ICOT Technical Report: TR-464

TR-464

並列プログラム変換／可視化システム
VISTAにおける可視化技術について

奥村　晃、藤田　博、上田　和紀、
長谷川隆三、吉田　紀彦(九大)、
相川　聖一、小野　越夫(富士通)

March, 1989

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

並列プログラム変換／可視化システムＶＩＳＴＡにおける可視化技術について

Visualizing parallel programs using the VISTA system

奥村 晃, 藤田 博, 上田和紀, 長谷川隆三,
Akira OKUMURA, Hiroshi, FUJITA, Kazunori UEDA, Ryuzo HASEGAWA,
(財) 新世代コンピュータ技術開発機構
ICOT Research Center

吉田紀彦, 相川勝一, 小野越夫
Norihiko YOSHIDA, Seiichi AIKAWA, Etsuo ONO
九州大学 富士通 (株)
Kyushu University Fujitsu Ltd.

GHC等の並列言語では通信し合う複数のプロセスによって計算が行われるため，プログラムの実行の様子を直感的に捉えることが逐次型言語に比べて困難である．現在構築中の並列プログラム変換／可視化システムVISTAは，並列プログラミングのための統合的支援環境の提供を目指しており，上記の問題の解消に向けて，プログラムの持つ構造や実行の様子をグラフィック表示する可視化機構を備えている．試作した可視化機構に付いて，その目的と機能を中心に紹介を行う．

It is rather difficult to understand behavior of GHC programs than of any sequential programming languages, because there exist so many communicating parallel processes in execution. The visualization system on VISTA (a VISualization and Transformation Apprentice for parallel logic programming) is a countermeasure for it. It shows program structures and execution graphically. In this article, we present a summary of it, especially about its purposes and functions.

1．はじめに

ICOTでは，並列論理型言語GHC[1]をベースに，仕様記述・検証，定理証明，プログラム変換，部分計算，プログラム可視化等の要素技術を統合した並列プログラミング支援システムを検討中である．VISTAはこれらの技術の内，プログラム変換，部分計算およびプログラム可視化技術をツール化した実験システムであり，当面並列プログラミングの教育や並列プログラムのデバグ支援等への応用を目指している[2]．VISTAの全体構成を図１に示す．

本システムは，プログラム変換／部分計算器に加えてプログラムの静的構造及び動的振舞をGHCのプロセス（ゴール）レベルで可視化する機能を備えており，レイヤードストリームプログラミングなどの並列プログラミングパラダイムの説明，部分計算によるプログラム構造の変換過程の説明など，様々な用途に用いることができる．

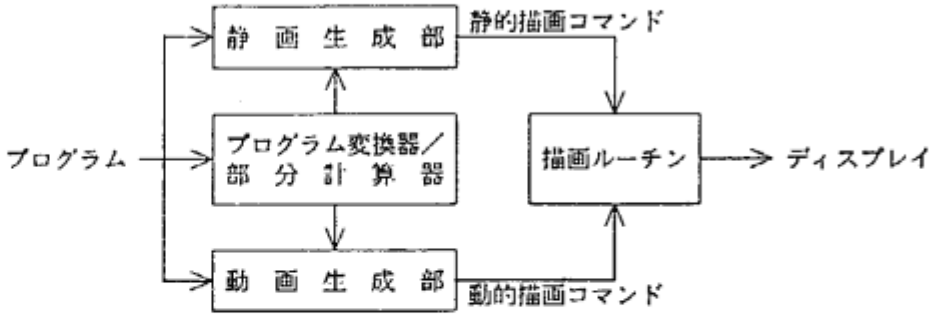

図1．ＶＩＳＴＡの全体構成

本稿では試作したVISTAシステムの可視化機構について，目的と現在実現されている機能を中心に概説する．試作段階であるので実現方式については省略する．

2．目的

GHCプログラムでは複数のプロセスが協調して計算を行うため，プログラムコードから全体の処

理の流れを読み取るのは，逐次型言語と比べて一般に困難である．このことは特に他人の作ったプログラムや，プログラム変換／部分計算で自動生成されたプログラム等を理解する上で障害となっている．

VISTAの可視化機構の目的は，この障害を少しでも回避するため，GHCプログラムの構造や実行の様子を視覚的に提示して，プログラム理解を支援することにある．具体的には，対象とするプログラムについて解析を行い，どのようなプロセスが発生しどことどこの間にプロセスの間通信が起こるかといったことについて，PSI-IIのビットマップディスプレイ上に線画出力するものである．

これによって，
- 実際に並列に計算が進行する様子が直感的に認識できるため，プログラミングの際のイメージ作りが容易になる．
- できたプログラムの動きを他人に説明するための道具となる．
- 起動されながらずっとサスペンドしたままのプロセス等，計算のボトルネックになっている部分の発見を助け，プログラムの最適化に寄与する．
- デッドロックの原因となっているプロセスの発生個所が掴み易くなるので，デバッグにも役立つ．
- 計算量のおおまかな見積もりや通信の集中する場所が把握できるので，プロセッサの割り当てを考える時の判断材料になる．
- プログラム変換や部分計算によってプログラムがどう変化したかを調べるのに，処理前と処理後のプログラム構造を可視化して比較検討することができる．

といった効果が考えられる．

3．対象とするプログラム

現在のVISTAで可視化の対象としているのは，プロセス間の入出力関係に循環のないGHCプログラムである．「循環のない」とは入出力関係に基づいてプロセスの間に半順序関係が定義できるということで，任意のプロセスを出発点として入力から出力へと順にプロセスを辿っていった場合に，決して最初のプロセスに戻ることがあってはいけない．これはストリームの流れる方向を左から右へ，上から下へという様に単純化して描画を容易にするために導入されている制限で，将来はこれを取り払う方向で検討している．

この条件を満たすプログラムの例として，レイヤードストリーム法[3]による４クイーンのプログラムを次に示す．以下ではこのプログラムに基づいて説明を行う．

```
fourQueen(Q4) :- true |
        q(begin,Q1),
        q(Q1,Q2),
        q(Q2,Q3),
        q(Q3,Q4).

q(In,Out) :- true |
        filter(In,1,1,Out1),
        filter(In,2,1,Out2),
        filter(In,3,1,Out3),
        filter(In,4,1,Out4),
        Out = [1*Out1,2*Out2,3*Out3,4*Out4].

filter(begin,_,_,Out) :- true | Out = begin.
filter([],_,_,Out) :- true    | Out = [].
filter([I*_|Ins],I,D,Out) :- true      | filter(Ins,I,D,Out).
filter([J*_|Ins],I,D,Out) :- D =:= I-J | filter(Ins,I,D,Out).
filter([J*_|Ins],I,D,Out) :- D =:= J-I | filter(Ins,I,D,Out).
filter([J*In1|Ins],I,D,Out) :- J \= I, D =\= I-J, D =\= J-I |
        D1 := D+1, filter(In1,I,D1,Out1),
        filter(Ins,I,D,Outs),
        Out = [J*Out1|Outs].
```

4．目標とする表示

表示するものは，基本的にはプロセスとプロセス間の共有変数の二つだけである．VISTAでは，プロセスは矩形で表し，共有変数は矩形間を折れ線で結合して表す．

表示の方式には大別して，プログラムの概略構造を示す静的表示と，実際の実行の様子を示す動的表示の二つがある．前者はトップゴールを指定されたレベルまで展開した結果を表示するもので，後者はプログラム実行時に発生するプロセスに付いて逐一表示を行うものである．

5．静的表示

静的表示では与えられたゴールを対象プログラムによってある程度展開し，実行時に発生するプロセスとプロセス間の入出力関係について解析・表示を行う。どこまで展開した段階での表示を行うかについては，ユーザが述語毎に指示を与えるものとする．この指示を描画指示と呼ぶ．

解析ルーチンは指示で要求されているレベルまでプログラムの構造を解析し，相対的配置を決める．それに基づいて描画ルーチンは表示の大きさや場所を決定し共有変数を示す結線とともに表示する．処理の流れを図２に示す．

静的表示からはプログラムの持つおおまかなプロセス構造を知ることができる．特にいくつかの永続的なプロセスが通信しながら処理を進めるようなプログラムを理解するのに有効であり，逆にプロセス構造が実行中にどんどん変化するようなものには向いていない．

４クイーンプログラムについての静的表示の例を図３に示す．これはfourQueen/1とq/2についてのみ展開を行いfilter/4については展開を抑制した場合の表示である．呼び出しの親子関係は矩形の入れ子で表現される．つまり，fourQueenの矩形の中にqの矩形が４つあるのは，fourQueenの呼び出しに対して４つのqがサブゴールとして発生することが示されている．図中の右向き三角形は出力同一化ゴールを示している．データの流れは左から右であり，各qにおいて４つのfilterからの出力がまとめられてqの出力となって隣のqに渡されるというプロセス構造が読み取れる．

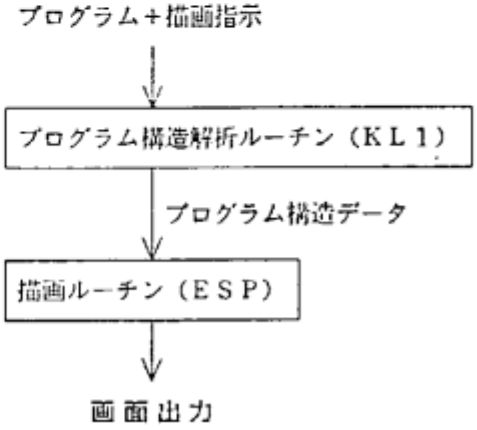

図２．静的表示における処理の流れ

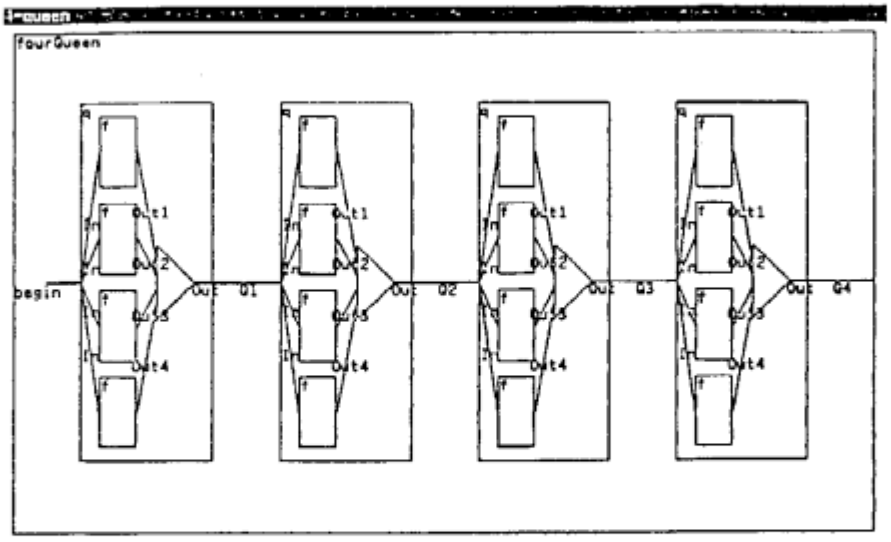

図３．静的表示の例

描画指示は述語毎に行うことになっているので，描画指示データの形式は述語指定と描画モードの対のリストの形をとる．

描画指示データ ::== [(述語名/引き数の数，モード),...]

描画モードには，always，once，never，throughの4つがあり，以下の様な意味を持っている．

always … その述語の呼び出しを示す矩形を表示し，さらにそのサブゴールについても処理を継続する．
once … その述語の呼び出しを示す矩形を表示する．もしその呼び出しが同じ述語の呼び出しからの再帰呼び出しでないならば，さらにそのサブゴールについても処理を継続する．
never … その述語の呼び出しについては矩形を表示するが，そのサブゴールについては何も表示しない．
through … その述語の呼び出しについては何も表示しないが，そのサブゴールについては処理を継続する．

この4つのモードを指定することにより望みの深さまでの解析が行われ，表示される．図3の表示を得るための描画指示は次の様になる．

```
[(fourQueen/1,always),(q/2,always),(filter/4,never)]
```

これによって，fourQueen/1やq/2はボディを展開して解析を行うが，filter/4についてはサブゴールの解析・表示を一切行わない．

静的表示においてプログラム構造はserial結合とparallel結合の再帰構造として扱われる．serial結合とは，入出力の依存関係が存在するため左から右へと横方向に配置すべきプロセスの結合状態である．それに対しparallel結合は依存関係が存在しないばあいのものであり，縦方向に配置される．図3の表示は描画ルーチンに渡されるプログラム構造データとしては，次の様に表される．尚，filterは省略してfにしてある．compは出力同一化ゴールに対応する．

```
goal(fourQueen([],[],[nil='Q4'],[]),
    serial([
        goal(q([v(begin)='In'],[],['Q1'='Out'],[]),
            serial([
                parallel([
                    goal(f(['In'=nil],[],['Out1'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out2'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out3'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out4'=nil],[]),nil) ]),
                comp(['Out1','Out2','Out3','Out4'],'Out') ])),
        goal(q(['Q1'='In'],[],['Q2'='Out'],[]),
            serial([
                parallel([
                    goal(f(['In'=nil],[],['Out1'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out2'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out3'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out4'=nil],[]),nil) ]),
                comp(['Out1','Out2','Out3','Out4'],'Out' )])),
        goal(q(['Q2'='In'],[],['Q3'='Out'],[]),
            serial([
                parallel([
                    goal(f(['In'=nil],[],['Out1'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out2'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out3'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out4'=nil],[]),nil) ]),
```

```
            comp(['Out1','Out2','Out3','Out4'],'Out') ])),
        goal(q(['Q3'='In'],[],['Q4'='Out'],[]),
            serial([
                parallel([
                    goal(f(['In'=nil],[],['Out1'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out2'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out3'=nil],[]),nil),
                    goal(f(['In'=nil],[],['Out4'=nil],[]),nil) ]),
                comp(['Out1','Out2','Out3','Out4'],'Out') ]))    ])).
```

描画ルーチンはプログラム構造データをもとにしてPSIのビットマップディスプレイ上に線画を生成する．表示の領域は最初にマウス操作等で指示された領域を親のゴールから順々に割り当てていくが，入れ子の一番深いレベルにおける表示の大きさが均等になるように調整される．プログラム構造データはトップダウンに解釈され概ね以下の様に出力を決定する．

goal(GoalInf,SubGoal) … 割り当てられた領域に矩形を表示する．GoalInfに基づいてゴール名の表示や他のゴールとの変数共有を示す結線を行う．矩形内の領域はSubGoalに分配される．
comp(Vars,VarName) … 右向き三角形を表示する．Varsで指示された変数に対応する結線を左底辺に行い，VarNameの変数の結線は右頂点から行う．
serial(SerGoals) … 領域を縦割りにしてSerGoalsの要素に左から順に割り当てる．
parallel(ParGoals) … 領域を横に分割してSerGoalsの要素に上から順に割り当てる．

この他にも，上向き，左向き，下向きの出力同一化ゴール用の三角形や，結線用の領域を確保するための仮想矩形等がある．

6．動的表示

動的表示では元のプログラムに描画ルーチンの呼び出しゴール（描画コマンドと呼ぶ）を挿入し，その描画コマンド付きのGHCプログラムを実際に走らせることにより描画コマンドが実行され，描画ルーチンに必要な情報（ゴールの分割，消滅など）が送られる．描画ルーチンはその情報に基づいて逐一表示を更新して動画を得る．処理の流れを図4に示す．

表示内容は静的表示の場合と共通する部分が多いが，呼び出しの親子関係に付いては矩形の入れ子ではなく，矩形の置き換えで示される．また，静的表示ではすべて解析してから表示の大きさを決定できるので，一番入れ子の深いレベルでの表示サイズの均等化が可能であったが，動的表示では，実際の実行に合わせて表示を変更するため，トップダウンに均等分割することにしている．

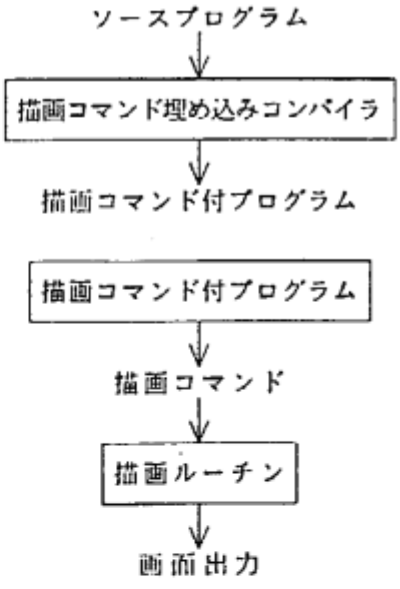

図4．動的表示における処理の流れ

描画コマンドは次の3つである．

:initialize(#visualizer,Options)
　　Optionに従って初期設定を行う．表示領域の指定等はここで行う．
:fork(#visualizer,OriginalGoal,ChildrenStructure)
　　OriginalGoalの表示を終了し，その領域を用いてChildrenStructureの表示を行う．
:vanish(#visualizer,Goal)
　　Goalがサブゴールを発生しないで計算を終了した場合で，Goalに対応した矩形の表示が消される．

先の4クイーンプログラムに描画コマンドを挿入した結果を次に示す．

```
fourQueen(Q) :- true |
    esp((:initialize(#visualizer,[
            size(1231,747),
            position(48,0),
            title("4-queen"),
            r(0.2,0.2),
            stream_name(off),
            value(clipping),
            data(on),
            layout(center) ]))),
    fourQueen([],nil=Q).

fourQueen(ID,A1=Q4) :- true |
    esp(    :fork(#visualizer,fourQueen(ID,[],[],[A1='Q4'],[]),
                serial([
                    q([v(begin)],[],['Q1'],[]),
                    q(['Q1'],[],['Q2'],[]),
                    q(['Q2'],[],['Q3'],[]),
                    q(['Q3'],[],['Q4'],[])]) ) ),
    q([1|ID],v(begin)=begin,'Q1'=Q1),
    q([2|ID],'Q1'=Q1,'Q2'=Q2),
    q([3|ID],'Q2'=Q2,'Q3'=Q3),
    q([4|ID],'Q3'=Q3,'Q4'=Q4).

q(ID,A1=In,A2=Out) :- true |
    esp( :fork(#visualizer,q(ID,[A1='In'],[],[A2='Out'],[]),
      serial([
        parallel([
            serial([filter(['In'],[],['Out1'],[]),data(1,'Out1','AST1')]),
            serial([filter(['In'],[],['Out2'],[]),data(2,'Out2','AST2')]),
            serial([filter(['In'],[],['Out3'],[]),data(3,'Out3','AST3')]),
            serial([filter(['In'],[],['Out4'],[]),data(4,'Out4','AST4')])]),
      comp(['AST1','AST2','AST3','AST4'],'Out') ]) ) ),
    filter([1|ID],'In'=In,1,1,'Out1'=Out1),
    filter([3|ID],'In'=In,2,1,'Out2'=Out2),
    filter([5|ID],'In'=In,3,1,'Out3'=Out3),
    filter([7|ID],'In'=In,4,1,'Out4'=Out4),
    Out = [1*Out1,2*Out2,3*Out3,4*Out4].

filter(ID,A1=begin,_,_,A4=Out) :- true |
    esp(    :fork(#visualizer,filter(ID,[],[],[A4='Out'],[]),
                comp([v(begin)],'Out') ) ),
    Out = begin,
    esp(    :vanish(#visualizer,
               comp([1|ID],[v(begin)],'Out') ) ).
filter(ID,A1=[],_,_,A4=Out) :- true |
    Out = [],
    esp(    :vanish(#visualizer, filter(ID,[],[],[],[]))).
filter(ID,A1=[I*_|Ins],I,D,A4=Out) :- true      | filter(ID,A1=Ins,I,D,A4=Out).
filter(ID,A1=[J*_|Ins],I,D,A4=Out) :- D =:= I-J | filter(ID,A1=Ins,I,D,A4=Out).
filter(ID,A1=[J*_|Ins],I,D,A4=Out) :- D =:= J-I | filter(ID,A1=Ins,I,D,A4=Out).
filter(ID,A1=[J*In1|Ins],I,D,A4=Out) :- J \= I, D =\= I-J, D =\= J-I |
    esp(    :fork(#visualizer,filter(ID,[A1='In1',A1='Ins'],[],[A4='Out'],[]),
                serial([
```

```
                    parallel([
                        serial([
                            filter(['In1'],[],['Out1'],[]),
                            data(J,'Out1','AST')]),
                        filter(['Ins'],[],['Outs'],[])]),
                    comp(['AST','Outs'],'Out')]) ) ),
    D1 := D+1,
    filter([1|ID],'In1'=In1,I,Di,'Out1'=Out1),
    filter([3|ID],'Ins'=Ins,I,D,'Outs'=Outs),
    Out = [J*Out1|Outs],
    esp( :vanish(#visualizer,comp([4|ID],[],[],[],[]))).
```

7．可視化例

可視化技術の応用として，部分計算[4]の効果を確認する例をあげる．ここでは並列パーサMeta-PAX[5]の部分計算を考えてみよう．Meta-PAXは与えられた入力文と文法から可能なあらゆる構文木をボトムアップに構成する汎用の構文解析インタプリタである．文法が固定された段階で，この汎用Meta-PAXはその文法専用のパーサに特化される．Meta-PAXのプログラム部分は次のように与えられている．

```
expand1(Cat,InS,OutS):-true|
  rules(Cat,Rules),
  expand1_1(Rules,InS,OutS).
expand1_1([Head|Rules],InS,OutS):-
    atomic(Head)|
  expand1(Head,InS,OutS1),
  expand2(Head,InS,OutS2),
  expand1_1(Rules,InS,OutS3),
  merge(OutS1,OutS2,OutS3,OutS).
expand1_1([(R->Head)|Rules],InS,OutS)
    :-true|
  expand1_1(Rules,InS,OutS1),
  merge([(R->Head)*InS],OutS1,OutS).
expand1_1([],_,OutS):-true|OutS=[].
```

文法は次のように与えられる．

```
rules(v,R):-true|
  R=[vp,(np->vp),(adv->vp)].
```

この文法規則について特化されたプログラム部分は次のようになる．

```
expand1(v,InS,OutS) :- true |
  expand2(vp,InS,OutS2_226),
  OutS=[(np->vp)*InS,(adv->vp)*InS|OutS2_226].
```

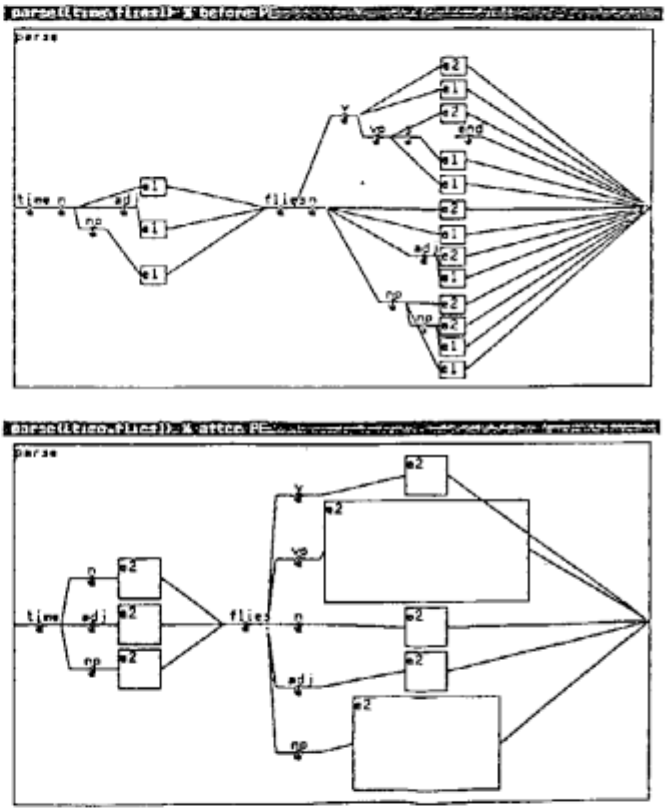

図5．部分計算前（上），後（下）のMeta-PAXの動作

部分計算前後のMeta-PAXパーサの動作はVISTAの動的表示機能を用いて図5のように表示される．表示から部分計算によって呼び出しの深さや数が減って計算量が少なくなっていることが容易に理解できる．

8．おわりに

並列論理プログラムの変換／可視化システムVISTA の可視化技術について述べた．現版のVISTAでは，プログラム構造の可視化方式として，トランスレータを介してプログラム中に可視化メソッドを埋め込む方式を採っているが，プログラム部分と可視化部分の分離を図るため，部分計算による可視化メソッドのコンパイル方式の導入を検討中である．

今後は，より理解性に優れた視覚化技法の検討を進めるとともに，制約プログラミング等のパラダイムをも扱えるシステムへ発展させていく予定である．

参考文献

[1] K. Ueda: "Guarded Horn Clauses", ICOT Tech Report TR-103, 1985 (revised in 1986). Revised version in Proc. Logic Programming '85, E. Wada (ed.), LNCS 221, Springer-Verlag, pp.168-179, 1986.

[2] 奥村 晃 他：「VISTA: 並列論理プログラム変換／可視化システム」，電子情報通信学会1989年春季全国大会 ＳＤ－６，1989.

[3] 奥村 晃，松本裕治：「レイヤードストリームを用いた並列プログラミング」，The Logic Programming Conference '87 論文集，pp.223-232，1987.

[4] 藤田 博：「FGHCプログラムの部分計算」，日本ソフトウェア科学会第5回大会論文集，pp.373-376，1988.

[5] Y. Matsumoto: "A Parallel Parsing System for Natural LAnguage Analysis", Proc. 3rd ICLP, 1986.