学 術 刊 行 物
情処研報 Vol. 90, No. 93

情報処理学会研究報告

９０－ ＮＬ －８０

１９９０年１１月２２日

社団法人　情報処理学会

自然言語処理 80－7
（1990．11．22）

並列時間差準破壊型単一化アルゴリズム実現の手法

藤岡 孝子[1]　苫米地 英人　古瀬 蔵　飯田 仁

ATR 自動翻訳電話研究所

梗概

単一化に基づく自然言語処理において、単一化手続きは最も処理時間の割合が大きく、これを高速化することが重要な課題となっている。これに対し、並列処理をすることによる高速化の手法を考える。まず、効率の良い並列化が可能な Tomabechi の時間差準破壊型単一化アルゴリズムについて考察し、これを並列に処理する手法を提案する。また、日本語文解析における単一化手続きにおいてこの並列アルゴリズムを用いて実験を行ない、並列化の効果と課題について述べる。

Parallelization Technique for Quasi-Destructive Graph Unification Algorithm

Takako FUJIOKA [2],Hideto TOMABECHI,Osamu FURUSE and Hitoshi IIDA

ATR Interpreting Telephony Research Laboratories

Abstract

A typical unification-based natural language processing system spends most of its processing times for graph unification. We have found existing graph unification algorithms to be hard to parallelize. It is essentially because synchronizations for each recurseve call into shared-arcs are required in the most existing algorithms and also due to the difficulty of efficient management of lock/unlock scheduling of simultaneous accesses to global shared data structures. We adopted the quasi-destructive graph unification algorithm as appropriate for effective parallization and propose the parallel quasi-destructive graph unification algorithm that avoids these two problems.

[1]早稲田大学大学院理工学研究科電気工学分野修士課程

[2]Department of Electrical Engineering,Waseda University

1 Graph Unification の並列化

本稿では、単一化に基づく自然言語処理を並列化するという目的において、従来の単一化アルゴリズムを考察した上で、高速化、効率化が期待でき、並列化に適していると思われるTomabechiの時間差準破壊型単一化アルゴリズムについて述べ、これを並列に処理するアルゴリズムを提案する。また、Earleyの日本語文解析パーザにおいてこれを適用した時の実験をおこなった結果から得られた並列処理における問題および効果について考察する。一般に効率の良い並列処理のためには並列プロセスに共有されるglobal変数のlock/unlockをできるだけ避けねばならない。また、ある程度のプロセス間の同期化も最小限度にする必要がある。この点から見ると、従来の手法はいずれも並列化するのが困難であると思われる。unificationのアルゴリズムとして従来提案されてきた主なものにはPereiraのアルゴリズム[Pereira, 1985]、structure sharing unificaiton アルゴリズム[Karttunen and Kay, 1985]、reversible unification アルゴリズム-[Karttunen, 1986]、non-destructive unification アルゴリズム[Wroblewski, 1987]などがあり、いずれもcopyのコスト削減に重点をおいた高速化を試みたものである。しかし、これらのアルゴリズムではいずれもcopyを避ける代わりにすべてのノードの情報をglobal変数の中に分散させているため、並列化した場合、複数のプロセスがこれにアクセスしようとするたびにlock/unlockのoverheadは膨大なものになると考えられる。また、Wroblewskiのアルゴリズムでは、ノードのcopyを作る時にそこから出発するすべてのarcをその1つのcopyノードにcopyするため、この部分はserial処理にならざるを得ない。これがノードの数だけincrementalにおこる為、シリアル化が本質的に避けられず、また、一つのノードへの成功した各sub-graphのcopyを並列的に付加する時のlock/unlockのコストは極めて大きくなる。

また、一般に、単一化アルゴリズムでは、shared-arcへの再帰的なunificationの結果を待たなければ現在進行中のunificationの結果を得ることができない。例えばWroblewskiでは、shared-arcへの再帰的unificationがすべて成功した時のみcopyノードに書き込みを行なうが、このためには現在進行中のunificationはすべての結果を待つこととなる。このように同期化の問題も重要なボトルネックとなっていると言える。

2 Tomabechiの時間差準破壊型単一化アルゴリズム

Tomabechiの時間差準破壊型単一化アルゴリズム[Tomabechi, 1991]では、unificationの高速化のポイントである2点、即ち失敗したもののcopyを避けることと、originalの情報を効率良く保持することをtime-stampを用いることにより効果的に解決している。

Tomabechiのアルゴリズムでは、まず、graphをunifyする時ノードの値にtime-stamp（整数）を属性として付加しておき、1つのunificationが終るとこれに1加えることによってoriginalとresultとの情報を区別する。また、すべての再帰的なunificationが成功した時点ではじめてcopyを作る。このコピーはgraphに対し2度目の再帰的処理をすることを意味するが、unificationの途中でのみ有効なforward（unificationの結果、他のノードと全く同じ構造を持つノードは、相手のノードとその構造を共有する）の情報が現在のtime-stampつきで各ノードのスロットに入れてあり、成功後のコピーはこのスロットのみを参照しながら迅速に行なわれる。一般に、自然言語処理においては1文の解析の間に行なわれるunificationの大半は失敗することを考えれば[Tomabechi, 1991]そのコストはover-copyingに比べて全体では少なくなるといえる。また、失敗した時にもやはりtime-stampが更新されこれらのスロットの値は次のunificationでは単に無視され、originalの情報のみがすぐに復活する。

この方法では、情報の退避と復活はすべてtime-stampのincrementで行なわれるため、graphの大きさに関わらずそのコストは常に一定で非常に小さい。さらに、再帰的に呼ばれたunificationがいずれか1つでも失敗すると次の再帰処理はもはや行なわず、すべてのunificationの実行を終らせる。従って各arcはそのsub-graphのunificationの結果を待つ必要がない。よってこの部分を自然に並列化することができこれによりさらに高速化することが期待できる。[Tomabechi, 1991]では、1文中の単一化の約6割から7割が失敗に終るような、一般的な文法[1]においてWroblewskiのアルゴリズムの約2倍の高速化が報告されている。以下図1、2にデータ構造[2]とアルゴリズムを示す（[Tomabechi, 1991]より）

3 Tomabechiの単一化アルゴリズムの並列化

Tomabechiのアルゴリズムでは、copyへの書き込みはgraph全体のunificationが成功した後のみに行なわれるので同期化の問題は本質的に存在しない。したがって、shared-arcにおける再帰的な呼びだしを並列に行なうことが可能である。しかしその場合新たに2つの問題点が考えられる。

1. ノードのforwardingはsub-graphのunificationがすべて成功してはじめて行なわれなければならないが、下のサブプロセスとは並列に処理が進んでいるので、この結果を待つという同期化を行なうと並列

---

[1] 例えば本研の実験で使われた文法である[Yoshimoto and Kogure, 1988]、[Kogure, 1989]など一般に単一化文法の解析では、8割以上の1文全体での成功率が記録されることはまれである。

[2] ノードのタイプにはatomic, leaf, complexがあり、leafノードはvariableを表している。

```
PROCEDURE UNIFY-DAG(DAG1,DAG2)
      RESULT:==UNIFY0(DAG1,DAG2)
      (但、FAIL を受け取ると NIL となる。)
      タイムスタンプを 1 増やし RESULT を返す。
ENDP

PROCEDURE UNIFY0(DAG1,DAG2)
      IFUNIFY1(DAG1,DAG2) が成功したら
        THEN新しいノードの arc-list に DAG1 の arc-list と
              タイムスタンプが有効であれば
              comp-arc-list の内容をコピーする。
      ENDIF
ENDP


PROCEDURE UNIFY1(DAG1,DAG2)
      DAG1:==DEREFERENCE(DAG1)
      DAG2:==DEREFERENCE(DAG2)
      IFDAG1,DAG2 が等しいならば
        THEN成功する。
      ELSE IF DAG1,DAG2 の一方が変数ならば
        THEN変数を表す DAG からもう一方へ
              タイムスタンプ付きの FORWARD 操作をして、成功する。
      ELSE IF DAG1,DAG2 がどちらも atomic ノード³ならば
        THENIF arc-list が等しいならば
                THEN成功する。
                      DAG2 から DAG1 へタイムスタンプ付きの FORWARD 操作を行なう。
                ELSE unification を中断し UNIFY0 に FAIL を返す
      ELSE IF DAG1,DAG2 の一方のみが atomic ノードならば
        THENunification を中断し UNIFY0 に FAIL を返す
      ELSE
        NEW:== COMPLEMENT-ARCS(DAG2,DAG1)
        SHARED:==INTERSECT-ARCS(DAG1,DAG2)
        FOR arc IN SHARED DO
        arc に対応する DAG1,DAG2 の arc 同士を
        再帰的に UNIFY1 で単一化する。
        IF    全ての再帰の結果が成功なら
              THEN
                DAG2 を DAG1 に FORWARD し、
                DAG1 の comp-arc-list に NEW を入れる。
                同時に comp-arc-list に現在のタイムスタンプをつける。
      ENDIF
ENDP
```

図 2: 時間差準破壊型単一化アルゴリズム

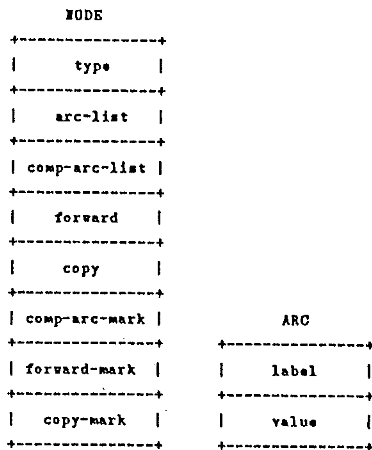


図 1: ノードおよびアークの構造

処理する意味が全くなくなってしまう。そこで、sub-graph の forwarding 操作を lazy evaluation を用いてすべての unification が終了するまで遅延 (delay) させる。そして、unification が成功した場合にのみこれを行なう。これにより、unification の並列プロセスが走っている間はノードへの同時書き込みが一切なくなるので、lock/unlock の問題は起こらない。ただし、もし graph が loop を含んでいる場合、forward の情報を利用せずに unification を進めるので、loop の数だけ余計に unification のチェックを行なわなければならない。しかし並列プロセッサの数が充分大きい場合はこれらは並列的にチェックが行なわれるので特に問題とならない。

2. ある unification プロセスが局所的に unification に失敗した時はトップのプロセスに失敗のメッセージを送り、他の実行中のプロセスをすべて中止させれば良いが、一方、unification が成功する場合、各ローカルな並列プロセスには全体の graph の大きさがわからない以上、全体の成功はどのプロセスにもわからない。よって成功はすべてのプロセスが正常に終ったことでしか判断不可能である。したがって各並列プロセスは成功したとき自分が最後のプロセスかどうかを調べ、他のすべてのプロセスが終っていればトップのプロセスに対し成功のメッセージを送るようにしなければならない。

メッセージを受け取ったトッププロセスは再び走りだすが、成功の場合は遅延させておいた forward の操作を行ない、成功した結果の graph を返すという操作を行なう必要がある。

以上のような考察を踏まえ、Tomabechi のアルゴリズムを並列化した手続きを以下図 3 に示す。

4 実験と考察

4.1 並列処理実験について

上に述べた並列アルゴリズムを並列マシンＳｅｑｕｅｎｔ (shared memory multiprocesser) 上の並列 Common Lisp 上にインプリメント[5]し、Earley のパーザの日本語文解析にＨＰＳＧ[6]を用いたシステムの単一化の部分にこの並列化 Tomabechi のアルゴリズムを用いて実験システムとした。

入力文として用いた日本語文は

(1) もしもし
(2) そちらは通訳電話国際会議事務局ですか。
(3) そうです。
(4) 私は会議に申し込みたいのですが。
(5) 分かりました。
(6) 登録用紙はありますか。
(7) いいえ。
(8) それではこちらからそちらに登録用紙をお送りいたします。
(9) お名前とご住所をお願いします。
(10) 大阪市北区茶屋町 23 米鈴木真弓です。
(11) こちらからそちらに登録用紙を至急お送り致します。
(12) わからない点がございましたら私どもにいつでもお聞き下さい。
(13) ありがとうございます。
(14) それでは失礼します。
(15) どうも失礼します。

の１６文であり、これらを解析した時の実験結果を表１にしめす。

ここで、R はトップの unification の結果であり ALL は全プロセス数であってトッププロセスを除いた全ての unification プロセスの数であり、グラフ中の shared-arc の総数に等しい。また、MAX-PP(最大並列プロセス数) は、１つの unification 中に作られた unification プロセス (総数 ALL) が最大いくつ同時に実行されたかを表す。また、並列化率はこの二つの比を取ったものである。１６文の解析全体でトップレベルの unification は１９６０回行なわれ、このうち何回の割合でこの比が得られたかが U-RATE(全単一化中の割合) によって示されている。これより、並列アルゴリズムにおける平均並列化率は

[5] 並列プロセスは、並列 Common Lisp 上では light-weight-process としてつくられている。これは、各プロセッサに対して１つずつ割り当てられるスケジューラプロセスがすべてのプロセスの実行状態を把握し、実際にどのプロセスをどのプロセッサで実行したり、止めたりするかなどの管理を一切まかなっており、必ずしも１つのプロセスが連続して１つのプロセッサで行なわれる訳ではない。したがって、実際にいくつのプロセッサを使用するかに依存せずに並列プロセスを作ることができる。

[6] Head-driven Phrase Structure Grammar, [Pollard and Sag, 1987]

```
PROCEDURE UNIFY-DAG(DAG1,DAG2)
    グローバル変数とそのロックをクリアする。
    MAKE-LWP⁴(UNIFY0(DAG1,DAG2))
    MESSAGE を受け取るまで待つ。
    IFMESSAGE が FAIL なら
      THENNIL を返す。
      ELSE 遅らせていた FORWARD をし、
           新しいノードに DAG1 の arc-list,
           comp-arc-list をコピーして返す。
    タイムスタンプを 1 増やす。
ENDP

PROCEDURE UNIFY0(DAG1,DAG2)
    (INCREMENT COUNT-OF-LWPS)
    MAKE-LWP(UNIFY1(DAG1,DAG2))
    (但、プロセス名にタイムスタンプを付ける。)
ENDP


PROCEDURE UNIFY1(DAG1,DAG2)
    DAG1:==DEREFERENCE(DAG1)
    DAG2:==DEREFERENCE(DAG2)
    IFDAG1,DAG2 のどちらも、
      complex ノードでないとき
      THENserial-quasi 同様に単一化を行ない、
           IF成功すれば
             THEN局所的にタイムスタンプ付きの FORWARD を行ない
                  (DECREMENT COUNT-OF-LWPS) が 0 であれば
                  SUCCESS の MESSAGE を送る。
             ELSEFAIL の MESSAGE を送り他のプロセスを全て殺す。
    ELSE
      NEW:== COMPLEMENT-ARCS(DAG2,DAG1)
      SHARED:==INTERSECT-ARCS(DAG1,DAG2)
      (ADD NUMBER-OF-SHARED to COUNT-OF-LWPS)
      FOR  arc IN SHARED DO
           MAKE-LWP(UNIFY1(arc に対応する DAG1,DAG2 の arc))
           但、現在のプロセス名を付ける。
      DAG2 から DAG1 への FORWARD を遅らせておく。
      DAG1 の comp-arc-list に NEW を入れる。
      同時に comp-arc-list にタイムスタンプをつける。
    ENDIF
ENDP
```

図 3: 並列化した時間差準破壊型単一化アルゴリズム

1. 結果が成功した場合 75.93%

2. 結果が失敗した場合 79.04%

であった。トップの unification が失敗する時は同時に走っているプロセスの終了を待たずにこれらを殺しているため並列度が高くなっている。

また、実際の並列化では 2 で述べたほかに次のような問題を考慮しなければならない。

1. 並列プロセスの世代管理

一つの unification が失敗した時には、トップのプロセスは失敗を知るとすぐに、現存しているサブプロセスのリストを参照して他のすべてのプロセスを強制的に終らせ、世代を更新して終る。次に新しいトップのプロセスが次の unification を始める。しかし実際にはサブプロセスが止まるまでにはスケジューラを介するために常にタイムラグがあり、殺されるサブプロセスが止まる直前に新しく生んでいる子プロセスは殺すことはできないという問題がある。これらの子プロセスや、またさらに生まれる孫プロセスは新しい unification の最中に fail や forward などの side effect を起こす可能性がある。この様子を図 4 に示す。

これを防ぐためには、side effect のある処理をプロセスが行なう前にプロセスがどの世代のトッププロセスに属するものかをチェックしなければならない。

2. 子プロセスへのデータ受渡し

再帰的に処理が進む場合でも、子プロセスを作る時は関数内の動的な変数の束縛などのローカルなスコープはすべて無効になる。つまり、引数を渡す場合でもプロセス間で共有できるデータ構造（グローバル変数、プロセス間メッセージなど）を用いなければならない。

3. 共有データの lock 管理

複数のプロセスが同じデータを使う時、read/read の衝突についてはマシンのアーキテクチャ上一般には問題にならないとされている。しかし、read/write, write/write の場合では、ハードウェアからコンパイラまでのいずれかのレベルで排他制御を行なうことが避けられないため、実際のアルゴリズムにはすべての共有データに対して同時のアクセスが起こり得ないことが極めて重要である。

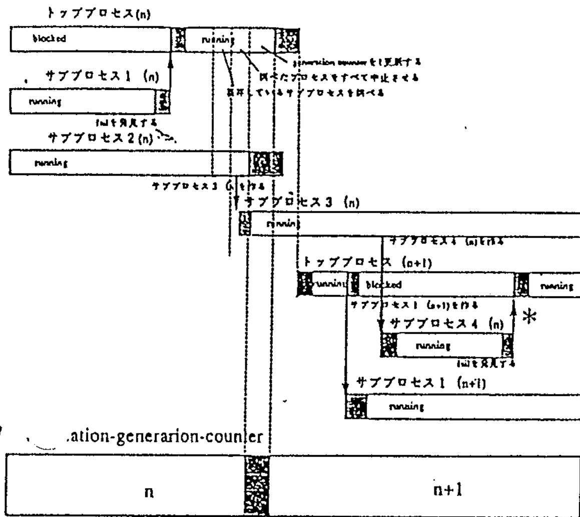
4. 局所的な forward の必要性

unification の最中では sub-graph が存在している場合は前述のように forward を遅延させるが、局所的に成功した場合（同一のノード同士の unification, 或はどちらか一方または両方が変数の時）は遅延させずにその場で forward しなくてはならない。もし

表 1: 並列処理実験の結果

R	ALL	MAX-PP	PP-RATE(%)	U-RATE(%)
T	4	3	75.00	26.36
T	5	4	80.00	4.31
T	3	2	66.67	4.15
T	2	1	50.00	3.38
T	5	5	100.0	2.21
T	1	1	100.0	1.44
T	6	5	83.33	1.18
T	4	4	100.0	1.07
T	7	6	85.71	0.41
T	8	7	87.50	0.26
T	6	6	100.0	0.15
NIL	6	5	83.33	15.18
NIL	5	4	80.00	10.31
NIL	4	3	75.00	8.15
NIL	2	1	50.00	6.31
NIL	5	5	100.0	4.87
NIL	3	2	66.67	3.49
NIL	7	6	85.71	2.62
NIL	8	7	87.50	1.54
NIL	6	6	100.0	0.87
NIL	2	2	100.0	0.72
NIL	4	4	100.0	0.31
NIL	10	9	90.00	0.21
NIL	7	7	100.0	0.21
NIL	3	3	100.0	0.21
NIL	9	8	88.89	0.10

<プロセス (n)はunification-generation n に属するプロセスであることをあらわす。
また▦の部分は処理が実行されるまでのタイムラグをあらわす。>



* サブプロセス1 (n)はunificationのfailを発見するとトッププロセス (n)にそれを知らせトッププロセス (n)は現存するサブプロセスm(n)をすべて調べ,中止させる。しかしサブプロセス2 (n)は中止させられる直前にサブプロセス3 (n)をつくり、これがgeneration-counterが更新されてからサブプロセス4 (n)をつくる。よってサブプロセス4 (n)がunificationのfailを発見するとトッププロセス (n+1)にfailを知らせてしまう。

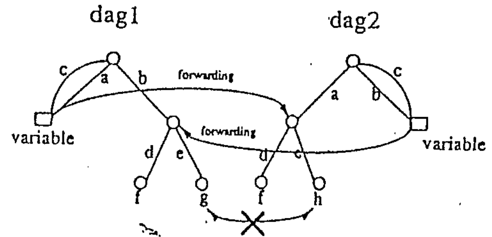
図 4: 並列プロセスの世代管理

この場合も遅延させるとそれによって unification の結果が誤ってしまうことが起こりうる。この例を以下の図 5 に示す。

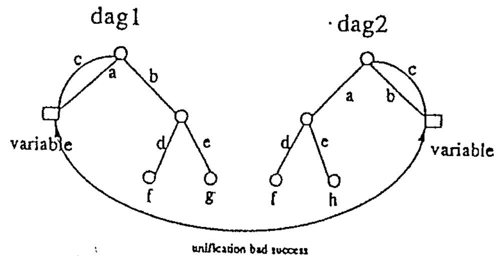
5 おわりに

Tomabechi の時間差準破壊型単一化アルゴリズムが並列化に適しているのは unification 中の graph の destructive な変更を lazy evaluation を使うことによって完全になくすことが可能であるため lock/unlock による問題を最小限にできるからである。また、各 unification は shared-arc への再帰的な呼び出しの結果を待たずに処理を進められるため unification 中の同期化の問題が発生しない。これらは、時間差準破壊型アルゴリズムの特徴であり、Wroblewski などのアルゴリズムでは本質的に実現不可能である。このように、Tomabechi のアルゴリズムを並列化した場合は各並列プロセスが sychronization やノードへの書き込みのコスト無しに、子プロセスを並列的に生みながら shared-arc を下っていけるので、unification の失敗を極めて早く発見することが可能である。このように並列化を行なうことにより Tomabechi のアルゴリズムの特徴である unification の失敗の早期発見がより効果的に行なわれる。以上のように、本稿で述べられたいくつかの手法を用いることにより Tomabechi

この場合arc a,arc bに対するunificationは片方がvariableなので局所的に成功する。



図(a)arc a,bに対するunificationが成功したあと局所的にforwardするとarc cについてunificationを行なった結果,全体で正しい結果(fail)が得られる.



図(b) arc a,bに対するforwardを遅延させるとarc cに対するunificationは両方variableなので成功してしまう.

図 5: 局所的 forward の必要性

のアルゴリズムを効果的に並列化できることがわかった。また現在、並列化によるスケジューリングと並列プロセスの同期化の over-head、 processor 数の変化といった因子と、graph の大きさや形などの変動要因の関係を実験中である。また、並列化に適する graph の特性を調べ、graph そのものを serial/parallel のどちらかに最適化するアルゴリズムも考えられるものと思われる。

参考文献

[Godden, 1990] Godden, K. "Lazy Unification" In *Proceedings of ACL-90*. 1990.

[Karttunen, 1986] Karttunen, L. *Development Environment for Unification-based Grammars*. Report CSLI-86-61. Center for the Study of Language and Information, 1986.

[Karttunen and Kay, 1985] Karttunen, L. and Kay, M. "Structure Sharing with Binary Trees". In *Proceedings of ACL-85*. 1985 .

[Kogure, 1989] Kogure, K. "Parsing Japanese spoken sentences based on HPSG". In *Proceedings of the International Workshop on Parsing Technologies,1989 (IWPT89)*.

[Pereira, 1985] Pereira, F. "A Structure-Sharing Representation for Unification-Based Grammar Formalisms". In *Proceedings of ACL-85*. 1985.

[Pollard and Sag, 1987] Pollard, C. and Sag, A. *Information-based Syntax and Semantics*. Vol 1, CSLI, 1987.

[Tomabechi, 1991] "Quasi-Destructive Graph Unification". In *Proceedings of The Second International Workshop on Parsing Technologies, 1991 (IWPT91)*.

[Wroblewski, 1987] Wroblewski, D."Nondestructive Graph Unification" In *Proceedings of AAAI87*. 1987.

[Yoshimoto and Kogure, 1988] 吉本啓、小暮潔 " 日本語端末間対話解析のための句構造文法 " 情報処理学会第３７回全国大会. 1988.