K-Prolog：並列マシン上でのPrologの実現

田村直之、有尾隆一、松田秀雄、金田悠紀夫、前川禎男
（神戸大学工学部システム工学科）

1. 概要

Prologの並列方式には大きく (1) OR並列 (2) AND並列 の2通りがある。OR並列（[1],[2]）は、すべての可能性について並列に実行するため、バックトラック処理を行う必要はない。しかし、並列に実行すべきプロセスの個数が増大し、実行不可能になる恐れがある。一方AND並列の場合は、並列に求めた解を統合する（たとえばP&QでPの解とQの解を並列に求めていき、P&Qを満たす解を選ぶ）点に困難がある。

そこで、ここではストリーム並列（バックトラック解の先取り探索）[3]、[11]に注目し、並列計算機（ブロードキャストメモリ結合形並列計算機[9]）上でのその実現方法について述べる。ここで言うストリーム並列とは、例えばP&Qにおいて、Pをproducer、P&Qをconsumerと考え、Pの解をパイプライン的にあらかじめ求めておく方法である。

ストリーム並列はセミ・コルーチン（親子関係のあるコルーチン）[8]を用いれば簡単に記述できる。[11] セミ・コルーチンはPrologインタプリタ本体（denoteと呼ぶことにする）の動作を考える上で、重要な概念である。

以下では、本システムのハードウェア構成およびソフトウェア構成について述べていく。

2. ハードウェア構成

図1にハードウェア構成を示す。[9] システムは1台のホスト・プロセッサ（Z80）と4台のスレーブ・プロセッサ（8086）で構成されている。

スレーブ・プロセッサ間は、データ・メモリおよびパケット・メモリと呼ばれる2種類の共有メモリを介して結合されている。また、ホスト・プロセッサからも任意のスレーブ・プロセッサのデータ・メモリもしくはパケット・メモリを自由に参照できる。

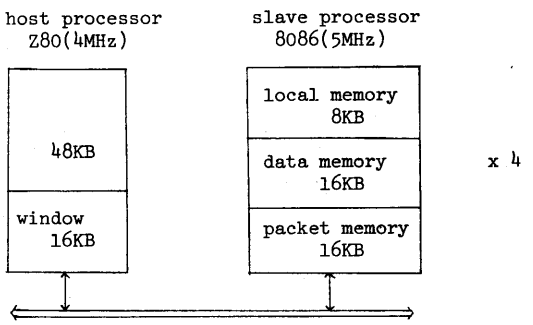


図1. ハードウェア構成

ローカル・メモリはスレーブ・プロセッサ毎に割当てであり、他のプロセッサからの参照は不可能である。データ・メモリもスレーブ・プロセッサ毎に独立した領域であるが、他のプロセッサから参照可能な共有メモリである。パケット・メモリは全プロセッサに共通したメモリであり、一台の書込み動作によりすべてのプロセッサのパケット・メモリに書込まれる（ブロードキャスト転送される）。

3. ソフトウェア構成

3.1 ソフトウェアの全体構成

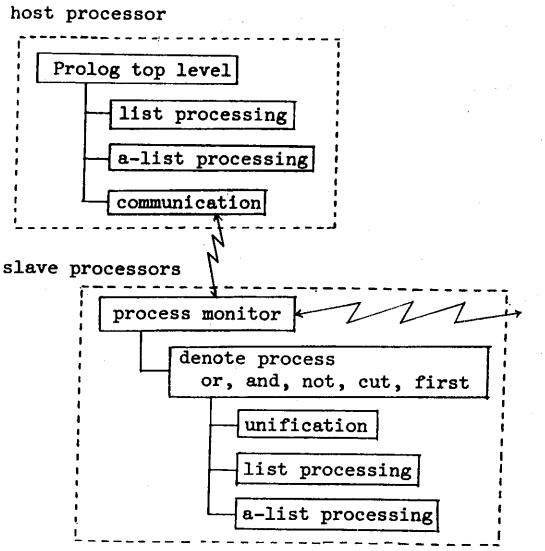


図2. ソフトウェア構成

ソフトウェア構成を図2に示す。ホスト・プロセッサは、Prologのトップレベル部分、すなわち節の入力と解の出力を担当する。解を求める動作は4台のスレーブ・プロセッサが担当する。各スレーブ・プロセッサには複数のプロセスが割当ってあり、その個数は実行時に動的に変化する。それらのプロセスの管理は各スレーブ・プロセッサのプロセス・モニタが行う。プロセスの実行に必要な情報（プロセスの状態、message buffer、変数の値など）は、そのプロセスを担当しているプロセッサのプロセスエリアに格納される。これは他のプロセスから参照可能になっている必要がある。また、新しく生成すべきプロセスはニュープロセスエリアに格納しておく。ニュープロセスエリアはすべてのプロセッサから参照される。

節は内部では、二進木の形で表現している。また、変数への代入は環境（連想リスト）による方法を用いた。

ハードウェア構成で述べた共有メモリの特徴を充分に生かすため、次のようにメモリを割当てた。

- ローカル・メモリ： プログラム、入力節
- データ・メモリ ： プロセスエリア
- パケット・メモリ： ニュープロセスエリア、連想メモリ

すなわち、他のプロセッサから参照する必要のないものをローカルメモリに、少数のプロセッサから参照されるものをデータ・メモリに、全プロセッサから共通に参照されるものをパケット・メモリに割当てている。

3.2 セミ・コルーチンとストリーム並列

Prologの動作は「論理式を真にする代入」を求める動作である。すなわち、ゴール文の右辺が論理式(註1) Fである場合、解σは $\forall F\sigma$ (註2) を満たす。[11]

(註1) ここでは、原子論理式とandとorのみから成る論理式を考えている。この時、論理式F中のすべての原子論理式が帰納的に可算ならば、Fも帰納的に可算である。

(註2) $\forall G$ はG中のすべての変数を全称記号で束縛した論理式である。

たとえば、次のような入力節

```
append([],Z,Z).
append([W|X],Y,[W|Z]) <- append(X,Y,Z).
```

に対して、ゴール節 ← append([a],U,V). を入力した場合に得られる結果は、代入 {V=[a|U]} であり、 $\forall U$; append([a],U,[a|U]) が真であることを意味する。

論理式Fを真にする代入σは、通常複数個存在するため、解σを求めるプロセスはコルーチン等で記述する必要がある。[4],[10],[11] また特にPrologインタプリタの場合、

(1) プロセス間に親子関係が存在する（セミ・コルーチン）。
(2) 子プロセスから親プロセスへ解を返すだけの一方通行である。
(3) 逐次処理あるいはストリーム並列の場合、親プロセスが解を要求するのは最後に生成したプロセス（末子プロセス）に対してだけである。

という特徴があり、一般のコルーチンよりも簡単に実現できる。

プロセス・モニタに必要な機能は次のようなものである。

- new プロセス名(引数$_1$,…,引数$_m$)： 新しいプロセスを末子プロセスとして生成する。生成するプロセスをscheduled状態としてニュープロセスリストに付加え、waitする。いずれかのプロセッサが受付けたならば、再びready状態になる。実際には、プロセスはdenoteの一種類だけである。
- receive： 末子プロセスから解を1つ受取る。末子プロセスのmessage bufferを調べ、空でなければ解を1つ取出す。空ならwaitする。
- send(解)： 親プロセスに解を送る。message bufferに解を付加える。message bufferがいっぱいになればwaitする。
- terminate： 末子プロセスを終了させる。末子プロセスが既にdoneを実行していれば、末子プロセスをterminated状態にする。まだ実行していなければwaitする。
- done： プロセスの実行を終了する。親プロセスが既にterminateを実行していればterminated状態になる。まだ実行していなければwaitする。
- abort： 末子プロセスおよびその子系のプロセスを強制的に終了させる。末子プロセスをaborted状態にする。

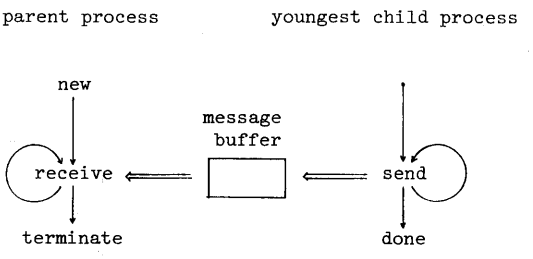


図3. プロセス間交信

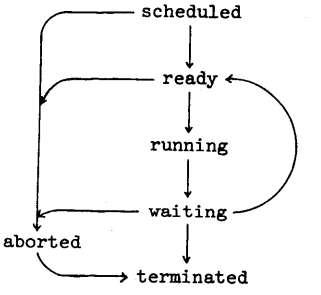


図4. プロセス状態遷移図

ストリーム並列は、生成されたすべてのプロセスを並列に実行することで達成できる。例えば、図5の場合プロセス1により生成されたプロセス2は、解を次々に求めていき、message buffer にそれらを貯えていく。逆にプロセス1は message buffer から解を次々に取り出す。プロセス1は最初の解 {X=a} に基いて q(a) のプロセスを生成し、p(X) & q(X) の解を求める。q(a) のプロセスが fail すればプロセス2の次の解 {X=b} に基いて同様の手順を繰り返す。プロセス2が fail すればプロセス1も fail する。

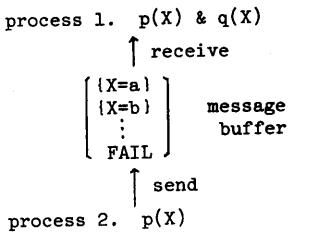


図5.

このように、ストリーム並列は逐次処理の動作と極めて近い並列方式である。すなわち、解の得られる順序は逐次処理の場合と同一であり、cut, not, first ([10],[11]) 等の処理も可能である。プロセスの並列度は message buffer の大きさに依存し、大きさが0の場合を逐次処理と見なすことができる。この時、プロセスの生成される順序および生成されるプロセスの個数は、ストリーム並列と逐次処理で同一になる。

セミ・コルーチンによるPrologインタプリタのアルゴリズムを、SIMULA 67風の擬似言語で記述したものをプログラム1に示す。またプログラム2にPrologのトップレベルのプログラムを示す。ただし、述語の定義やcut処理などの部分は省いてある。andとorの引数についても2引数に固定してある。上で述べたように、ストリーム並列を実現するには生成されたプロセスを並列に動作させるだけでよい。

次に示すPrologのプログラム（リストの昇順ソート）を例にとって、ストリーム並列の動作をもう少し詳しく説明する。（図6）

```
sort(X,Y) <- perm(X,Y) & ord(Y).
perm(X,[Q|Y]) <- append(P,[Q|R],X) & append(P,R,Z) & perm(Z,Y).
ord([]).
ord([X]).
ord([X,Y|Z]) <- X =< Y & ord([Y|Z]).
append([],Z,Z).
append([W|X],Y,[W|Z]) <- append(X,Y,Z).
<- sort([1,3,2],Y).
```

まずゴール節によりプロセス denote(sort([1,3,2],Y)) が生成される（図6(1)）。(1)の実行により(2)のプロセス denote(perm([1,3,2],Y) & ord(Y)) が生成される。プロセス(2)はプロセス(3) denote(perm([1,3,2],Y)) を生成し、解が得られるのを待つ。最初の解 {Y=[1,3,2]} に基いて、プロセス(2)はプロセス(4) denote(ord([1,3,2])) を生成する。この時点でプロセス(3)は、プロセス(4)と並列に動作しており、他の解を探索している。また、プロセス(1)と(2)はそれぞれの末子プロセス(2)と(4)からの解を待っている。プロセス(4)がFAILを返してくると、親プロセス(2)は実行を再開し、プロセス(4)を終了させてからプロセス(3)に次の解を要求する。次の解 {Y=[1,2,3]} を得たのち、それに従ってプロセス(5) denote(ord([1,2,3])) を生成する。(5)が成功し解{}を返してくると、プロセス(2)も成功し解 {Y=[1,2,3]} をプロセス(1)に返す。待ち状態にあったプロセス(1)は実行を再開し、解として {Y=[1,2,3]} を得る。

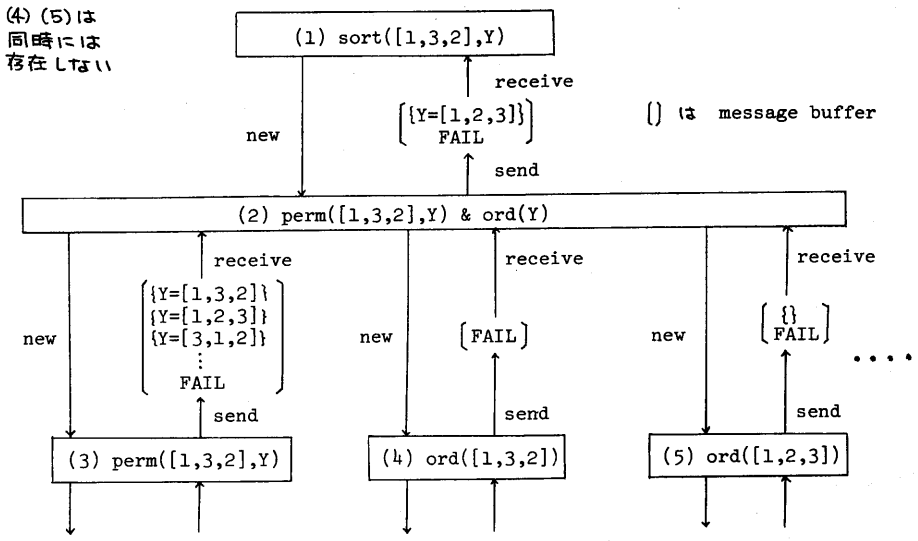


図 6.

この例でもわかるように、ストリーム並列はバックトラックが頻繁に起こる問題に対して有効である。バックトラックが全く起こらない場合は、逐次処理と同程度の速度になる。

3.3 リスト処理

入力した節は二進木に変換される(図7)。なお、入力の際、前置、挿入、後置の記法が使えるようにパーザを用意した。and の記号として，（コンマ）ではなく & を使用していること、および ":-" でなく "<-" を使用していることを除けば DEC 10 の Prolog [5] と同じ文法を採用している。

入力

```
p(X) <- q(X) & r(Y) & s(Z).
```

内部表現

```
(<- (p X) (& (q X) (& (r Y) (s Z))))
```

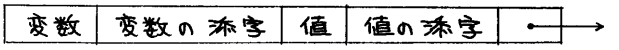
図 7.

変数と定数（リテラルアトム）とを簡単に区別できるように、異なる種類のアトムとして登録している。

次に述べるように、変数への代入状況を連想リストで表現しているので、実行中に変化する節は存在しない。従ってスレーブ・プロセッサに必要なリスト処理関数は car, cdr, atom 等だけであり、cons およびガベージコレクションは必要ない。また、セミ・コルーチンによるインタプリタであるため、導出形を作成する必要もない。

3.4 連想リスト

変数への代入状況は次の様な連想リスト（変数、変数の添字、値、値の添字の四つ組のリスト）で表現する。（[10]で binding environment と呼んでいるもの）



従って、項は<構造、添字、連想リスト>の三つ組で表すことができる。例えば連想リストが（$Z_1 = [b]_0$, $Z_0 = [W|Z]_1$, $W_1 = a_0$）の時、Z_0 は [a,b] を指す。

新しい代入は常に連想リストの先頭に付加わるので、trail [6] の必要がなく、複数のプロセスで共有して使用できる。ただし、代入されている値の探索に時間がかかるのが欠点である。また、連想リストのガベージコレクションを行う必要がある。

4. おわりに

本稿では、ブロードキャストメモリ結合形並列計算機上での、ストリーム並列 Prolog システムの実現方法について述べた。

なお、本システムは現在作成中である。

参考文献

[1] 相田仁： 並列 Prolog システム "Paralog" について，
Prolog Conference、1982.

[2] 梅山伸仁： 論理プログラムの並列実行について，
知識工学と人工知能研究会資料 46、1982.

[3] J.S.Conery, et al: Parallel Interpretation of Logic Programs,
Proc. of the 1981 Conference on Functional Programming Languages
and Computer Architecture.

[4] K.Furukawa, K.Nitta, Y.Matsumoto: Prolog Interpreter based on Concurrent
Programming, Prolog Conference, 1982.

[5] D.Warren, et al: User's guide to DECsystem-10 PROLOG,
Dep. of A.I. Univ. of Edinburgh, 1978.

[6] D.Warren: Implementing PROLOG,
D.A.I. Reserch Report No.39, 1977.

[7] R.Boyer, J.Moore: The Sharing of Structure in Theorm Proving,
Machine Intelligence Vol.7, 1972.

[8] O.-J.Dahl, E.W.Dijkstra, C.A.R.Hoare: Structured Programming,
Academic Press, 1972.

[9] 小畑、金田、前川他： ブロードキャストメモリ結合形並列計算機の試作、
信学技報、EC81-37、1981.

[10] 田村、金田、前川： PROLOG の制御構造についての一提案、
情報処理学会第24回全国大会、1L-6、1982.

[11] 田村： 述語論理型プログラミング言語の研究、
神戸大学修士論文、1982.

プログラム 1.

```
class denote(t);
  term t;
begin


  procedure or(t1,t2);
    term t1,t2;
  begin
    substitution s;

    new denote(t1);
    loop
      s:=receive
      exit when s=FAIL;
      send(s)
    end loop;
    terminate;
    new denote(t2);
    loop
      s:=receive
      exit when s=FAIL;
      send(s)
    end loop;
    terminate
  end;


  procedure and(t1,t2);
    term t1,t2;
  begin
    substitution s1,s2;

    new denote(t1);
    loop
      s1:=receive
      exit when s1=FAIL;
      new denote(substitute(t2,s1));
      loop
        s2:=receive
        exit when s2=FAIL;
        send(composite(s1,s2))
      end loop;
      terminate
    end loop;
    terminate
  end;


  procedure not(t1);
    term t1;
  begin
    substitution s;

    new denote(t1);
    s:=receive;
    if s=FAIL then send({});
    abort
  end;


  procedure first(t);
    term t;
  begin
    substitution s;

    new denote(t);
    s:=receive;
    if s≠FAIL then send(s);
    abort
  end;


  begin
    term c;
    substitution mgu,s;

    if null(t)
          then send({})
    else if functor(t)=AND
          then and(arg1(t),arg2(t))
    else if functor(t)=OR
          then  or(arg1(t),arg2(t))
    else if functor(t)=NOT
          then not(arg1(t))
    else if functor(t)=FIRST
          then first(arg1(t))
    else begin
      for every input clause c do
        if unify(t,head(c)) then begin
          mgu:=unifer(t,head(c));
          new denote(substitute(body(c),mgu));
          loop
            s:=receive
            exit when s=FAIL;
            send(composite(mgu,s));
          end loop;
          terminate
        end
    end;
    send(FAIL);
    done
  end
end;
```

プログラム 2.

```
loop
  clause:=read;
  new denote(clause);
  loop
    s:=receive
    exit when s=FAIL;
    print(s)
  end loop;
  terminate
end loop
```