ソフトウェア基礎論19－9
プログラミング言語 9－9
（1986. 12. 12）

ＴＡＯによる並列問題解決プログラミング

Parallel-Problem-Solving Programming in TAO

尾内理紀夫　竹内郁雄
Rikio Onai　Ikuo Takeuchi

（ＮＴＴ電気通信研究所）
NTT Electrical Communications Laboratories

あらまし　solver間の通信を許す動的manager-solvers階層を基本とする並列協調型問題解決モデルを提示するとともに、複合パラダイム言語ＴＡＯの並列・分散プロセス間通信・同期機能について述べる。また、それらＴＡＯの通信・同期機能による同報通信、同期通信をはじめとするいくつかの既存の通信・同期法および多重レベル通信の実現を示す。これは、ＴＡＯの通信・同期機能の強力さの提示であると同時に、プログラミングに際しプログラマの好みによりパラダイムを選択できるという、複合パラダイム言語ならではの有用性の提示でもある。最後に、ブロック移動問題を例に取り、manager-solvers階層モデルによる並列協調型問題解決プログラミングを知能処理用ワークステーションＥＬＩＳ上のＴＡＯにより試みる。

Abstract　We firstly present parallel-coordinated problem solving model based on dynamic manager-solver hierarchy which allows communications among solvers, then describe mechanisms of communication among parallel-distributed processes by a multiple paradigm language TAO. Then, we implement some existing communication methods, such as broadcast and synchronized communication etc., and multi-level communication using TAO mechanisms. This implementation shows not only power of the TAO communication mechanisms but also such usefulness of multiple paradigm language that programmers can select their favorite paradigms on programming. We also try parallel-coordinated problem solving for moving blocks in TAO on AI workstation ELIS.

１．はじめに

近年の素子技術の発展はここで改めて述べるまでもなく極めて急速である。この結果、マイクロコンピュータはより高機能に安価に、メモリはより小さく安価になってきたし、これからもその傾向は続く。また、ネットワークも発展を続けている。このため、この世の多くのものに、高機能なマイクロコンピュータとある程度の容量のメモリとネットワークインターフェースを安価に具備させることが可能になってきた。即ち、飛行機や船だけでなく、自動車にもエレーベータにも、そしてロボットにも通信機能付きのマイクロコンピュータが搭載されることがあたり前の時代がくるだろう。いや、そのような時代を迎えるためには、各コンピュータが自律的に通信を行い、互いの知識を交換し、協力しながら、全体として問題を解決する技術を確立しなければならない。

この技術は、広い分野に関連している。多数のコンピュータが頻繁に通信を行なえば、ネットワークが過負荷になってしまう。よって、各コンピュータは、なるべくローカルな知識を用いて各自の担務である問題を解決しなければならない。ただ、常にローカルな知識のみで問題が解決できる訳ではない。そうだとするならば、グローバルな知識をどこに置き、どのようにアクセスするかという課題を解決しなければならない。また、各コンピュータは知識を交換するが、これは別の見方をすれば、コンピュータが他のコンピュータから知識を獲得する。即ち、学習するということであり、これまた、知識の衝突あるいは知識の非単調増加をどう処理するかという重要かつ未開拓な課題である。

他には、複数のコンピュータが一つの大きな問題を解決する際に、全体として、解空間が狭まっていくことの保証をどう与えるかという課題もある。

本研究はこのような課題を含む並列協調型問題解決システム構築のため第一歩である。 本報告では、並列協調型問題解決モデルと、並列協調型問題解決に必要な複合パラダイム言語TAO(オブジェクト指向＋シンボル操作＋論理型＋数値計算)[Take 83][Suzu 85] の各種通信同期機能を提示する。そして、その各種通信同期機能によるいくつかの既存通信手法 (ex. 1対1通信、1対多通信、無限長/有限長バッファ通信、Now型/Past型/Future型通信、同期/非同期通信etc.)と多重レベル通信の実現を示す。 また、ブロック移動問題を例にとり、TAOによる並列協調型問題解決プログラミングについて述べる。

2. 並列協調型問題解決モデル

これまでにもいくつかの並列協調型問題解決モデルが提案されてきた [Film 84]。しかし、黒板モデルといわれる、共有メモリを前提とする方式以外にはハードウェアを意識したシステムは少ない。

黒板モデルは、黒板への書き込み、読み出しを実行する存在が、数個から十数個の分野には適しているかもしれないが、それ以上の個数の分野には、黒板へのアクセスが衝突するため不向きとなってしまう。

本研究の目ざすところは、極めて多数の存在が有機的に結合され、通信し合い、全体として処理が円滑に進んでいくシステムを構築することであるから、黒板一枚という訳にはいかない。それでは、黒板を複数にできるかというと、黒板モデルの黒板は受け身的な存在であり、自ら積極的に他に働きかけることはできない。そのため、黒板をノードした階層構造を構成することは好ましくない。

本モデルではすべての存在は能動的であり、CPU処理速度の差、メモリ容量の差はあるが、質的には同一の知的な存在(Intelligent Agent, 以後略してIA)である。 このIAはある時には、managerとなりその配下にあるIA(solvers)に対して問題を付与しその解決進行を管理するが、ある時にはsolverとなり他のIA(manager) から送られてきた問題をできるかぎりローカルな知識で解決しようとする。

即ち、問題解決モデルとしてmanagerあるいはsolverの役割及び関係の動的な変更を許す階層モデルを考える。これを図1に示す。ただし図1は単にある局面でのmanagerとsolverの関係を示すだけであり、solver間の通信も可能であるし、manager-solver関係の動的逆転も可能である。よって、本モデルの実現アーキテクチャが木構造という訳ではない。

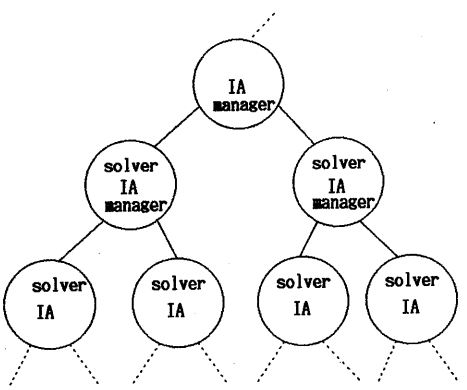

図1 manager-solvers階層

本モデルの実現にあたってはmanager-solver間通信を極力抑制するように、即ち、solverはできるかぎりローカルな知識で問題解決をはかるように問題解決戦略を構築していくので、問題によってはモデルの部分的な物理アーキテクチャとして各IAを共有バスで連結したものも可能である。

具体的なネットワーク・アーキテクチャをどうするかということは、本モデルを適用する分野に依存することである。逆に、どんな応用分野に対しても適したネットワーク・アーキテクチャというものも存在しないと考える。ただ、ここで重要なことは、ネットワーク・トラフィックを極力低減すること、全体として解を求める方向にシステム全体が向っていることを保証することである。

本研究では、一つのIAを一台の知能処理用ワークステーションELIS [Yama 86]と考える。現時点において、ELISは、自動車やロボットに搭載するには大きすぎるが、ハードウェア技術の進歩を考えれば、ごく近い将来には自動車はおろか、電子レンジや洗濯機に組み込むこともハードウェア的には可能になるだろう。また、問題解決戦略、通信指示をはじめとするソフトウェア記述言語としては、複合パラダイム言語TAOをその出発点とする。

3. TAO通信・同期機能

TAOは、Lisp、Prologオブジェクト指向機能等を調和平均的に融合させたプログラミング言語である。調和平均的という意味は、単にLisp、Prolog、オブジェクト指向機能をとり入れたというのではなく、Lisp関数中で論理変数が使用できたり、逆にPrologのリテラルとしてLisp関数が使用できたり、メソッドを実行中のオブジェクトに対してthrowによって割込みをかけることができるなど、各種言語機能が有機的に結合しているということである。

このように TAOは各種パラダイムの自然な融合を目指しており特殊なプリミティブを新たに導入することなくLispパラダイム内に論理型パラダイムを埋めこむことも、その逆も可能である。 両方の場合をプログラム例により示す。

```
•Logic in Lispの例
(defun logic-in-lisp
      (&aux _x _y x)          ; _を付加すると論理変数
      (== _x _y)              ; ==はunification
      (!x 1)                  ; !は代入
      (== _x  ,x)             ; ,は項を評価
      (write _y))             ; 1が出力される

•Lisp in Logicの例
 (assert (lisp-in-logic)   ;Logic, head predicate
         (== _x 1)         ;first body predicate
         (let (x)    ;Lisp,second body predicate
            (!x _x)
            (write x)))    ;1が出力される
```

本章では、TAOの機能のうち、並列協調型問題解決に必要な並列・分散プロセス間の 通信・同期に関する機能について述べる。 TAOにおいて、プロセスはシステム定義クラスprocessのインスタンスである。そして、本並列協調型問題解決モデルのIAへの割付け単位をプロセスとし、特にwork-processと呼ぶ。もちろん、1つのwork-processの中で動的にプロセスを生成することは可能である。

3.1 メイルボックス

TAOの通信機構として、クラスmailboxがサポートされている。mailboxによる通信とは、荒っぽく言えば、普通の家庭にある郵便箱による通信と同じである。

インスタンス変数としては次のものがある。

- sys:mailbox-process-queue

メイル（郵便物）を待っているプロセス（たとえて言えば、郵便箱に新聞をとりに行ったが、まだきていないので待っている人)は、このsys:mailbox-process-queueという待ち行列につながれる。

- sys:mail-queue

プロセスにより受理されるのを待っているメイル（たとえて言えば郵便箱の中に入れられた手紙)は、このsys:mail-queueという待ち行列につながれる。

関数としては次のものがある。

- (receive-mail メイルボックス)

receive-mailにより、メイルボックスに送られてくるメイルを取り出す。 receive-mailは該当メイルボックスにメイルが送られてくるまで値は返ってこない。すなわち、受信プロセスはメイルを受け取るまで待ち続ける。

なお、これに等価なメソッドも用意されている。 この場合には［メイルボックス receive-mail］となる。 即ち、［……］はオブジェクトへのメッセージ・パッシングであり、［ のすぐ右がオブジェクト、続いてメッセージ・セレクタと引数である。

- (receive-mail-with-timeout n メイルボックス)

receive-mail-with-timeoutは、メイルボックスよりメイルを取り出す際に、n*(1/60)秒間（これをtimeoutという）しか待たない。 そして、その間にメイルが送られてこなければnilを返す。

- (send-mail メイルボックス　メイル)

メイルボックスに対しメイルを送る。

mailboxオブジェクトは、いわば、生産者プロセスと消費者プロセスの間の無限長バッファである。生産者プロセスは、send-mailにより sys:mail-queueにメイルを送り込み、消費者プロセスは、各種receive-mailによりそこからメイルを取り出す。送り込みと取り出しは非同期であり、生産者プロセス、消費者プロセスの数は１つでも複数でもよい。よって１プロセス対１プロセス、多プロセス対１プロセス、１プロセス対多プロセス、多プロセス対多プロセスの非同期かつ非決定的通信がこの mailboxオブジェクトにより実現できる。

なおTAOには、cpuを割り当てられていたプロセスがそのcpuを返す関数(Smalltalk-80のyieldに相当する)、process-allow-scheduleがある。

- (send-personal-mail メイルボックス 受信オブジェクト メイル)

受信オブジェクトを指定する。 send-mailによってメイルボックスに送られたメイルはそのメイルボックスに対して receive-mailを実行したオブジェクトに渡されるが、send-personal-mailによって送付されたメイルは、指定されたオブジェクトがreceive-mailを実行した時にはじめて渡される。

- (receive-personal-mail 送信オブジェクト メイルボックス)

メイルの送信オブジェクトを指定する。 receive-mailの場合は、メイルボックス内にメイルがあればそれを取り出してしまうが、 receive-personal-mailは、指定した送信オブジェクトからのメイルを取り出す。

- (get-back-mail メイルボックス メイル)

これはいったん出したメイルを取り返すためのものである。値はnil（すでにメイルボックスからメイルが取り出されてしまった)あるいは、メイル(get-back成功)である。

- (send-mail-with-timeout n メイルボックス メイル)

これは、メイルは出すが、n*(1/60)秒以内にメイルボックスからとり出されなければそのメイルを送信オブジェクトに送り返す。よって値は、メイルか t(他のオブジェクトによって受信された)である。

- (send-round-mail 相手のメイルボックス メイルid メイル)
  (receive-round-mail メイルid 自分のメイルボックス)

これらは往復メイルを表す。単に相手からのメイルを待つのでなく、自分の出したあのメイルの返事を待つという場合に使用する。

互いのメイルボックス、即ち、いわばアドレスをどうやって知るかと言えば、最初に出すメイル中に自分のアドレスを書いてもいいし、オブジェクトidを送ればアドレスを教えてくれるaddress-noteオブジェクトを生成してもよい。メイルidとしては、発信オブジェクトidと時刻（get-universal-timeにより生成できる）を組にする。

3.2 プロセス間クロージャとプロセス割込み

前述したようにプロセスはクラスであり、そのインスタンス変数のなかにプロセス間通信のためのクロージャinterprocess-closureがある。interprocess closureはプロセスがフォークするときにinitial-functionとinitial-argumentsにapplyされる。

たとえば、初期値0の変数s-flagを２つのプロセスtest-process1とtest-process2で共有したければ

```
(defun make-common-closure ()
  (let ((s-flag 0))
    (declare (special s-flag))
    (closure '(s-flag)
             #'(lambda (fn args) (apply fn args)))))

(!env (make-common-closure))

(!test-process1 (make-process 'test-process1
                 :interprocess-closure env))

(!test-process2 (make-process 'test-process2
                 :interprocess-closure env))
```

とし、これら2つのプロセスをフォークすればよい。(なお、Common Lispでは、スペシャル変数をクロージャに入れられないことに注意)

プロセス割込み process-interruptのシンタクスは、

(process-interrupt プロセス fn args flag)

である。プロセスは割り込みたい相手プロセスである。 flagが t ならば、相手プロセスが待ち状態の時でも割込み、argsに fn を applyする。flagが nilならばプロセスが走行中に fn が実行され、fn がプロセスの実行状態を破壊しないかぎり、fnの実行終了後、元の状態に戻って計算を続ける。

process-interruptが実行されると、相手の環境で fnが評価される。 よって、args内にcatch tagを指定することによって、プロセス間Ｔｈｒｏｗを実現できる。

なお、TAO特有の機能としてcatcher-caseとthrowablepがある。

catcher-caseのシンタクスは

```
(catcher-case form (tag1 receiver11 receiver12 ・・・)
                          :
                   (tagi receiveri1 receiveri2 ・・・)
                          :
                   (tagN receiverN1 receiverN2 ・・・))
```

である。 form中で、例えば (list tagi vari)がthrowされれば、receiveri1以降に制御が移る。receiveri1以降ではtagiは変数として束縛される。tagiの初期値はvariである。tagNをtにしておけば、throwのcatch tagがtag1 ~ tagN-1に一致しない場合、制御はreceiverN1以降に移る。

throwablepのシンタクスは、(throwablep tag)である。一致するtagがあれば t、なければ nil を返す。

3.3 セマフォ

クラスsemaphoreがある。 セマフォに対する関数（またはメソッド）はp-semとv-semである。 セマフォ test-semを生成したければ

```
(!test-sem (make-instace 'semaphore))
```

とすればよい。なお、TAOのセマフォは2値セマフォであり、インスタンスが生成された時には１にセットされている。

４．ＴＡＯによる通信・同期法の実現

本章では、前章で述べたTAOの通信・同期機能による、既存の通信・同期法及び多重レベル通信の実現について述べる。なお、send-mail、receive-mail等を使用することにより、１対１通信、無限長バッファ通信、普通通信（緊急でないという意味での普通）が実現できることは容易に理解できると思う。

4.1 同報通信（１対多通信）

同報通信は、窓口メイルボックスroot-boxをインスタンス変数としてもつbroadcastオブジェクトに次に示すメソッド呼び出しstartを実行することにより実現できる。

送信する時は

[[broadcastインスタンス root-box] (list メッセージ 受信メイルボックスリスト)]

とすればよい。

```
(defmethod (broadcast start)
  (loop
    (&aux x message mailboxes)
    (!x (receive-mail root-box))
    (!message (car x))
    (:until (equal message 'stop-broadcast))
    (!mailboxes (cadr x))
    (loop (:until (null mailboxes))
          (send-mail (pop mailboxes)  message))))
```

(mailboxesは各受信オブジェクトのメイルボックスのリスト)

4.2 有限長バッファ通信

プロセス間クロージャとセマフォを使用する。メイルボックスboxと、セマフォbuffer-semと、バッファサイズmax-sizeを定義し、そのバッファを使用する複数プロセスのプロセス間クロージャとしておく。そして、各プロセスから以下の関数を呼ぶ。 (sys:semaphore-process-queueはP操作待ちのプロセスを格納するインスタンス変数)

```
(defun send-buffer (message)
   (sys:without-interrupts
    (cond ((= max-size (length [box sys:mail-queue]))
           (p-sem buffer-sem))))
   (send-mail box message))

(defun receive-buffer (&aux x)
   (sys:without-interrupts
    (!x (receive-mail box))
    (cond ([buffer-sem sys:semaphore-process-queue]
           (v-sem buffer-sem))))
   x)
```

4.3 Occam同期式通信

C.A.R.Hoareにより提案された CSP (Communicating Sequential Processes)[Hoar 85]をベースにした並行プロセス記述言語がOccam[Inai 86]である。 このOccam同期式通信がTAOのmailboxオブジェクトにより実現できることを例を用いて示す。

２つのプロセスがある。生産者プロセスは 値iniから始まる整数を次々にチャネル経由で出力し、消費者プロセスはチャネルを通して受け取った整数を出力装置に出力するものとする。同期式だから、生産者プロセスがチャネル経由で１つ整数を送り、消費者プロセスがそれを受け取ってから生産者プロセスは次の整数を生成する。 mailboxオブジェクトによりチャネルを実現した TAOプログラムを次に示す。

```
(defun generate (ini out ack-channel)
 (loop (&aux n)
       (:init (!n ini))  ;内部変数nに初期値iniを設定
       (:until [n = 101] t)     ;nが101になったら終了
       [out send-mail n]        ;ﾒｲﾙﾎﾞｯｸｽ outへnを送出
       [ack-channel receive-mail]
              ;ﾒｲﾙﾎﾞｯｸｽ ack-channelより確認信号を受理
```

```
      (!n (+ n 1))))          ;nの値を更新

(defun receive (in ack-channel)
  (loop (&aux i) (:until [i = 100] t)
        (!i [in receive-mail])
                              ;ﾒｲﾙﾎﾞｯｸｽ inから値を取る
        [ack-channel send-mail 'ack]
             ;ﾒｲﾙﾎﾞｯｸｽ ack-channelへ確認信号ackを送出
        (write i)))                ;出力装置へ出力

(!channel (make-instance 'mailbox))
          ;ｸﾗｽmailboxのｵﾌﾞｼﾞｪｸﾄchannelを生成
(!ack-channel (make-instance 'mailbox))

(!generate-process (make-process 'generate-process)
(!receive-process (make-process 'receive-process)
(defun synchro-para-start ( )
  (process-fork-go generate-process
                   '(generate 1 channel ack-channel))
  ;generateﾌﾟﾛｾｽをﾌｫｰｸﾌﾟﾛｾｽとして生成し､走行させる。
  ;1,channel,ack-channelはそれぞれ実引数
  (process-fork-go receive-process
                   '(receive channel ack-channel)
                   :standard-output *standard-output*))
```

本プログラムは(synchro-para-start)により起動される。

4.4 ＡＢＣＬ通信

ABCL(An Object-Based Concurrent Language)[Yone 85][Suzu 85]は、並列･分散処理をはじめから考慮に入れたオブジェクト指向言語であり､Actorモデルをそのベースとしている。本節では､ABCLの通信・同期機能の大きな特徴である3種類のメッセージ伝達モード(Past型､Now型､Future型)をmailboxに対する各種関数を用いて記述する。

･Past型

２つのオブジェクト間でsend-mailとreceive-mailを実行することにより実現できる。

･Now型

```
  (send-mail 相手のメイルボックス メイル)
  (receive-personal-mail 送信オブジェクト
                         自分のメイルボックス)
```

あるいは、

```
  (send-personal-mail 受信オブジェクト
                      相手のメイルボックス
                      メイル)
  (receive-personal-mail 送信オブジェクト
                         自分のメイルボックス)
```

あるいは、

```
  (send-round-mail 相手のメイルボックス
                   メイルid  メイル)
  (receive-round-mail メイルid 自分のメイルボックス)
```

とすればよい。

･Future型

Future型メイルを送ったオブジェクトは、相手オブジェクトからの返事をまたずに自分の仕事を続け、相手からの返事が返ってきた段階でその結果をある変数に代入する。もし返事が到着する以前に、この変数を参照する必要が生じたなら、メイルを送ったオブジェクトはそこで返事待ちとなる。このようなFuture型は

```
  (send-mail 相手のメイルボックス メイル)
        :
        :
  (receive-mail 自分のメイルボックス)
```

あるいは、

```
  (send-round-mail 相手のメイルボックス メイルid
                   メイル)
        :
        :
  (receive-round-mail メイルid
                      自分のメイルボックス)
```

などにより実現できる。

また、このFuture型通信は以下のように、TAOの論理型パラダイムを用いても実現できる。

```
(assert (process1 (_ch1 . _ch2))
   (== _ch1 ,(message-making))
   (do-something-A)
   (loop     ;_ch2に値が結合されるまで､つまり
             ;_ch2のﾃｰﾀﾀｲﾌﾟがundefでなくなるまで待つ。
     (:while (undefp  _ch2) t)
     (process-allow-schedule))
   (write (do-something-B _ch2)))

(assert (process2 (_ch1 . _ch2))
   (loop (:while (undefp  _ch1) t)
         (process-allow-schedule))
   (== _ch2 ,(do-something-C _ch1)))

(defun start ( )
  (let (_ch)
    (process-fork-go ABCL-process1 '(process1 _ch)
                     :common-variables '(_ch))
    (process-fork-go ABCL-process2 '(process2 _ch)
                     :common-variables '(_ch))))
```

本プログラムは(start)により起動される。またOccam同期式通信も論理型パラダイムにより実現できる。

このように、TAOでは､問題をプログラム化する際に、プログラマの好みにより適宜パラダイムを選択することができ、スムーズなプログラム作成を実現することができる。これは、TAOの複合パラダイム言語ならではの有用性である。

4.5 緊急通信

これはいわば、電報あるいは電話に類する通信である。

電報通信は、単に相手オブジェクトのメイルボックスにメイルを送るのではなく、相手オブジェクトに割込みをかけ、メイルが来たことを知らせる。

TAOでは、process-interruptとcatch&throwを用いて次のように実現できる。

```
work-process0
    :
  (process-interrupt work-process1
    #'throw (list 'telegram メイル) nil)
    :

work-process1
   (!telegram-message
     (catch 'telegram
        :
     )
```

一方、電話通信は相手オブジェクトに割込みをかけ、電話がかかってきたことを知らせるとともに通信用回線を割当てる。電報通信と同様 process-interruptと catch&throwを用いる他に、電話回線の割り当てを担当するtelephone-managerオブジェクトを必要とする。

電話通信の受信側

```
(!line
  (catch 'telephone
    (  :
       :
    )))
```

電話通信の送信側

```
(!line (will-call-up 相手電話番号))

(defun will-call-up (your-phone-number)
  (send-mail box-of-telephone-manager
             (list my-box my-phone-number
                          your-phone-number))
  (receive-mail my-box))
```

クラスtelephone-managerの定義

```
                :
                :
 (!message (receive-mail box-of-telephone-manager))
 (!line (get-line (cdr message)))
 (process-interrupt
     (pick-up-process-name (caddr message))
    #'telephone-calling (list line) nil)
 (send-mail (car message) line)
                :
;get-lineは回線の確保
;pick-up-process-nameは電話番号からプロセス名を探す

;telephone-calling定義
 (defun telephone-calling (line)
   (throw 'telephone line))
```

4.6 多重レベル通信

電報電話等緊急通信にもいろいろな種類があり、また受信側からすればすぐには受けつけたくない通信もある（例えば、風呂に入っているときは電話には出たくない）。ここで述べる通信は各種緊急通信と普通通信及びその受け方を統一的に扱おうという通信法である。

主として、プロセス間Ｔｈｒｏｗとcatcher-caseを使う。

受信側処理

```
        :
 (declare
  (special
    note-behind inhibited-message allowable-message))
 (catcher-case  ;allowable-messageとinhibited-
   (            ;message には受けつける通信、拒否する
        :       ;通信を入れる。note-behindには受けつ
    (bath-room) ;けられなかった通信による書き置きが
    (dining-room)                              ;入る。
    (study)
        :
   )
   (telephone
     (  :
     ))
   (telegram
     (  :
     )))

(defun bath-room (&aux allowable-message)
 (declare (special allowable-message))
 (!allowable-message nil)  ;どんな通信も拒否
        :
)

(defun study (&aux inhibited-message)
 (declare (special inhibited-message))
 (!inhibited-message '(telephone))     ;電話は拒否
        :
)
```

緊急通信の直接の送信者として、多重レベル通信managerを設ける。各プロセスは、緊急通信時にこのmanagerを利用する。多重レベル通信managerはプロセス間Ｔｈｒｏｗを用いるが、process-interruptの第２引数に次の処理を指定し、プロセス割込時に実行する。

緊急通信が電話通信の時

```
 (cond ((or (member 'telephone inhibited-message)
            (null allowable-message))
        (!note-behind
          (cons (list 'telephone my-tele-number)
                note-behind))))
       ((and (or (equal 'allowable-message 'all)
                 (member 'telephone
                         allowable-message))
             (throwablep 'telephone))
        (throw 'telephone line)))
```

telephoneをtelegramに、lineをmessageに変えれば電報通信の時である。

次のような check-note-behindを適宜実行する（例えばbath-roomから出たあと）ことにより緊急通信の到着を知り、それに対応する処理をすることができる。また、普通通信の到着を知るためには適宜 receive-mail-with-timeoutを実行すればよい。

```
(defun check-note-behind (note-behind)
 (cond ((null note-behind))
       ((member 'telephone
                (mapcar #'car note-behind))
                :
                :                          )
       ((member 'telegram
                (mapcar #'car note-behind)
                :
                :                          )))
```

5．並列問題解決プログラミング

本章では、solver間の通信を許すmanager-solver階層モデルによる並列問題解決法を示す。例としてはブロック移動問題[Sace 75][Take 85]を取り上げ、TAOの 各種 send&receive-mailによる普通通信とプロセス間Ｔｈｒｏｗによる緊急通信を用いたプログラミングにより実現する。
ブロック移動問題をここでは次のように定式化する。

- ・移動できる複数のblockがある。
- ・移動できない複数のplaceがある。
- ・placeあるいはblockの直上には高々一つのblockしか存在できない。
- ・問題は初期状態から目標状態へとblockを移動させることである。

初期状態と目標状態の例を図２に示す。 ここではplaceの数は4、blockの数は4としている。blockの数4には特に意味はない。一方、placeの数を4としたのは、placeが4以上の時には同時に2個の block移動が可能であり、並列協調効果を期待できるからである。

これを協調解決するためのmanager-solvers階層モデルを図３に示す。

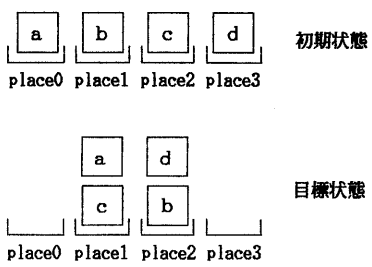

図2 placeとblock

各solverは各placeに対応する。つまり、ひとつのplaceがひとつのIA内work-processに相当する。これは、見方を変えれば、各placeに1台のELISがあり、blockをやりとりしあうと見なすことができる。 そして、全体の制御、たとえば、デッドロックの検出と回避を行なうためにmanagerを置き、これもIA内work-processとする。

さて、協調問題解決において考慮しなければならない重要なことは以下の三点である。

ポイント1. できるだけローカルな知識で問題を解決する。
ポイント2. デッドロックを防止する。
ポイント3. 全体として解の求まる方向に処理を進めていく。

ブロック問題は簡単な問題であるが、これらポイントを考察するのに適している。 以下３つのポイントを念頭に置き並列協調型解法について述べていく。

最初、各solverに与えられる知識は、各placeごとの初期状態(例えばsolver0には、block aがあるということのみで他のplaceの状態は知らされない)、全体の目標状態、managerと各solverの通信用メイルボックスidだけである。managerとsolver0~3の初期共通部品はプロセス間クロージャとされる。

処理が開始されると、各solverは普通通信によるメイル送受を開始する。 solver同志が送受するのは、次の5種類のメイルのみである。ここで強調すべきことは、これらのメイルは、deadlock?(後述)のみによりmanagerがデッドロックを検出した際にsolverが送出する do-not-send-because-of-deadlockメイル以外は、solverがローカルな知識に基づいて送受できるということである（solverがローカルな知識に基づいて do-not-send-because-of-deadlockメイルを送出する場合もある)。(ポイント１)　なお、これらメイル送受により、副次効果として他のsolverに関する知識を獲得することができる。

will-send-block
目標状態と自分のblock状態を比べ、目標placeへ自分のblockひとつを送りたいという意思を伝えるメイル

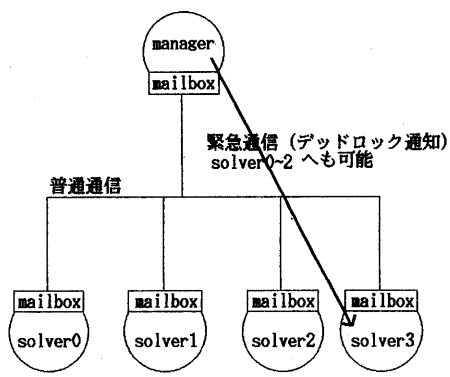

図3 manager-solvers 階層

will-send-block-for-escape
自分のblockを一時、目標place以外のplaceへ待避させたいというメイル

try-to-move
これは、will-send-block あるいはwill-send-block-for-escapeに対する返答メイルであり、送りたいというblockの送出許可である。

block
これはtry-to-moveメイルに対する返答メイルであり、該当blockの送出である。

do-not-send-because-of-deadlock
これは will-send-block あるいは will-send-block-for-escapeに対する返答であり、送りたいというblockの送出不許可メイルである。

以上はsolver間の普通通信であった。一方、ポイント2より、デッドロックを防止しなければならない。 solverから managerへの普通通信は、このデッドロックの検出(solverが目標状態に達したことの通信及びいったん目標状態に達しながら再びblockの移動があったことの通信を含む)に関するものである。

デッドロックには、solverでローカルに検出可能なもの(例えば図2のplace1とplace2、これをdeadlockとする)とローカルには検出不可能なもの(例えば図2のplace0とplace3、これをdeadlock?とする)とがある。

solverがdeadlockを検出した時は、それをmanagerに普通通信する。managerは全体情報に基づいてdepth-firstに解を探索し、その結果を各solverへ緊急通信する。

solverはwill-send-blockあるいはwill-send-block-for-escapeメッセージを送出したにもかかわらず、返答が一定時間以上ない時は managerにdeadlock?を普通通信する。managerは deadlock?を受け取ると全体を見渡し(必要な情報が solverから到着していない時は待つ)、deadlockの場合にはmanager自ら解を探索し、その後solverに割込みをかけdeadlockと解手順を緊急通信する。

この並列協調問題解決法では、deadlockあるいはdeadlock?が発生した場合には、全体による協調解法をmanagerによる逐次解法に切り換えている訳で、解が存在するならばそれに到達できる(ポイント3)。

6. おわりに

TAOの豊富な通信・同期機能とそれを基礎にしたいくつかの通信・同期法の実現を示した。TAOプログラミングでは、問題により、あるいはプログラマの好みによりTAOの各種パラダイムの中から好みのパラダイムを選択でき、これにより、解こうとする問題とプログラムとの間のセマンティック・ギャップを狭めることができる。なお、文法的なデコレーションはLispの持つマクロ機能により簡単に付加できる。また、Future型、同報通信をはじめとする各種通信も複数の方法により実現できる。これらは、TAOの複合パラダイム言語ならではの有用性と考えることができる。 本稿では記述できなかったが、Concurrent Prolog[Shap 83]、Concurrent Smalltalk[Yoko 85][Suzu 85]の通信同期機能のTAOによる実現についても試みた。これについては、機会を改めて報告することとしたい。

また、ブロック移動問題を例にとり、TAOの通信・同期機能を用いたmanager-solver階層モデルによる並列協調型問題解決プログラミングを試み、並列協調型問題解決に対し、TAOの通信・同期機能が有効に作用することを確認した。

そして、並列協調型問題解決において重要な
・いかにしてローカルな知識で問題を解決するか
・いかにして解空間を狭めるか
・いかにしてデッドロックを防止するか
という点についてひとつの指針的な解法を示した。これについてはさらに今後、定量的、理論的検討をするつもりである。

現在のところ前述したブロック移動問題解決プログラムは1台のELIS上で走行しており、並列環境はprocess-allow-schedule等のプロセス・スイッチにより擬似的に実現されている。よって本稿で述べた各種通信法も、ネットワーク上へのパケット送出レベルまで細かくインプリメントされているわけではない。今後はELIS複数台を接続したシステムを構築し、実際に各種通信をこのシステム上にインプリメントし、物理的な並列・分散環境における協調問題解決を試みる予定である。

最後に、御討論頂いたTAOグループ諸氏に深謝する。

<参考文献>

[Film 84] R.E.Filman and D.P.Friedman: "Coordinated Computing", McGraw-Hill, 1984.(訳本 協調型計算システム、 マグロウヒルブック)
[Hoar 85] C.A.R.Hoare: "Communicating Sequential Processes", Prentice-Hall, 1985.
[Onai 86] 尾内: "Occamとトランスピュータ",共立出版, 1986.
[Sace 75] E.D.Sacerdoti: "The Non-linear Nature of Plans", Proc. of IJCAI 4,1975.
[Shap 83] E.Y.Shapiro: "A Subset of Concurrent Prolog and its interpreter",ICOT Technical Report TR-003,1983.
[Suzu 85] 鈴木則久編: "オブジェクト指向 解説とWOOC'85からの論文",共立出版,1985.
[Take 85] 竹内彰一: "Moving Block World", ICOT Working Group PPS資料, 1985.
[Take 83] I.Takeuchi,H.G.Okuno and N.Osato: "TAO-A harmonic mean of Lisp,Prolog and Smalltalk" ,ACM SIGPLAN Notices,Vol.18,No.7,1983.
[Yama 86] 山田,大野,日比野,竹内: "AIワークステーションELISの検討" ,情報処理学会研究報告(マルチメディアと分散処理), 86-MDP-31-2, 9月,1986.
[Yoko 85] 横手,所: "並行オブジェクト指向言語 Concurrent Smalltalk", コンピュータソフトウェア, vol.2,No.4,1985.
[Yone 85] 米澤: "オブジェクト指向並列モデルとその記述言語ABCL",昭和60年電気・ 情報関連学会連合大会, 35-6,1985.