高級言語指向分散処理システムの構成法

半田剣一*・浜田 喬
（東京大学生産技術研究所）

1．はじめに

近年、分散処理システムは計算機システム技術の重要課題として認識が深まっているが、その背景にはハードウェア技術の進歩とユーザからの要求がある。

この動きへのソフトウェア側からの対処はというと、多少出遅れた感はあるが、単一計算機内での並行処理の手法を元として発展した。ソフトウェア危機に対する一つの解答は高級言語の採用であったが、複雑な並行処理においては実行時のテストが非常に困難であり、高級言語の使用が記述性はもちろん信頼性の面からも必須と言える。そして R.W. Dijkstra のセマフォ (1968)[i]、C.A.R. Hoare のモニタ (1973)[ii] などの概念を経て P. Brinch Hansen の Concurrent Pascal (1975)[iii] や N. Wirth の Modula (1977)[iv] などの並行処理記述用高級言語が実用化されてきた。Brinch Hansen による Solo システム (1977)[v] は高級言語で記述された最初の大きな並行処理システムと言えよう。

これらの研究が分散処理に進むには、共有メモリを持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。Hoare は C.S.P. (Communicating Sequential Processes－1978)[vi] 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を提案し、OCCAM (1982) という言語に活かしている。Brinch Hansen はモニタとプロセスの概念を結合した D.P. (Distributed Processes－1978)[vii] を提案し、それに基づいた Edison (1981) を発表した。その他にも Ada，*HOD、CP、PLITZ など分散処理の記述能力がある高級言語が続々と発表されてきている。

我々の研究室においても、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 DPL の設計[viii] とそのコンパイラの作成[ix] を行なってきた。今回実際の計算機上に DPL を実装し、簡単な実行テストを行なったので、その経過を報告する。

2．分散処理記述用言語 DPL

DPL (Distributed Processing Language) はプロセスを単位とした分散処理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、信頼性・記述性に富んだ高級言語となっている。

DPL で分散処理システムを記述する際は、システム内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 Fig.1 のように２つの階層に分割しているためである。上位レベルは DPL プログラム自身であり、下位レベルは各ノード上でプログラムの実行をサポートとする仮想計算機である。

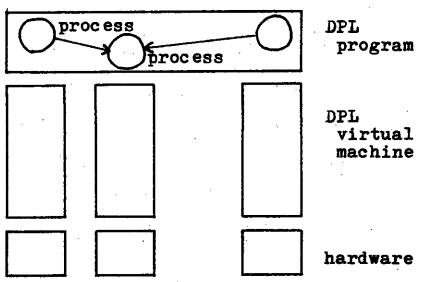


Fig.1 DPL system architecture

* 現在電総研

ＤＰＬはConcurrent Pascalの言語的特徴を多く継承しているが、その主な点を以下に示す。

- プログラムに抽象データ型の概念を取り入れており、プロセス・クラスは型として定義される。
- 各プロセスは初期プロセスを用いて、親プロセスから動的に生成され、生成された後は他のプロセスと並行した処理を始める。
- プロセス間の通信は、生成時に親プロセスから譲り受けた情報（アクセス権）に従って手続き呼出し型で行なわれる。
- 各抽象データ型は入れ子構造が許され、名前の有効範囲規則はConcurrent Pascalのそれに従う。

そして新しい特徴としては以下がある。

- プロセス間の通信は、モニタを介する事なく直接他のプロセス内の手続きを呼び出す事によってなされる。（Brinch HansenのD.P.に類似している）
- 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Brinch Hansenによるguarded regionを採用している。
- 例外処理、誤り処理を記述できる。（Adaの影響を受けている）
- ＤＰＬ仮想計算機とのインタフェースを定義でき、高級言語では記述が難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に任せられる。
- 変数を特定のアドレスに割り付ける事ができ、上記の機能と合わせる事で、周辺装置の制御等はＤＰＬで直接記述できる。

以降、ＤＰＬの文法について簡単に説明する。

(1) プログラムの構成

ＤＰＬのプログラムは

- definition module 部
- 宣言部
- 初期プロセスの本体

からなる（Fig.2）。

```
definition module DOORS;
  const NUL=(:0:); CR=(:13:);
  procedure LAMP(KIND:integer);
end DOORS;

type PROC_A=process
  type CLASS_A=class
    ----
    begin --- end
  end CLASS_A;

  begin --- end
end PROC_A;

var PROCESSA:PROC_A;

begin
  init PROCESSA; ----
end.
```

Fig.2 example of DPL

definition module 部では、プログラム全体に共通な定数・型・例外名を定義できる他に、仮想計算機内のルーチンとのインタフェースを定義できる。

宣言部では、型宣言（抽象データ型を含む）、定数・変数宣言（初期プロセス用）を行なう。抽象データ型にはクラスとプロセスがあるが、ともにConcurrent Pascalのそれと同じ機能を持つ。プロセス・クラスともに入れ子構造が可能である。

初期プロセスは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言う。ＤＰＬシステムは1つの親ノードと複数の子ノードで構成される。

(2) プロセスの生成

Fig.3 のような init 文を実行する事により、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 へのアクセス権が渡される。

```
const NODE=3;
var PA:PROCESSA;
    PB:PROCESSB;
begin
  init PA,(NODE,PB(PA));
  ----
end;
```

Fig.3 init statement

(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 及び procprocess を定義できる（Fig.4）。

```
type PROCESSA=process
  visible procmail PM1,PM2;
          procprocess PP1 end;

  procmail PM1;
    -----
  procmsil PM2;
    -----
  procprocess PP1;
    -----

  begin
    select FLAG1:PM1; FLAG2:PM2 end;
    ----
  end
end PROCESSA;

type PROCESSB=process(PA:PROCESSA)

  begin
    PA.PM1;
    PA.PP1;
    ----
  end
end PROCESSB;

var PA:PROCESSA; PB:PROCESSB;
begin
  init PA,PB(PA)
end.
```

Fig.4 interprocess communication

1つのプロセス内の procmail はそれぞれ排他的に実行される。プロセスは自分の持つ procmail に対して複数プロセスから呼び出しを受けても、一度には1つのプロセスにのみその実行を許し、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 を持つプロセスは select 文を実行する事により、真の条件部に対応する procmail を呼び出しているプロセスのうちの1つのみと通信を行ない procmail の実行を許す。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 は最大呼び出したプロセスの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部分に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 文の中に入ったプロセスは条件部が真の文を1つだけ非決定的に選択してそれを実行する。

(4) 例外・誤り処理

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起こすものと、プログラム中で assert 文や raise 文により強制的に起こせるものがある。

```
EXCEPTION EX1,EX2;
BEGIN
  EXCEPTION UPON EX1 DO ---- ;
            UPON EX2 DO ---- ;
  END;
  ---
  STATEMENT_LIST
  ---
END
```

(5) その他

- 変数を任意のアドレスに割り付ける事ができる。

```
var A:array(.1..5.) of real at #200;
```

上の例は配列Aを 200(16進)番地からの連続した領域に割り付ける。

- 関数や手続きの仮引数の属性には in(参照のみ)、in out(更新)、out(新たに設定) の3種がある。

```
procedure P(X:in real; Y:out real);
```

以上がDPLの特徴的なところである。現在、佐藤・堀によって作成されたDPLコンパイラが東大生研共用計算機 HIPO(富士通) 上で動いている。このコンパイラはD-codeと呼ばれるコードを出力するが、これは Concurrent Pascal コンパイラが出力する Con-rede に類似した仮想命令コードである。下にD-codeの一部の例を示す。

```
LOCALADDR(18)
LOCALADDR(-4)
COPYWORD
INITREMOTE(#50)
PUSHCONST(4)
LOCALADDR(14)
PUSHCONST(4)
LOCALADDR(10)
```

Fig.5 example of D-code

4. DPL仮想計算機 dove

dove (distributed operating virtual machine executive) はミニコン U1400 (PANAFACOM) 上に作成された親ノード用のDPL仮想計算機である。dove はアセンブリ言語で記述された 13 Kバイト強のプログラムであり、その設計に際しては次の点に留意した。

- 他の計算機上にも仮想計算機を作成する事を考えて移植性の高いものにする。
- 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・縮少を簡単に行なえるようモジュール化を徹底する。

dove のプログラムとしての構造は Fig.6 のようになっている。

- プログラムモジュール群

  処理の対象となるデータによって13個に分割されており、それぞれが3種の構成要素(プロセス・スーパバイザルーチン・プロシジャ)を数個づつ含んでいる。

- コア部

  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。

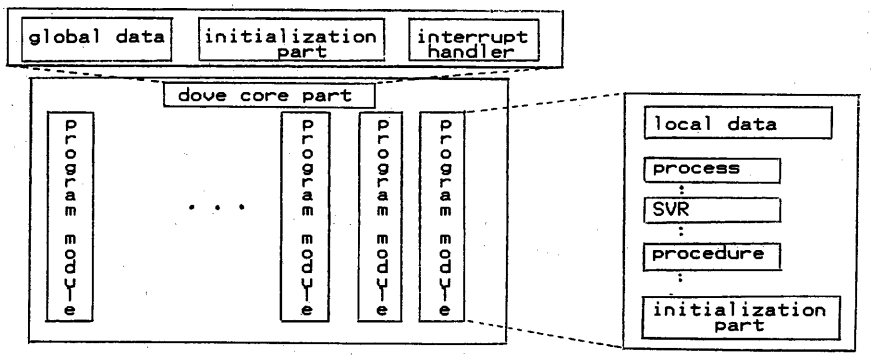


Fig.6 structure of dove

3種の構成要素は実行の形式によって次のように区別されている。

- プロセス
  他のプロセスと並行して実行できる処理、実行中に何らかの事象待ちが必要な処理はプロセスとして記述されている。doveが持つ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）と同様の処理をうける。
- 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度に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）を受けて排他的に走り始める。自ノードのSVRのみではなく、他のノードのSVRに対してもrequestを行なえる。前者をlocal SVR request、後者をremote SVR requestと呼ぶ。
- プロシジャ
  他の構成要素から呼び出され、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Fig.7のよう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。以下に各機能ごとのプログラムモジュールの処理概要を示す。

interpreter
kernel high level
kernel low level

Fig.7 function of dove

(1) インタプリタ機能

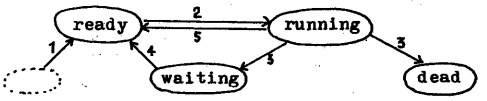
D-codeを実行する。ただしプロセスにまたがった処理やハードウェアを意識した処理は下の機能に任せる。次のプログラムモジュールが処理にあたる。

- interpreter
  システムプロセスが起動される際には目的のプロシジャに制御が移りD-codeが解釈実行される。CPM（Concurrent Pascal Machine）のインタプリタと同じ手法が使われている。
- system-service-routines
  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interpreterから呼出される。

(2) カーネル上位機能

プロセス生成、プロセス間通信等まだプロセスという概念を意識した上での種々の処理を提供する。

- process_manager
  プロセスの生成、中断、起動などの処理を行なう。これらに伴うプロセスの状態変遷をFig.8に示す。図の右側はそれぞれの処理を行なう構成要素名である。カーネルプロセスであるcreatorはシステムプロセスから新たなプロセス生成の通信を受けてはプロセスの生成登録を行なってその情報を返信する。



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

Fig.8 process status

◦ code-manager

DPLシステムでは、D-codeがまず親ノードにロードされ、その後各子ノードにプロセスが生成されていく毎に、必要なD-codeが転送されていく。カーネルプロセス code-receiver はシステムプロセスと通信を行ないながら Fig.9 に示すD-code分配の処理を行なう。D-codeは入れ子になった抽象データ型単位で転送される。

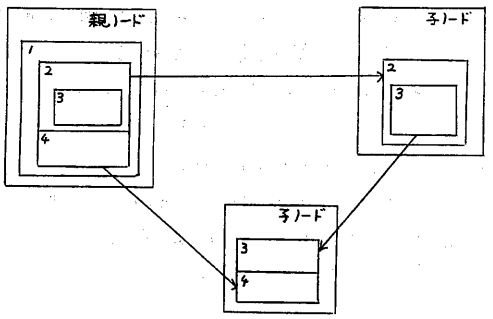
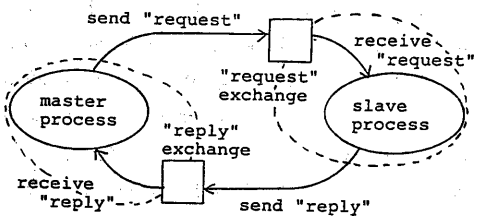


Fig.9 distribution of D-code

◦ exchange-manager

Fig.4 に示した procmail や procprocess の呼び出しは、コンパイラによって、相手プロセスの持つ exchange への send 要求と自分の exchange への receive 要求に分解される(Fig.10)。




```
PUSHCONST(4)                    PUSHCONST(4)
LOCALADDR(14)                   GLOBADDR(6)
PUSHCONST(4)                    PUSHCONST(4)
LOCALADDR(10)                   GLOBADDR(2)
PUSHLOCAL(18)                   MYPLACE
PUSHNAME                        GLOBADDR(-28)
FIELD(-28)                      PUSHCONST(2)
PUSHCONST(2)                    RECEIVE
SEND ------------------------>  POP(14)
 POP(14)                        GLOBADDR(2)
 LOCALADDR(10)                  PUSHNAME
 PUSHNAME                       PUSHCONST(0)
 PUSHCONST(0)                   SEND
RECEIVE <----------------------  POP(6)
 POP(6)
```

Fig.10 communication by exchange

exchange に対するこれらの処理を行なうものとして3つのSVR(¥send, ¥receive, ¥communicate)がある。それぞれ、send要求の処理、receive要求の処理、同一exchangeに対するsend要求とreceive要求がマッチした時のデータ転送、を担当する。Fig.11-1は同一ノード内の、Fig.11-2は別々のノードにあるプロセス同士が通信を行なう際のSVR requestの流れを示している。両者で異なる点は、各requestがlocalであるかremoteであるかのみである。さらに、同期のみを目的とした通信の場合メッセージ数は0なのでFig.11-3のようにして効率向上をはかっている。

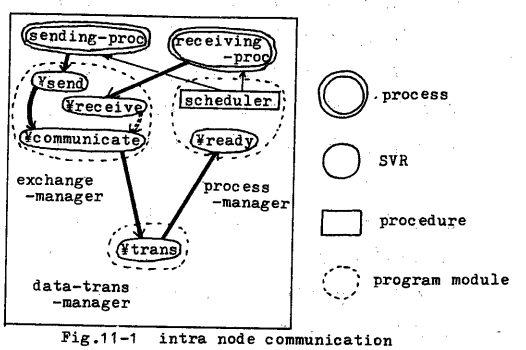


Fig.11-1 intra node communication

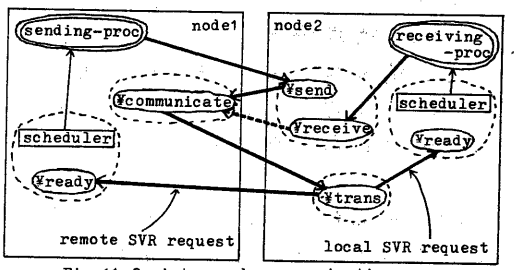


Fig.11-2 inter node communication

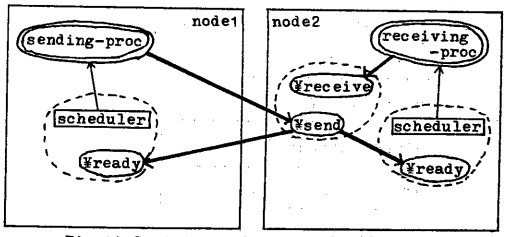


Fig.11-3 inter node communication
(message = 0)

◦ gate-manager

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当てられ、そのプロセス内のprocprocessを呼び出してwhen文に出っかったプロセスは一度に一つづ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。

(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なう。

◦ request-manager

SVR requestが発せられて目的SVRに制御が移るまでの処理を行なう。各構成要素は、メモリ上にFig.12の形式のrequest messageを作成してプロシジャrequest-handlerに制御を移す。request-handlerは、local SVR requestであれば直接目的SVRを呼び出し、remote SVR requestであればrequest messageをrequest queueにつなぐ。その後プロセスからのrequestであればschedulerに、SVRからのrequestであればそのSVRに制御を戻す。

◦ network-manager

2つのネットワークプロセスsenderreceiverを持ち、実際にノード間のデータ通信を行なう。それぞれの処理内容は

- sender: 前述のrequest queueからrequest messageを1つ取り出し、それにindirect parameterを付加して(modified request message)ネットワークへ送り出す。
- receiver: modified request messageを受信して、その内容をもとに今度はlocal SVR requestを行なう。

このようにノード間でやりとりされるデータはすべてが独立したrequest messageに統一されている。remote SVR requestが行なわれた時の制御の流れをFig.13にまとめる。

◦ memory-manager

doveのメモリ管理を行なう。現在のインプリメントでは64kbyteの連続した領域を扱える。

◦ timer-manager

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ロセス切換を行なっている。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32msecに設定してある。

以上が各プログラムモジュールの処理

SVR node
SVR index
direct param bytes
direct parameters
indirect param counts
param1 address bytes
param2 address bytes
⋮

Fig.12 request message

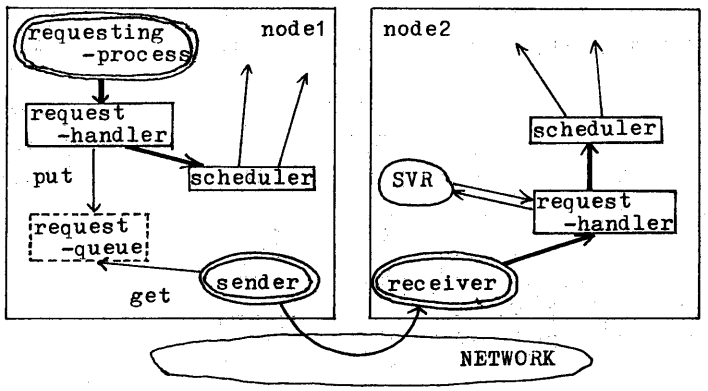


Fig.13 remote SVR request

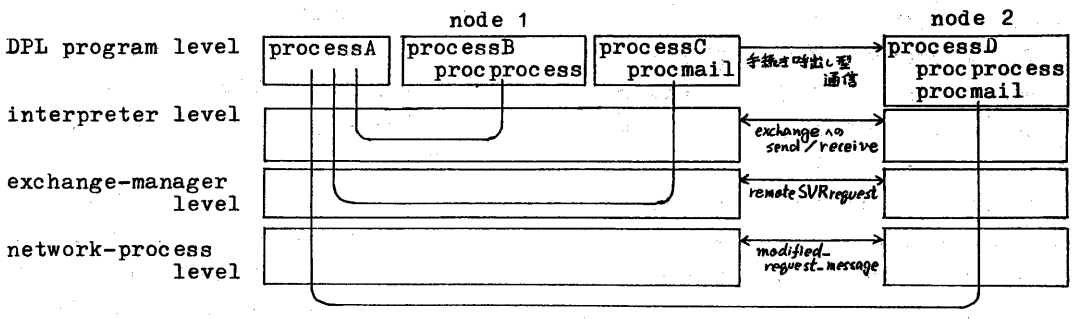


Fig.14 DPL system protocol

概要である。

前述のようにDPLプログラムにおける手続き呼出し型のプロセス間通信は、最終的にはmodified request messageとなってネットワークへ送出される。Fig.14にDPLシステムの通信プロトコル階層をまとめた。前の説明では触れたが、効率を上げるために同一ノード内のprocprocess呼び出しについては、exchangeを介さないで呼び出し元プロセスが直接procprocessを実行するようになっている。

4. DPL・doveの実行テスト

まずDPLを実装したU1400周辺のハードウェア環境をFig.15に示す。ネットワークプロセスはSIO(Serial Interface Adapter)を介してデータの入出力を行なう。インテルのマイコンMDS210は、doveから見れば他のノードであるが、実際は、受け取ったデータを表示する／キーボードからの指示とメモリ上のデータを送信する、というプログラムが走っているのみである。

テスト用に作成したプログラムは紙テープのコピーを行なうもので、5つのプロセスが並行処理を行なっている(Fig.16)。実行中任意の時点でコピーの中断、再開、終了の指示を行なえる。Fig.17にptrプロセスの記述部分

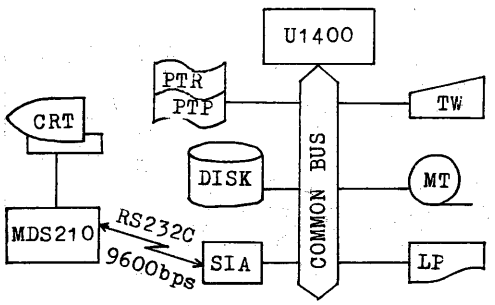


Fig.15 hardware environment

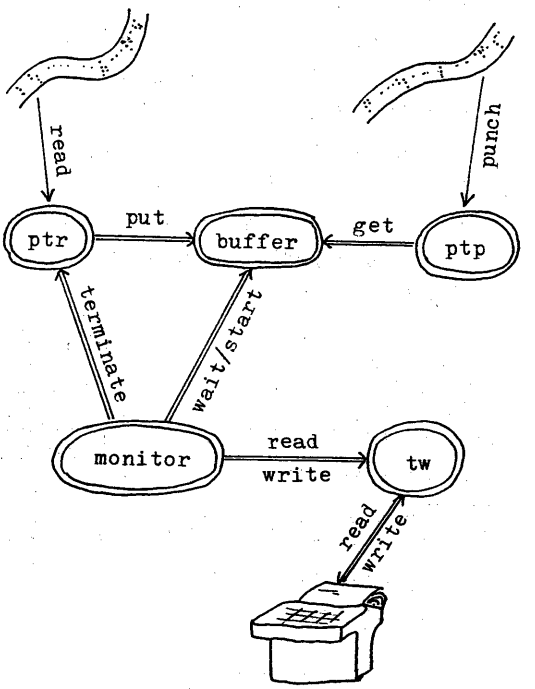


Fig.16 5 processes of doors8

```
TYPE PTR_PROCESS=PROCESS(BUF:BUFFER_PROCESS)
  VISIBLE PROCPROCESS TERM END;
  CONST PTR_PUNO=#0220;
        READ_CMD=#8000;
        BUSY=0; OPCALL=1;
  VAR   DSR:INTEGER AT #0220;
        BFR:INTEGER AT #0224;
        CMR:INTEGER AT #0226;
        TERMINATION:BOOLEAN; C:CHAR;

  PROCEDURE READ(C:OUT CHAR);
    VAR I:INTEGER;
  BEGIN
    WHILE TEST_BIT(BUSY,DSR) DO;
    CMR:=READ_CMD;
    WHEN NOT TEST_BIT(BUSY,DSR): I:=BFR END;
    IF I>=0 THEN C:=CHR(I DIV 256)
            ELSE C:=CHR(I DIV 256 + 128)
  END;

  PROCPROCESS TERM;
  BEGIN
    TERMINATION:=TRUE
  END;

  BEGIN
    GATE_TRIGGER_SET(PTR_PUNO);
    TERMINATION:=FALSE;
    REPEAT
      READ(C);
      BUF.PUT(C)
    UNTIL (TERMINATION=TRUE) OR (C=EM);
    IF C<>EM THEN BUF.PUT(EM)
  END
END PTR_PROCESS;
```

Fig.17 ptr_process

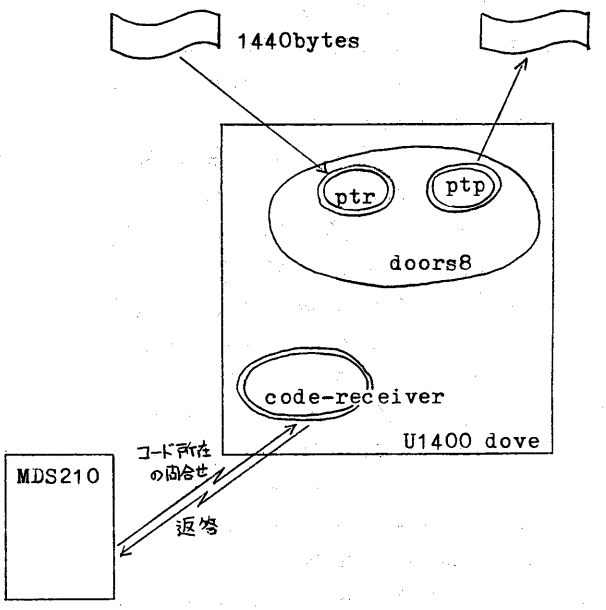


Fig.18 DPL,dove execution test

を例としてあげる。

procedure read は直接紙テープリーダのハードを制御して一文字づつデータを読み込む。

procprocess term は monitor プロセスから呼び出され、フラッグ termination を真にする。

ptr プロセスの本体は emコードを読み込むか termination が真になるまで、一文字読み込んではbuffer プロセスにそれを送るという動作を繰返す。

このテストプログラム(doors 8)の正常動作を確認した後、dove の性能評価のために Fig.18 のようなテストを行なった。すなわち MDS 210 のメモリ上に、code-receiver (dove のカーネルプロセス) に対してある単位の D-code の所在を問合わせるための request_message 列を作成する。そして doors 8 が 1440 byte のコピーを行なっている間に MDS 側から remote SVR request を行ない、コピー終了時での各プロセスの消費時間を測定した。問合せと返答で1サイクルの通信(この両側から5計9回の remote SVR request が行なわれる)として、コピーや 0～160 サイクルの通信を行なった結果を Fig.19 に示す。

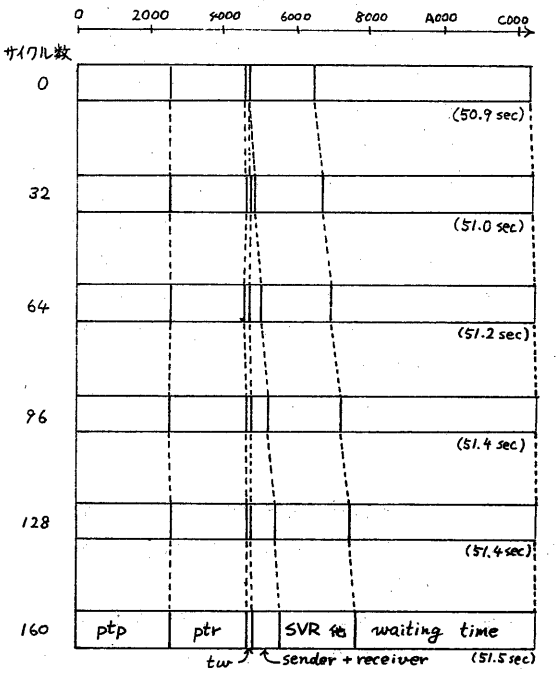


Fig.19 running time of processes

この結果からは次の事が言える。

- 通信サイクル数が増加するにつれ両ネットワークプロセスの消費時間は確実に増加していっているが、トータルの消費時間はほとんど増加していない。これは、外からの通信割込みの処理がうまくCPUの空き時間内に吸収されている事を示している。
- 紙テープパンチャの処理速度は50 byte/sec であり、1440 byte のパンチに最低でも29秒かかる。doorsBの処理内容から考えるとトータルで50秒強という処理時間は許容範囲内であろう。

すなわちdoveの実行効率はこのままでもある程度実用になると考えられる。

またDPLの記述性については次の事が言える。

- doorsBは全部で230行のプログラムである。その機能（ハードウェアの制御も含む）を考えると、アセンブリ言語で記述した場合に較べて極めて短い。DPLの記述性の良さの証しである。
- doorsBのコーディングとデバッグ合せて半日以下であった。この間文法エラーのためと実行時エラー（紙テープ制御法についての誤り）のために3回コンパイルしなおし、その後は正常に動作している。これはDPLで記述する事の効率の良さと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事を大きな目標として、

- 親ノード用のDPL仮想計算機doveの作成
- DPLおよびdoveの性能を評価するのに有効な実行テスト

を行なってきた。現在はまだ、単一計算機上で論理的に並行処理を行なうプログラムをテストした段階であるが、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ために

- 他の計算機上にもDPL仮想計算機を移植する。
- それらの計算機を結ぶネットワークを構成する。
- より下位レイヤの通信プロトコルを規定する。

といった課題を解決していく必要がある。

参考文献

i) E.W. Dijkstra, "Co-operating Sequential Processes" programming languages, edited by F.Genuys, pp.43-112 Academic Press 1968

ii) C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", CACM Vol.17 No.10 pp.549-557 Oct.1974

iii) P. Brinch Hansen, "The Program Language Concurrent Pascal", IEEE Trans. SE-1 No.2 pp.199-207 Jun.1975

iv) N. Wirth, "Modula: A Language for Modular Multi-programming", SOFTWARE-PRACTICE AND EXPERIENCE Vol.7 No.1 pp.3-35 1977

v) P. Brinch Hansen, "The Architecture of Concurrent Programs", Prentice Hall 1977

vi) C.A.R. Hoare, "Communicating Sequential Processes" CACM Vol.21 No.8 pp.666-677 Aug.1978

vii) P. Brinch Hansen, "Distributed Process: A Concurrent Programming Concept", CACM Vol.21 No.11 pp.934-941 Nov.1978

viii) 佐藤, "分散処理システム記述用言語に関する研究", 修士論文 1980

ix) 堀・佐藤・渋田, "分散処理システム記述用言語DPLとその処理系概要", 電子通信学会全国大会 1981