研　究　速　報

UDC 681.3.068

分散処理システム記述用言語DPLの実装

Implementation of DPL

浜田　喬*・半田剣一*・宮内　宏*

Takashi HAMADA, Kenichi HANDA and Hiroshi MIYAUCHI

1. はじめに

近年のハードウェア技術の進歩により，従来の中央集中処理システムに対して，処理能力・性能価格費・信頼性の向上が可能な分散処理システムの重要性が増大している．一方，分散処理システムのハードウェアを十分に生かせるだけのソフトウェアを効率よく記述するためには，信頼性・記述性の高いプログラミング言語が必要である．このため，単一計算機内の並列処理の手法を発展させた方法が考案されてきている．単一計算機内の並列処理記述言語としては，Concurrent Pascal や Modula 等が作られた．Concurrent Pascal による Solo システムは高級言語で記述された最初の大きな並列処理システムである．これらの研究が分散処理に進むうえで，共有メモリをもたないプロセッサ同士をいかに協調させて効率の良い処理を行わせるかという問題があった．そのため Hoare の C.S.P. (Communicating Sequential Processes)によってプロセス同士が共有変数を使用せずに，メッセージのやりとりによって通信する手法や B. Hansen のモニタとプロセスの概念を結合した D.P. (Distributed Processes)による手法等が発表されている．これらの分散処理システムのプログラミング技術に基づいて，＊ MOD, PLITS, Edison, OCCAM, Ada 等の分散処理の記述が可能な言語が開発されてきている．

本研究室でも，Concurrent Pascal をベースとして，これに分散処理のための拡張，改良を加えたプログラミング言語である DPL (Distributed Processing Language)の設計，コンパイラの作成，仮想機械の製作を行ってきた．本稿では，DPL の言語の概要，仮想機械の構成につき述べる．

2. DPLの特徴

DPL はプロセスを単位として分散処理システムを従来の並列プログラミングの手法を用いて記述できる，信頼性，記述性に富んだ高級言語である．DPL で分散処理システムを記述する際は，システム内の各ノードの結合形態や，それぞれのノードにあたる計算機のハードウェアを特に意識する必要はない．これは，DPL システムを DPL プログラム自身である上位レベルと各ノード上でプログラムの実行をサポートする仮想計算機である下位レベル2つの階層に分割しているからである．

DPL は，Concurrent Pascal の言語的特徴を多く継承しており，Concurrent Pascal になかった特徴としては以下のようなものがある．

・プロセス間の通信は，モニタを介することなく直接他のプロセス内の手続きを呼び出すことによって行われる．

・各プロセスは DPL システム内の任意のノード上に生成できる．

・共有資源にアクセスしようとするプロセス間で同期をとる機構として B. Hansen による guarded region を採用している．

・例外処理，誤り処理の記述ができる．

・DPL 仮想計算機とのインタフェースが定義でき，高級言語では記述がむずかしい低レベルの処理は仮想計算機内に組み込まれたルーチンを呼び出すことによって行う．

・変数を特定のアドレスに割り付けることができる．これと DPL 仮想計算機とのインタフェース機能により，周辺装置の制御などは DPL で直接記述できる．

3. DPLの言語仕様

3.1 プログラムの構成　DPL プログラムは，definition module 部・宣言部・初期プロセスの本体から構成される．definition module 部では，プログラム全体で使用する定数・型・例外名を定義するとともに，仮想計算機内のルーチンとのインタフェースも定義できる．宣言部では，型宣言(プロセス，クラスの宣言を含む)，初期プロセス用の定数・変数の宣言を行う．プロセス，クラスは Concurrent Pascal におけるそれと同じ機能を持ち，いずれも入れ子構造が可能である．初期プロセスは

＊ 東京大学生産技術研究所　第3部

プログラムの実行開始とともに自動的に生成されるプロセスで，これが生成されるノードを親ノードという．DPLシステムは，1つの親ノードといくつかの子ノードからなる．

3.2 プロセスの生成　図1のようなinit文を実行することによりNODEの示す番号のノード上にPROC_A型のプロセスPROCESSAとPROC_B型のプロセスPROCESSBが生成され，PROCESSBにはPROCESSAへのアクセス権が設定される．

3.3 プロセス間通信と同期　各プロセスは他のプロセスから呼び出せる手続きとして，procmailおよびprocprocessを定義できる．1つのプロセス内のprocmailはそれぞれ排他的に実行される．プロセスが自分の持つprocmailに対して複数プロセスから呼出しを受けた場合，一度には1つのプロセスにだけその実行を許し，他のプロセスは待たせておく．この同期と排他性はselect文によって制御される．procmailを持つプロセスがselect文を実行すると，条件部が真であるprocmailのなかから1つを選んでこれと通信を行い，そのprocmailの実行を許可する．そのようなプロセスがない場合はselect文中で待っている．

これに対しprocprocessは，呼び出した数だけの並列度で非同期に実行される．ただし，共有資源にアクセスするなどでプロセス間の同期が必要なときは次のような構文のguarded region（when文）を使用する．

WHEN B1：…；B2：…；B3：…END

1つのプロセス内のwhen文の中身は1度に1つのプロセスしか実行できない．when文を持つプロセスは条件部が真のものを1つ非決定的に選択して，実行する．

3.4 誤り・例外処理　DPLでは，あらかじめ宣言しておいた例外名に対応する例外が生じた場合の処理を，exception handler部に記述できる．例外には，仮想計算機が自動的におこすものと，プログラム中で，assert文やraise文で強制的に起こされるものがある。

3.5 その他　変数を任意のアドレスに割付けられる．

VAR A：ARRAY（.1..5.）OF REAL AT # 0200；

上の例では配列Aを0200番地からの連続した領域に割り付ける．

```
const NODE = 3 ;
var PROCESSA : PROC_A ;
    PROCESSB : PROC_B ;
begin
  init PROCESSA ;
  init (NODE, PROESSB(PROCESSA)) ;
  ------
end
```

図1　init文

関数や手続きの仮引数の属性にはin（参照のみ），out（新に設定），in out（更新）の3種がある．

以上が，DPLの概略である．現在，DPLコンパイラはFACOM M-180AD上で動いている．このコンパイラは，Concurrent Pascalコンパイラが出力するCon-codeに類似した仮想命令コードD-codeを出力する．

4. DPL仮想計算機dove

dove（distributed operating machine executive）は，ミニコンPANAFACOM U1400上に作成された親ノード用のDPL仮想計算機である．doveはアセンブラ言語で記述された13キロバイト強のプログラムであり，その設計に際してはつぎの点に留意した．

・各種バッファ領域等は必要になった時点で動的に獲得されるようにし，プログラム自体の大きさは最小限に留める．

・各種機能の拡張・縮小を簡単に行えるようにモジュール化を徹底する．

doveのプログラム上の構成は，処理の対象となるデータによって13個に分割されたプログラムモジュールと全プログラムモジュールから参照されるデータ，そのノードについての初期設定部と割込処理部からなる．さらに，各プログラムモジュールは，実行の形成によって区別される次の3種の構成要素を持っている．

（1）モジュールプロセス　他のプロセスと並行して実行できる処理，実行中になんらかの事象待ちが必要な処理はモジュールプロセスとして記述されている．doveが持つモジュールプロセスにはネットワークプロセスとカーネルプロセスの2種があり，プロセスマネージャによって前述のDPLで記述されたプロセス（以後システムプロセスと呼ぶ）と同様の扱いを受ける．

（2）スーパバイザルーチン（SVR）　プロセスとは異なり1つのノード上では1つのSVRしか実行できない．各SVRはプロセスもしくは他のSVRから処理依頼（SVR request）を受けて排他的に動作する．自分のノードのSVRだけでなく他のノードのSVRに対しても要求が行われる．前者をlocal SVR request，後者をremote SVR requestと呼ぶ。

（3）プロシジャ　他の構成要素から呼び出されて，その構成要素の一部として実行される．自ノードのプロシジャしか呼び出せない．

doveの機能は上位から，インタープリタ機能，カーネル上位機能およびカーネル下位機能の3つに階層化されていて，それぞれの機能ごとに数個のプログラムモジュールが処理にあたっている．以下では，各機能ごとのプログラムモジュールの処理概要を示す．

4.1 インタプリタ機能　D-codeの実行を行う．ただし，プロセスにまたがった処理やハードウェアを意識した処理は下の階層にまかせる．処理にあたるプログラムモジュールを次に示す．

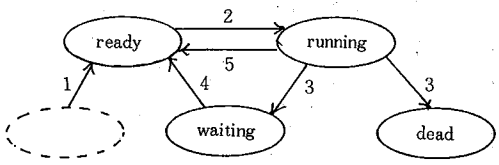
(1) インタプリタ　D-codeの解釈実行を行う．CPM (Concurrent Pascal Machine)のインタプリタと同じ手法を使用している．

(2) システムサービスルーチン　DPLの definition module 部で定義されたプロシジャの本体からなり，必要に応じてインプリタに呼び出される．

4.2 カーネル上位機能　プロセスの生成，プロセス間通信など，プロセスを意識した上での種々の処理を提供する．

(1) プロセスマネージャ　プロセスの生成，中断，起動などの処理を行う．これらに伴うプロセスの状態遷移を図2に示す．

(2) コードマネージャ　DPLシステムでは，D-codeがまず親ノードにロードされる．その後各子ノードにプロセスが生成されるごとに，必要となるD-codeが子ノードへ転送される．コードマネージャ内のカーネルプロセス Code-receiver はシステムプロセスと通信を行いながら図3に示す処理を行う．D-codeは抽象データ型単位で転送される．



1 : creator (kernel process)
2 : scheduler (procedure)
3 : wait (procedure)
4 : ready (SVR)
5 : postpone (procedure)

図2　プロセスの状態遷移

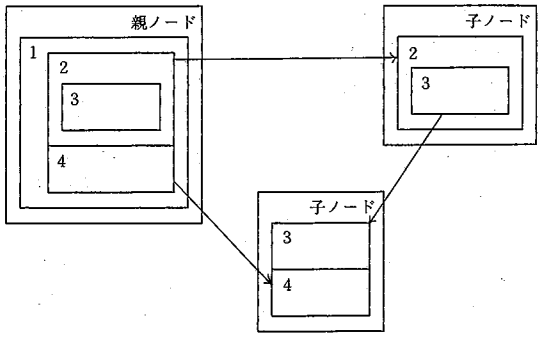


図3　D-codeの転送

(3) エクスチェンジマネージャ　procmailやprocprocessの呼出しは，コンパイラによって，相手プロセスの持つエクスチェンジへのsend要求と自分のエクスチェンジへのreceive要求に分割される．エクスチェンジに対するこれらの処理を行うものとして3つのSVR (¥send, ¥receive, ¥communication) があり，それぞれ，send要求の処理，receive要求の処理がマッチしたときのデータ転送を行う．

(4) ゲートマネージャ　前述のwhen文の機能を実現するためにゲートというデータ構造が使用される．各プロセスは生成時に独自のゲートを割りあてられ，そのプロセス内のprocprocessを呼び出してwhen文にぶつかったプロセスは一度に1つずつそのゲートに入れられていく．こうしてプロセスの同期と排他制御が実現されている．これらの処理のために3つのSVR (¥gate-enter, ¥gate-wait, ¥gate-exit) がある．

4.3 カーネル下位機能　ネットワーク管理やハードウェアに依存した処理を行う．

(1) リクエストマネージャ　SVR request が発せられて目的SVRに制御が移るまでの処理を行う．各構成要素は，メモリ上に要求先や種々のパラメータを持ったリクエストメッセージを作成してリクエストマネージャ内のプロシジャrequest-handlerに制御を移す．request-handlerは local SVR request であれば，直接目的SVRを呼び，remote SVR request であれば，リクエストメッセージをリクエストキューにつなぐ．

(2) ネットワークマネージャ　2つのネットワークプロセス sender と receiver を持ち，実際にノード間のデータ通信を行う．

(3) メモリマネージャ　doveのメモリ管理を行う．現在のインプリメントでは64 Kbyteの連続した領域を扱うことができる．

(4) タイママネージャ　1つのノード内の各プロセスの並列処理を模擬するために，タイマ割込みによるプ

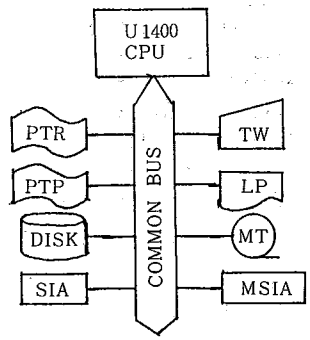
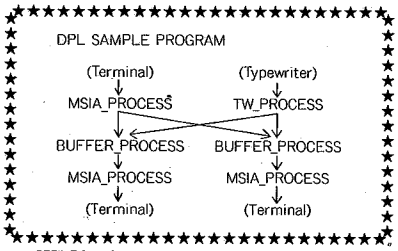


図4　ハードウェア構成

研究速報




```
DEFINITION MODULE TRANSFER ;
  CONST LF='(10)' ; EM='(25)' ;
        EOF='(26)' ; TB='(9)' ;
        CR='(13)' ; CAT='(64)' ;
        MAX_LINE=132 ;
        MAX_BUF=2 ;
        R_DSR=0 ; R_BCR=1 ; R_MAR=2 ; R_CMR=3 ;
        W_DSR=4 ; W_BCR=5 ; W_MAR=6 ; W_CMR=7 ;
  PROCEDURE GATE_TRIGGER_SET (PUNO : INTEGER) ;
  PROCEDURE DUMMY ;
  FUNCTION TEST_BIT (BIT, I : INTEGER) : BOOLEAM ;
END TRANSFER ;

TYPE LINE=ARRAY (.1..MAX LINE.) OF CHAR ;
TYPE REGISTER=ARRAY (.0..31.) OF INTEGER ;
TYPE MSIA_BUF=ARRAY (.0..15.) OF CHAR ;

(*********************)
(*    BUFFER PROCESS    *)
(*********************)

TYPE BUFFER_PROCESS=PROCESS
  VISIBLE PROCPROCESS PUT, GET END ;
  CONST BUF_SIZE=32 ;
  TYPE BUF_INDEX_TYPE=1..BUF_SIZE ;
  VAR TOP, BOTTOM : BUF_INDEX_TYPE ;
      BUFFER : ARRAY (.BUF_INDEX_TYPE.) OF CHAR ;
      FULL, EMPTY : BOOLEAN ;
      BUSY_FLAG : BOOLEAN ;

  FUNCTION NEXT (I : BUF_INDEX_TYPE) : BUF_INDEX_TYPE ;
  BEGIN
    IF I<BUF_SIZE THEN NEXT := I+1
                  ELSE NEXT := 1
  END.

  PROCPROCESS PUT (C : IN CHAR) ;
  BEGIN
    WHEN NOT FULL : BEGIN
      IF NOT EMPTY THEN BOTTOM := NEXT(BOTTOM)
                   ELSE EMPTY := FALSE ;
      BUFFER (.BOTTOM.) := C ;
      IF NEXT (BOTTOM)=TOP THEN FULL := TRUE
    END END
  END ;
  PROCPROCESS GET (C : OUT CHAR) ;
  BEGIN
    WHEN NOT EMPTY : BEGIN
      C := BUFFER (.TOP.) ;
      IF TOP=BOTTOM THEN EMPTY := TRUE
                    ELSE TOP := NEXT(TOP) ;
      FULL := FALSE
    END END
  END ;

  INITIAL
    TOP := 1 ; BOTTOM := 1 ;
    EMPTY := TRUE ; FULL := FALSE ;
    BUSY_FLAG := FALSE
  END

    BEGIN END
  END BUFFER_PROCESS ;

(*********************)
(*    TW PROCESS     *)
(*********************)

(*********************)
(*    MSIA PROCESS   *)
(*********************)

(*********************)
(*       MAIN        *)
(*********************)

VAR BUF1, BUF2 : BUFFER_PROCESS ;
    TW : TW_PROCESS ;
    MSIA1, MSIA3 : MSIA_PROCESS ;
    MSIA4 : MSIA_PROCESS ;

BEGIN
  INIT BUF1, BUF2 ;
  INIT TW (BUF1, BUF2) ;
  INIT MSIA1 (BUF1, BUF2) ; MSIA1. MINIT (1) ;
  INIT MSIA3 (BUF1, BUF2) ; MSIA3. MINIT (3) ;
  INIT MSIA4 (BUF2, BUF1) ; MSIA4. MINIT (4) ;
END.
```

図5　DPL プログラム例

ロセス切換えを行う．現在各プロセスが連続して CPU を占有できる時間の上限は 32 msec に設定してある．

5. DPL dove の実行例

図4に DPL を実装した U1400 のシステム構成を示す．ネットワークプロセスは SIA (Serial Interface Adapter) を介してデータの入出力を行うが，実際には他のノードでは，DPL マシンは未実装であるのでネットワークの代わりに，まとまったデータの送信及び受信が可能なマイクロコンピュータを接続してテストを行った．

図5に，MSIA (SIA が4個接続されたもの) を通して，図に示されている2つの通信を並列に行うプログラムを示す．システムプロセスは，TW_PROCESS　1個，BUFFER_PROCESS 2個，MSIA_PROCESS 3個の計6個である．TW_PROCESS はタイプライタの入力を制御するプロセスで，タイプライタから1文字入力しては，バッファに出力する．メインルーチンに示されるように，TW_PROCESS は通信先を変えられるように，2つのバッファにたいしてアクセス権が設定されている．BUFFER_PROCESS は，2つの procprocess put と get を持ち，他のプロセスは，これらを呼び出すことにより，BUFFER_PROCESS 内の変数にアクセスできる．バッファへの書込みと読出しは排他的に行われなければならないので，when 文を使って同期をとっている．MSIA PROCESS は MSIA から1文字入力してバッファに出力する機能とバッファから1文字入力して MSIA に出力する機能を備えている．

このプログラムを実行させつつ，ネットワーク側から，コードマネージャ内の Code-receiver に対してある単位の D-code の所在を問い合わせるための remote SVR request を送り，dove の動作を確認した．

6. おわりに

現在までに，単一ノードの仮想計算機の作成・テストまで進んだわけであるが，今後の課題としては次のような点があげられる．

- 他の計算機に dove を移植すること
- 複数の計算機を結ぶネットワークを構成すること
- より下位のレイヤの通信プロトコルを規定すること
- 実際にプログラミングした際にでてくる DPL の使用しにくい点に対する検討

(1985年1月21日受理)

参考文献

1) 浜田・佐藤，“分散処理システム記述用言語—DPL”生産研究 1980.2
2) 半田・浜田，“高級言語指向分散処理システムの構成法” 情報処理学会分散処理システム研究会 18-4, 1983