ICOT Technical Report: TR-427

TR-427

知識処理向き並列推論エンジン

北上　始、横田　治夫、服部　彰(富士通)

October, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

知識処理向き並列推論エンジン

Knowledge Processing Oriented
A Parallel Inference Engine

北上　始、横田治夫、服部　彰
Hajime KITAKAMI, Haruo YOKOTA, Akira HATTORI
㈱富士通
Fujitsu Limited

あらまし

ＡＮＤストリーム型並列言語ＧＨＣを使って全解探索用の並列推論エンジンの実現方法について述べる。実現に当たっては、ＯＲ並列とＡＮＤ並列の機能が双方必要であるが、ＧＨＣではＯＲ並列の機能を備えていないことからＯＲ並列を実現する際の問題点を分析し、その解決策について述べる。また、ＯＲ並列とＡＮＤ並列で実現した全解探索用の並列推論エンジンの測定結果についても述べる。

１．はじめに

知識処理システムでは、今後、大規模なデータを扱うような分析や診断問題・膨大な組み合わせを扱うような計画や設計問題・多大な計算を必要とするようなシミュレーション問題などを実時間で解く要求が高まってくるものと考えられている。これに答えるために、並列マシンの力を効率良く引き出すアプローチで、知識処理を高速化する研究が進められている。

知識処理は、人間の知識をデータとして知識ベースに登録し、①知識ベースから必要なデータを検索した後、②それを分析・加工する処理と見なすことができる。従来の並列処理では、これらを並列化するために、ＯＲ並列に力点を置く研究とＡＮＤ並列に力点を置く研究が別々に進められてきた。

ＯＲ並列に力点を置く研究では、ゴール列（g1, g2, ..., gn）のゴールを並列に実行するがゴール間に共有変数があると、各ゴールgiに対して、Ni個の解を求めた後に、それらの解集合に対して同じ値を見つける処理を行わなければならない。この処理では、同じ値を見つけるのに逐次処理で、o（N1×N2×....×Nn）回の判定が必要になり〔Kasif et al. 1983〕、明らかに効率が悪い。これを解決する方法としてハッシュを使って高速化したという研究〔Rong Yang 1987〕があるが、これも、所詮、逐次処理なので、効率がいいとはいえない。

一方、ＡＮＤ並列に力点を置く研究では、ＯＲ処理をＡＮＤストリーム型並列言語（GHC）へコンパイルする方式〔竹内　1987, 上田　1986〕があるが、この方法では、インクリメンタルに増減する知識ベースを、直接、管理することが難しい。また、この研究では、コンパイル時に変数の入/出力宣言をしなければならないので、利用者にとっては煩わしい制限になっている。このような方式の不都合を回避するために、ＡＮＤストリーム並列言語ＧＨＣを使って、ＯＲ並列とＡＮＤ並列を組み合わせる方法が考えられる。この方法が実現できれば、インクリメンタルに増減する知識ベースを対象とした全解探索用の並列推論エンジンが容易に実現され得る。

本報告では、知識処理システムをＯＲ並列とＡＮＤ並列の両方で並列化することにより、知識処理システムが十分に並列化され得ると考え知識処理システムの代表的な機能である全解探索用推論エンジンをＯＲ並列とＡＮＤ並列の両方で実現する方法とその測定結果について報告する。また、ここで使用した並列マシンは、現在、実験システムとして使っているSEQUENT社のSymmetry（8台のプロセサが共有メモリーで接続された構成）であり、インプリメンテーション言語は、Commited-Choice型言語GHCである。

ＧＨＣは単一代入言語なので、Prologのバックトラック機構に対応する変数の書き換え機構などを持たないので、知識ベースを持つ推論エンジンをインプリメントすることが難しいとされていた。ここでは、ＧＨＣに新しいデータタイプとして、$変数を導入する方法により、ＯＲ並列とＡＮＤ並列を旨く組み合わせることについて述べる。知識ベースには、$変数で表現されたホーン節の集合が格納されており、これに対してＯＲ並列とＡＮＤ並列の両方の機能をもつ全解探索用の推論エンジンの実現方法について報告する。さらに、ＧＨＣで実現した全解探索用推論エンジンの性能を知るために、家系図の問題を取り上げ、並列性能比の測定を行った。その結果、８台のプロセッサで約６倍の性能を得た。

本研究は、制約型論理プログラミング言語・知識獲得や学習などで必要とされる知識ベース管理・確信度付きの推論エンジン・時間論理を持つ推論エンジン・意味ネットワーク用の推論エンジン・プロダクションシステム用推論エンジンなどをＧＨＣで並列効果を出しながらプログラミングする際の指針を与えている。

２．全解探索用推論エンジン

知識処理は、知識ベース検索と検索された知識の分析・加工から構成されるが、これらの処理をまとめた例として論理型プログラミング言語 Prolog （Bowen 1981）による推論エンジンの研究がある。

このエンジンは、図１に示されるようにPrologの機能拡張を必要とすることからメタプログラミングの方式により実現されているが、メタプログラミング方式では、データとプログラムを区別しないプログラミングを行っているので知識ベースのデータも推論エンジンを作成するためのプログラムも同じホーン節の集合として表現されている。必要なデータ（知識）は、３番目のclause述語により検索され、知識の分析は２番目のプログラムによって行われる。検索では、帰結部Ｐを与え、それと単一化するホーン節が(P:-Q)として得られる。分析はＱについて進められる。３番目のclause述語が条件部を持たないホーン節を検索したとき、Q=trueとなり、１番目の停止条件を満たすようになる。

従って、図２に示すように、知識ベースに格納される知識としてのホーン節を表すための変数は、Prologの変数がそのまま使われているので推論エンジンで使われる変数と同じである。

```
solve(true):- !.
solve((P,Q)):- solve(P),solve(Q).
solve(P):- clause(P,Q),solve(Q).
```

図１．推論エンジンの例

```
ancestor(X,Y):-
            parent(X,Y).
ancestor(X,Y):-
            parent(X,Z),ancestor(Z,Y).

parent(f1,f2).
parent(f2,f3).
parent(f3,f4).
```

図２．知識ベースの例

以上のような推論エンジンを使うと、取敢えず解答を１つ求めることができるが、この処理は探索木のある一本の経路を辿る処理なので、沢山の経路を同時に辿るような意味での並列性がない。例えば、?- solve(ancestor(X,Y)) を実行するとX=f1,Y=f2 が計算されるが、目立った並列性がない。従って、これを拡張し、全解を求めるための推論エンジンbagof-solve を実現すると、それは、並列に複数解を求める処理として並列化が可能である。

３．ＧＨＣによる並列化

全解を求める推論エンジンを並列化する方法について述べる。この推論エンジンは、知識ベースから必要なデータを検索する処理と得られたデータを分析する処理から構成されるが、以下では、それらの処理をＧＨＣで並列化する際の問題点について述べる。

３．１　問題点

知識ベース検索をＧＨＣ上で実現する場合はＧＨＣのストリーム機能と合わせるために、知識ベース検索の結果をストリームで返す機能を実現する必要があり、そのためには、次のような処理が必要である。

①ストリームで解答をデータ転送
②必要なデータを探すときの単一化処理
③変数名の新規作成

①の処理は、ＧＨＣがもつ機能を利用することにより実現できるが、②の処理は、ＧＨＣの単一代入制限により、実現できない。即ち、２件以上のデータを探すことが出来なくなる。ま

た、③の機能は、ＧＨＣに存在しない機能であり、新機能として作成する必要がある。

以上により、②が大きな問題点になる。この処理で、単一代入制限を回避する方法として、ＧＨＣに項のコピー機能を持たせる方法があるが、この方法では、項の中の変数をコピーする時に変数の排他制御が必要であり、オーバーヘッドが大きいと考えられる。

この問題点は、全解探索用の推論エンジンにおいて、複数のデータを分析していく過程でも生じるので、解決しなければならない問題点である。

３．２　解決策

変数を含む項のコピーは、オーバヘッドが大きいが、変数を含まない項のコピーは特に問題が生じないことに着目し、データを表現する変数を特殊な定数で表現し、ＧＨＣの変数と区別する方法を考えた。この特殊な変数をＳ変数と呼び、知識ベースのデータ表現や知識ベース検索の条件指定の表現に利用することにした。図３に知識ベースの表現例を示す。知識ベース検索の条件指定は、次のように表される。

```
?- bagof-clause(ancestor($10,$11),Stream),
 Stream=
   [(ancestor($12,$13):-parent($12,$13)),
    (ancestor($14,$15):-...)]
```

データの変数をＳ変数で表すと、項の単一化処理をもはやＧＨＣの単一化命令で行うことが

```
ancestor($1,$2):-
            parent($1,$2).
ancestor($1,$2):-
         parent($1,$3),ancestor($3,$2).

parent(f1,f2).
parent(f2,f3).
parent(f3,f4).
```

図３．知識ベースの例

出来なくなるので、以下のような$変数を含む項を扱うための命令をＧＨＣで実現した。

① unify(TM1, TM2, NewTM, OutBinf)
TM1(項)とTM2(項)を単一化し、その結果をNewTM に返す。 OutBinfには、単一化の際に行われた$変数の代入状況を返す。単一化が終わっても、TM1 とTM2 の$変数は、書き換わることがない。

② substitute(InBinf, TM, NewTM, OutBinf)
InBinfの代入情報をもとにTM内$変数を書き換えNewTM にその結果を返す。OutBinf にはその時に行われた代入状況を返す。代入が終わっても、TM1 とTM2 の$変数は、書き換わることがない。

③ generate-newVL(InVList, OutVList)
InVList 中の$変数要素分だけ、新しい$変数を生成し、それらをOutVListに返す。

４．インプリメンテーション

４．１　システム構成

第４図は、ＧＨＣで作成した全解探索用の並列推論エンジンの構成図である。図４中の並列推論ドライバーは、与えられた問題（ゴール列）をＯＲ並列やＡＮＤ並列で解くための指示を行う。並列推論ドライバーがゴール列を受け取ると、それをＡＮＤ並列処理部に渡す。

ＡＮＤ並列処理部では、ゴール列を分解し、ゴールを並列推論ドライバー経由でＯＲ並列処理部に渡し、ゴールを実行する（ＯＲ並列処理部での実行の結果、複数の解答が得られる）。ゴール列の中に、p($1),q($1,$2)で示される共

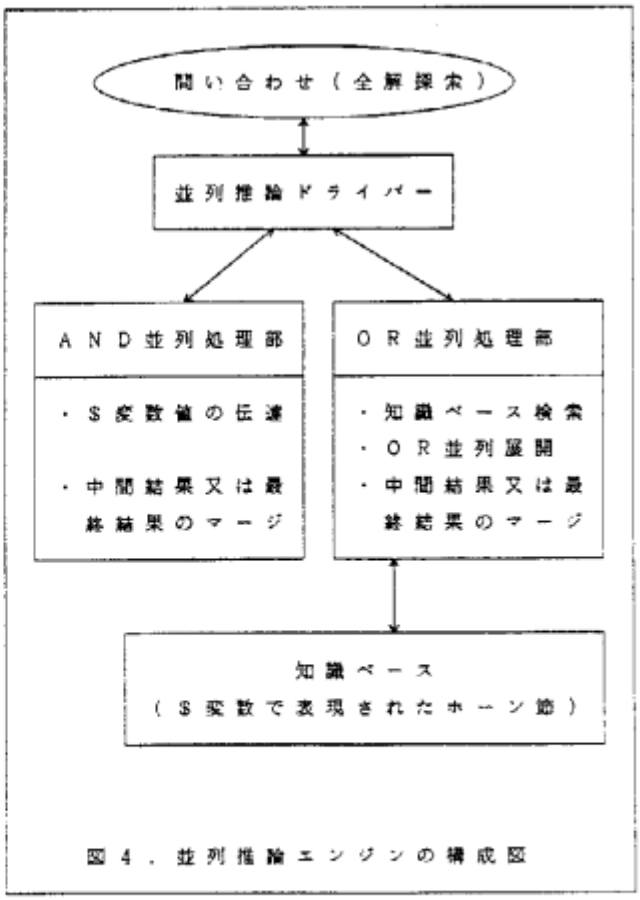

図４．並列推論エンジンの構成図

有変数（$1）があるとp($1)の複数解（例：$1=1,2,3,...）をq($1,$2)に伝達しなければならないので、そのための処理を行う。以上によりゴール列の複数解が求まるので、それらをマージする処理が行われる。

ＯＲ並列処理部では、並列推論ドライバーから渡されたゴールと単一化するデータ（知識）を知識ベースから検索し、検索結果として得られる複数のデータをストリームで、受け取る。その後、個々のデータを解釈するために、個々のデータを並列処理できるように並列展開を行う。

このようにして並列推論ドライバーは、ＯＲ並列とＡＮＤ並列により、並列推論を進め、ゴールが無くなるまで実行を進める。求められる結果は、

＜解答，Ｓ変数の代入状況，推論履歴等＞

の集合である。

付録に、本処理系による全解探索エンジンの作成例を示す。

第５図は、この実現プログラムの主要部分について着目した呼び出し関係グラフである。

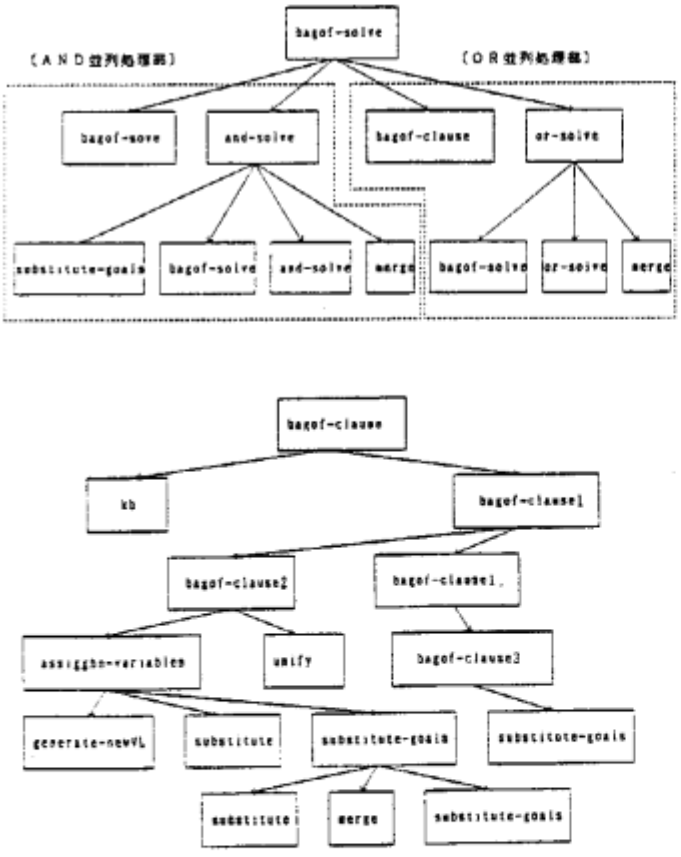

図５．並列推論エンジンの実現例

bagof-solveが全解探索を主目的とする並列推論エンジンのドライバーに相当する。

ＡＮＤ並列処理部の中のsubstitute-goalsは前述したＳ変数値の伝達を行う処理である。and-solveから呼び出されるbagof-solveとand-solveにより、ゴール列の論理積の関係が処理される。また、mergeでは、中間結果又は最終結果として、複数解がマージされる。

ＯＲ並列処理部の中のbagof-clauseが、Ｓ変数の単一化検索に相当する。bagof-solveとor-solveにより、複数解の並列実行を行い、並列実行により得られた個々の中間結果又は最終結果として、複数解がmergeによりマージされる。

４．２　並列動作

図３の知識ベースに対して、図６に示す問い合わせ"anc($10,$11)の全解探索"を行う問題について、ＯＲ並列処理及びＡＮＤ並列処理により解く手順を、図６から図８を用いて具体的に説明する。

図６において、①は、問い合わせ"ancestor($10,$11)の全解探索"が図４の並列推論ドライバーに入力された状態を示す。ancestor($10,$11)は単一ゴールなのでＯＲ並列処理部に渡され、ancestor($10,$11)に関係する知識ベース検索が行われる。

②は、ancestor($10,$11)に関係する知識ベース検索の結果として得られるデータの条件部を示す。処理を行うときに変数名による矛盾が生じないようにするために、新しい変数名が割当てられた条件部parent($12,$13)が示されている。parent($12,$13)はＯＲ並列処理部に渡され、parent($12,$13)に関係する知識ベース検索が行われる。これにより、③から⑤に示すＯＲ並列処理の解答が図示のように得られる。そして、(A)に示すように、②に対する複数解として、これら③から⑤の解をマージした結果がストリームとして得られる。

以上の②の処理と並行して、⑥の処理が行われる。⑥では、ancestor($10,$11)に関係する知識ベース検索の結果として、"ancestor($14,$15):-parent($14,$16),ancestor($16,$15)"が得られるので、その条件部が示されている。この条件部はゴール列であるので、これはＡＮＤ並列処理部に渡される。ＡＮＤ並列処理部では、先ず、ＯＲ並列処理部へparent($14,$16)のゴールを送り、⑦で⑧から⑩の解答を得る。これにより、($14,$16)の値として、(f1,f2),

(f2, f3), (f3, f4) の３つの値が得られるので、ゴール列内の共有変数$16に着目してancestor($16, $15) を書換え、ancestor(f2, $15), ancestor(f3, $15), ancestor(f4, $15)のそれぞれについて、⑪以下の処理を行うと、($16, $15) の値として(f2, f3), (f2, f4), (f3, f4) の３つの値が得られる。そして、⑧では、($14, $16) と($16, $15) の値を組み合わせた($14, $15) の値として、(f1, f3), (f1, f4), (f2, f4) が求まる。

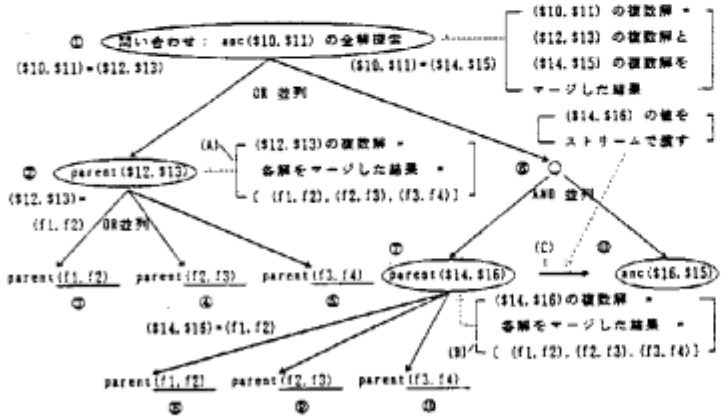

図６．並列動作例－１

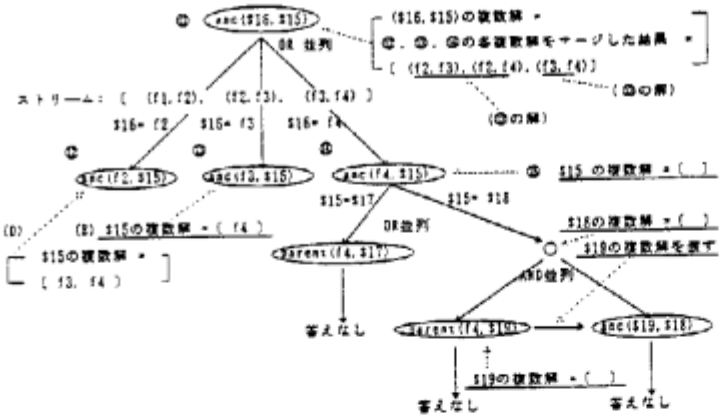

図７．並列動作例－２

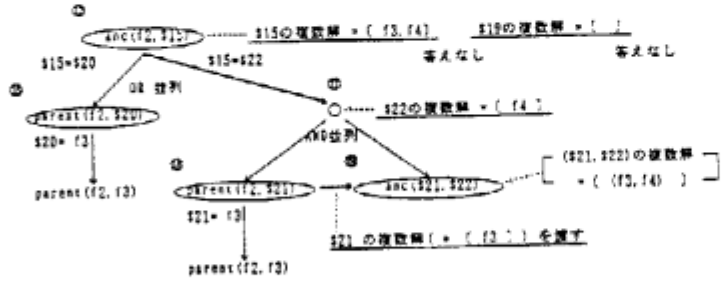

図８．並列動作例－３

以上から、①の問い合わせの解答を出力するために、ＯＲ並列処理部では、②と⑥の解答をマージする。これにより、次のような複数解が得られる。

($10, $11)=(f1, f2), (f2, f3), (f3, f4),
(f1, f3), (f1, f4), (f2, f4)

以下、($16, $15) の解答を得るための⑪以下の処理について、図７と図８を用いて、具体的に説明する。図７において、⑫から⑭は、ストリームとして渡された各解答をもとに設定された新たなゴールである。(D), (E) は、それぞれ⑫と⑬の解答である。⑭の解答は、図に示すように、存在しない。⑮は、図８で、⑮から⑲に示すように、前述したＯＲ並列処理とＡＮＤ並列処理を行うことにより、解(D) を得る。同様に、⑬に対して、解(E) を得る。また、⑭に対しては、解答が存在しないという結論を得る。このような結果から、(D), (E) をマージすると前述した($16, $15) の解答が得られる。

図８は、図７の⑫の解答を得る手順を示す。これは、図６で説明したのと同じ方法で順次解答を求めている。

４．３　測定結果

図３の知識ベースを使い、図６で示される問い合わせの台数効果を測定した。この知識ベースでは、ＯＲ並列処理部で２１回の知識ベース検索の要求が出されており、８６％以上がＯＲ処理であることが分かった。図９に台数効果を表す測定結果を示す。さらに、この問題は、データ量を増やすと、並列度が高くなることに着目し、並列性能がどの程度上がるかの測定を行った。その結果を図１０に示す。

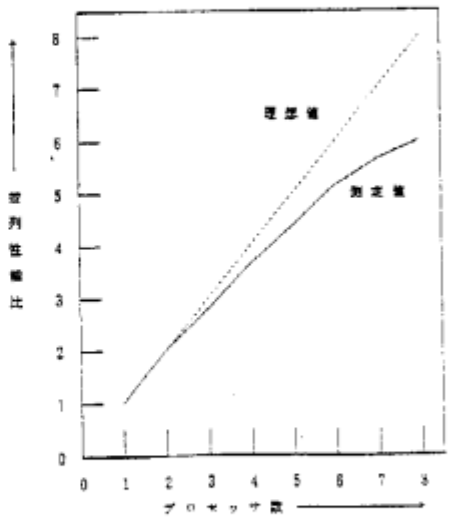

図９．台数効果

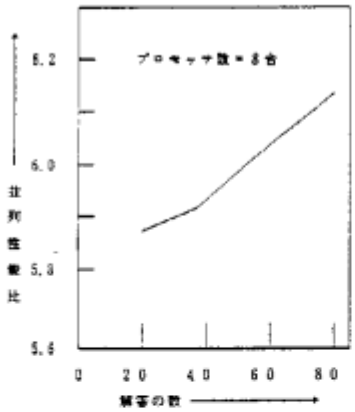

図１０．問題のサイズに対する並列性能比

５．まとめ

ＧＨＣを使い、ＯＲ並列処理部とＡＮＤ並列処理部をもつ全解探索用の並列推論エンジンを試作！　並列効果が出ることを確認した。しかし、Ｓ変数を含む項の単一化や代入は、ＧＨＣ自身で書いているため、知識ベース検索のスピードに問題がある。また、知識ベースのデータ量が増大するにつれ、益々、重要な問題になってくる傾向がある。

現在、この知識ベース検索機能を、ＧＨＣよりも下位の言語で作成しており、これを、ＧＨＣと結合を行っている。

〔謝辞〕

本研究に当たり、日頃から有益なコメントを下さったＩＣＯＴ第三研究室・伊藤室長、岡富士通研究所・林人工知能研究部長、ＩＣＯＴのＫＢＭメンバーの方々に深謝致します。

〔　参考文献　〕

〔１〕Bowen, D.L.: DEC-10 PROLOG USER'S MANUAL, University of Edinburgh (1981).

〔２〕S.Kasif, M.Kohli and J.Minker: 'PRISM: A Parallel inference system for Problem Solving', Proc. of the Logic Programming Workshop 83, pp.123-152 (1983).

〔３〕K.Ueda: Guarded Horn Clauses, Technical Report TR-103, ICOT (1985).

〔４〕S.Takeuchi: 並列問題解決用言語 ANDOR-Ⅱ, KP-87-10 (1987).

〔５〕K.Ueda: " Making Exhaustive Search Programs Deterministic Ⅱ ", Proc. of 3rd Conf. of Japan Society of Softoware Science and Technology (1986).

〔６〕Rong Yang: P-Prolog: A Parallel Logic Programming Language, Series in Computer Science - Vol.9, World Scientific Publishing Co Pte Ltd. (1987).

〔付録〕全解探索用の並列推論エンジンのプログラム例

```
kb(testkb,X):-true|
    X=[[anc($(1),$(2)),parent($(1),$(2)),[$(1),$(2)]],
       [anc($(3),$(4)),(parent($(3),$(5)),anc($(5),$(4))),[$(3),$(4),$(5)]],
       [parent(f1,f2),true,[]],
       [parent(f2,f3),true,[]],
       [parent(f3,f4),true,[]]].

test(Goals,VL,Results,N):-true|
    bagof_solve(testkb,Goals,VL,[],[],Results,1,N).

bagof_solve(KB,true,VL,Binf,His,Result,PE,N):-true|
    Result=[[VL,Binf,His]].
bagof_solve(KB,(P,Q),VL,Binf,His,Result,PE,N):-true|
    bagof_solve(KB,P,VL,Binf,His,Result1,PE,N),
    and_solve(KB,Q,Binf,His,Result1,Result,PE,N).
bagof_solve(KB,P,VL,Binf,His,Result,PE,N):-PF=(Q,R)|
    bagof_clause(KB,P,ClauseList),
    or_solve(KB,ClauseList,VL,Binf,His,Result,PE,N).

and_solve(KB,Q,Bindis,History,[[R1,Binf,His]|RT],Result,PE,N):-wait(Q)|
    substitute_goals(Binf,Q,Q1,_),
    bagof_solve(KB,Q1,R1,Binf,His,Result1,PE,N),
    and_solve(KB,Q,Bindis,History,RT,Result2,PE,N),
    merge(Result1,Result2,Result).
and_solve(KB,Q,Bindis,History,[],Result,PE,N):-wait(Q)|Result=[].

or_solve(KB,[[(Head:-Body),Inf]|ClauseList],VL,Binf,His,Result,PE,N):-true|
    replaceVL(Inf,VL,NewVL),
    append(Inf,Binf,BinfNew),
    append(His,[(Head:-Body)],HisNew),
    bagof_solve(KB,Body,NewVL,BinfNew,HisNew,Result1,PE,N),
    or_solve(KB,ClauseList,VL,Binf,His,Result2,PE,N),
    merge(Result1,Result2,Result).
or_solve(KB,[],VL,Binf,His,Result,PE,N):-true|Result=[].

substitute_goals(Sub,Goals,SGoals,BindOut):-wait(Sub)|
    substitute_goals(Sub,Goals,SGoals,[],BindOut).

replaceVL([Inf,[Y|YL]],NewVL):-true|
    my_element(Y,Inf,NY),replaceVL([Inf,VL,NewVL],
    merge([NY],NewVL1,NewVL).
replaceVL([Inf,[]],NewVL):-true|NewVL=[].

substitute_goals(Sub,(G1,G2),SGoals,BindHis,BindOut):-true|
    substitute(Sub,G1,SG1,Bind),
    SGoals=(SG1,SG2),
    merge(BindHis,Bind,BindHis1),
    substitute_goals(Sub,G2,SG2,BindHis1,BindOut).
substitute_goals(Sub,G,SGoal,BindHis,BindOut):-GY=(G1,G2)|
    substitute(Sub,G,SGoal,Bind),
    merge(BindHis,Bind,BindOut).

bagof_clause(KB,Head,SetofClause):-true|
    kb(KB,ClauseList),
    bagof_clause1(Head,ClauseList,SetofClause).


bagof_clause1(Head,[C|CL],SetofClause):-true|
    bagof_clause2(Head,C,SetofClause,SetofNewClause),
    bagof_clause1(Head,CL,SetofNewClause).
bagof_clause1(Head,[],SetofClause):-true|SetofClause=[].

bagof_clause2(Head,[H,B,YL],SetofClause,SetofNewClause):-true|
    assignVLs(YL,H,B,VL1,H1,B1),
    unify(Head,H1,Inf),
    bagof_clause3(Head,H,B1,Inf,SetofClause,SetofNewClause).

bagof_clause3(Head,H,B,Inf,SetofClause,SetofNewClause):-
    PF=fail|
    substitute_goals(Inf,B,B1,_),
    SetofClause=[[H,-B1],Inf]|SetofNewClause].
bagof_clause3(Head,fail,H,Inf,SetofClause,SetofNewClause):-true|
    SetofNewClause=SetofClause.

assignVLs(VL,H,B,VL1,H1,B1):-wait(H)|
    generate_newVL(VL,VL1),
    substitute(VL1,H,H1,_),
    substitute_goals(VL1,B,B1,_).

generate_newVL([$(N)|V1],VL1):-true|
    genvar(N1),
    VL1=[b(N,$(N1),0)|V1],
    generate_newVL(V1,V1).
generate_newVL([],YL):-true|VL=[].
```