自然言語処理 117－11
（1997. 1. 20）

並列 HPSG パーザーに向けて

二宮 崇　鳥澤 健太郎　田浦 健次朗　辻井 潤一
東京大学 理学部 情報科学科
{ninomi,torisawa,tau,tsujii}@is.s.u-tokyo.ac.jp

本研究では、鳥澤の HPSG パージングアルゴリズム [1][2] の並列化に関する考察を行い、その部分的な並列化を行った。鳥澤のアルゴリズムはフェーズ 1 とフェーズ 2 という 2 つのフェーズからなる。フェーズ 1 は HPSG の辞書項目からコンパイルされた CFG を用いたパージングであり、フェーズ 2 ではコンパイルされた CFG ではカバーしきれない制約を素性構造を用いて計算する。本研究ではフェーズ 1、つまり、HPSG のコンパイルによって得られた CFG によるパージングの並列化アルゴリズムを実現した。実現は超並列計算機 AP1000+ (256 ノード、1 ノードは Super SPARC 50Mhz 相当) 上で並列オブジェクト指向言語 ABCL/*f*[3] を用いて行われ、新聞からとられた文を用いた実験を行った。50 語以下の文 (平均 19 語) をパーズした結果、一文にたいして可能なすべての構文木を数え上げるのに要した時間は、98 ミリ秒であった。

Towards a Concurrent HPSG Parser

NINOMIYA Takashi　TORISAWA Kentaro　TAURA Kenjiro　TSUJII Jun'ichi
Department of Information Science, University of Tokyo
{ninomi,torisawa,tau,tsujii}@is.s.u-tokyo.ac.jp

This paper describes an attempt to develop a parallel parsing algorithm for Torisawa's parsing algorithm for HPSG. Torisawa's algorithm consists of two phases. At Phase 1, a parser enumerates possible parse trees using CFG rules compiled from lexical entries in HPSG. The constraints uncovered by the CFG are solved at Phase 2, using feature structures and a variant of unification, partial unification. We realized a parallel parsing algorithm for Phase 1, on a highly parallel computer AP1000+ (256 Super Sparc 50Mhz) with concurrent object-oriented programming language ABCL/*f*. The average parsing time for the sentences consisting of less than 50 words was 98 msec.

1　はじめに

言語情報処理研究の流れには、言語学的に妥当な処理体系と、必ずしも言語学的に妥当とは言えないが、効率、および、ポータビリティーを重視した処理体系とが混在する。近年、前者の枠組みとしては、その高い記述力、および柔軟性の観点から HPSG をベースとするものが注目を浴びているが、一般に理論的研究の域を超えているものはごく少数である。我々の最終的な目標は、HPSG をベースとして、言語学的には妥当とは見なせない技術により処理されているアプリケーションでの使用に耐えるパージング、文法、自動的知識獲得などのフレームワークおよび、環境を開発する事である。このような目標に向け、まず、効率、頑健性を主眼におき、鳥澤らが HPSG の為のパージングアルゴリズム [1, 2] を開発した。本研究の目標はその並列版アルゴリズムを並列計算機上に実現し、より高速なパージングシステムを開発、今後の研究の基礎とする事である。

鳥澤のアルゴリズムは、二つのフェーズからなっており、最初のフェーズであるフェーズ 1 では、前もって辞書項目から計算されたオートマタを用いてボトムアップ・チャート・パージングを行ない、フェーズ 2 ではオートマタに反映されていない制約を素性構造を用いて計算する。本稿ではフェーズ 1 パージングは CFG によるパージングと見做せる事を示し、並列化の観点から見た鳥澤のアルゴリズムの性質を考察し、超並列計算機である AP1000+ 上で並列オブジェクト指向言語 ABCL/*f*[3] を用いて高速なフェーズ 1 パーザーが実現されることを示す。

一般に逐次アルゴリズムを並列化する際において重要なのは、いかに逐次アルゴリズムを分割し、その分割された部分を並列に動作するプロセスにマップするかである。直観的には一度に動作するプロセスの数、すなわち、並列度が大きいほど並列化の効果は高いと考えがちであるが、無用に並列度が高いと並列プロセス間の通信量が増えるということ、通信の処理は各プロセスに到達した時点で逐次処理をされてしまうことをあわせ、並列化の効果は低くなり、極端な場合には逐次アルゴリズムの方が高速になることもある。つまり、無用に並列度を高くするような逐次アルゴリズムのマップは多くの場合、良い結果を生まない。重要なのは、逐次アルゴリズムおよび処理系の特性にあわせ、並列度とプロセス間通信とのバランスがとれ、並列化の効果が最大になるようなトレードオフをうまく見つけてやることである。

本研究では、鳥澤のアルゴリズムのフェーズ 1 パージングは CFG パージングともみなせ、その並列パージングアルゴリズムとしては、他の既存の並列アルゴリズムに比して、並列 CKY アルゴリズムが最良であることを実験を通して示す。また、並列 CKY アルゴリズムは CFG 文法をそのまま利用できる、つまり、LR アルゴリズムのように CFG を別の形式に変換する必要がないということと、HPSG からコンパイルされた CFG 規則が非常に大量なものであることからも好ましいものである。

また、従来、並列パーザーのアルゴリズムは、並列論理型言語の処理機構に合わせた形で記述されることが多く、記述が容易な反面、並列処理の細かな部分のコントロールが困難であった。本研究ではそのようなストラテジーをとらず、並列オブジェクト指向言語 ABCL/*f* を用いることにより、一般には相反する、高級言語による記述の容易さと細かな並列処理の記述が可能であることの二点を同時に利用することができ、比較的容易に高効率を達成できたといえる。

以下、次節で今回並列化の対象となる鳥澤の HPSG パーザーのアルゴリズムについて述べる。3 節では HPSG パーザーの並列化の方法について述べ、フェー

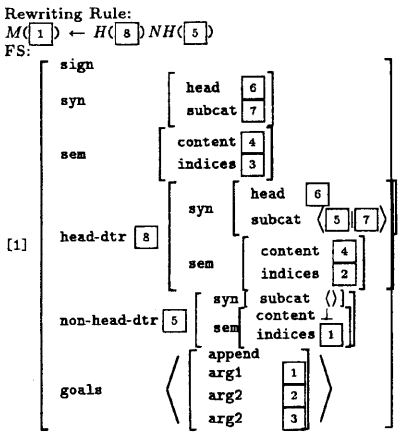

図 1: ルールスキーマの例

ズ1の並列化の実装、および、フェーズ2の並列化の見通しについて述べる。4節では実装、実験の結果及び、それらへの考察を加える。

2 鳥澤のアルゴリズム

本節では、ターゲットとしている鳥澤のアルゴリズムについて、特に今回並列化の対象となったフェーズ1に焦点をしぼって説明をする。

2.1 概要

鳥澤のアルゴリズムはHPSGをCFGとCFGではカバーできない部分にコンパイルしていると言える。一般のボトムアップHPSGパーザーの仕組みを簡単に説明すると、主辞の素性構造 H と主辞でない子の素性構造 NH と親の素性構造 M に対し、ルールスキーマ[1]が H, NH, M の間での制約となって、構文木を作り情報を伝えることになる。

例えば、図1のようなルールスキーマ R があった時、H, NH, M の関係は以下の記述のようにしている。

$$M = \begin{bmatrix} \text{head-dtr} & H \\ \text{non-head-dtr} & NH \end{bmatrix} \sqcup R$$

鳥澤のアルゴリズムを概観すると、まず、

1. 辞書項目に書かれている制約のうち構文木の骨格に大きく影響を及ぼす部分をパーズに先立ち計算して有限オートマタに展開しておき、
2. フェーズ1ではそのオートマタを用いボトムアップチャートパーザーでパーズし、
3. フェーズ2ではフェーズ1で計算されなかった残りの差分の素性構造を計算している。

以下、鳥澤のアルゴリズムのコンパイルの手法、および、パーズのアルゴリズムについて詳細を述べる。

[1]オリジナルのHPSGには、ルールスキーマというコンポーネントは存在しないが、ここでは、プリンシプル、IDスキーマなどの素性構造で記述された非語彙的な文法規則をすべて単一化により統合したものをルールスキーマと呼ぶ。

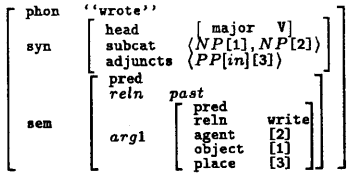

図 2: 辞書項目 (Lexical entry)

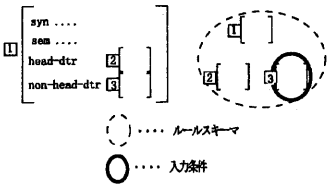

図 3: ルールスキーマの別表現

2.2 オートマタへのコンパイルとCFGへの変換

まずオートマタへのコンパイルとCFGへの変換について述べる。オートマタへのコンパイルとはすなわち、部分的な単一化操作をパーズ前に実行し、HPSGをCFG部分とそうでない部分とに変換する操作と言える。以下詳細を述べる。

各辞書項目 w に対し、

1. w の素性構造 (Lexical Sign) を主辞とし、素性構造 [⊥] を主辞でない子としてルールスキーマを適用する。新しく生成された 親の素性構造を m とする。
2. m に対して同様に m を主辞とし、[⊥] を主辞でない子として繰り返しルールスキーマを適用する[2]。

上の操作では、主辞でない子として素性構造 [⊥] しか指定しないが、主辞とルールスキーマの単一化の結果、例えば、主辞の subcat 素性の値などが、主辞でない子の値と構造共有される。この時、生成された親、あるいは、辞書項目をそれぞれ状態とし、主辞に対応する状態から親に対応する状態への遷移弧をつくる。ここで、ルールスキーマとの単一化により空でない値が定まった主辞でない子を遷移弧の入力条件とすれば、これを有限状態オートマトン[3]とみなせる。また、生成された親、あるいは辞書項目のうち、同値な素性構造をもつものは同一の状態とみなす。

例えば、"wrote" の Lexical Sign は、図2のようになっている。前述のルールスキーマを図3の左側とした場合、その右側のようにも表記される。すると、この Lexical Sign に対して図4のようにルー

[2]ここでは、この状態が常に有限であると仮定する。有限でない場合は [2] を参照のこと。

[3]正確には入力が有限であると仮定しないと有限状態オートマトンではない。しかし、一度全ての辞書項目に対し、全ての状態を作ったあと、その状態にアルファベットを割り付けるならば、そのアルファベットは有限であり、そのアルファベットを入力とするならば、入力は有限ということになる。

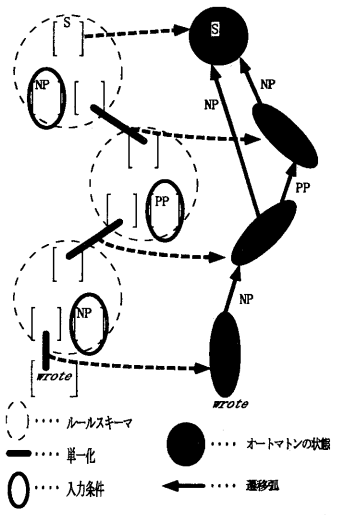

図 4: "wrote" に適用されるルールスキーマ

ルスキーマが適用される。また、この "wrote" に対しては他にもルールスキーマの適用の可能性があって、入力条件としてPPをとらないルールスキーマの場合もある。これら二つをあわせると、図4の右側のようなオートマトンができる。ルールスキーマおよび素性構造とオートマトンの状態の対応は破線で書かれた矢印が表している。

ここで、全ての状態に対しIDをふり、全ての可能な遷移弧の入力条件と全ての状態との間で単一化可能性のチェックを行ない、それが可能であれば、以下のようなCFGルールを生成する。鳥澤のアルゴリズムにおけるルールスキーマは2分木の場合のみを扱っているため、変換されたCFGは binary rule もしくは unary rule になる。

遷移弧の行き先の状態の ID
→ 遷移弧の出た状態 (主辞) の ID,
入力条件と単一化可能な状態の ID

2.3 パージングアルゴリズム

オリジナルの鳥澤のアルゴリズムでは、フェーズ1はボトムアップのチャートパーザーによるパーズになっており、主辞に対応する状態から発する遷移弧の入力条件と、文中の位置において隣合うエッジにアサインされた状態（これは実は素性構造である）が単一化可能ならば親を生成する。また、フェーズ1ではまだオートマタに対する入力、つまり主辞でない子の素性構造は構造共有を通じて親に伝搬されていないので、その差分を補完しなければならない。この計算はフェーズ2において、フェーズ1で完成した構文木に沿いながら、主辞と主辞でない子から素性構造の差分を親に伝搬させることによっておこなわれる[4]。

[4]詳細は [2] を参照のこと。

3 鳥澤のアルゴリズム フェーズ 1 の並列化

以上に述べた鳥澤のアルゴリズムの並列化はフェーズ1とフェーズ2で独立に行なわれる。まず、フェーズ1での並列化についてのべるが、前に述べたように辞書項目からコンパイルされたオートマタはCFGと変換され得るので、フェーズ1でのパージングをCFGパージングとみなし、CFGパーザーを並列化することによって高速化を実現する。以上にのべたように、本研究では、並列CKYパージングアルゴリズム [4] をベースとし、その改良したものでフェーズ1パージングをおこなう。

並列CFGパーザーのアルゴリズムに関しては、既にいくつかの研究がなされているが、大雑把にいって、それらは以下の3つに分類できると思われる。

1. 入力文中の位置にしたがって、パージング・プロセスを分割するもの。つまり、文中の位置毎に異なる並列プロセスが存在するもの。
2. 文法ルールあるいは、CFGの非終端記号毎に、異なる並列プロセスが存在するもの。
3. その両方を行なうもの。つまり、入力文中の位置と文法ルールあるいは非終端記号の異なるペア毎に並列プロセスが存在するもの。

並列CKYパージング・アルゴリズムは1.の代表的な例であり、米澤の並列パーザー [5] は2.の代表的な例である。また、PAX[6] はAND並列のみの実行の場合は1.に属するが、OR並列まで考慮すると、3.に属すると思われる。本研究では、鳥澤のアルゴリズムのフェーズ1を実現するために、並列CKYアルゴリズムをベースとして選択した。我々の文法は大量のCFGルールを持つので、2.のカテゴリに属するアルゴリズムは必要以上の並列度がでて不適切である。また、AND並列のみを用いたPAXも並列CKYパージングアルゴリズムによくにた戦略で並列化をはかっているといえるが、並列化が不十分で十分な台数効果がえられていない。

3.1 他の並列パージングアルゴリズム

以下では、まず、並列CKYアルゴリズム以外の米澤のアルゴリズムおよびPAXのアルゴリズムについてより詳しい考察を加え、並列CKYアルゴリズムを選択した理由をより詳細に述べ、フェーズ1アルゴリズムの実現および、並列CKYアルゴリズムの改良について説明する。

3.1.1 米澤の並列パージングアルゴリズム

米澤のアルゴリズムでは、まず、CFGの各書き換え規則にたいして、並列オブジェクトの集合を生成する。図5は生成されたオブジェクトの集合の例を示す。灰色のボックスが書き換え規則の左辺に対応し、白いボックスが書き換え規則の右辺に対応し、各ボックスは並列オブジェクトを表す。CFGの各ルールに対応するオブジェクト群が集まり、ネットワークを構成する。

パーズは基本的に、このようなオブジェクト間で部分的な構文木をメッセージとして投げ合うことによって行なわれる。つまり、白いオブジェクトが構文木の受け口となっており、ある書き換え規則の中の白いオブジェクト全てにお互い隣接する文字列をカバーする構文木が流れてきたら、それらを結合

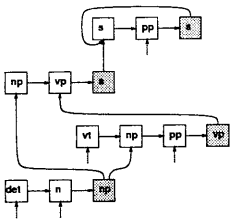

図 5: 書き換え規則によるネットワーク

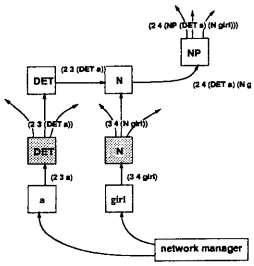

図 6: 実際のパーズの流れ

し、灰色のオブジェクトにメッセージとして渡す。灰色のオブジェクトは、文法全体に対応するネットワーク中に存在する同じ非終端記号を表す白いボックスにその結合された新しい構文木を配る。(図6参照)

現実に存在するような並列計算機上で HPSG パーザーを実現するという観点からすると、米澤パーザーは3つの点で非現実的である。すなわち、

- ある非終端記号を根にもつ部分的な構文木がメッセージとして投げられる宛先の数が非常におおきくなる。現実にこのような通信は非常に高価である。
- 頻繁に使われる CFG ルールに対応するオブジェクト群にメッセージが集中し、その部分が逐次処理になってしまう。
- 文法ルールが多いため、生成されるオブジェクトの数が非常に大きい。

部分的な構文木のメッセージの宛先をパーズの中間結果を共有している並列なプロセスと見做し、一番目の欠点を言い替えると、パーズの中間結果を共有すべき並列プロセスの数が非常に大きくなるということになる。この並列プロセスの数は、現在の文法では、ある非終端記号をもつ CFG ルールの数は最大で 700 近くなることがあり、一つのオブジェクトがそれら全てにメッセージをなげると仮定すると、ボトルネックが生じる。また、二番目の欠点も同様にクリティカルである。これはまた文法ルールの数が多いことも合わせ、余計な並列度がでるということも意味する。これは自然言語の文法では非終端記号ごとに頻度に差があるが、米澤のアルゴリズムではそれらを平等に扱うことによって生じている。これは逆に、各プロセスが利用するパーズの中間結果の数に偏りがあるということである。これも先の欠点と同様にボトルネックとなる。

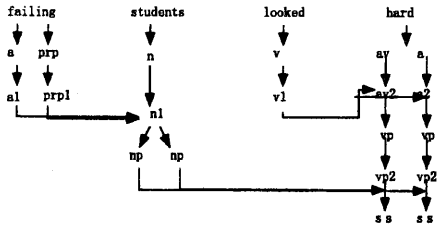

図 7: PAX のデータの流れ

3.1.2 PAX

ここでは AND 並列のみを使用した時の PAX について検討する[5]。この場合においては、PAX 中の並列プロセスは、文中の位置ごとに存在する。(図7参照) すなわち、パーズ開始時においては、各単語ごとにプロセスが存在し、その後生成されたパーズの中間結果である部分的な構文木は、その構文木がカバーする constituent の開始位置に対応するプロセスが保持、処理を続ける。

この枠組においては、プロセス間通信は、あるプロセスと、そのとなりの位置に対応するプロセスとの間でしか生じない。つまり、ある並列プロセス A が中間結果を生成すると仮定すると、その中間結果を必要とするプロセスは、A のとなりの位置に対応するプロセスだけである。これはデータの共有という観点からは望ましいが、反面、十分な並列度が得られないという欠点がある。実際、PAX でえられる最大の並列度は文の長さに等しいので、平均 20 語程度の長さの文をパーズすることを考えると並列度が低過ぎる。このような並列度の低さにより、報告された実験では望ましい台数効果を得られていない [7]。

3.2 並列 CKY アルゴリズムと並列フェーズ 1 パージングの実現

並列 CKY パーザーは以前から研究されていて、Nijholt[4] による提案もその一つである。並列 CKY パーザーでの並列プロセスは逐次 CKY パーザーの三角表の各要素に対応する。ここでは三角表の要素のことを便宜上セルと呼ぶことにする。

並列度の観点からすると、ある意味で並列 CKY パーザーは、米澤のアルゴリズムと PAX との中間に位置を占める。というのも、米澤のアルゴリズムでは、文のパーズが開始された時点でシステム中にあるオブジェクトの数は各文法規則中の非終端記号 (我々の文法では、$\sim 10^5$ 個) の総数にほぼ等しく、PAX では入力文の長さにほぼ等しい。それに対して、並列 CKY では三角表のセルの数、すなわち、

[5]OR 並列を実現したときには、最大で文法ルール単位で並列実行がおきるので、我々の文法を使用すると、米澤のアルゴリズムと同様に過度の並列度が出る可能性がある。

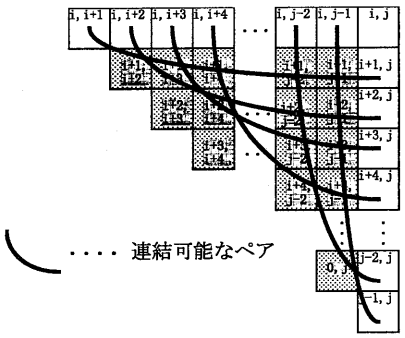

図 8: $t_{i,j}$ を計算する過程

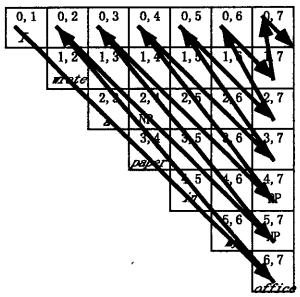

図 9: 逐次での CKY パーズの過程

入力文の長さを n とすると、$n(n+1)/2$ 個の並列プロセスが存在する。我々の現在のアルゴリズムでは最大並列度がおおむね、n から $n(n+1)/2$ の間におさまっており、新聞などに現れる文の平均長が 20 語前後であること、現在使用可能な並列計算機のノード数が 64 ノードから 256 ノード程度であることなどを考えあわせると、並列 CKY での並列プロセスの数は適当なものであるといえる。

また、各セルに格納される共有データ、あるいは constituent は、米澤のアルゴリズムでは文法規則ごとに配置され、PAX ではその constituent の先頭位置ごとに配置しておかれている。文法規則に依存した配置では前述したように出現する非終端記号の頻度に偏りがあるためデータの共有の度合が高い。PAX は先頭位置に対して共有データの配置が決まるが、それに対し CKY アルゴリズムでは constituent の先頭位置、後尾位置に対して、共有データの配置が決まるため、共有データは比較的分散され、共有の度合が低く、高い並列度が得られる。

まず逐次の CKY アルゴリズムを説明する。CKY アルゴリズムは基本的なデータ構造として、三角表と呼ばれる表を持つ。入力文の長さを n とした時に表の各セルは $t_{i,j}(0 < i < j < n)$ で表される。あるセル $t_{i,j}$ にある非終端記号 X が存在するのは、入力文のうち、i から j であらわされる部分文字列をカバーするような部分構文木で、その根のラベルが非終端記号 X で表されるものが存在するということである。

ここで、書き換え規則が binary rule からなるとすると、左端が i、右端が j である表のセル $t_{i,j}$ に存在する非終端記号を全て得るためには、$\{\langle t_{i,k}, t_{k,j}\rangle | i < k < j\}$ という全てのペアに対し、$t_{i,k}$ と $t_{k,j}$ を binary rule における子供として binary rule に適用し、新しい親の非終端記号が得られるかどうか見ればよい、ということになり、得られたならば、それが $t_{i,j}$ の非終端記号ということになる。(図 8の連結可能なペアに対し binary rule を適用すれば $t_{i,j}$ の非終端記号が得られる。) ここで、パーズは表の $t_{i,j}$ に対し、図 9の矢印の順にパーズされる。また、鳥澤のアルゴリズムで用いられるルールスキーマは二分木を生成するようになっているため、HPSG から展開された CFG の書き換え規則は全て binary rule もしくは unary rule になっている。従って、そのまま CKY アルゴリズムを適用することができる。

並列アルゴリズムの実現に関しては、表の各セルを ABCL/f における並列オブジェクトとみなし、表の各セルが並列に処理される。このように並列化することによって、共有すべきデータが、文中の位置に対してできる限り分散されたことになる。つまり、$t_{i,j}$ が必要とするデータはセル $\{\langle t_{i,k}, t_{k,j}\rangle | i < k < j\}$ にあるものであり、これらペアを見に行くという処理は各々並列に行える。また、各非終端記号の組を binary rule に適用する際、上記の組合せの子供に対して全て適用可能であるため、子供同士が隣あう位置にいるかどうかのチェックをする必要はない。また、binary rule の適用は配列に対して高々 1 回のアクセスで処理できるため、逐次に処理される部分も高速に処理される。

また並列 CKY では、同一の非終端記号を持つエッジをまとめるという操作が容易に行える。ここでエッジとは表中のセル $t_{i,j}$ に格納される非終端記号のことを指す。つまり、位置情報と非終端記号との組であるとも言える。また、ここで、セル $t_{i,j}$ に対応するオブジェクトを $o_{i,j}$ とし、非終端記号 X があったならば、エッジを $\langle i, j, X\rangle$ と表記する。オブジェクト $o_{i,j}$ に $\langle i, k, Y\rangle$ と $\langle k, j, Z\rangle$ が来て、さらに $X \rightarrow Y, Z$ というルールがあったならば、このオブジェクト $o_{i,j}$ に $\langle i, j, X\rangle$ というエッジが生成される。ここでさらに $\langle i, l, V\rangle$ と $\langle l, j, W\rangle$ というエッジが来て、さらに $X \rightarrow V, W$ というルールがあった場合にエッジをまとめないとするならば、もう一つ $\langle i, j, X\rangle$ を作ることになるが、エッジをまとめるのならば、二つめの $\langle i, j, X\rangle$ は生成されない。特に今回用いる HPSG の文法では曖昧性が非常に大きいため、エッジをまとめるという操作が非常に効いてくる。論理型言語でパーザーを記述するならばこの操作は容易ではないが、並列オブジェクト指向言語ならばこの記述は容易である。

さて、Nijholt の方法ではこの $t_{i,j}$ が表中の他の計算結果を取りにいくとき、ヒューリスティックスに従って一般に早く計算が終る順にデータをとりにいくようにしている。我々はこの部分を並行[6]に処理し、一般に早く終ることが期待される順に取りにいくのではなく、実際に早く計算が終ったペアから順に処理をするようにした。つまり、データフローに近い形で処理が行なわれるが、完全なデータフローと違って、計算結果を取りにいく側でフロー制御が

[6]並列ではない。データを取りにいくという操作に順序が指定されていない、ということ。つまり、このオブジェクト中では全ての瞬間においてエッジをとりにいくプロセスはただ一つである。

行なえるためシステムが不安定な状態にはならない。

4 鳥澤のアルゴリズム フェーズ2の並列化

フェーズ2の並列化は現時点では終了していないが、本節では、フェーズ2の並列化に対する見通しを述べる。基本的にはフェーズ1とフェーズ2は異なる方法により並列化される。

逐次アルゴリズムのフェーズ2で、フェーズ1で完成された構文木にそって子ノードから親ノードに差分の素性構造を伝搬させる際に、主辞と主辞でない子に対応する素性構造の差分は、それぞれ互いに独立に計算できる。またことなる計算プロセス間でのデータの共有は、あるノードが複数の親ノードの間で共有されている時に生じるが、このような状況は、フェーズ1で生じ得るデータの共有に比べれば、共有するプロセスの数が少なく、ボトルネックは生じにくいと予想される。フェーズ2の並列アルゴリズムを要約すると次のようになる。

- 並列アルゴリズム中のフェーズ2では、フェーズ1で完成した構文木の根からその子孫を再帰的に手繰っていく際に、あるノードの主辞と主辞でない子に対し、素性構造の差分を求める手続きをそれぞれ独立な計算プロセスとして起動する。

5 実験および考察

今回AP1000+上に実現したHPSGのための並列CKYパーザーの性能を評価するために、いくつかの実験を行った。

実験1は小規模な文法を用いた1プロセッサ上での実験である。これにより、AP1000+ 1プロセッサの性能と他の逐次マシンの基本的性能の差、および、我々のCKYアルゴリズムにおける台数効果を得るための基本的データを得る。

実験2ではやはり小規模な文法を用いた場合の台数効果を見る。

実験3では我々の最終目標であったHPSGパーザーのフェーズ1を想定した実験を行なう。すなわち、実際にHPSGからコンパイルされた大量のCFGルールを使って複数プロセッサ上で我々のアルゴリズムを実行した場合とC++でインプリメントされたフェーズ1のアルゴリズムの逐次における実現を比較検討する。最終的な我々の得たい結果は実験3で示される。

また、実験1、2、3を通して用いられるC++のCKYパーザーは東京大学の光石氏により実装されたものを用いている。SAXは、用いたPrologはSicstus Prolog 2.1であり、下降予測は使っていない[7]。

5.1 実験1

実験は

- [8]にてGrammar IVと記載されていた文法規則419、単語数241のCFGによる英語文法、および、

台数	(i)	(i)'
1	12475	446
2	17,180	614
4	9,025	322
8	4,949	177
16	2,878	103
32	2,055	73
64	1,633	58
128	1,441	51
256	1,365	49

(i) 並列CKYアルゴリズムによる解析時間(msec)
(i)' 並列CKYアルゴリズムの一文あたりの平均解析時間(msec)

表2: Grammar IVによる28文の解析時間における台数効果

	257文の解析時間 (msec)	1文あたりの平均時間 (msec)
AP1000+(ABCL/f 256プロセッサ,CKY)	25,079	98
SS10(C++,CKY)	378,280	1,472
Dec Alpha Station	44,301	172

表3: HPSGからコンパイルされたCFGによる257文の解析時間

- 平均語長15.79の28文からなる英語のコーパス

を用いて行なった。

ここではAP1000+ 1プロセッサの性能と逐次マシンの性能とを比較し、ABCL/fによるオーバーヘッドを測るため同じ逐次マシン上で、ABCL/fとC++の実装によるCKYパージングアルゴリズムの性能差を測る。また、他のパージングアルゴリズムとの比較のため、一般に広く使用されているパーザーであるSAXとも比較を行なった。評価に用いた逐次マシンはSun SPARC Station 10 80Mbytes(SS10)である。

以上の実験結果を表1に示す。

この結果から、SS10とAP1000+ 1プロセッサはおよそ同じ速さであるということ、およびC++に比べABCL/fの並列処理によるオーバーヘッドはほとんどない、ということがいえる[8]また、SAXが遅いのは前述のエッジをまとめていないことが大きな原因であると推測する。

5.2 実験2

実験2ではCKYパーザーによる台数効果の評価を行ない、表2の結果が得られた。用いた文法、コーパスは実験1と同じである。この表によると、CKYパージングアルゴリズムにおける台数効果は256台で9.14倍であった。実験に用いた文の長さが短いこと、および、SAXの並列版であるPAXの台数効果は[7]によると、およそ、2～3倍程度である、とい

[7] 奈良先端大学院大学 松本研にあるSS10で実行した場合およそ2倍程度速い結果がでている。いずれにせよ、下降予測を使えばおよそ3～4倍速くなると期待される。

[8] ここで、C++で記述されたCKYパーザーがABCL/fより遅いのは、メモリの解放部分がまだ未完成であり、SS10に搭載されている実メモリ量をパーザーが使い切ったためである、と推測される。

計算環境	AP1000+ 1PE ABCL/f CKY	SS10 ABCL/f CKY	SS10 C++ CKY	SS10 Prolog SAX
28 文の計算時間 (msec)	12,475	11,075	12,470	134,780
1 文あたりの平均 (msec)	446	396	445	4,813

表 1: Grammar IV による 28 文の解析時間

うことを考慮に入れると比較的高い台数効果が得られていると言える。

5.2.1 実験 3

次に HPSG で記述した日本語文法での実験結果を示す。

- 使用した文法は、ルールスキーマ 5 個、50 個の辞書項目のクラス、43 個の辞書項目からなり、これらをコンパイルした CFG は 746 個の非終端記号、178,535 個のルールからなる。
- 朝日新聞の社説 485 文中パーズに成功した 257 文に対して実験を行った。

その結果は表 3のようになった。また、257 文の平均語長は 19 である。台数効果に関しては同じ AP1000+ 上の 1 プロセッサで動く C 言語で書かれたプログラムと比較するべきではあるが、文法が非常に大規模であり、1 プロセッサ上に入りきらなかったため、前述の実験 1 よりほぼ同じ速さとみなせる SS10 と比較して台数効果を求める。すなわち、

$$\frac{\text{実験 2 での SS10, C++ の時間}}{\text{実験 2 での ABCL/}f\text{,AP1000+256 台の時間}}$$

とすると、得られた台数効果は 15.08 倍である。以上の実験により、CKY パージングアルゴリズムは他のアルゴリズムに比べ、特に HPSG などで書かれた複雑な文法記述からコンパイルされた CFG において、実用に耐え得る高速なアルゴリズムであると言える。

6 結論

鳥澤の 2 フェーズ HPSG パージングアルゴリズムの並列化の手法を示し、そのフェーズ 1 の実験を行った。実験、および考察の結果から、現在使用可能な並列計算機環境においては並列度 CKY アルゴリズムが有効であるといえる。特に HPSG からコンパイルされた CFG は 18 万ものルールからなっており、ルールを並列の単位とするのは非現実的なアプローチであるのに対し CKY アルゴリズムはルールを並列の単位としないためと言える。実験は、平均 19 語からなる 257 文をパーズした結果、一文にたいして可能なすべての構文木を数え上げるのに要した時間は、98 ミリ秒であり、得られた台数効果は 256 台で平均約 15 倍であった。使用した文法は、ルールスキーマ 5 個、50 個の辞書項目のクラス、43 個の辞書項目からなり、これらをコンパイルした CFG は 178,535 個のルールからなっていた。

謝辞

本研究のために PAX の情報を提供して下さった津邑氏、SAX についての情報をいただいた奈良先端大学院大学の宮田助手に深く感謝致します。並列計算機環境を提供していただいた米澤研の方々に深く感謝致します。最後に C++ により実装された CKY パーザーの実験に協力していただいた光石氏に感謝の意を表します。

参考文献

[1] Kentaro Torisawa and Jun'ichi Tsujii. Off-line raising, dependency analysis and partial unification. In *Third International Conference on HPSG*, pages 14–23, 1996. In the proceedings of TALN '96.

[2] Kentaro Torisawa and Jun'ichi Tsujii. Computing phrasal-signs in HPSG prior to parsing. In *COLING 96*, pages 949–955, 1996.

[3] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pages 275–292. American Mathematical Society, 1994.

[4] Anton Nijholt. *Parallel Natural Language Processing*, chapter Parallel Approaches to Context-Free Language Parsing, pages 135–167. Ablex Publishing Corporation, 1994.

[5] Akinori Yonezawa. *Parallel Natural Language Processing*, chapter Object-Oriented Parallel Parsing for Context-Free Grammars, pages 188–210. Ablex Publishing Corporation, 1994.

[6] 松本 裕治 and 杉村 領一. 並列構文解析. 情報処理学会自然言語処理研究会, 1986.

[7] Tsumura, Goshima, Mori, Nakajima, and Tomita. An improvement of a parallel parsing system pax by or-parallelizing. In *SWoPP '96*, 1996.

[8] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.