Vol. 26 No. 3 情報処理学会論文誌 May 1985

分散処理システム記述用言語 DPL とその実装法†

半田剣一††* 浜田喬††

高級言語の使用により，信頼性の高い分散処理システムを効率的に記述できる．本論文は分散処理システム記述用の高級言語 DPL と，そのプログラムを目的計算機上で実行するための仮想計算機 DOVE について述べている．DPL はプロセスを単位とした分散処理プログラムを従来の並列プログラミングの手法を使って記述できる Pascal 系の言語である．DPL は任意のノード上に動的にプロセスを生成でき，それらプロセス間の通信はノードの違いを意識せずに手続き呼出し型の形式で記述できる．またプロセスの同期には guarded region を採用し，システム機能に必要な柔軟な処理が記述可能である．DPL システムは，システム全体を統合的に記述した DPL プログラム自身と，各ノードにあたる計算機のアーキテクチャの相違を吸収しプログラム実行をサポートする仮想計算機とに階層が分かれる．後者は DPL システム内のすべてのノード上に実装されるべきソフトであり，DOVE はとくに親ノード上の仮想計算機としての機能をもっている．実際のノード間データ通信も DOVE がサポートするものであるが，データ型式の統一によりその処理は非常に簡潔なものとなった．実行テストは単一計算機でのシミュレートであるが，十分 DPL および DOVE の有用性を確かめられるものであった．

1. はじめに

並行処理を高級言語で記述するための研究は，セマフォ[3]，モニタ[5]等の概念を経て，Concurrent Pascal や Modula 等の言語を実用化してきた．これらの研究はさらに分散処理へと進み，CSP[6] や DP[1] 等の概念が提案され，Edison[2]，OCCAM[7]，Ada[4] 等の分散処理の記述能力をもつ高級言語が次々と発表されてきている．

著者らも分散処理システム記述用の言語 DPL の設計[8]とそのコンパイラの作成[9]を行ってきたが，今回 DPL の記述性・信頼性を評価するために DPL を実際の計算機上にインプリメントしたので，その経過について述べる．

2. 分散処理システム記述用言語 DPL

DPL (Distributed Processing Language) は Concurrent Pascal をベースにして，これに分散処理用の拡張，改良を加えた言語であり，比較的大きなプロセスを単位とした分散処理プログラムを記述することを目的としている．

2.1 DPL システムの概要

DPL が記述の対象とするシステムは，研究室単位あるいは建物単位の，ミニコンやマイコンクラスの計算機による LAN 上のオペレーティングシステムである．このためには，

- 低レベルから高レベルの処理までの幅広い記述性
- LAN のノードとなる各計算機の資源の有効利用
- コンパイラを通った後のプログラムの信頼性
- 小メモリのマイコン上でも動くコンパクト性

等が必要であるが，これまでこれらすべてを兼ね備えた言語はなかったので，DPL の設計に際してはこれらの点をつねに念頭に置いた．

DPL システムは独立した計算機を一つのノードとし複数のノードを通信回線で結んだものである．システムの記述に際しては各ノードの結合形態や計算機アーキテクチャをとくに意識する必要はなく，システム全体を一つのプログラムとして記述できる．これは DPL システムを図1のように二つの階層に分割しているためである．上位層は DPL プログラム自身であり，下位層は各ノード上でのプログラム実行をサポートする仮想計算機(ソフト)である．これにより，プログラムは各ノード上で動くプロセスすべてを統一的に記述できるのでシステムの見通しがよくなるうえ，各ノードに必要なソフトは DPL プログラムのオブジェクトコードを実行する小さな仮想計算機のみであり，システムの拡張性，移植性がよくなる．

DPL システムは一つの親ノードと複数の子ノードからなり，親ノードが，システム立ち上げ時に初期プロセスを走らせて必要なだけのオブジェクトコードを子ノードに分配する機能をもつほかは，親ノードと子ノードの違いはない．このようなコード分配方式を採ったのは，子ノードの仮想計算機をコンパクトにするのと，システム変更の際の手順を簡単にするためであ

† DPL and Its Implementation for Describing Distributed Processing by KEN'ICHI HANDA and TAKASHI HAMADA (Institute of Industrial Science, University of Tokyo).

†† 東京大学生産研究所
* 現在 電子技術総合研究所パターン情報部推論システム研究室

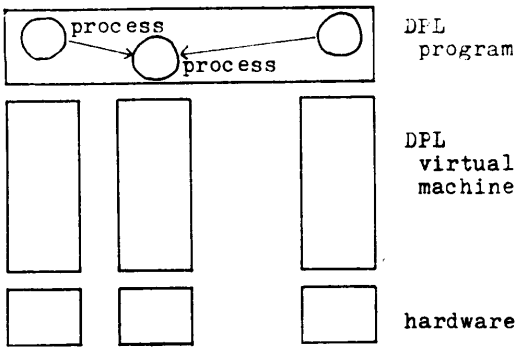
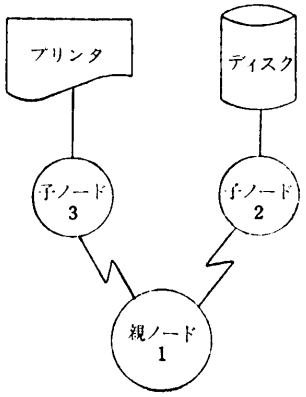


図 1 DPL システムのアーキテクチャ
Fig. 1 DPL system architecture.




```
definition module DOORS;                          ┐
  procedure LAMP (KINO: integer);                 ┘ 定義モジュール部
end DOORS;
type DISK_DRIVER=process                          ┐
  begin ··· end                                   │
end DISK_DRIVER;                                  │
type PRINTER_DRIVER=process                       │ 宣言部
  begin ··· end                                   │
end PRINTER_DRIVER;                               │
var DISK: DISK_DRIVER;                            │
    PRINTER: PRINTER_DRIVER;                      ┘
begin                                             ┐
  init (2, DISK);                                 │ 初期プロセス
  init (3, PRINTER);                              │
  ···                                             ┘
end.
```

図 2 DPL プログラムの構成
Fig. 2 Structure of DPL program.

る.

2.2 分散処理 OS 記述用の言語としての特徴

DPL の Concurrent Pascal からのおもな変更点は，あるノードから他のノードに向けてプロセス生成の命令を実行できる点と，どのノードのプロセスに対してもそのプロセス内の手続きを呼び出すことによってプロセス間通信ができる点である.

まず，プロセス生成について DPL ではプログラマがノードを明示して（各ノードは一連の番号を振られている）そこにプロセスを作るという直接的な方法を採ったが，これは先に述べたように DPL の記述対象を研究室内あたりの LAN としたためである．この場合，ノードとなる計算機はそれぞれ特色のある資源をもつことが多く，プログラマはそれを熟知して各ノードに合ったプロセスを生成し資源の活用を図ることが重要となるが，そのためにはこの方法が確実でありプログラマにとっても記述しやすいと思われる.

またプロセス間通信であるが，言語としての記述性・信頼性を考えて，Ada や OCCAM のようなメッセージ通信型ではなく Edison のような手続き呼出し型を採用した．一般に後者は前者に比べると記述能力が劣るといわれるが，通信の際に同期をとる機構の拡充により十分柔軟な処理を記述できるようにし，またメッセージ通信型では無理な，同期をとらないプロセス間通信も可能とした.

その他，OS 記述用の言語にとって重要な例外処理や誤り処理の機能は Ada を参考にして付加した．また変数のアドレス割付け等によりハードウェアに依存した処理もかなり記述できるようになっているが，それでも高級言語では記述がむずかしい部分は仮想計算機内のルーチンに任せて，DPL プログラムではインタフェースを“定義モジュール部”で定義することによってそのルーチンを呼び出すことができるようにした.

図 2 に DPL プログラム全体の構造を示す．これは大容量のディスクをもつマイコンと高品質プリンタをもつマイコンをそれぞれディスクサーバ，プリンタサーバとして機能する子ノードとする DPL システムの例である．定義モジュール部の手続き定義は親ノードの表示ランプを駆動するためのもので手続き本体は仮想計算機内にある.

さて DPL の分散処理機能を特徴づけるものは上に述べたプロセス生成とプロセス間通信であるが，以下ではこれらについて詳しく説明する.

2.3 プロセス生成・プロセス管理

“宣言部”は入れ子になった抽象データ型であるが，プロセスもここで抽象データ型として定義される．図 3 ではAやBがそうしたプロセス型である．そして実際にプロセスを生成するにはプロセス変数の宣言を行った後に init 文を実行すればよい．Concurrent Pascal と異なるのは init 文のパラメータにプロセスを生成するノードの番号が含まれる点だけであり，他のプロセスへのアクセス権の渡し方などは変わっていな

```
type A=process                    /*  Aのプロセス型定義  */
  ...
end A;
type B=process (PA : A)           /*  Bのプロセス型定義  */
  ...
end B;
const NODE 1=1; NODE 3=3;         /*  ノード番号の定義  */
var PA: A; PB: B;                 /*  プロセスの宣言  */
begin
  init (NODE 1, PA);              /*  PA をノード1に生成  */
  init (NODE 3, PB (PA));         /*  PB をノード3に生成し
  ...                                 PA へのアクセス権を渡す  */
end.
```

図 3 init 文によるプロセス生成
Fig. 3 Process creation with init statement.

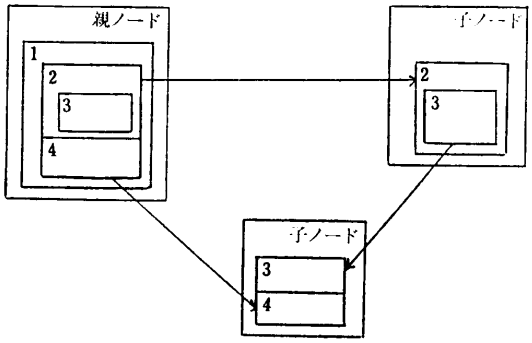


図 4 D-code の分配
Fig. 4 Distribution of D-code.

い.

init 文の実行は次の手順で行われる.

●相手ノードに対して生成するプロセス型のオブジェクトコードがすでにあるかどうかの問合せをする.

●コードがなければそのプロセス型の中に入れ子になっている抽象データ型のオブジェクトコードも含めてコード転送を行う.

●相手ノードに対して指定プロセスの生成を依頼する.

●プロセスにアクセス権を渡す必要があるときは,すでに生成されて動き始めたプロセスとの通信 (implicit に行われる) によってそれを渡す.

2番目の手順で,入れ子になったコードはすべて送るようにしたのは,相手ノードに生成されたプロセスがさらに別のプロセスを生成しようとするときに必要となるコードがすべてそのノードにあることを保証するためである. 図4にコード分配の例を示す.

一度あるノード上に生成されたプロセスはそのノードの固定されたメモリをコードおよび作業領域として占め以後その位置は変更されない. このような静的なプロセス管理はメモリの使用効率という点ではあまり好ましくない. すなわちあるプロセスが終了してそのプロセス用のメモリが不必要になってもそのメモリを再利用できないからである. しかし DPL で記述するものはおもに OS であり, OS の機能の一部を果たすプロセスには不必要になって終了するものはあまりないはずである. それよりはこの静的なプロセス管理による種々の処理の簡略化のメリットのほうが大きいと考えられる. たとえばプロセスに関する情報 (プロセスの通信用ポートの位置等) はすべてプロセス生成時に決定され以後変更されないので, ノード番号とアドレスの組で覚えておくだけでよい. 異なるノード上のプロセスへの通信もこの組の情報を頼りに行われる.

その他にプロセス生成について言及すべきことに,プロセス生成のタイミングの問題がある. Concurrent Pascal では送信プロセスも受信プロセスもモニタの手続きを呼び出すことにより通信を行うので, とくに受信プロセスの場合であるが自分の準備が整う前に通信を受ける心配はなかった. しかし DPL では直接受信プロセス内の手続きを呼び出すため (場合によっては同期をとらずに), 受信プロセスは自分の初期化が終了するまで呼び出しを受けないようにする必要がある. この点は次のように処理した.

● Concurrent Pascal で採用されたアクセス権の手法により, あるプロセスに通信できるのはそのプロセスを生成したかあるいはそのプロセスへのアクセス権をもらったプロセスに限られる.

●プロセスは初期化のための手続きを記述するための initialize 部をもち, これをもつプロセスは生成されるときに initialize 部を実行する. この実行が終了するまでは生成元プロセスの init 文も終了しない.

●したがって init 文の後では, 生成したプロセスに対していつでも通信を送ることができるし, そのプロセスへのアクセス権を他のプロセスに渡しても通信

```
type A=process
  visible procmail GET, PUT end;
  var BUF: array [1..10] of char;
                /* 長さ10のリングバッファ */
      HEAD, LENG: integer;
  procmail GET (C: out char);
        /* バッファから1 char 取り出す */
  begin
    C: =BUF [HEAD];
    LENG: =LENG-1;
    if HEAD=10 then HEAD=1 else
                              HEAD: =HEAD+1
  end;
  procmail PUT (C: in char);
            /* バッファに1 char 詰める */
    var I: integer;
  begin
    I: =HEAD+LENG; if I>10 then I:= I-10;
    BUF [I]:=C;
    LENG: =LENG+1
  end;
  begin
    HEAD: =1; LENG:=0;
    while true do
      select LENG>0: GET; LENG<10: PUT end
    end
  end A;
```

図 5 procmail の記述例
Fig. 5 Example of procmail.

が保証されている.

2.4 プロセス間通信

分散処理におけるプロセス間通信は，モニタ等の共有変数を利用できないため基本的にはメッセージ通信型で行わねばならない．しかし DPL では先に述べた理由により記述形式としては手続き呼出し型を採用し，コンパイラによってこれをメッセージ通信型に展開するようにした．そして Concurrent Pascal におけるモニタは存在意義を失ったため，BrinchHansen が示した DP の考えを取り入れ Edison のように直接通信先プロセスの手続きを呼び出すことにした．このため Concurrent Pascal におけるモニタがもっていたものに代わる新たな通信同期機構が必要となるが，DPL では次の二つの方法を採った.

第一は，手続きと同期処理を分離しプロセスの本体 (body) の select 文によって他のプロセスから呼ばれる手続き (procmail) の同期制御を行うものである．select 文は次の形式をしており

select B_1: PM_1; B_2: PM_2; .. B_i: PM_i .. end;

論理演算式 B_i が真となりかつ PM_i という名の procmail が他のプロセスから呼び出されている選択肢を一つ非決定的に選び出してその PM_i 本体を実行する．条件を満たす選択肢が存在しない場合この select 文を実行しているプロセスは中断され，いずれかの procmail か次に述べる procprocess に呼出しが来た時点で再び選択肢を選び始める．一つの procmail に複数のプロセスが呼出しをかけている場合は最初に呼び出したプロセスのみが通信の対象となり，他のプロセスは再び select 文が実行されてその procmail を含む選択肢が選ばれるまで待たされる．図5に procmail を使用したリングバッファの記述例を挙げる.

第二の方法は，他のプロセスから呼ばれる手続き自体をメッセージを受け取っては返す能動的なプロセスと見なし (procprocess)，必要な場合のみ Edison でも使用された when 文によってこれらのプロセス間の同期を取るものである．これは自分はプロセス変数の値を更新していきながら，他のプロセスにその変数の値を参照させるようなプロセスの記述に非常に便利である．通常 procprocess は他のプロセスからの通信を受け取れる状態で待っているが，そこに呼出しが来るとプロセス本体やそのプロセス内の他の procprocess とは独立して手続きを実行し始める．when 文はプロセス本体や各 procprocess 内に次の形式で記述する.

when B_1: ST_1; B_2: ST_2; .. B_i: ST_i .. end;

ここで各 ST_i は通常の文もしくは複文である．一つのプロセス内では一時に一つの when 文しか実行できないので，それぞれ並行して実行されているプロセス本体や procprocess も when 文を実行しようとするところで同期が取られる．すなわち when 文を実行しようとしたときすでに同一プロセス内の他の部分で when 文が実行されている最中なら，その when の実行が終了あるいは中断されるまで待たされる．when

```
type B=process
  visible procprocess GET, PUT end;
  var MAX: integer;
  procprocess GET (I: out integer);
  begin
    I:=MAX              /* 最大値を返す */
  end;
  procprocess PUT (I: in integer) <limit 10>;
   /* 値を受け取る procprocess を10個作る */
  begin
    when MAX <I: MAX: =I; MAX>=I: end
  end;
  initialize MAX:=0 end;
  begin end
end B;
```

図 6 procprocess の記述例
Fig. 6 Example of procprocess.

```
type A=process
  visible procmail GET, PUT;
        procprocess START, STOP end;
  var BUF: array [1..10] of char;
      HEAD, LENG: integer;
      FLAG: boolean;
  procmail GET (C: out char);
  begin
    C:=BUF [HEAD];
    LENG:=LENG-1;
    if HEAD=10 then HEAD=1 else
                          HEAD: =HEAD+1
  end;
  procmail PUT (C: in char);
    var I: integer;
  begin
    I: =HEAD+LENG; if I>10 then I: =I-10;
    BUF [I]: =C;
    LENG: =LENG+1
  end;
  procprocess START;
  begin
    when true: FLAG: = true end
  end;
  procprocess STOP;
  begin
    when true: FLAG: = false end
  end;
  initialize FLAG:=false end;
  begin
    HEAD: =1; LENG: =0;
    while true do if FLAG then
      select LENG>0: GET; LENG<10: PUT end
  end
end A;
```

図 7 procmail, procprocess の混合使用例
Fig. 7 Example of mixed usage of procmail and procprocess.

文は次の手順で実行される.

- 各 B_i のうち真となるものを非決定的に選ぶ.
- B_i が選ばれたら ST_i を実行し when 文を終了する.
- 真となるものがなければ一時 when 文の実行を中断し, 他の部分での when 文の実行が終了するか同一プロセス内の procmail への呼出しが終了するのを待って再び選び始める.

リミットパラメータにより一つのプロセス内に同じ処理を行う procprocess を複数個作成することも可能であり, 複数のプロセスが同一の procprocess を呼び出したときでもその procprocess の数だけの並列度で実行できる. ただし呼出しの数が上回った場合は後からの呼出しは先の呼出しのどれかが終了するまで待たされる. **図 6** にいくつかのプロセスから送られてきた数のうち最大のものを他のプロセスに送り返すプロセスの例を示す.

procmail と procprocess を比べると後者のほうが記述能力は高い. しかし図 5 の例のように他のプロセスからの呼出しはすべて排他的である必要がある場合も多く, その際 procprocess を用いると when 文が各所に分散され記述が繁雑になるおそれがある. procprocess は, その手続きのなかに他と並行して実行できる部分が多い場合や他とまったく同期を取る必要のない場合に有効となる. **図 7** に一つのプロセスが procmail と procprocess をもつ例を示す. これは図 5 のバッファプロセスを他のプロセスから制御できるようにしたものである.

2.5 DPL コンパイラ

現在, 佐藤・堀によって作成されたコンパイラ (Sequential Pascal で記述) が FACOM M 180 上で動いている. このコンパイラのオブジェクト出力はスタックマシンを対象とした D-code と呼ばれる仮想命令コードであり, DPL 仮想計算機の上で実行される.

3. DPL 仮想計算機 DOVE

DOVE (Distributed Operating Virtual machine Executive) はミニコン PANAFACOM U 1400 上に作成された DPL 仮想計算機である. DOVE は DPL システム内のすべてのノードに実装される必要があるので, とくに移植性と保守性を考えて作成された.

3.1 DOVE の概要

DOVE は機能的には D-code のインタプリタ部とカーネル部に分かれ, さらにカーネル部はプロセス管理やプロセス間通信などプロセス単位の処理を行う上位部とメモリ管理・タイマ管理などハードウェアに依存した処理やネットワークを扱う処理を行う下位部に分かれる. DOVE のプログラムとしての構造は**図 8** のようになるが, この 13 個あるプログラムモジュールが数個ずつに分かれて上の各部の処理を司っている.

各モジュール内の構成要素にはプロセスと SVR がある. プロセスは DPL で記述されたプロセスと同様に他のプロセスと並行して実行されたり何らかの事象待ちとなったりするが, SVR (supervisor routine) は一度には一つしか実行されず一度実行が始まれば他から割込みを受けない. 次のサイクルで処理が進む.

- いくつかのプロセス (DPL で記述されたプロセスを含む) が並行して実行されている.

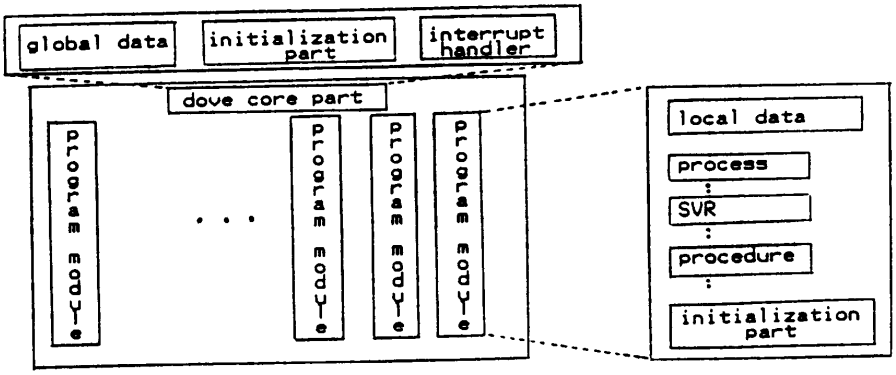


図 8 DOVE のプログラム構造
Fig. 8 Structure of DOVE.

● あるプロセスが SVR に処理依頼 (request) を行うとその SVR が排他的に走り始め，各プロセスは中断される．

● SVR はさらに他の SVR に request を行い，最終的にはモジュール process manager 内の SVR ¥ready に request が行きつく．

● ¥ready は最も優先度が高いプロセスを再起動する．

request は自ノードに限らず他ノードの SVR に対しても行われ，前者を local request 後者を remote request と呼ぶが，ノードにまたがったプロセス間通信は最後には remote request となってネットワークに送信されることになる．

3.2 DOVE によるプロセス間通信処理

DPL の手続き呼出し型プロセス間通信はコンパイラによって送信とそれに対する確認を1組として2組のメッセージ通信に展開される．

呼出し側	procprocess 側
相手の procprocess を呼び出す	呼出しを待つ
	呼出し受付の確認
受付の確認を待つ	
	手続き内の処理
procprocess 終了の合図を待つ	終了の合図を送る
合図受信の確認	受信の確認を待つ

各通信は生成時にプロセスに与えられる交換子 (exchange) によって同期をとって行われる．交換子には各 procprocess procmail につき一つずつ与えられる要求交換子と，必ず一つ与えられる返答交換子がある．通信は，送信プロセスが相手プロセスの交換子に受信プロセスが自分の交換子に通信要求を出し，同一交換子にこの両要求が揃った時点で実際のデータ転送を行う，という方法で同期をとられる．送信要求ばかりが一つの交換子に溜っても一度に受信要求と結ばれるのは早いものから順に一つずつであり，逆もそうである．通信は必ず要求と返答が対になって行われ，それぞれの場合，要求交換子と返答交換子が使われるがその様子を図 9 に示す．

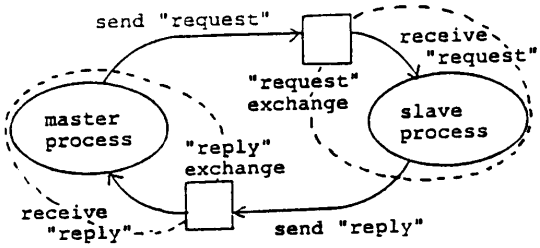


図 9 Exchange を介してのプロセス間通信
Fig. 9 Inter process communication via exchange.

この手法は，手続き呼出しを展開したため呼出し時と呼出し終了時の2回通信が必要なことと，同期をとるのとデータを送るのを別々のタイミングにしたことにより，ノードにまたがる通信が行われた場合にはノード間での実際のデータのやりとり回数が多くなるという欠点をもっている．しかし後者についてはこうしたことにより通信バッファのサイズを小さくできたという利点もある．すなわち通信回数を減らすためには送信要求をするときに送信データも送ってしまう方法もあるが，こうすると相手ノード上で受信プロセスの準備がまだできていない場合は送信プロセスの ID のほかにそのデータ自身も交換子内のキューに溜めておく必要がある．複数のプロセスが同時に送信しようとしたらキューに必要なメモリはかなり大きくなりマ

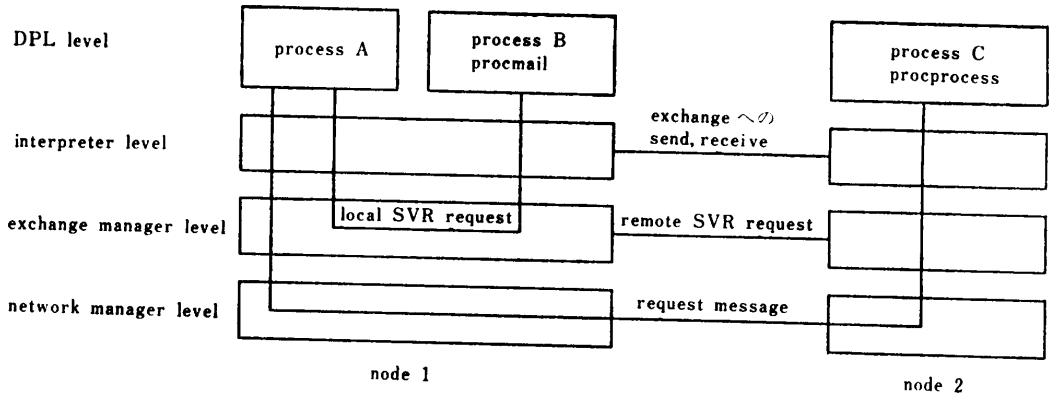


図 10 DPL システムの通信プロトコル
Fig. 10 DPL system protocol.

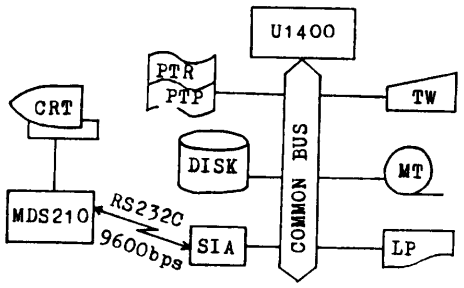


図 11 U 1400 のハードウェア環境
Fig. 11 Hardware environment of U 1400.

イコンクラスのノードにとっては負担になる可能性がある.

DOVE 内では，これらはインタプリタ部，カーネル上位部のモジュール exchange manager，さらにノードにまたがる通信であればカーネル下位部のモジュール network manager へと順にレベルをおとされながら処理されていくが，その各レベルでの通信プロトコルをまとめると**図 10** のようになる.

4. DPL・DOVE の実行テスト

DPL の言語としての記述性のよさ，DOVE の実行効率を評価するために，実際に DPL でプログラムを記述して実行テストを行った.

まず DPL を実装した U 1400 周辺のハードウェア環境を**図 11** に示す. MDS 210 は DOVE から見ると DPL システム内の他のノードであるが，DPL 仮想計算機は未実装である. これには DOVE から受け取ったデータの表示/メモリ上のデータの送信だけの機能をもたせて，DOVE の応答の検証およびノード間通信のシミュレートに利用した.

テストプログラムは紙テープのコピーをターミナルキーボードからの指示によって行うもので，三つのハードウェア制御プロセス（紙テープリーダ，パンチャ，キーボード）・バッファプロセス・バッファモニタプロセスの計五つのプロセスが並行動作する. 実行テストは，このプログラムが走って一定量のテープコピーをしている間に MDS から DOVE 内のプロセスへの通信を繰り返させて，各プロセスの消費時間を計測するという形でなされた. MDS から DOVE 内プロセスへの 1 回の通信（あるコードの有無をモジュール code manager 内のプロセス process creator に問い合わせる）は，MDS 側から 5 回 DOVE 側から 4 回の remote request となってノード間通信が行われ，各 remote request ごとに平均 40 バイトのデータが実際に通信回線を流れる. 通信回数 0 の時と 160 の時（9600 bps の回線での限界）とで，通信を司るプロセスの消費時間と CPU 割込み待ちをしている時間の全体に対する割合は次の表のようになる.

通信回数	処理時間 (sec)/%		
	total	network	waiting
0	50.9/100	0/0	24.9/49
160	51.5/100	4.1/8	20.6/40

通信回数が増すとその処理にさかれる時間が増す代りに割込み待ちの時間が減って，全体では 1% 強しか処理時間がふえていない. これは，DOVE のプロセススケジュールがうまく働き，回線からの通信割込みに対する処理等は CPU の空き時間に吸収されていることをしめしている. この程度の回線を使用するシステムなら前節で述べた通信回数増大については心配い

らないことがわかる．

またこのテストプログラムは各種ハードウェア制御を含んでいるにもかかわらず総行数は230行と短い．さらにこのコーディングに用いた時間は半日以下でありデバッグも3回の再コンパイルで終了した．これはDPL の記述性のよさ，信頼性の高さを示している．

5. おわりに

以上，高級言語で記述された分散処理システムを実現することを大きな目標として，

- 分散処理システム記述用言語 DPL の設計
- DPL 仮想計算機 DOVE の作成
- DPL と DOVE の実行テスト

を行ってきた．現在はまだ単一計算機上で論理的に並行処理を行うシステムをテストした段階であるが，その有用性は明らかである．今後複数ノードにまたがる大きな分散処理システムを作成するには，

- 他の計算機上にも DOVE を移植する．
- 各計算機を結ぶ適切なネットワークを作成する．
- より下位レイヤの通信プロトコルを整備する．

といった課題が残されている．

謝辞 本研究に関し多大なご援助・ご討論をいただいた研究室の皆様に深く感謝します．

参考文献

1) BrinchHansen, P.: Distributed Processes: A Concurrent Programming Concept, *CACM*, Vol. 21, No. 11, pp. 934-941 (1978).
2) BrinchHansen, P.: Edison—A Multiprocessor Language, *Softw. Pract. Exper.*, Vol. 11, No. 4, pp. 325-361 (1981).
3) Dijkstra, E.W.: *Co-operating Sequential Processes*, Academic Press, New York (1968).
4) United States Department of Defense: Reference Manual for the Ada Programming Language (July, 1980).
5) Hoare, C. A. R.: An Operating System Structuring Concept, *CACM*, Vol. 17, No. 10, pp. 549-557 (1974).
6) Hoare, C. A. R.: Communicating Sequential Processes, *CACM*, Vol. 21, No. 8, pp. 666-677 (1978).
7) INMOS Limited: occam プログラミングマニュアル，啓学出版，東京 (1984).
8) 佐藤文一：分散処理システム記述用言語に関する研究，東京大学修士論文 (1980).
9) 堀 健一，佐藤文一，浜田 喬：分散処理システム記述用言語 DPL とその処理系概要，電子通信学会昭和56年度全国大会，p. 1481 (1981).

(昭和58年 9 月 2 日受付)
(昭和59年11月15日採録)