Vol. 28 No. 4　　情　報　処　理　　Apr. 1987

解　説

4. システム技術

4.4 分散処理言語†

浜田　喬††*

1. はじめに

分散処理システムは多くの利点をもつが，その反面，ソフトウェアの規模が比較的大きく複雑となり，また論理的あるいは地理的に処理が分散しているためその正当性を保証するのが困難であるという問題点をもつ．すなわち，逐次処理プログラムが一連の単一処理による処理アルゴリズムであるのに対し，分散処理プログラムは，複数のプロセッサ上で同時に行われる処理を記述しなければならない．そのため，時間依存性の誤り，プロセス間の通信と同期，デッドロックの防止，共有資源の管理など，逐次処理プログラムにはみられない問題に直面することになる．このような問題は，オペレーティングシステムの問題と共通する部分が多く，いわゆる並列問題として早くから検討されており，分散処理プログラムの諸問題は並列処理プログラムの延長上にあるものと考えることができる．ここでは，信頼性，記述性が高く，分散処理システムを統一的に記述できるプログラミング言語を実現するための方法論と，具体的な分散処理言語とを紹介する．

2. 分散処理のためのプログラム方法論

並列動作を行うプログラム，すなわちプロセスの間の通信方式は，プログラミングの形式上次の二つに分類できる．

（1）　共有変数による通信

（2）　メッセージによる通信

ここではまずはじめにオペレーティングシステムに関係の深い共有変数形通信の特徴について概観する．

Dijkstra は，共有変数を操作する領域に対しては，同時にはただひとつのプロセスのみにしかアクセスを認めないように管理するための道具として semaphore を提案した．semaphore S に対して二つの操作 P, V が定義され，次のような意味をもつ．

P(S)：もしSがビジーならば，プロセスはSにて用意された待ち行列に入り，待ち状態となる．Sが空きならば，Sをビジーにしてプロセスの実行を続ける．

V(S)：Sを空き状態にする．このときP(S)操作の実行によって待ち行列に入っていたプロセスがあれば，その先頭を取り出して実行可能とする．V(S)を実行したプロセスも実行を継続できる．

セマフォは，基本的で強力な道具であるが，プログラムが書きにくく理解しにくいことや，プログラムミスによってPまたはV操作が行われないと，デッドロックを起こしてしまうというような欠点をもつ．

これに対して critical region は，セマフォのP，V操作を括弧と同様の対にしたもので，変数を，プロセス固有の変数と，複数のプロセスに共通の共有変数とに分け，共有変数に対しては critical region の中からしかアクセスできないようにするとともに，critical region 内では同時には一つのプロセスのみしか実行できないという特徴をもったものである．このような規則は，コンパイル時にチェックできるため，プログラムのミスを防止することができる．

critical region を具体的な言語の様式として提案したのが Hoare と Brinch Hansen による monitor であり，これは共有変数とそれに対する動作を，一つのモジュールにまとめたものである．各プロセスは monitor の手続きをとおしてのみ共通変数にアクセスすることができる．この手法は，ある型の変数とそれに対する操作とをまとめて抽象データ型であると考える抽象的プログラミングの手法とも一致し，そのプログラミングの技法に大きな影響を与えた．monitor と process の概念に基づいて設計された最初の言語として Brinch Hansen の Concurrent Pascal と Wirth の Modula とがあることはよく知られるとこ

† Languages for Distributed Processing by Takashi HAMADA (Institute of Industrial Science, University of Tokyo).

†† 東京大学生産技術研究所
* 現在　学術情報センター

ろである．

さらに，プロセスが critical region を実行する場合の論理式の評価の際に非決定性を導入し，共有変数がある条件を満足するまで待つことができるようにした conditional critical region, guarded command (GC), guarded region (GR) などが提案されている[1]．GC は

if B1：S1|B2：S2|…**end**

do B1：S1|B2：S2|…**end**

の構文をとるもので，if 文の場合，条件 B1, B2, … のいくつかが真である場合，そのひとつ Bi が非決定的に選択されて，対応する Si が実行される．すべての条件が偽であるとプログラムは停止する．do 文の場合，真である条件がある間対応するSを次々に実行し，なくなれば do 文を終了して次の文へ進む．GR は

when B1：S1|B2：S2| …**end**

cycle B1：S1|B2：S2| …**end**

の構文をとるもので，when 文の場合，条件の一つ Bi が真になるのを待って対応する Si を実行する．cycle 文は when 文の無限繰り返しである．

次に，メッセージによる通信及び同期の方法について述べる．メッセージによる基本的な方法は，send と receive のコマンドを用いて，プロセス間の通信を行い，かつこれによってプロセス間の同期をとるもので，次のようになる．

```
process A
  send X to B
end
process B
  receive X from A
end
```

この手法は，semaphore と同様に基本的で強力な道具であるが，コンパイル時における安全性のチェックが困難で，危険が分散してしまう恐れがある．また，通信の際にお互いの名前を知らなければならないのは，プログラムの汎用性に欠ける．

通常のシステムでは，送り側がメッセージを送出した後も，送り側がそのまま実行を続ける非同期処理が多いので，メッセージを蓄える領域が問題となる．受信側がメッセージを取り込むまでの間，メッセージを蓄えるために mail box をプログラムで定義することができる．上述の send, receive による送受信は，次のように書き換えることができる．

```
var M : mailbox
process A
  send X to M
end
process B
  receive X from M
end
```

また pipe で mail box を定義しているオペレーティングシステムもよく知られている．

以上に述べた，共有変数型の考え方とメッセージ通信形の記述方式とは，本来機能的には同様のものであり基本的な差異があるものではない．しかし，分散処理を実際に実現する場合には，共有変数をもたずにメッセージによるプロセス間通信を行うことが多い．さらに，あるプロセスがとりうる動作は，これと通信を行う他のプロセスの動作によって左右され，プログラムの記述上，決定的な動作を指定できなくなる．このような考えから，Hoare はメッセージ通信の手法に，前述の guarded command による非決定性を導入した Communicating Sequential Processes (CSP) を提案し[2]，またほぼ同時に，Brinch Hansen も同様の言語概念である Distributed Processes (DP) を発表した[3]．

CSP では，入出力がプログラミングの基本的要素となり，通信をし合うプロセスを並列に配することによりプログラムの構造を与えることができるという考えに立っている．いま，P1, P2, …をプロセスとするとき，これらの並列処理を，並列コマンド [P1‖P2‖…] で記述する．各プロセス内では，＊[B1→S1‖B2→S2 ‖…] によって前述の cycle 文を表す．また入出力コマンドはそれぞれ

＜相手のプロセス名＞？＜入力値を入れる変数＞

＜相手のプロセス名＞！＜出力値の式＞

の形式をとり，二つのプロセスが入力および出力コマンドを相互に出し合ったときに通信が成立する．このうち，入力文は，条件 Bi として記述でき，通信が成立したときに true となる．CSP によるプログラムの一例として，プロセスXから文字列を入力し，1行 120 文字ずつプリントするものを次に示す[2]．

```
lineimage : (1..120) character ;
i : integer : i :=1 ;
＊[c : character ; X?c→lineimage (i) :=c ;
  [i<120→i :=i+1
  ‖i=120→lineprinter! lineimage ; i :=1
```

```
] ]
[i=1→skip
‖i>1→*[i<120→lineimage (i):=space;
i:=i+1];
lineprinter! lineimage
]
```

一方，DP はプロセスとモニタとを合わせたような記述方式をとるもので，あるプロセスが他のプロセスの内部の procedure (**proc**) を呼ぶことによって通信が行われる．非決定性は，前述の when 文と同様の構文によって記述される．一例として，前述の semaphore を DP で記述すると次のようになる[3]．

```
process semaphore; s: int
proc P when s>0:␣ s:=s-1 end
proc V; s:=s+1
s:=0
```

CSP と DP は，Ada を始めとするその後の多くの言語に影響を与えている．

3. 分散処理言語の構成手法

以上のような種々の手法の考え方にそって，具体的な言語も数多く提案されてきた．これらの大まかな系譜を示すと図-1 のようになろう．同図でブロック内に示したのが提案ないしは実用化された言語であり，他は方法論や言語概念である．プログラムの記述の形式に応じて，critical region, monitor に端を発する系と，メッセージ通信系の言語とに分けているが，大きな意味はなく，特に近年では，この二つの機能を併せもった言語も多い．以下では，従来より開発されてきた主要な言語を例にとり，分散処理用言語の構成手法について概観する．

3.1 Star MOD[6]

分散処理言語の初期の代表的なものに Star MOD がある．これは，Robert P. Cook が，N. Wirth による並列処理用言語 Modula を分散処理用に改造したもので，Modula がそうであるように，モニタを用いモジュールの概念をベースとしている．次のプログラムは4つのモジュールが一つのセンタモジュールを

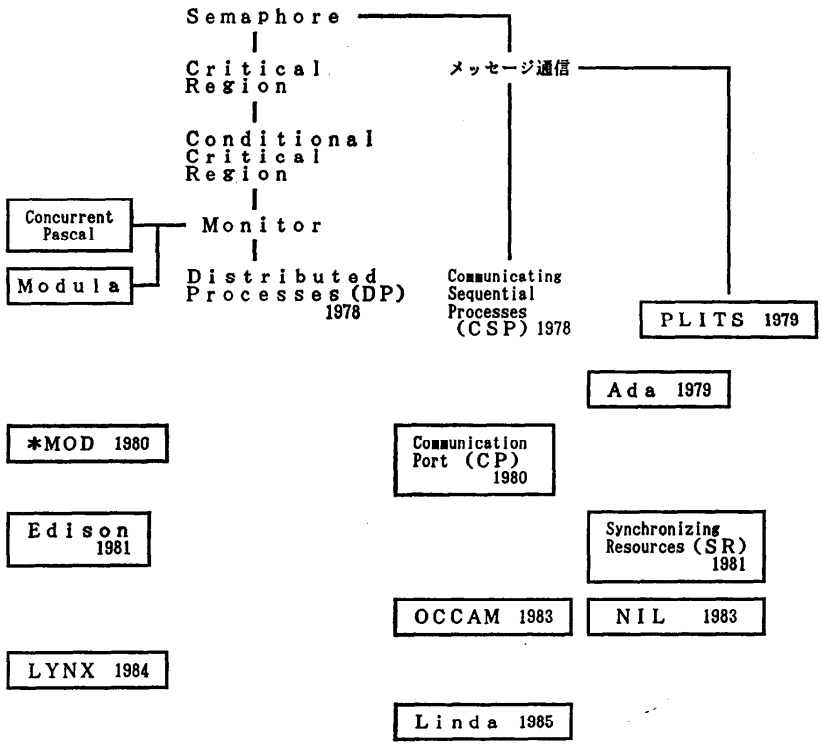

図-1 分散処理言語の系譜

介して通信するシステムの記述例である.

```
network module star=(center, pr), (pr, center)
  const NOPROCESSORS=4;
  processor module center;
    define communicator;
    process communicator (……)
            :
      pr[i]. communicator (……)
            :
    end communicator;
  begin (*initialization*)
  end center;
  processor module pr[NOPROCESSORS];
    define communicator;
    import center;
    process communicator (……);
            :
    center. communicator (……);
            :
      end communicator;
    begin (*initialization*)
    end pr
end star.
```

network module はプロセッサ間の通信チャネルを定義し，また，各 processor module に共通の型，定数，手続きなどを宣言する．processor module は複数の並行動作プロセスからなり，また，他のモジュールとのインタフェースを宣言する．

3.2 Communication Port[7]

T. W. Mao と R. T. Yeh は，Brinch Hansen の DP を拡張して，互いに通信し合えるプロセスを明確に定義し，さらに通信を終了させるタイミングを指定できる言語として，CP (Comunication Port) を発表した．

CP では，マスタプロセスと複数のサーバントプロセスが，CP を介して通信を行うという形態をとる．次のプログラムは，三つの sample プロセスから非同期に送られるデータをもとに，各プロセスを制御する，リアルタイムシステムである．

```
process sample [i]; (*i=1, 2, 3*)
  var tolerable: boolean;
      data: real;
  begin
    cycle
      read-sample (data);
      connect control. valuate (data: tolerable);
      if not tolerable then do-proper-op;
    end cycle
  end
process control;
  var mean, saferange: real;
  begin
    set-mean-and-saferange;
    cycle
      port evaluate (d: real; #result: boolean);
      servant sample [1], sample [2], sample [3];
      begin
        result:=abs(d-mean)<=saferange;
        !!;
        recompute-other-statistics;
      end
    endcycle
end
```

control プロセスは，port 文において，servant に示したプロセスのうち最初に connect 文を実行したプロセスと通信を行い，下から5行目の!!文の実行によって同期をとく．その際，アウトプット・パラメータが sample プロセスへ返される．CP はポート内の!!文の位置によって，抽象データ型やモニタを実現することも，メッセージ通信型の同期を実現することもできる．

3.3 Edison[8)~10)]

言語 Edison は，Brinch Hansen が DP を実現すべく 1979 年より開発したマルチマイクロプロセッサ用言語であり，①簡潔さ，②概念の分離，の二つの方針のもとに設計されている．①は Concurrent Pascal にみられる種々の制約を取り払ったこと，ステートメントの数を大幅に減らしたことなどがあげられ，②はモジュール化，並行動作，同期の概念を分離したことがあげられる．Edison によるバッファの例を次に示す．

```
module "buffer"
  var slot: char; full: bool;
  *proc put (c: char)
  begin
    when not full do
```

```
      slot:=c; full:=true
    end
  end
  *proc get (var c: char)
  begin
    when full do
      c:=slot; full:=false
    end
  end
begin full:=false end
```

Edison では，モジュールが抽象データ型になっており，モジュール内のプロシジャのうち*をつけたものがモジュール外から参照できる．

Edison における同期は，conditional critical region の形式 with r when B do S end を簡略化して，when B do S という構文をとる．マルチプロセッサによる並行プロセスは Edison では，

```
cobegin 1 do SL1
also 2 do SL"
   ..
also n do SLn end
```

という構文をとる．1…n は，プロセッサの番号である．並行プロセスは動的に生成，消去される．

3.4 SR (Syncronizing Resources)[11]

SR も，並列に動作する多数のプロセスを処理するためのプログラミング言語で，共有メモリ型システム用言語とネットワーク型システム用言語とのギャップを埋めることを設計目標にしているといわれる．プロセス内の入力文によって，オペレーションが定義されるが，入出力文は次の構文をもつ．

```
input: in operation_command_list ni
```

ただし，operation command は次の構成である．

```
op_id (fomal_name)
    synchronization scheduling → command
    body
```

synchronization は，キーワード and で始まる論理式で，scheduling はキーワード by で始まる算術式である．op_id から synchronization までが guard となる．synchronization の論理式が true であるとき guard が真になり，入力文中の operation command の guard のどれかが真であるとき，そのどれかが非決定的に選択され command body が実行される．選択されたオペレーションの呼び出しが複数であった場合は，scheduling の算術式を最小にするものが選ばれる．

SR には通常の変数の型のほかに，オペレーションをフィールドとしてもつレコードとしてケーパビリティがある．たとえば，主プロセスがケーパビリティ型の変数を引数として渡すことによって，従プロセスは呼び出した側の名前を知ることができ，処理が終了したのちに結果をどのオペレーションに返せばよいかが分かる．SR の入力文は手続き呼び出し型，メッセージ通信型の両方をサポートする．オペレーションは手続きとして call で呼び出すこともできるし，メッセージ通信を送るために send で呼び出すこともできる（後者では，呼び出し元のプロセスは次の処理を続行できる）．入力文によりオペレーションの排他的アクセスが保障され，同時に，同期，スケジューリングが行われる．

3.5 NIL[12)~14)]

NIL は，IBM Yorktown で開発された言語であり，システムが logical view と physical view に分けられ，logical view には NIL の仮想マシンが対応する．logical view における一つのプロセスは，コンパイラによって，physical view における複数のプロセッサに割り当てられる．プログラマにとっては，logical view しか見えないので，プロセスの概念だけでプログラムが可能である．

データは一つのプロセスにだけ所有されるが，その所有権を引き渡すことによってデータの受け渡しが行われる．プロセスは，そのデータの処理を終了するか，他のプロセスに所有権を渡すかのどちらかしか選択できない．このように，logical view ではデータの共有がないので，言語レベルでは，従来の逐次処理の考え方でよいといわれている．

通信は，メッセージの返ってくるのを待つか否かによって非同期式 (message passing) と同期式 (calling) とがある．次のプログラムは非同期式通信の例である．

```
A: process uses (C)
declare
  AMSG: MTYPE
  APORT: PTYPE sendport
  S: EBCDICSTRING
begin
    ...
  allocate AMSG;
    AMSG. DATA='Hello';
  send AMSG to APORT;
```

```
    ...
  end A;
  B: process uses (C)
  declare
    BMSG: MTYPE
    BPORT: PTYPE receiveport
  begin
    ...
    receive BMSG from BPORT:
    if BMSG. DATA='Hello' then ...
    ...
  end B;
  C: definitions
    ...
  MTYPE is message
      (DATA: EBCDICSTRING)
  PTYPE is send
      interface of MTYPE
  end C;
```

基本は，send (queue)→receive (dequeue) 型である．メッセージが BPORT の行列に並ぶと，所有権が移されもはや AMSG は読み書きできない．

同期式通信では，送り側(A)は，call 演算子によって，call-message を受け側(B)へ送る．call-message 中にあったAの実引数は，Bの仮引数に代入され，return 演算子によって，call-message は再びAに戻り，Aの所有権が回復する．どちらの方式にしろ，常にデータの所有権が一つのプロセスに対応しているので，同時にデータを共有することがない．

通信路は実行時に動的に生成される．まず，input port が，自分へのアクセス権を，ケーパビリティとして「発行」する．ケーパビリティは，他のプロセスへ渡され，そのプロセスは connect 演算子により output port を，先の input port と結びつける．

3.6 Linda[16]

Linda は，今までの言語とは異なる生成型通信用の言語である．メッセージの送受信をタプルの追加，回収によって行っている．これにより，空間的にだけでなく，時間的にも分散化を実現しており，ネットワーク環境に整合のよいシステムを実現できる．

プロセス間通信のために，タプルを追加，回収するための仮想的なグローバル領域のことをタプルスペース TS という．プロセスAとプロセスBが通信する際，Aはまずタプルを生成し，TS に加え，次にBがそれを回収するという方法をとる．どんなに多くのプロセスがあろうとも，TS は一つであるということが大きな特長である．

通信用関数として，次の3種類がある．

(1) out (N, P2, ···, Pj)

(2) in (N, P2, ···, Pj)

(3) read (N, P2, ···, Pj)

ここで，引数は次のとおりである．

N: TYPE NAME を示す実引数

P2, ···, Pj: パラメータリスト (実引数 or 仮引数)

(1)の out (N, P2, ···, Pj) により，タプル N, P2, ···, Pj が TS に追加される．out 文はタプルの送信に相当する．ここでは話を簡単にするために，P2, ···, Pj を実引数とする．(2)の in (N, P2, ···, Pj) はタプルの受信に相当する．ここでは，P2, ···, Pj を仮引数とする．この in 文が実行されると，もし，TS 内の TYPE NAME がNに一致し，引数型も一致するタプルが存在すれば，それを TS から取り出し，in 文の仮引数パラメータ P2, ···, Pj を，取り出したタプルの実引数パラメータに代入する．この処理は，Prolog のユニフィケーションに似ている．もし，一致するタプルがなければ，in 文は，out 文によって，一致するタプルが TS に入るまで待機する．(3)の read (N, P2, ···, Pj) は，in 文と同様に，TS 内で一致するタプルを見つけて，自身の仮引数にタプルの実引数を代入するが，タプルを TS からは引き出さない．

一致するタプルを見つけられずに待機中の in 文，read 文があるときに，ある out 文が実行され，TS にタプルがおかれた場合，そのタプルに一致する in 文あるいは read 文が複数個あるときには，タプルを引き渡される in 文あるいは read 文は非決定的に決められる．

4. おわりに

以上，分散処理言語の方法論と，具体的な言語について概観した．なお，言語 Ada も分散処理言語として重要な位置を占めるが，Ada については本誌でも特集[17]や解説[18]も多いのでここでは割愛した．

Communicating Sequential Processes や Distributed Processes の概念はすでに多くの言語に取り入れられており，さらに新しい言語の手法も発展しつつある．たとえば最近の言語を従来の言語と比較してみると，従来プログラマが固定的に割り当てていた通信路が，最近の言語では実行時に動的に生成されるこ

と，同時に同じ変数を分けあうという共有変数の概念を排除していること，親プロセス，子プロセスの概念をなくし，呼び出された側が新たな制御権をもつというコルーチン的な考え方が，多く採用されてきていることがあげられる．

今後の問題も多いが，たとえば，コンパイラの機能の問題をあげれば，ユーザ側に並列性をあまり意識させずにプログラムさせる方法も検討されているが(NILなど)，プロセッサの配置状態についてユーザが知りうる場合，ユーザにも，プロセスの割り付けに介入の余地をもたせたほうがよいのかどうかという問題があり，また，タイプチェックを実行時にすべきであるのかという問題などがあげられる．

参 考 文 献

1) Dijkstra, E. W.: Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, Comm. ACM, Vol. 18, No. 8, pp. 453-457 (1975).
2) Hoare, C. A. R.: Communicating Sequential Processes, Comm. ACM, Vol. 21, No. 8, pp. 666-677 (1978).
3) Brinch Hansen, P.: Distributed Processes: A Concurrent Programming Concept, Comm. ACM, Vol. 21, No. 11, pp. 934-941 (1978).
4) Hull, M.E. C. and McKeag, R. M.: Communicating Sequential Processes for Centralized and Distributed Operating System Design, ACM Trans. Program. Lang. and Syst. Vol. 6, No. 2, pp. 175-191 (Apr. 1984).
5) Feldman, J. A.: High Level Programming for Distributed Computing, CACM, Vol. 22, No. 6, pp. 353-368 (June 1979).
6) Cook, R. P.: *MOD-A Language for Distributed Programming, IEEE Trans. SE, Vol. SE-6, No. 6, pp. 563-571 (Nov. 1978).
7) Mao, T. W. and Yeh, R. T.: Communication Port: A Language Concept for Concurrent Programming, IEEE Trans. SE, Vol. SE-6, No. 6, pp. 194-204 (Mar. 1980).
8) Brinch Hansen, P.: Edison-A Multiprocessor Language, Software-Pract. and Exper., Vol. 11, No. 4, pp. 325-361 (Apr. 1981).
9) Brinch Hansen, P.: The Design of Edison Software-Pract. and Exper., Vol. 11, No. 4, pp. 363-396 (Apr. 1981).
10) Brinch Hansen, P.: Edison Programs, Software-Pract. and Exper., Vol. 11, No. 4, pp. 397-414 (Apr. 1981).
11) Andrews, G. R.: The Distributed Programming Language SR-Mechanisms, Design and Implementation, Software-Pract. and Exper. Vol. 12, No. 8, pp. 719-753 (Aug. 1982).
12) Strom, R. and Yemini, S.: NIL: An Integrated Languge and Systems for Distributed Programming, SIGPLAN '83 Symposium on Programming Language Issues in Software Systems (June 1983).
13) Strom, R. and Halim, N.: A New Programming Methodology for Long-Lived Software Systems, IBM Res. Dev., Vol. 28, No. 1, pp. 52-59 (Jan. 1984).
14) Strom, R. and Yemini, S.: The NIL Distributed Systems Programming Language: A Status Report, SIGPLAN Not., Vol. 20, No. 5, pp. 36-44 (May 1985).
15) Scott, M. L. and Finkel, R. A.: LYNX: A Dynamic Distributed Programming Language, Proceedings of the 1984 International Conference on Parallel Processing, pp. 395-401 (1984).
16) Gelernter, D.: Generative Communication in Linda, ACM Tran. Program. Lang. and Syst., Vol. 7, No. 1, pp. 80-112 (Jan. 1985).
17) 小特集：プログラミング言語—Pascal と Ada，情報処理，Vol. 22, No. 2, pp. 86-148 (1981).
18) 古谷立美：高級言語による並列処理の記述，情報処理，Vol. 21, No. 9, pp. 949-958 (1980).

(昭和62年3月19日受付)