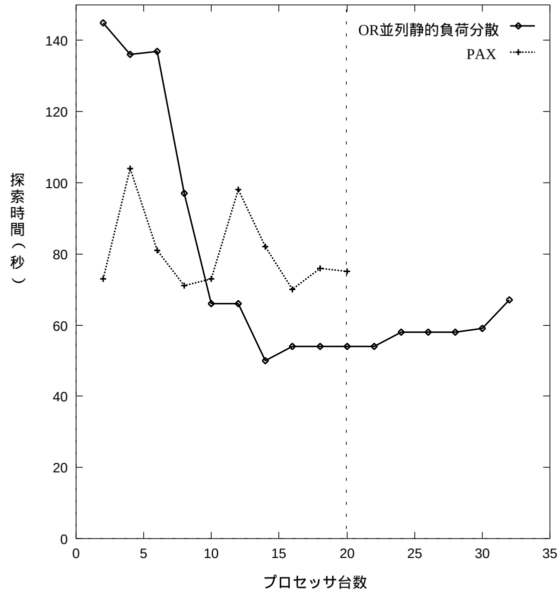


図 12: 文 (2) による評価結果

というように負荷の偏りを発生させる可能性を多く含んでいるため, 実行時間は単調に減少するとは限らない。

これに対し OR 並列による静的負荷分散版 PAX はプロセサ数に応じてある程度単調に実行速度が向上しているが, 20 台付近でピークとなり, そのあとは徐々に実行速度が低下している (表 5, 図 12)。

解析実行時でのゴール中断回数と発生メッセージ数を測定した結果をそれぞれ表 6, 7に示す。通信低減版 PAX の台数を 20 にしてあるのは, 文 (1) の時に述べたのと同じ理由からである。

オリジナル PAX, 通信低減版 PAX ともに AND 並列に基づく負荷分散を用いているのに対し, 今回実装を行った分散方式は OR 並列性に基づいているため, 各プロセッサ間のデータ依存性が少ない。実際は, 負荷の偏りにより仕事をほとんど行っていないようなプロセスがある程度存在しているため, 平均値はこの表に示した値より悪くなると考えられる。しかしそれを考慮にいれても, 依存性の低下によりゴール中断回数とメッセージ数が大きく減少しているのは明らかである。

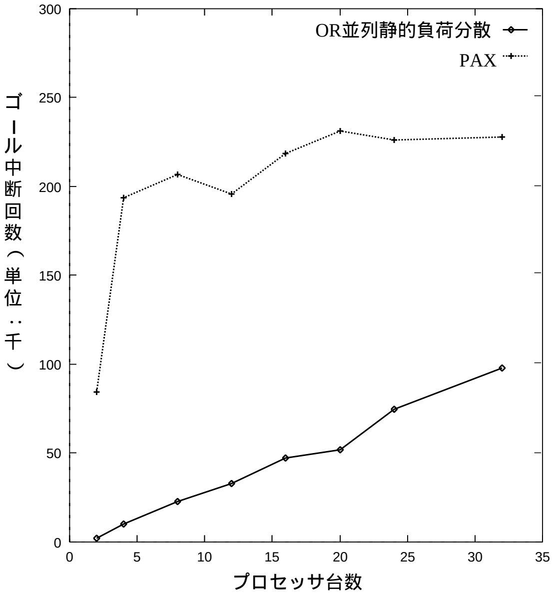


図 13: ゴールの中断回数

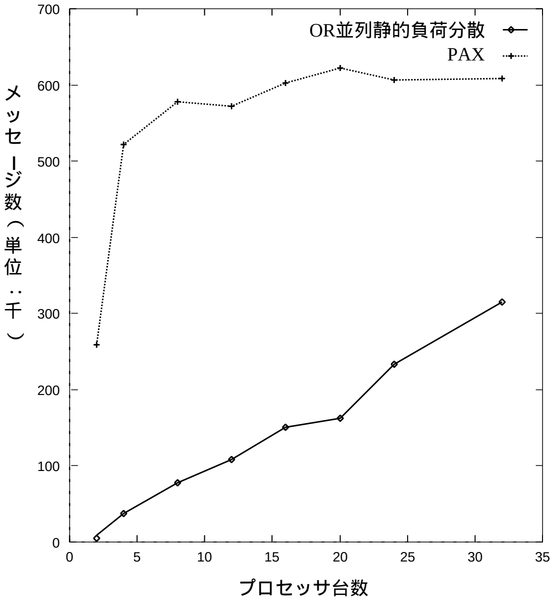


図 14: メッセージ数

表 2: 文 (1) の解探索に要した時間 (sec)

PE 台数	2	4	8	11	16	32	63
オリジナル PAX	48.20	36.34	34.77	26.15	26.17	26.14	26.06
通信低減版 PAX	7.97	7.15	7.07	6.37	6.21	6.33	6.27
OR 並列静的負荷分散	4.46	3.21	5.87	6.14	4.93	6.13	5.20
OR 並列マスタスレーブ	15.52	7.36	7.27	7.25	7.18	7.15	7.19

表 3: ゴール中断回数とメッセージ数

	ゴール中断回数		メッセージ数	
	総計	平均	総計	平均
オリジナル PAX (11 台)	88,417	8,038	291,551	26,505
通信低減版 PAX (11 台)	11,135	1,012	73,146	6,650
OR 並列静的負荷分散 (11 台)	9,474	861	41,049	3,732
OR 並列マスタスレーブ (11 台)	1,814	165	76,260	6,933

表 4: 文 (2) の解探索に要した時間 (sec)

PE 台数	2	4	8	12	16	20	32
通信低減版 PAX	73.50	104.97	71.67	98.67	70.87	75.85	76.02
OR 並列静的負荷分散	145.35	136.23	97.52	66.42	54.61	54.91	67.10

表 5: 台数効果

PE 台数	2	4	8	16	20	32	63
OR 静的負荷分散	1.07	1.15	1.61	2.91	2.90	2.39	1.91

表 6: ゴールの中断回数

	総数	プロセッサ平均
通信低減版 PAX (20 台)	231,048	11,552
OR 並列静的負荷分散 (20 台)	51,788	2,590
OR 並列静的負荷分散 (32 台)	97,907	3,060

表 7: メッセージ

	メッセージ数		サイズ (KByte)	
	総計	平均	総計	平均
通信低減版 PAX (20 台)	622,259	31,113	18,787	939
OR 並列静的負荷分散 (20 台)	162,181	8,109	5,118	256
OR 並列静的負荷分散 (32 台)	314,807	9,838	17,680	281

6 おわりに

本論では, 並列構文解析システム PAX の OR 並列性を利用した静的/動的負荷分散について述べ, その有効性を AP1000 版 KLIC 上で実行し評価した。OR 並列性を用いたことによってプロセッサ間のデータ依存関係を軽減することができ, 結果として KL1 プログラムのゴール中断回数とメッセージ量が減ることで高い並列性を引き出すことができた。

構文解析の結果として得られる解の個数は少ないものであるが, 各解は構文木を表現しているためある程度の大きさを持つ。オリジナル PAX や通信低減版 PAX は解が解析の過程で 1 つのプロセッサに集まるようプロセスがマッピングされているが, OR 並列で並列処理を行った場合, 解析終了時に解を特定プロセッサ上に集める必要がある。今後の課題としては, より効率を上げるために, この解の収集方法にも工夫が必要であろう。

また, ゴール中断回数の減少を見ても明らかなように, OR 並列性を利用しているためプロセス起動時にその入力データがすべて決定している。よって KLIC の一括送信機能 [15] を利用すれば、細粒度通信の発生を回避できると考えられる。

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚な謝意を表します。適切なご指導を賜った中島浩助教授, 森眞一郎助教授に心から感謝致します。

また貴重なご助言を頂いた大野和彦氏, AP1000 版 KLIC の環境を整えて頂いた伊川雅彦氏をはじめとして, 日頃御討論頂いた京都大学工学部情報工学科富田研究室の諸兄に感謝致します。

最後に, 今まで私に何らかの形で関わりのあったすべての方々に感謝致します。

参考文献

[1] Matsumoto, Y.: A Parallel Parsing System for Natural Language Analysis, *New Generation Computing*, オーム社 (1987).

[2] 松本裕治, 杉村領一: 論理型言語に基づく構文解析システム SAX, コンピュータソフトウェア, Vol. 3, 日本ソフトウェア科学会.

[3] Chikayama, T.: *Introduction to KL1* (1994).

[4] 佐藤裕幸: 並列自然言語構文解析システム PAX の改良, *Proceedings of KL1 Programming Workshop* (1990).

[5] 近山隆: KLIC ユーザーズ マニュアル (1995).

[6] 六沢一昭, 仲瀬明彦, 近山隆, 藤瀬哲朗: KLIC 分散メモリ処理系の設計と初期評価, 並列処理シンポジウム JSPP '95 論文集 (1995).

[7] Kasami, T.: An efficient recognition and syntax algorithm for context-free languages, *Science Report*, No. AFCRL-65-758, Air Force Cambridge Research Laboratory (1965).

[8] Kay, M.: Algorithm Schemata and Data Structures in Syntactic Processing, Technical Report CSL-80-12, Xerox PARC (1980).

[9] Pereira, F. and Warren, D.: Definite clause grammars for language analysis – A survey

of the formalism and a comparison with augmented transition networks, *Artificial Intelligence*, Vol. 13 (1980).

[10] Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H. and Yasukawa, H.: BUP: A Bottom-Up Parser Embedded in Prolog, *New Generation Computing*, Vol. 1, No. 2, オーム社 (1983).

[11] Matsumoto, Y., Kiyono, M. and Tanaka, H.: BUP 構文解析システムの諸機能, 自然言語理解と論理プログラミング, 近代科学社 (1990).

[12] Aho, A. and Ullman, J.: The Theory of Parsing, Translation, and Compiling, *Parsing*, Vol. 1, Prentice-Hall (1972).

[13] Okumura, A. and Matsumoto, Y.: Parallel Programming with Layered Streams, *Proc. International Symposium on Logic Programming*, pp. 224–232 (1987).

[14] 松本裕治, 奥村晃: 並列論理型言語による探索問題のプログラミング —層状ストリーム法の拡張—, 情報処理学会論文誌, Vol. 32, No. 7 (1991).

[15] 伊川雅彦, 大野和彦, 森眞一郎, 中島浩, 富田眞治: 並列論理型言語処理系 KLIC における通信の高速化 (1994).