輪講資料
並列処理言語

小川 敏一

平成 8 年 4 月 25 日

はじめに

近年、ハードウェアの技術の進歩によって計算機の速度はますます速くなっている。一方、処理したい問題も大規模なものになってきている。そのため、並列計算機や並列言語の研究開発が盛んになっている。従来この分野の研究は、数値計算用に開発された逐次言語である FORTRAN を中心に行なわれてきた。そこでは、DO 文をコンパイラでベクトル命令に変換し、ベクトル計算機で実行するという方式が一般的である。しかし、最近では数値計算ばかりでなく、人工知能の分野で大規模ソフトウェアを高速に処理することが要求されてきている。この種のソフトウェアでは、記号処理的な計算が多く現れ、あらかじめプログラムのどの部分の計算にどのくらいの手間がかかるかが、予想しにくいことが多い。従って、これらを記述する言語としては、いろいろなアルゴリズムを簡単に試すことができ、その修正、変更が容易である必要がある。また、高速実行のためには、並列化し易いことも重要である。この様な要求に応えるためには、(数値計算用の)逐次言語を用い、それで書かれたプログラムの一部を並列実行するという今までのアプローチでは不可能である。

1　並列処理言語

並列処理言語に必要な機能は、潜在的な並列性を表現できること、その結果として生じる同期と通信の問題を解決する手段、そして非決定性である。以下、それぞれについて簡単にではあるが説明する。

1.1　並列性

並列性を明示するには、大きく分けて二つの方法がある。一つは各プロセスを一つのモジュールとして宣言し、並列分岐によって記述する方法。もう一つは、一つのモジュール内で並列実行可能な部分を指定する方法である。また、この二つの分類の中でもプロセスの起動方法によって分類することができる。これは、プロセスを動的に起動できる、つまり起動されるプロセスの数が入力データによって変化し得る場合と、静的にしか起動できない、つまり起動されるプロセスの数が固定されている場合とである。

一プロセスを一モジュールとして宣言する方法には、cobegin/coend 構文と、fork/join 構文がある (2ページ図 1)。しかし、これらの構文を用いてプロセスを記述すると、問題が生じることがある。それは、並列に動作するべきプロセスが、オーバーラップするデータを持ち、しかもこのデータが変化する時に混乱が生じるというものである。これは、プロセスの相対的な速度に依存している(レース条件がある)ために期待している結果が出てくるとは限らない。

これに対し、各プロセスを一つのモジュールとして記述する方法は、プログラムの階層的分割が可能で、変数の管理も容易である。しかし、並列動作部をすべてモジュール化する必要があるため、細かい並列動作の記述には不向きである。

プロセスを一つのモジュールとして記述する場合、その起動方法にも静的な場合と動的な場合とがある。プロセスの定義を含むモジュールが実行を開始した時点で、自動的にそのプロセスが起動されるのが静的な起動であり、起動されるべきプロセスの定義を参照できるモジュール内で起動命

令を陽に実行するのが動的な起動である。

```
cobegin                    .....
   Stmt1                   FORK
   Stmt2                   .....
   Stmt3                   JOIN
   ....                    .....
   StmtN            label: Stmt
coend                      .....
                           QUIT
```

図 1: cobegin/coend 及び fork/join の記述

1.2 同期

並列に動作するプロセスが協力して処理を進めるためには、プロセス間で必要な情報を交換する必要があるとともに、プロセスの実行を調整して正しい結果となる実行順序を保証しなければならない。

制御情報の交信、すなわち同期は、正しい結果になるように実行の順序を保証する。同期の使われ方には、条件同期と相互排除がある。条件同期とは、あるプロセスが他のプロセスによってある条件を満たされるまで待つことである。一方相互排除とは、あるプロセスが共有データに対して操作している最中に、他のプロセスがその共有データに対して同時に操作しないように制御することである。共有データに対する更新操作のように不可分に実行されなければならないプログラム部分をきわどい区間というが、相互排除は、このきわどい区間が不可分に実行されることを保証する。変数を共有するタイプのプロセス間の通信では、同時にアクセスされる共有変数が存在するため、条件同期と相互排除をともに解決するための機能が必要になる。一方メッセージ転送タイプのプロセス間通信では共有する変数が存在せず、直接プロセス間でデータを移動させるので相互排除の問題は生じない。

1.3 非決定性

非決定性とは実行可能な命令系列が複数存在することである。これには二つの解釈が可能であり、一つは、個々の命令系列が出す結果の中で与えられた条件を満たすものを全体の答とする場合であり、もう一つは、個々の命令系列から生じる結果の中でどれを全体の答としても良い場合である。

非決定性の考えを進めたものにガード付コマンド（2ページ図2）がある。次に実行すべき文の選択をするために各文の前に論理式をおき、その論理式が真の時だけ後ろの命令が実行される。一般の条件文と違うのは、それぞれの条件が互いに排他的である必要がないという点である。

```
＜ガード１＞->＜実行文＞
＜ガード２＞->＜実行文＞
＜ガード３＞->＜実行文＞
 ...................
＜ガードN＞->＜実行文＞
```

図 2: ガード付コマンドの形式

非決定的記述を一つのプロセス内だけでなく他のプロセスとの関係に用いることができると、相手の応対に応じて自分の態度を変えるような記述ができるようになり情報交換による deadlock の危険は減少する。

2 プロセス間通信

個々のプロセス間で行なわれる情報交換をプロセス間通信という。これは一般に共有変数とメッセージ転送という二種類の通信モデルに分類される。ここでは、それぞれの通信モデルに基づく通信機構について述べる。

2.1 共有変数による通信機構

共有変数を介したプロセス間通信は、送信プロセスがデータを特定の共有変数に書き込み、受信プロセスがその共有変数を読み出すという具合に簡単に行なうことができる。ただし、これらの動作は非同期に起きるため、送信プロセスの書き込み終了前に受信プロセスが読み出したりする可能

性がある。そこで、同期操作を明示的に行なわなければならないが、そのためには、セマフォ、モニタなどの同期プリミティブを用いれば良い。以下、セマフォ及びモニタについて簡単に触れる。

2.1.1　セマフォ

セマフォs は、負でない値をとることができる整数変数である。この s に対して次のように定義される 2 つの操作のみが許されている。

`P(s)`:　`s>0` ならば `s:=s-1` を実行し、そうでなければ、`P(s)` を呼び出したプロセスの実行を一時停止する。

`V(s)`:　待ち状態にあるプロセスが存在すれば、そのプロセスを実行状態に移し、存在しなければ、`s:=s+1`。

このとき、セマフォの値は使用されていない資源の数を表す。つまり初期値としては、資源の数を代入しなければならないことになる。また、P、V 操作は、それぞれ不可分に実行されなければならない。セマフォを用いると、同期と相互排除の両方を簡単に記述できるが、deadlock に陥り易いという欠点がある。

2.1.2　モニタ

モニタは、相互排除と同時にデータの抽象化の概念を含み、共有資源の初期化操作をまとめて記述したものである。このため、共有資源を使用する際にはその内部の詳細を知る必要がなくなる。より正確にいうと、モニタとはで 3ページ図 3に示すような monitor 型の定義であり、この monitor 型の変数が実際の共有資源となる。

モニタはしばしば、相互排除されるべき共有資源を取り囲んだ塀と表現される。共有資源は塀の外から直接アクセスすることはできず、モニタの手続きを介してのみアクセス可能である。塀には入口がついており、共有資源にアクセスしたいプロセスは単に手続きに対応するエントリを呼び出すだけでよい。また、共有資源への相互排除アクセスはモニタが保証する。つまり、モニタの手続きのうちどれか一つでも他のプロセスに呼び出されて実行されている間は、モニタ内に入ることはできない。しかし、条件同期の機能は含まれていない。

```
モニタ名:monitor;
   begin
     <変数宣言>
     procedure<手続き名>(<仮引数>)
     begin
       <手続き本体>
     end;
     <他の手続きの定義>
           ......
     <他の手続きの定義>
     <初期化操作の定義>
     end;
```

図 3: モニタの定義

2.2　メッセージ転送による通信機構

共有変数が存在しない時、プロセス間で情報交換をするための方法として、メッセージ転送がある。メッセージ転送では、データも制御情報もすべて同一経路を通して同じ方法で渡される。これを実現する代表的な通信プリミティブが send/receive で、一般に 3ページ図 4の形式をしている。

```
send <式の並び>to <受信側指定子>
receive <変数の並び>from<送信側指定子>
```

図 4: send/receive の形式

＜送信側指定子＞と＜受信側指定子＞の組（通信チャネル）は、お互いに相手を指定しない限り確立されず、正しくメッセージを受け渡すことができない。このため、通信プリミティブの設計では通信チャネルの指定方法が重要となる。この通信チャネルの指定方法を分類するとき、二つの分類法が考えられる。一つは通信チャネルとして何を指定するかという分類であり、もう一つは、通信チャネルを指定する時点による分類である。以下それぞれについて説明する。

1. 通信チャネルとして何を指定するか。
    (a) 相手プロセスを直接指定する方法。一対一通信に適している。実現が容易であり、プロセスは他のプロセスからのメッセージを位置受信するか制御することができる。しかし、一対多、多対多の通信には適さない。
    (b) ポートを用いる方法。
    一対多や多対多の通信に対応するため、送信側指定子及び受信側指定子として大域的な通信チャネル名をつける方法である。この方法は、通信相手を間接的に指名できるので、柔軟性がある。しかし、間接指定なので通信相手を特定できず誤りの原因となるおそれがある。
2. 通信チャネルをどの時点で指定するか。
    (a) 静的チャネル指定。コンパイル時に通信チャネルが確定し、実行効率がよいが、実行時に通信チャネルの選択ができず、柔軟性に欠ける。
    (b) 動的チャネル指定。実行時に通信チャネルを決定することができ、しかも実行の状況に応じて通信チャネル選択が行なえる。しかし、効率的な実現が難しい。

3　並列論理型言語 KL1

これまでに述べてきた、並列処理言語の機能を考えながら並列論理型言語 KL1 について述べる。

3.1　シンタックス

KL1 プログラムは次のような次のようなシンタックスを持つ。

$$H_1 : -G_{11}, ...G_{1m_1} | B_{11}, ..., B_{1n_1}.$$

$$............$$

$$H_l : -G_{l1}, ...G_{lm_l} | B_{l1}, ..., B_{ln_l}.$$

ここで、各行はクローズ（節）、H、G、B は、各々クローズヘッド、ガードゴール、ボディゴールと呼ばれる。オペレータ”|’は、コミットメントバーと呼ばれ、クローズ中で、これより前にある部分をガード部、続く部分をボディ部と呼ぶ。ここでガードゴールは組み込み述語しか書けないという制限がある。また、ボディゴールには組み込み述語とユーザ定義述語の二つの種類が書ける。この形式は、1.3 で述べた、ガード付命令を拡張したものである。

3.2　プログラムの実行規則

次に、KL1 のプログラムがどのようにして実行されるかについて説明する。今、あるゴール H が与えられたとする。この時、ゴール H の実行は以下の規則に則って行なわれる。

1. コミットメントバーの左に書いてあること、せいかくには、
    - 候補に上がっているクローズのヘッダと、そのクローズを呼び出したゴール H の同一化 (Unification)
    - ガードゴールの実行

    は、複数の候補クローズに対して並列に実行してよい。しかし、それらは H を具体化してはならない。具体化しようとしたら、その具体化の原因となる同一化が中断させられる。
2. コミットメントバー の左に書いてあること、つまりクローズのボディゴールは、そのクローズを呼び出したゴール H を具体化してよいが、それができるのは、H が複数の候補クローズの中からただ一つ選ばれたクローズに限られる。

3.3　KL1 によるプログラミング

KL1 のプログラムは、プロセスと呼ばれる実行の基本要素を、ストリームと呼ばれるプロセス間通信のための経路を用いて組み合わせることによって構成されることが多い。

プロセスの動作は、KL1 では一つの述語として表される。プロセスを表す述語は、内部状態の保持のために他のプロセスから参照されない局所変数を用い、プロセスとして存在するために、再帰呼び出しを用いる。ストリームは、リストを使って実現される。

ストリームとして論理変数そのものではなくてリストを用いるのは次の理由による。つまり、KL1 のような論理型言語では、一度ある値に具体化された論理変数は、別の値に再び具体化することができない (単一代入規則)。したがって一つの論理変数を用いた通信は一回しかできない。このような不都合を解消するために、

[< メッセージ > | < 新しい未具体化変数 >]

のような Car 部に伝えたいメッセージ、Cdr 部に新しい論理変数を持つようなリスト構造で具体化する。

また、KL1 は、ゴールごとに実行するプロセッサを指定することができる。

```
:- module main.

(1)     main:- prod(3,L),
               cons(L,K)@node(1),
               io:outstream([print(K),nl]).
(2a)    prod(T,L):- T > 0 |
               L = [T|NewL],
               NewT := T - 1,
               prod(NewT,NewL).
(2b)    prod(T,L):- T = 0 |
               L = [].
(3a)    cons(J,K):- J = [N|NewJ] |
               cons(NewJ,NewK),
               append(NewK,[N],K).
(3b)    cons(J,K):- J = [] |
               K = [].
(4a)    append(X,Y,Z):- X = [] |
               Y = Z.
(4b)    append(X,Y,Z):- X = [W|NewX] |
               Z =[W|NewZ],
               append(NewX, Y, NewZ).
```

図 5: KL1 のプログラム例

ここで、図 5のプログラム例を用いて実行の様子を説明する。図のプログラムは、整数の降順に生成された線形リストを昇順に並べ替えて表示するプログラムである。このプログラムは、整数の線形リストを生成するプロセス (prod/2)、そのリストを昇順に並べ替えるプロセス (cons/2)、並べ替えの際に利用するプロセス (append/3) から構成されている。今トップレベルのゴールとして、prod(3,L)、cons(L,K)、io:outstream([print(k),nl]) が与えられると、これは 3 までの線形リストを昇順に表示することを意味し、実行後 [1,2,3] というリストが表示される。この計算過程を以下に説明する。

まず、このトップレベルのゴールが与えられると、クローズ (2a) のガード部の実行に成功し、ボディ部が実行される。この時新たに prod(NewT,NewL) というゴールが生成され同様にクローズ (2a) が実行されて行く。そのうち、prod/2 の第一引数の具体値が 0 になり第二引数がリスト [3,2,1] となり、クローズ (2b) が実行されこのプロセスは終了する。また、クローズ (2a) のボディ部を実行によってリストの Car 部が具体化される度に、クローズ (3a) のガード部の実行に成功し、ボディ部が上で述べたプロセスに並列に実行される。以下、ガード部の実行に成功したクローズはそのたびにボディ部を実行させて行き、そうでないものは中断して待ち状態にある。

参考文献

[1] M. ベン-アリ 著. 渡辺 栄一 訳. 並列プログラミングの原理 -プロセス間通信と同期への概念的アプローチ-. 啓学出版,1986.

[2] リチャード.C. ホルト 著. 大野 豊 監訳. 伊藤 潔, 広田 豊彦 訳. 並行処理と UNIX. 啓学出版,1985.

[3] 富田眞治, 末吉敏則. 並列処理マシン. オーム社,1989.

[4] 木村 康則. 並列論理言語 KL1 の処理系に関する研究.