研究速報
UDC 681.326=092

分散処理システム記述用言語—DPL

A Language for Distributed Processing Systems

浜田　喬*・佐藤文一*
Takashi HAMADA and Fumikazu SATO

1. はじめに

近年，複数のプロセッサが有機的に結合した分散処理システムが，続々と現れ，その将来性が脚光をあびる様になってきた．この理由としては，第1に，大形計算機の様な過度の集中により，その複雑さによる弊害が大きくなって，行き詰まりをもたらし，集中化に対する反省がでてきたことである．第2に，LSI技術の進歩により，マイクロプロセッサなどが，低価格化・高性能化し，その実現が経済的に可能になったことである．分散処理システムの主な利点として，並列動作による処理能力，特にスループットの向上，性能・価格比の向上，信頼性の向上，性能増加が容易な点，資源の共有，などがあげられる．しかし，多くの場合，そのハードウェアができあがっても，それを十分生かせるだけのソフトウェアが立ち遅れていた．従来のソフトウェアは，主に単一プロセッサ上のマルチプログラミングによる並列問題，あるいは，コンピュータ・ネットワーク上のNOS（Network Operating System）の問題として研究がなされてきた．前者として，Concurrent Pascal[1]やModula[2]の言語があるが，これらは主記憶共有形マルチプロセッサの言語として採用することはできても，主記憶分散形システムに適した言語とはいえない．後者のNOSは，すでに存在する種々のOSを，全体としていかにうまくつなげ調和させるかを主眼にしたものであって，最初から，システム全体を統一的に記述することを目的としてはいない．したがって，いずれの場合も，種々のハードウエア構成をもつ分散処理システムを，統一的・効率的に記述できる言語には適していない．分散処理システムでは，ソフトウェアの規模が比較的大きく，複雑であるため，また論理的あるいは地理的にその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．このため，信頼性・保守の面からも高水準言語で記述することが必要であり，そのための言語の開発が望まれていた．近年分散処理のためのプログラミング技術が，幾つか発表される様になってきた．HoareのCSP[3]（Communicating Sequential Process）によって，プロセス同士が共有変数をもたないで，直接メッセージによって通信を行う手法が再認織される様になった．さらに，Brinch Hansenは，プロセスとモニタの概念を結合したdistributed process[4]の概念を提案している．最近発表された，米国国防省（DOD）の言語ADA[5]において，その影響が表れている．筆者らの提案するDPL（Distributed Processing Language）もこの流れをくんだものである．

2. 分散処理

分散処理とは，並列に動作しうる複数個の処理（プロセス）が，論理的あるいは地理的にも分散し，かつそれらが協調して仕事を行っていく形態である．この場合，並列に処理されることから生じる並列問題として，

（i）複数のプロセスの進行速度の違いにより，その結果が異なってくる時間依存性の誤り

（ii）プロセスが永久に実行されない状態になるデッドロック

（iii）プロセス間の通信・同期の問題

（iv）共有資源の管理

がある．これらのために，プログラムのちょっとしたミスが，システム全体を破局的な動作に導いてしまう危険性がある．また，その実行時のテストが，ほとんど不可能なため，誤りを見つけるのが困難である．分散処理の場合，これらに加えて次の点が問題になる．

（i）システム全体の情報を把握するのは不可能であるため，局所的情報に基づいて制御しなければならない．

（ii）共有資源への多重アクセスを管理する部分は，効率が良くないと，システム全体の効率を下げ，ボルトネックになる恐れがある．また，通信による遅延も無視できない．

3. DPL

3.1 DPLの特徴　DPLは，分散処理システムを効率良く管理でき，信頼性・記述性に富んだ高水準言語になることを目的とした言話である．その特徴，および設計方針を列記すると，以下の様になる．

- プログラムを階層化されたモジュール構造により記述できる．
- DPLは，仮想機械上を動作するが，これとのインタフェースが定義できる．これにより，プリミティブな命令が拡張可能である．必要によっては，インタフェ

*東京大学生産技術研究所　第3部

ースの部分で，プロセスの優先度や，消滅のための命令を付け加えることにより，迅速な処理が要求されるプロセスや，動的なプロセスを記述できる.
- プロセス間の通信・同期が記述できる.
- ハードウェアに依存する入出力機器の制御が記述できる.
- 例外処理・誤り処理が記述できる.
- 高水準言語の採用により，記述性・信頼性・拡張性に富む.

3.2 DPLの言語仕様 DPL言語の記述は, N.Wirthによって設計されたプログラミングPASCALを土台にし，これに新たな機能を付加したものである．以下，DPLの特徴的なところを中心に述べる.

3.2.1・プログラムの構成 プログラムは，definition module と module とから構成される．moduleの中で，さらに他のmoduleやprocessの定義を含むことができる.

module は，機能的に密接な関係にある処理手続きの集合と，それらの手続きが取り扱うデータ，module の初期値を設定する initial ルーチンから成る．module は可視性を制御するための壁の役割をもつ．module 間のつながりを示すのが，visible 宣言と use宣言である．前者は，外部に対して使用を許可した名前を定義するための宣言である．後者は，この情報を使用するための宣言である．これらにより，必要最小限の情報を提供し，その他の不必要な情報は，他から見ようとしても見えない様になっている．コンパイル時に，正しく使用されているかどうかチェックでき，データの保護・信頼性の増加に役立つ．なお，visible 宣言によって，外部から使用することのできるモジュール変数は，外部からは読み出しだけが許される.

process 宣言は，他に宣言された process と同時に実行できる遂次的なアルゴリズムを宣言するためのものである，構造は moduleに類似している．プロセスの実行開始は，activate 文によって行われる．activate 文により実行されたプロセスが，その初期値設定ルーチンを終えると，他のプロセスは，このプロセスと通信可能な状態になる.

definition module は，DPL をインプリメントする際，その仮想計算機とのインタフェースを定義するための構文である．仮想計算機は，ターゲット・プロセッサ上で，DPL で記述されたシステム・プログラムの実行をサポートするためのものである．その構成は，DPLのコンパイラで生成されるオブジェクト・コードを実行するインタプリタと，プロセスの生成やスケジューリングなどの基本的な機能を提供する制御プログラム核（kernel）とから成る.

```
MODULE FIFO;
  VISIBLE  FUNCTION ARRIVAL, DEPARTURE END;

  VAR HEAD, TAIL, LENGTH: INTEGER;
  FUNCTION ARRIVAL: INTEGER;
    BEGIN
      ARRIVAL: =TAIL;  TAIL: =TAIL MOD L +1;
      LENGTH: =LENGTH+1
    END;
  FUNCTION DEPARTURE: INTEGER;
    BEGIN
      DEPATURE: =HEAD;  HEAD: =HEAD MOD L +1;
      LENGTH: =LENGTH-1
    END;
  INITIAL HEAD: =1;  TAIL: =1;  LENGTH: =0 END;
END FIFO;
```

図1　module の例

```
DEFINITION MODULE SYSTEM;
  EXCEPTION OVERFLOW;
  EXCEPTION DIVIDE_ERROR;
  TYPE DEVICE= (CONSOLW, PTR);
  TYPE OPERATION= (INPUT, OUTPUT);
  PROCEDURE IO (D: DEVICE;  OP: OPERATION;
  C: IN OUT CHAR);
END SYSTEM;
```

図2　definition module の例

3.2.2 プロセス間通信と同期 プロセス間の通信方式は，基本的には，共有変数による通信とメッセージによる通信の2つに分類できる．前者の手法は，各プロセスが，直接その変数にアクセスできることを前提にしているので，主記憶分散形システムの記述には適していない．後者が，環境に左右されない通信の手法といえる．この最も基本的な形式が，send と receive のコマンドを用いて通信・同期を行う手法である．DPLでは，信頼性・記述性の面からも，コマンド方式で行うのではなく，あたかも手続き呼び出しの形で実行できるようにした.

DPL でのプロセス間通信は，他のプロセス上で定義されている procmail, procprocess を呼び出すことによって行われる．外部からは，どちらの方法で通信が行われているかを気にかける必要はない.

ここで，生産者・消費者問題の DPL での記述を考えてみる．この問題は，生産者プロセスがデータを生産し，消費者プロセスがそのデータを消費するのである．この場合，次の様な同期のための条件が満たされなくてはならない．バッファが一杯の時に，生産者がデータをバッファを入れようとした時には，消費者がバッファからデータを取り出すことによって，バッファに空きができるまで，生産者は待たされる．また，バッファが空きの時に，消費者がデータを取り出そうとした時には，生産者がバッファにデータを入れるまで，消費者は待たされる. 図3は，procmailを用いて，この問題を記述したものである.

図4は，図3のプログラムにおける情報の流れを示したものである．

process Tが，process S の procmail P を呼び出した場合の実行は次の様に行われる．

実パラメータを評価し，要求(パラメータ＋ヘッダ)を相手プロセスSに転送する．この時，Tは，S から返事が送られてくるまで待つ reply wait 状態になる．

select 文は，この様な複数のプロセスから送られてる要求を，自分自身の内部状態と照らし合わせながら，どれを選択するかを決定するための文である．この方式は，自分の所へ到着する局所的情報(要求)のみを用いて制御しているので，分散処理に適している．

select 文の構文は

$SELECT\ B_1 : PM_1 ; B_2 : BM_2 ; \cdots\cdots B_n : PM_n\ END$

である．ここで B_i は論理式，PM_i はprocmail名である．論理式 $B_i \wedge PM$. mailbox ($1 \leq i \leq n$)の中で，真の値のものを一つ非決定的に選び出し，対応するprocmailの本体を実行する．ここで，PM_i. mail boxは，PM_i のmailboxに蓄えられている要求が空のとき偽，それ以外では，真の値をとる．すべての論理式が偽のとき，要求待ちの request wait 状態になる．procmail 本体の手続きの実行を終えると，呼び出したプロセスに対して返事を送り，これを受け取ったプロセスは実行可能な状態になる．なお，論理式を評価している間は，mail box はロック状態にある．

さらに効率的な通信を実行するための道具として，procprocess の概念を導入した．これは，procedure と process を結合した概念である．図5は，図3の buff-erprocess を procprocess を用いて記述したものである．processT が processS の procprocessP を呼び出すと，次の様に実行される．

TとSが同一計算機内にある時は，直接アクセス可能なので，Tは(Sの内部に入り)Pをprocedureとして実行する．TとSが異なる計算機内にある時は，TはSの要求(パラメータ＋ヘッダ)を送り，Pを実行するための内部プロセスをS内に生成または獲得する．

したがって，procprocess を呼んだプロセスの数だけの並列度で実行できる．図6の buffer の内部においては，本体を実行するプロセスと，procprocess を呼んだ複数個のプロセスが存在すると見ることができる．この場合，slot と full は，内部共有変数となるので，これらプロセス間の同期をとるための機構が必要となる．DPLではこのための道具として，Brinch Hansen の guarded region を採用した．これは，一度にはたった1つのプロセスしか実行できない領域に対してプロセスが入ろうとする際，ある条件が成立するまで待つ様にしたものである．

```
MODULE PRODUCER_CONSUMER;
  TYPE LINE=ARRAY (.1..12.) OF INTEGER;

  PROCESS BUFFER;
    VISIBLE  PROCMAIL PUT, GET  END;

    VAR SLOT: LINE;  FULL: BOOLEAN;
    PROCMAIL PUT (M: IN LINE);
      BEGIN SLOT: =M;  FULL: =TRUE END;
    PROCMAIL GET (M: OUT LINE);
      BEGIN M: =SLOT;  FULL: =FALSE END;
    INITIAL FULL: =FALSE END;
    BEGIN
      LOOP
        SELECT  NOT FULL: PUT;  FULL: GET  END
      END
    END
  END BUFFER;

  PROCESS PRODUCER;
    USE BUFFER;
    VAR M: LINE;
    BEGIN LOOP BUFFER. PUT (M)  END END
  END PRODUCER;

  PROCESS CONSUMER;
    USE BUFFER;
    VAR M: LINE;
    BEGIN LOOP BUFFER. GET (M)  END END
  END CONSUMER;

  INITIAL
    ACTIVATE BUFFER, PRODUCER, CONSUMER;
  END;
END PRODUCER_CONSUMER;
```

図3　生産者消費者問題

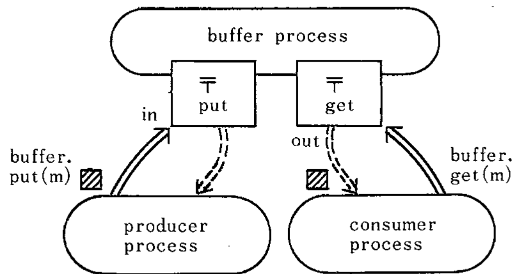

図4　プロセス間の情報の流れ

構文は

$WHEN\ B_1 : S_1 ; B_2 : S_2 ; \cdots\cdots ; B_n : S_n\ END$

で，論理式 $B_1 \sim B_n$ の中から真のものを1つ非決定的に選び出し，対応する文を実行する．

3.2.3　例外処理　プログラム実行中に生じる誤りや，例外を扱うための構文を説明する．使用するすべての例外は，前もって宣言されてなければならない．

EXCEPTION OVERFLOW

本体の実行中に，例外が生じた時，それを処理するため

```
PROCESS BUFFER;
  VISIBLE  PROCPROCESS PUT, GET  END;

  VAR SLOT: LINE;  FULL: BOOLEAN;
  PROCPROCESS PUT (M: IN LINE);
    BEGIN
      WHEN NOT FULL: BEGIN SLOT: =M;
      FULL: =TRUE END END
    END;
  PROCPROCESS GET (M: OUT LINE);
    BEGIN
      WHEN FULL: BEGIN M: =SLOT;
      FULL: =FALSE END END
    END;
  INITIAL FULL: =FALSE END
END BUFFER;
```

図5　buffer process

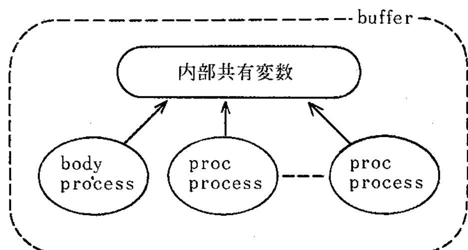

図6　buffer process の内部構造

の実行を定義したのが，exception handler 部である．

```
BEGIN
  ............
  X: = X+I
  ...............
EXCEPTION UPON OVERLOW DO X := 0 END
END
```

なお，例外は，raise文により強制的に生じさせることができる．

```
RAISE OVERFLOW;
```

3.2.4 その他の記法

```
PROCESS READER－WRITER;
PROCPROCESS READ(V: IN ELEM)<LIMIT 4>
PROCMAIL WRITE (E:OUT ELEM)<LIMIT 4>
```

procprocess や procmail を実行時に呼ぶことができる最大のプロセスの数を，limit により定義する．これにより，mail box や内部プロセスを要求のたびに生成するのではなく，あらかじめ必要なだけを生成しておき，要求のたびに割り当てるという静的管理が可能となる．静的管理は，動的管理に比べて，メモリ占有量が大になる恐れはあるが，管理の簡素化，デッドロックの危険性を防ぐという利点がある．

```
ACTIVATE READER－WRITER ON NODE 1;
```

この文により，プロセスを，どのノード(計算機)で作成するかを決めることができる．

```
PROCESS PHILOSOPHER(. 1.. 5.);
ACTIVATE PHILOSOPHER(. 3.);
```

プロセスを配列と同じ様に宣言することにより，同時に複数個定義できる．

```
GENERIC (L: INTEGER)
MODULE FIFO;
....................
END FIFO;
....................
MODULE FIFO1 = NEW FIFO(5);
```

generic 宣言により，process や module を一つの型の様に扱うことができる．

```
BUFFER. GET (M) DURING  I := I+1;
I := I * 10 END
```

プロセス間通信において，要求転送によって処理を依頼する場合，通信による遅延が無視できない場合がある．この場合，during 節によって，すぐに reply wait 状態になるのではなく，この節の実行が終えてから，reply が来てるかどうか調べるようにする．

4. おわりに

現在，以上の仕様のDPL のコンパイラを，Brinch Hansen が設計した言語 Sequential Pascal で作成中である．今後コンパイラができしだい，インタプリタ，および分散処理用の核（Kernel)を作成し，DPL を評価していきたい．なお，DPL 言語の設計の際に，特に考慮しなかったが，次の点も検討してみたい．

（i）プロセッサ間の負荷が均等になる様に，また，ボトルネックプロセッサが発生しない様にするための機構

（ii）システムを完全に止めて，ソフトウェアを入れ変えることができない場合，実行時にある特定のプロセスを修正したり，新しいプロセスを付加できるための機構

最後に，日頃御指導いただいている渡辺勝先生，藤田長子先生，および，渡辺・浜田研の方々に感謝する．

（1979年10月25日受理）

参　考　文　献

1) P. Brinch Hansen "The Architecture of Concurrent Programs" Prentice Hall 1977.
2) N. Wirth "Modula: A Language for Modular Multiprogramming" Software Practice and Experience vol. 7 No. 1 1977 pp 3－35.
3) C. A. R. Hoare "Communicating Sequential Processes" CACM vol. 21 No. 8 Aug. 1978 pp 666－677.
4) P. Brinch Hansen "Distributed Processes; A Concrrent Programming Concept" CACM vol. 21 No.11 Nov. 1978. pp 934－941.
5) "Preliminaly ADA Reference Manual", Rational for the Design of the ADA Programming Language" ACM SIGPLAN vol. 14 No. 6 June 1979.