Vol. 29 No. 10 情報処理学会論文誌 Oct. 1988

ゴール群に基づく並列論理型言語の情報資源管理方式†

藤村 考†† 栗原正仁†† 加地郁夫††

GHC などの committed-choice 型の並列論理型言語において，データベースなどの情報資源を統一的かつ容易に蓄積し利用するための基本的な枠組みを論じる．情報資源を従来方式によりプログラム節や内部状態を持つ永久プロセスとして蓄積した場合，いくつかの問題点（例えば，多数のプロセスに分散した情報資源に対する一括した処理が複雑になる等）がある．本論文では，ゴール群と呼ぶ新しい概念に基づいて情報資源を管理することにより，これらの問題を解決する．ゴール群は各資源に対応して存在し，各ゴール群はプロセスの集合として統一的に表現される．応用システムから各ゴール群にアクセスする手段としてプロセス・ディレクトリを提案し，その仕様と実現法の概略を述べる．また，本方式は特定のプロセスへのトランザクションの集中によるボトルネックを生じる可能性があるが，これを避けるためのプログラミング技法について述べる．

1. はじめに

並列論理型言語は論理型言語における並列性の研究から生まれた言語である．並列論理型言語の歴史は浅いが，現在までに多くの言語が提案，議論されてきた．その中でも特に注目されている言語には，PARLOG[1]，Concurrent Prolog[2]，GHC[3] などがある．これらの言語は，最近，committed-choice language（以後，CCL と略）と総称されている．CCL はいずれも論理型言語の枠組みを基にガードと呼ばれる制御構造を導入し，与えられたゴールに対する解を複数個生成する機能を放棄している．この結果として，CCL は Prolog のような関係データベースとの密接な関係を失った．そのため，CCL では大量のデータを有する応用システム（application）が書き難く，そのための言語の拡張やプログラミング技法の開発が望まれている．

本論文は，CCL における情報資源を統一的かつ容易に蓄積し利用するための基本的な枠組みについて論じる．ただし，情報資源とは，応用システムの実行過程で参照あるいは獲得されるデータで，いくつかの応用システムの共有資源として蓄積し利用され得るものを呼んでいる．例えば，エキスパート・システムにおける知識ベース，データベース，ファイルなどは情報資源の一例である．本論文で論じる内容はきわめて一般的なものであり，具体的なデータモデルに基づいた特定のデータベースの実現技法などを論ずるものではない．

以下，2章では，CCL における情報資源を従来方式により，プログラム節や内部状態を持つ永久プロセスとして蓄積した場合に生じるいくつかの問題点（例えば，データのバックアップがとる操作が複雑になる等）を示し，それを解決するために，「ゴール群」と呼ぶ概念に基づいて統一的かつ容易に情報資源を蓄える基本方式を述べる．3章では，ゴール群を名前で集中的に操作できる「プロセス・ディレクトリ」と呼ぶ機構を提案し，その仕様と実現法の概略を述べる．4章では，本方式の問題点となりやすい「特定のプロセスへのトランザクションの集中」を避けるためのプログラミング技法について述べる．そして，5章では，本方式を取り入れた処理系をインプリメントする際のポイントであるゴールのスケジューリング方法の概要を示す．

なお，本論文で提案する方式はどの CCL でも有効であるが，本論文中のプログラムの記法は Flat GHC[6] に従う．また，特に明記しない限り専門用語は文献 7) に従う．

2. 情報資源管理の問題点とゴール群

2.1 情報資源管理の問題点

Prolog のプログラミングでは，情報資源をプログラム節（特に，単位節すなわちボディが空の節）として蓄積する方法がしばしば使用される．この方法は，組み込み述語 assert などを用いてプログラムを書き換える必要が生じる．これらの述語は副作用を使う非論理的なものであるため，プログラム変換や合成などの高度なプログラミング・システムの開発を妨げる．このため，このような述語の使用は避けることが望ましい．しかし，関係データベースの組（tuple）を単位

† Information Resource Management by Goal Groups in Parallel Logic Languages by KOU FUJIMURA, MASAHITO KURIHARA and IKUO KAJI (Department of Information Engineering, Faculty of Engineering, Hokkaido University).

†† 北海道大学工学部情報工学科

節に対応させることにより，データベースの検索などの処理を行うプログラムが非常に容易に記述できるため，実用上この方法が用いられることが多い．

一方，CCL のプログラミングでは，以下の問題があり，情報資源をプログラム節として蓄積することは現実的でない．

(A1) プログラムを書き換える assert, retract などの組み込み述語を許すと，実行順序の非決定性のため，ゴールの展開木のどの時点でプログラムが書き換わるか一意に決まらない．そのため，プログラムの意味が複雑になる．

(A2) CCL が与えられたゴールに対する解を複数個生成する機能を放棄したことにより，複数の単位節を列挙してもコミットされたただ一つの単位節しか検索できない．すなわち，CCL における実行過程は，Prolog と異なり，関係データベースの検索とうまく対応していない．

そこで CCL では，情報資源を内部状態を持つ永久プロセス (perpetual process)[2),7)]として保持するという技法を用いるのが一般的である．これは，CCL プログラムのプロセス的解釈[1),2),4),5)]に基づき，プロセスの引数のうち他のプロセスと共有していないものをそのプロセスの内部状態とみなし，内部状態に対する更新などのトランザクションはストリームを介してそのプロセスに送ることによって実現するものである．この技法は，①副作用を用いないでデータを蓄えられる，②外部からデータ構造を隠ぺいすることができる，③入出力資源もプロセスとみなせるため情報資源と入出力資源を統一的に扱える，という優れた特徴がある．また，この技法は，CCL でオブジェクト指向プログラミングする際の基本となっている[8)]．しかし，このプログラミング技法も次のような問題がある．

(B1) 応用システムの実行によって得られた情報資源は，応用システムの終了後も処理系に蓄えられている必要がある．しかし，応用システムの実行の終了をゴール節が空節になることとする従来の実行方式では，プロセスとして蓄えた情報資源を応用システムの終了後に存在させることができない（プロセスは論理的にはゴールにほかならないため，空節はプロセスが存在しない状態を意味する）．したがって，従来の CCL プログラミングでは，応用システムの終了時に情報資源を入出力装置などに書き込む必要があった．

(B2) 大量の情報資源を一つの永久プロセスに保持した場合，トランザクションが集中し，そのプロセスの処理がボトルネックになる．

問題点 (B2) は，情報資源を多くのプロセスによって協調して保持し，並列度を高めることにより解決できる（4章で例を示す）．しかし，この場合，次のような別の問題が生じることになる．

(B3) 多くのプロセスに分散して情報資源を保持すると，ファイルに情報資源のバックアップをとったり，情報資源をまとめて削除するなど操作が複雑になる．例えば，情報資源のバックアップをとるには，その情報資源を保持しているすべてのプロセスにメッセージを送り，情報を集めなければならない．

本論文では，CCL の処理系にプロセスを管理する新しい機構を導入し，問題点(B1)と(B3)を克服する．また，(B2)を解決するために情報資源を分散するプログラミング技法を示す．これにより，情報資源をプロセスとして蓄えるプログラミング技法をより実用的なものにすることができる．

2.2 ゴール群

これらの問題点を解決するための準備として，"ゴール群" と呼ぶ概念を導入する．ゴール群の定義を示す前に，ここでゴール節を定義しておく．

定義 ゴール節とは

$$\leftarrow B_1, \cdots, B_n$$

の形をした節で，結論部は空である．各 B_i $(i=1, \cdots, n)$ をゴールあるいは場合によりプロセスと呼ぶ．

$y_1, \cdots, y_r$ をゴール節 $\leftarrow B_1, \cdots, B_n$ の中の変数とすると，この表現は，論理式

$$\forall y_1 \cdots \forall y_r(\neg B_1 \vee \cdots \vee \neg B_n)$$

の略記である．

定義 ゴール節 $G = \forall y_1 \cdots \forall y_r(\neg B_1 \vee \cdots \vee \neg B_n)$ 中の1個以上の任意のリテラルの選言

$$\neg B_{i_1} \vee \cdots \vee \neg B_{i_j} \quad (j>0)$$

を G のゴール群と呼ぶ．

ゴール節 G 中の n 個のリテラルを分割してできる k 個のゴール群を $G_1, \cdots, G_k$ とすると，

$$G = \forall y_1 \cdots \forall y_r\ (G_1 \vee \cdots \vee G_k).$$

例 ゴール節 $G = \leftarrow p(X, Y),\ q(Y), r(X)$，$G_1 = \neg p(X, Y) \vee \neg q(Y)$，$G_2 = r(Y)$ とすると，G_1, G_2 はそれぞれ G を分割してできたゴール群である．

直感的には，ゴール群はあるゴール節の断片 (fragment) である．ゴール節とゴール群との論理的な違い

は，ゴール節がすべての変数の出現 (occurrence) が束縛 (bound) されているのに対し，ゴール群はすべての変数の出現が自由 (free) であることである．したがって，ゴール群は全称閉包 (universal closure) を付け加えることにより，ゴール節に変換できる．

例 ゴール群 $G_1=\neg B_1\vee\cdots\vee\neg B_n$ とすると $\forall(G_1)=\leftarrow B_1,\cdots,B_n$ となる．ただし，$\forall(F)$ は F の全称閉包である．

定義 ゴール節 $G=\forall(G_1\vee\cdots\vee G_i\vee\cdots\vee G_k)$，ゴール群 $G_i=\neg B_1\vee\cdots\vee\neg B_m\vee\cdots\vee\neg B_n$ とする．CCL の計算規則に従い G を実行し，G_i 以外のゴール群のいくつかのゴールと B_m が展開されて，G_i が $G_i'=(\neg B_1\vee\cdots\vee\neg B_{m-1}\vee\neg A_1\vee\cdots\vee\neg A_r\vee\neg B_{m+1}\vee\cdots\vee\neg B_n)\,\theta'\theta^{EX}$ となったとき，G_i' を G_i の子のゴール群と呼ぶ．ここで，θ' は B_m の展開によって生じた代入，θ^{EX} は G_i 以外のゴール群のゴールの実行によって生じた代入とする．同様に，$G_i'', G_i''', \cdots$ を G_i の子孫のゴール群と呼ぶ．

直感的には，ゴール群 G_i の中のゴールを1回以上展開したゴール群が G_i の子孫のゴール群である．

2.3 ゴール節の実行とゴール群

ゴール群はゴールの管理上の概念で，プログラムの意味はゴール節の群への分割によって変わらない．つまり，ゴールの計算規則はゴール群に依存しない．ただし，本論文では，2.1 節で示した問題点(B 1)を克服するため，応用システムの起動と終了を従来と異なった方式で行うことを提案する．

Prolog などの一般的な論理型言語の処理系では，応用システムの起動は，ユーザがゴール節を（端末などから）入力することによって行われる．一方，本論文では，ゴール節 G を入力するのではなく，G の断片であるゴール群を入力することによって起動すると考える．つまり，処理系があらかじめいくつかのゴール群 $GS_1,\cdots,GS_n$ を保持しており，ユーザがゴール群 GM を入力すると，ゴール節 $G=\forall(GM\vee GS_1\vee\cdots\vee GS_n)$ を構成し，これを応用システムを動かすゴール節として実行することにする．直感的には，各 GS_i は各資源に，GM は応用システムの実行プロセスに対応している．各 GS_i をシステム・ゴール群，GM をメイン・ゴール群と呼ぶが，これらの詳細は3章で述べる．

応用システムの実行の終了とは，ゴール節 G を実行していく過程で，メイン・ゴール群の子孫のゴール群が空節になることとする．つまり，G が $\forall(\square\vee GS_1^*\vee\cdots\vee GS_n^*)\,\theta$ となることである．ここで，各 GS_i^* を GS_i の子孫のゴール群，θ をこの実行によって得られた解代入 (answer substitution) とする．

本論文では，このように，ゴール節を空節になるまで実行するメイン・ゴール群と空節になるまで実行しなくてもよいシステム・ゴール群とに分け，システム・ゴール群を処理系に残すことで，2.1 節で示した問題点(B 1)を克服している．また，ゴール群に基づいて多くのプロセスを一括して管理する機構を実現することにより，2.1 節で示した問題点(B 3)を克服することができる．次章ではこのゴール群の管理方式を述べる．

3. ゴール群の管理方式

3.1 メイン・ゴール群

メイン・ゴール群とは，応用システムを駆動するために，ユーザが（端末などから）入力したゴール群およびその子孫のゴール群である．したがって，メイン・ゴール群が入力されると応用システムが起動し，メイン・ゴール群が空節になると応用システムが終了する．

メイン・ゴール群には，directory という特別の述語記号を持つ一引数のゴールが高々一つ存在してもよい．これは，他のゴール群（システム・ゴール群）と通信するために用いられるプロセスで，プロセス・ディレクトリと呼ぶ．プロセス・ディレクトリの詳細は 3.3 節で述べる．

なお，本論文では簡単化のため，メイン・ゴール群は同時に複数個存在しないこととする．しかし，マルチ・ユーザに対応するため，複数個存在することを許すように拡張することは容易であると思われる．

3.2 システム・ゴール群

システム・ゴール群とは，各資源に対応して存在するゴール群およびその子孫のゴール群である．システム・ゴール群は，通常，応用システムの起動前に既に処理系が保持しており，応用システムの終了後，その子孫のゴール群が新たなシステム・ゴール群として再び処理系に保持される．ただし，3.3 節で述べるシステム・ゴール群の生成削除の機構により，応用システムにより動的に発生させたり，削除することも可能である．

システム・ゴール群には，①プログラミング・システムによって提供される“組み込みのゴール群”と②ユーザがゴールを与えて後述の機構により発生させた

ゴール群がある．①の“組み込みのゴール群”には，さらに入出力の論理デバイスに対応するプロセスとして存在する仮想的なゴール群とゴールとして実在するものがある．

これらの様々なゴール群は，名前で統一的に管理する．また，各システム・ゴール群はその外部からのメッセージを受け取るためのストリームとして用いる特別な変数を必ず一つ持つよう構成することにする．この変数をそのシステム・ゴール群の入力ストリームと呼ぶ．各システム・ゴール群は，3つ組 $GS_i=\langle i, G_i, V_i\rangle$ で処理系が管理する．ここで，i を名前，G_i をゴール群，V_i を GS_i の入力ストリームとする．また，処理系が管理しているすべてのシステム・ゴール群の名前の集合を N で表す．

3.3 プロセス・ディレクトリ

メイン・ゴール群からの資源のアクセスは，各資源に対応するシステム・ゴール群の入力ストリームにメッセージを送ることによってなされる．この場合，そのメッセージを送るための機構が必要となる．また，応用システムがシステム・ゴール群を動的に生成削除する機構も実用上必要である．これらの操作は本質的に逐次性が要求される．本論文では，逐次性を確保するために，システム・ゴール群に対する操作を集中的に受け付ける特別のプロセスを導入し，それをプロセス・ディレクトリと呼ぶ．

プロセス・ディレクトリはメイン・ゴール群において一引数の組み込み述語 directory (S) を実行することにより，メイン・ゴール群中に高々1個発生させることができるプロセスである．ここで，S はこのプロセス・ディレクトリへのメッセージの入力ストリームである．プロセス・ディレクトリは S に [ ] を送ることによって消滅する．

プロセス・ディレクトリは内部状態として，すべてのシステム・ゴール群の名前 i とストリーム V_i とのペア $\langle i, V_i\rangle$ の集合を保持している．この集合をプロセス・テーブルと呼ぶ．プロセス・テーブルはプロセス・ディレクトリが消滅する際に処理系に保存され，プロセス・ディレクトリが発生する際に処理系から与えられる．プロセス・ディレクトリは以下のような機能を持つ．

① メッセージ send (i, M) を受け取ると，V_i にメッセージ M を送る．例えば，プロセス・ディレクトリに GS_i 宛の k 個のメッセージが send (i, M_1), send (i, M_2), …, send (i, M_k) の順で送られてきた場合，V_i に $M_1, M_2, \cdots, M_k$ の順でメッセージを転送する．

② メッセージ create (i, G, V_i) を受け取ると，処理系に新たなゴール群 $GS_i=\langle i, G, V_i\rangle$ を登録し，プロセス・テーブルに $\langle i, V_i\rangle$ を追加する．V_i は GS_i の入力ストリームとなるので，G には必ず V_i が含まれなければならない．

③ メッセージ delete (i) を受け取ると，処理系のメタレベルの機構により，そのゴール群のすべてのゴールの実行を中止 (abort) し，プロセス・テーブルから $\langle i, V_i\rangle$ を取り除く．

プロセス・ディレクトリの基本的な仕様は次のプログラムで表現できる．

```
directory (S) :-true|
    {プロセス・テーブル PT を設定する},
    directory (S, PT).
directory ([send (I, M)|S], PT) :-true|
    send (I, M, PT, PT1),
    directory (S, PT1).
directory ([create (I, Goal, V)|S], PT) :-true|
    {ゴール Goal を実行する新たなゴール群 I を
     生成する}.
    directory (S, [(I, V)|PT]).
directory ([delete (I)|S], PT) :-true|
    {ゴール群 I のすべてのゴールの実行を中止
     (abort) する},
    delete (I, PT, PT1),
    directory (S, PT1).
directory ([ ], PT) :-true|
    {プロセス・テーブル PT を保存する}.
send (I, M, [(J, V)|PT], PT1) :-I=J|
    V=[M|V1], PT1=[(J, V1)|PT].
send (I, M, [(J, V)|PT], PT1) :-I≠J|
    PT1=[(J, V)|PT2], send (I, M, PT, PT2).
send (I, M, [ ], PT1) :-true|PT1=[ ].
delete (I, [(J, V)|PT], PT1) :-I=J|PT1=PT.
delete (I, [(J, V)|PT], PT1) :-I≠J|
    PT1=[(J, V)|PT2], delete (I, PT, PT2).
delete (I, [ ], PT1) :-true|PT1=[ ].
```

このほかにも，様々なメッセージを許すことが考えられる．例えば，

④ copy (i, j)：ゴール群 i のすべての変数を新しい変数に取り替えたゴール群をつくり，そのゴール群を名前 j として登録する．

⑤ frozen_goals (i, GL, Y): ゴール群 i のすべてのゴールのリスト（ただし，ゴール中のすべての変数に '$var' (0) などの特別な基礎項を代入したもの）を GL に代入し，このリストの中の項で V_i に対応する基礎項を Y に代入する．この命令は，例えば，二次記憶上にゴール群のバックアップを保存する場合に使用できる．

⑥ dir (X): プロセス・ディレクトリに既に登録されているゴール群の名前などの情報を X に代入する．

なお，directory 述語のプログラムの一部を文章（{ } で囲まれた部分）で記述したのは，これらの処理がゴールをデータとして扱うメタレベルの操作で，CCL プログラムでは記述できないからである．これらの操作はプロセス・ディレクトリが処理系にその処理を委託する．directory 述語は組み込み述語として処理系が提供する．

directory 述語はこのように一階述語ではない“非論理的”な述語であるわけであるが，メイン・ゴール群中で高々一度しか呼び出されないため，メイン・ゴール群を

$\neg B_1 \vee \cdots \vee \neg B_n \vee \neg$directory ($S$)

（実際には，:-$B_1, \cdots, B_n$, directory (S). という記法で入力する）

として与えることにより，$B_1, \cdots, B_n$ を展開する応用システムのプログラムコードとしては非論理的な述語を使わなくても済む．

3.4 使 用 例

ここでは，簡単な例により，プロセス・ディレクトリの基本的な使用法を示す．

ファイルなどの入出力資源は，書き込まれている情報を内部状態とする仮想的なプロセスとみなすことができる．これらのプロセスはあらかじめプロセス・ディレクトリに登録してある“組み込みのゴール群”として提供される．しかし，以下のように，ファイルの仕様をプログラムで記述し，そのプログラムによって発生させたゴール群を処理系に登録することにより，実在のゴールとして実現することもできる．これをプロセス・ファイルと呼ぶ．

```
file_instance ([open_input (Xs)|S], C) :-true|
  file_input (Xs, C), file_instance (S, C).
file_instance ([open_output (Xs)|S], _) :-true|
  file_output (Xs, C), file_instance (S, C).
file_input ([X|Xs], [A|C]) :-true|
  X=A, file_input (Xs, C).
file_input ([X|Xs], [ ]) :-true|
  X=end_of_file, file_input (Xs, [ ]).
file_input ([ ], _) :-true|true.
file_output ([X|Xs], C) :-true|
  C=[X|C1], file_output (Xs, C1).
file_output ([ ], C) :-true|C=[ ].
```

このプログラムの第1引数は，そのファイルへのトランザクション・ストリームであり，open_input (S), open_output (S) などのメッセージを受け付ける．第2引数はファイルの内容を保持するのに使われている．

プロセス・ファイルはメイン・ゴール群：

```
:-directory (S),
  S=[create (name 1, file_instance (X, [ ]), X)].
```

を入力し，処理系に登録しておくと，delete メッセージにより削除しない限り，システム・ゴール群として存在し続ける．したがって，ユーザから見ると物理的なファイルとして実現されているものと全く区別なく使うことができる．例えば，CCL の例題として有名な素数生成のプログラムによって発生した素数列ストリーム Ps をプロセス・ファイル prm 100 に書き込むには，メイン・ゴール群：

```
:-directory (S), S=[send (prm 100,
  open_output (Ps))], primes (100, Ps).
```

を入力すればよい．次に，このデータをプロセス・ファイルから再び読みだし，window 1（組み込みゴール群）に表示させるには，メイン・ゴール群：

```
:-directory (S),
  S=[send (prm 100, open_input (Is)),
      send (window 1, open_output (Os))],
  type (Is, Os).
```

を入力すればよい．ただし，type (Is, Os) はストリーム Is に変数を送ることによって，データを要求し，その結果得られたデータをストリーム Os に送る簡単なゴールである（付録参照）．

3.5 ゴール群間の通信

各ゴール群は基本的に**図1**のようなストリームによって連結されている．ゴール群間の通信は，メイン・ゴール群 GM から各システム・ゴール群 GS_i ($i \in N$) に結合しているストリーム V_i ($i \in N$) にメッセージを送ることによって開始する．したがって，システム・ゴール群同士での通信やシステム・ゴール群からメイン・ゴール群の向きの通信などをシステム・

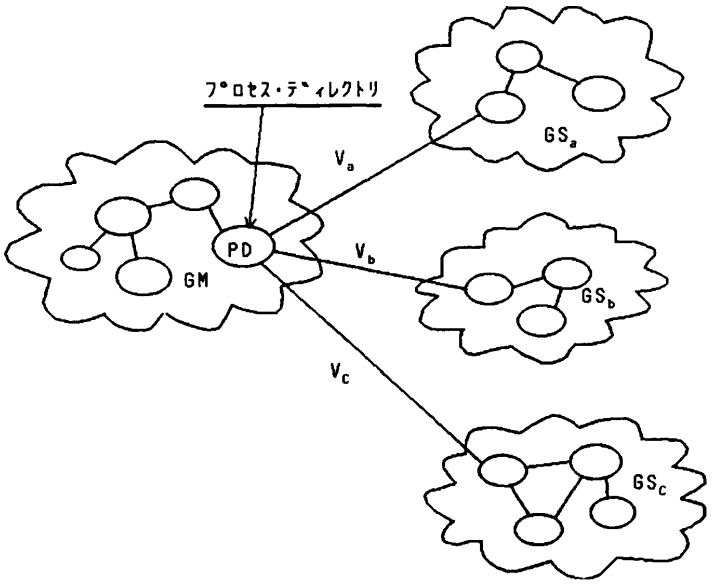

図 1 ゴール群とプロセス・ディレクトリ
Fig. 1 Groups of goals and a process directory.

ゴール群が能動的に行うことはできない．能動的に通信を行えるゴール群は唯一メイン・ゴール群のみである．

GM と 各 GS_i はプロセス・ディレクトリを介して，共有変数 V_i を常に1つ持っている．本論文で提案するゴール群の管理方式では，各 GS_i に対するメッセージはすべてこの V_i を通して送られてくるように制限していることが1つの重要な特徴である．つまり，V_i は GS_i とその外界とを結ぶ唯一の窓口であり，そのゴール群の外界（他のゴール群）で観測される GS_i 中の変数は，少なくとも初期状態では，V_i 以外に存在しない．ただし，ストリーム V_i に対して変数を含んだメッセージを送ることにより，間接的に多くの変数がそのゴール群とその外界で共有されることは可能である．次章ではこの具体例を示す．

4. 資源分散のプログラミング技法

4.1 ASP

情報資源を永久プロセスとして蓄えるプログラミング技法は，特定のストリームへのメッセージの集中によるボトルネックを引き起こす可能性がある．ここでは，連想記憶という簡単な例を通じ，これを避けるためのプログラミング技法（の一例）を示す．

連想記憶（associative store）とは，ロケーションの指定がアドレスではなく，その内容によって行われる記憶である．本論文では，Key と Value のペア（〈Key, Value〉で表す）から成る集合 A（抽象データ型）に対して，次の3種類の命令の列を実行するためのデータ構造を連想記憶と呼ぶ．

① member (K, X): $\exists V\langle K, V\rangle \in A$ ならば，X に V を代入し，$\neg\exists V\langle K, V\rangle \in A$ ならば，X に nil を代入する．

② insert (K, V): $\exists V'\langle K, V'\rangle \in A$ ならば，A を $A-\{\langle K, V'\rangle\}\cup\{\langle K, V\rangle\}$ で置き換える．$\neg\exists V'\langle K, V'\rangle \in A$ ならば，A を $A\cup\{\langle K, V\rangle\}$ で置き換える．

③ delete (K): $\exists V\langle K, V\rangle \in A$ ならば，A を $A-\{\langle K, V\rangle\}$ で置き換える．

連想記憶を CCL で実現するために，集合 A を内部状態として保持するプロセスを生成し，ストリームを介して，member, insert, delete などのトランザクションを送ることとする．このプロセスを連想記憶プロセス（以後，ASP: Associative Store Process と略す）と呼ぶ．ASP のプログラムは以下のようになる．

```
asp([member(K,X)|S],A) :-true|
    member(K,X,A), asp(S,A).
asp([insert(K,V)|S],A) :-true|
    insert(K,V,A,A'), asp(S,A').
asp([delete(K)|S],A) :-true|
    delete(K,A,A'), asp(S,A').
```

ただし，member, insert, delete 述語は抽象データ型 A に基づき適当に定義されているものとする．

いま，asp 1 という名前のシステム・ゴール群として，あらかじめ ASP_A^1 が登録されているものとする．asp 1 を利用しようとするメイン・ゴール群のプロセス p が asp 1 にトランザクションを送るには，各トランザクションを1つ1つ send メッセージで包み（例えば，send (asp 1, member (foo, X))）プロセス・ディレクトリを介して送ることもできる．この方法では，1つの ASP に対して多くのプロセス p_i $(i=1, \cdots, k)$ がアクセスしようとした場合は，

```
:-p1(S1), ..., pk(Sk),
    merge([S1, ..., Sk], SM), directory(SM).
```

（ただし，merge $([S_1, \cdots, S_k], S_M)$ は k 本のストリーム S_1 から S_k をマージして，ストリーム

S_M とするゴール)

としてストリームをマージすることとなるが，このとき，次のような問題が生じる．

(1) asp 1 に送るすべてのトランザクションをプロセス・ディレクトリが1つ1つ宛先を確認しなければならない．このため，プロセス・ディレクトリがボトルネックとなる可能性が生じる．

(2) asp 1 の処理能力を超えるトランザクションがプロセス p_i $(i=1,\cdots,k)$ から送られる場合，asp 1 の処理がボトルネックとなる．(これは，2.1節の(B 2)で示した問題の具体例である)

そこで，member, insert, delete の命令に加えて，ASP への新たなトランザクション・ストリームを確保する1引数の命令 open を許し，

:-$p_1(S_1),\cdots,p_k(S_k)$,
S=[send (asp 1, open (S_1)),…, send (asp 1, open (S_k))], directory (S).

のようにメイン・ゴール群を与えることにする．これにより，各 p_i と asp 1 とを直接結ぶストリームができるため，問題点(1)は解決できる．S_i $(i=1,\cdots,k)$ を流れるメッセージは，前例と異なり，send メッセージで包まずに送るものとする．このような複数のトランザクション・ストリームを持つ ASP をマルチ・トランザクション・ストリーム連想記憶と呼ぶ．この例に限らず，プロセス・ディレクトリがボトルネックとなる場合の多くは，資源に対するトランザクション・ストリームが動的に複数個得られるように資源を設計することが有効である．

4.2 集中実現法

この ASP の最も簡単な実現法は，既に示した ASP のプログラムに次のような open メッセージを受け付けるための節を付け加えることである．

asp ([open (S_1)|S], A) :-true|
merge (S_1, S, S_M), asp (S_M, A).

この方法は，実現の容易さの点で優れている．しかし，ASP 本体のトランザクションの処理能力が向上していないので，問題点(2)を解決していない．

4.3 分散実現法

問題点(2)を解決するためには，集合 A を分割し，複数個の ASP で協調して集合 A を保持することにより，並列度を高めることが必要である．そこで，Key を入力とする1引数の述語の列を適当に定め，D_i $(i=1,\cdots,n)$ とする．これにより集合 A を次のように $A_{0\cdots0}$ から $A_{1\cdots1}$ の 2^n 個に分割し，これに対応して ASP を分割することにする．

$A_{\langle 0\rangle}=A$
$A_{0\langle i-1\rangle}=\{\langle K,V\rangle \mid \neg D_i(K) \text{ and } \langle K,V\rangle \in A_{\langle i-1\rangle}\}$
$A_{1\langle i-1\rangle}=\{\langle K,V\rangle \mid D_i(K) \text{ and } \langle K,V\rangle \in A_{\langle i-1\rangle}\}$
(ただし，$\langle i\rangle$ は長さ i の 0,1 の列を表す)

述語 D_i としては様々のものが考えられるが，例えば，Key の i 番目のビットが1のとき真となる述語を用いれば容易に実現できる．

ここで，2つの入力ストリーム S_a と S_b をマージし，そのマージして得られたメッセージ列をメッセージ中のキー K が上記の述語 D_i を満たすか否かによって分類し，$D_i(K)$ が真ならば出力ストリーム S_1 へ，偽ならば S_0 へ送り出す 2-2 分配プロセス (2-2 DP : 2-input 2-output Distribute Process と略) '2-2 DP' (S_a, S_b, S_0, S_1) を導入する．このとき，2-2 DP と分割された ASP によって，例えば，8つのストリームをもつ連想記憶は，**図2**のような binary n-cube のプロセス・ネットワークとして構成できる．図2において，D_i は述語 D_i による分類を行う 2-2 DP であり，$A_{\langle 3\rangle}$ は分割された ASP を表している．一般に，N 本のトランザクション・ストリームの場合，ASP を N 個に分割し，$\lceil \log_2 N \rceil$ 段の 2-2 DP のネットワークを構成できる (ただし，$\lceil X \rceil$ は X の天井 (ceiling) を表す)．

このネットワークは，メイン・ゴール群からの open 命令による新たなストリームの要求に応じて，ASP およびネットワーク自身により動的に構成できる．基本的には，open (X) のメッセージを ASP あるいは 2-2 DP が受け取ると，必要に応じて ASP を

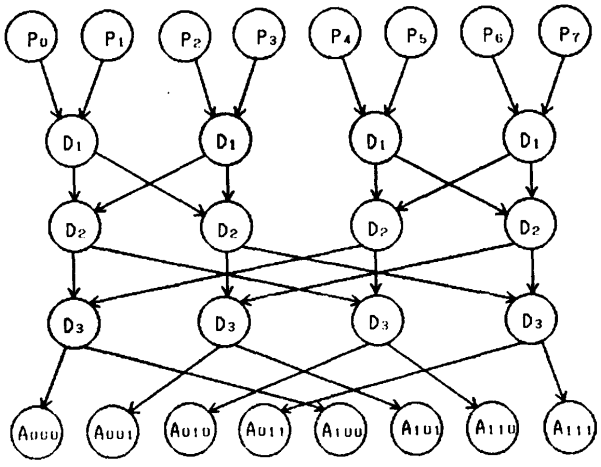

図 2 2-2DP によるプロセス・ネットワーク
Fig. 2 A process network using 2-input 2-output distribute processes.

分割したり，2-2 DP をコピーすることによって，変数 X を新たなトランザクション・ストリームとして使用できるようにする．ASP と 2-2 DP が open(X) を受け取ったときに行う処理のアルゴリズムはプログラムで表現すると次のようになる．

```
asp ([open (X)|S], A) :-true|
    divide (A, A0, A1), '2-2 DP' (S, X, S0, S1),
    asp (S0, A0),  asp (S1, A1).
'2-2DP' ([open (X)|Sa], Sb, S0, S1) :-true|
    S0=[open (S0^2)|S0^1],
    S1=[open (S1^2)|S1^1],
    '1-2 DP' (Sa, S0^1, S1^1),
    '2-2 DP' (X, Sb, S0^2, S1^2).
'2-2 DP' ([ ], Sb, S0, S1) :-true|
    '1-2 DP' (Sb, S0, S1).
'2-2 DP' (Sa, [ ], S0, S1) :-true|
    '1-2 DP' (Sa, S0, S1).
'1-2 DP' ([open (X)|S], S0, S1) :-true|
    '2-2 DP' (S, X, S0, S1).
'1-2 DP' ([ ], S0, S1) :-true|
    S0=[ ], S1=[ ].
```

このプログラムにおいて，2-2 DP が open メッセージを受け取った際に行う処理は複雑なので，**図 3** にプロセス・ネットワークが変化するようすを示す．ここで，'1-2 DP' (S, S_0, S_1) は，2-2 DP と同様に，S から送られるメッセージを出力ストリーム S_1 と S_0 へ分配するプロセスである．なお，2-2 DP や 1-2 DP は open 以外のメッセージを受け取った際に，D_i (Key) を調べる必要があるので，この i を内部状態として，引数に保存しなければならないが，本論文ではその処理は省略している．例として，メイン・ゴール群：

:-$p_1(S_1)$, $p_2(S_2)$, $p_3(S_3)$,

S_0=[send (asp 1, open (S_3), send (asp 1, open (S_2)), send (asp 1, open (S_1))],
directory (S_0).

を与えたとき，ネットワークが構成されるようすを**図 4** に示す．この例では，トランザクション・ストリームがプロセス・ディレクトリからのを含めて 4 本の場合であるが，任意の数のストリームの場合でも同様に構成できる．また，このプログラムには open で得ら

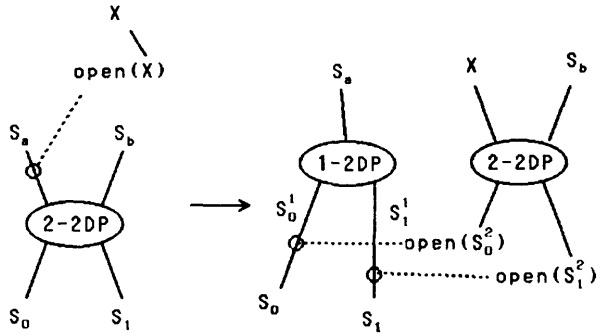

図 3 2-2DP の分割
Fig. 3 Division of a 2-2DP.

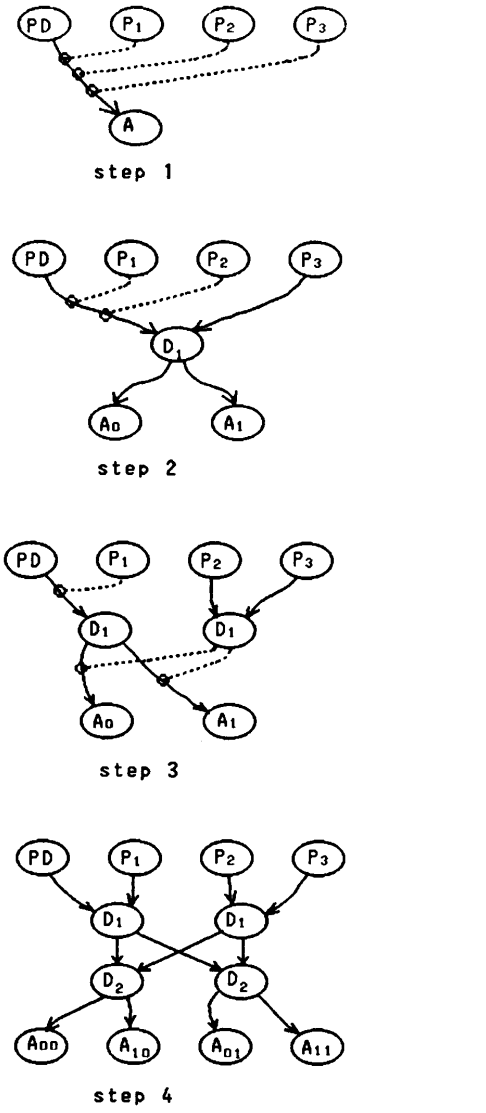

図 4 2-2DP によるネットワークの動的構成例
Fig. 4 An example of dynamic generation of a 2-2DP process network.

れたストリームを削除（ストリームを［］に具体化することで行う）した場合の処理も記述している．入力ストリームの削除は，2-2 DP の片方の入力ストリームが［］になるとそのプロセスを 1-2 DP とし，1-2 DP の入力ストリームが［］になるとそのプロセスを消滅させることにより，実現している．（ただし，open で得られたすべてのストリームが［］になると 1-2 DP と分割された ASP から成るバランスのとれた二進木になるが，いったん分割された ASP は再び1つには戻らない．）

4.4 ASP の性能解析

このように分散実現法では，情報資源を多くの ASP によって協調して保持することで並列度を高めるとともに，分割された各 ASP を binary n-cube ネットワークで結合し，特定のストリームへのメッセージの集中を避けている．このため分散実現法は，集中実現法より高効率になることが期待できる．しかし，①プロセッサ数，②プロセッサ間の通信コスト，③プロセス生成のコストなどにより，実際の効率は集中実現法より劣る場合もある．ここでは，処理系を

- 十分な数のプロセッサがあり，各プロセスには専用のプロセッサが割り当てられる．

と仮定した上で，ASP 1 個による集中実現法と ASP k 個による分散実現法におけるトランザクション処理の平均応答時間を待行列理論に基づいて見積る．いま，

- ASP を利用するプロセスが k 個 p_i $(i=1,\cdots,k)$ 存在し，各 p_i でトランザクションが単位時間当り λ 個ランダムに発生する．
- ASP でのトランザクション処理時間は平均 $E(t_a)$，変動係数 C_a の一般分布に従う．
- 2-2 DP でのトランザクション処理（述語 D_i の判定＋メッセージ長に比例した通信時間）は平均 $E(t_d)$，変動係数 C_d の一般分布に従う．
- ASP および 2-2 DP の入力側に無限長バッファがある．

と仮定すると，各 ASP および 2-2 DP は $M/G/1$ 待行列系[9]でモデル化でき，ネットワーク全体は $M/G/1$ 待行列網でモデル化できる．さらに簡単のため，

- トランザクションがランダムに与えられたとき，述語 D_i が真および偽となる確率はともに 1/2 である．
- $E(t_a)$, C_a は，集合 A の分割数 k によらず一定．（実際は k の増加とともに $E(t_a)$ の減少が期待できる．）
- k の値はちょうど2のべき乗になっている．

とする．$M/G/1$ 待行列の平均応答時間（ASP を利用するプロセス1個，ASP 1個）は，

$$\text{平均応答時間}=\text{平均サービス時間}+\text{平均待時間}$$
$$=E(t_a)+(1+C_a^2)\rho E(t_a)/2(1-\rho)$$

となる[9]．ただし，ρ は平均サービス時間 $E(t_a)$ とトランザクションの平均到着時間間隔 λ^{-1} との比，$\rho=\lambda E(t_a)$．

集中実現法の平均応答時間 $E(t_s)$ は，単位時間当り，$k\lambda$ 個のトランザクションが1個の ASP に集中するので，

$$E(t_s)=E(t_a)+(1+C_a^2)k\rho E(t_a)/2(1-k\rho) \quad (1)$$

となる．（ストリームのマージのコストは無視）

分散実現法の平均応答時間 $E(t_p)$ は，λ 個/時間のトランザクションが各 ASP および各 2-2 DP に到着し，2-2 DP のプロセス網全体のオーバヘッドは段数 $\lceil\log_2 k\rceil$ に比例するので，

$$E(t_p)=E(t_a)+(1+C_a^2)\rho E(t_a)/2(1-\rho)$$
$$+\lceil\log_2 k\rceil\{E(t_d)+(1+C_d^2)\rho' E(t_d)/2(1-\rho')\} \quad (2)$$

となる．（ただし，$\rho'=\lambda E(t_d)$）

いま，プロセス数 k 以外のパラメータを一定にし，k を増加させることによりトランザクション数を増加させてみる．（1）式より，集中実現法の応答時間が，ASP での待ち時間が $k/(1-k\rho)$ に比例して増加することが，（2）式より，分散実現法の応答時間が分配プロセスの段数 $\lceil\log_2 k\rceil$ に比例して増加することがわかる（**図 5**，ただし，便宜上 k は連続量として図示）．

一般に，台数 k が小さいうちは，集中処理の応答時間の増加は緩やかであり，分散処理による通信のオーバヘッドが大きい．台数 k が大きくなると，集中処理

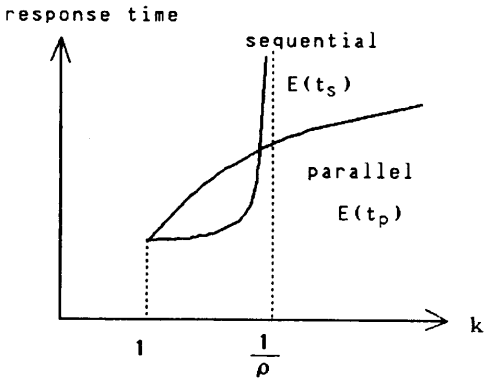

図 5 ASP の分散/集中実現法の性能
Fig. 5 Performance of parallel/sequential implementation of ASP.

の応答時間は急激に増加する．$k=\rho^{-1}$ では応答時間は無限大となり，事実上のシステムダウンとなる．しかし，本方式の分散実現法では，k の増加に応じて，動的にネットワークが拡大していくので，$\lceil \log_2 k \rceil$ に比例したわずかなオーバヘッドのみで，応答時間の増加は緩やかである．また，実際には k の増加に応じて A が分割され，$E(t_a)$ の減少が期待できるので，応答時間の増加はさらに緩やかとなる．

5. スケジューリング技法

本章では，複数のゴール群に分かれているゴールの簡単なスケジューリング技法を提案する．ここで示す技法は逐次マシン上での実現を念頭においているが，並列マシン上で実現する際にも有効である．

本論文の主旨である情報資源をゴール群として蓄えるプログラミング・スタイルは，記憶のためのゴールを大量に発生させることになる．また，記憶のためのゴールはサスペンド状態にある確率が高いという特徴がある．このため，ゴールのスケジューリングを工夫し効率化することが重要である．各システム・ゴール群 GS_i $(i\in N)$ は通常，デッドロック状態（そのゴール群のすべてのゴールがサスペンドしている状態）にあり，プロセス・ディレクトリからのメッセージが V_i を介して送られてはじめて実行される可能性が生じる．このようなゴール群の実行可能性を検出し，そのゴール群のすべてのゴールをはじめからスケジューリングの対象から除くことができれば，かなりの効率化が期待できる．以下でこの方法を示す．

CCL の逐次処理系の多くは，ゴールのスケジューリングのための待行列（以後，Queue と記す）を1つしか持っていない．しかし，ゴール群ごとの実行制御を行うために，

(1) ゴール群ごとに Queue を用意する．
(2) Queue 自体の待行列を用意し，複数の Queue の間での実行をスケジュールする．

こととする．このような Queue の待行列を群待行列と呼び，ゴール群の間でのスケジューリングを群スケジューリングと呼ぶ．

群スケジューリングを実現するため，各 GS_i $(i\in N)$ を4つ組 $R_i=\langle i, Q_i, V_i, E_i\rangle$ で管理する．ただし，i はゴール群の名前，Q_i は名前 i のゴール群の Queue，V_i は名前 i のゴール群とプロセス・ディレクトリとを結ぶストリーム，E_i は V_i を介して間接的に運ばれた変数の集合である．E_i については後で詳しく述べる．R_i をゴール群 i のレコードと呼び，$\{R_i|i\in N\}$ をゴール群レコードと呼ぶ．ゴール群レコードは，3章で述べたプロセス・テーブル（$\langle i, V_i\rangle$ の集合）を含んでいるため，ゴール群レコードを用意すれば，プロセス・テーブルは不要である．また，ゴール群レコードはスケジューラだけではなく，プロセス・ディレクトリからもアクセスされる．

準備として，ゴール群のオープンとクローズという概念を導入する．システム・ゴール群 GS_i がオープンしているとは，GS_i とメイン・ゴール群 GM との間に V_i 以外の共有変数を持つことと定義する．また，GS_i がクローズしているとは，GS_i がオープンしていないことと定義する．ゴール群 i がオープンしているか否かを検出するには，前述の E_i を使用する．E_i は初期状態を空とし，V_i を介して運ばれた変数を E_i に加え，そして，E_i に加えた変数が基礎項 (ground term) に具体化した場合その変数を取り除くこととする．例えば，V_i を介してメッセージ $m(X)$ が送られると，E_i を $E_i\cup(X)$ に置き換える（この更新はプロセス・ディレクトリが行う）．また，$X\in E_i$ が $f(X_1, X_2)$ に具体化したときは E_i を $E_i-\{X\}\cup\{X_1, X_2\}$ で置き換えることとする（この更新は後で述べる群スケジューラが行う）．ゴール群のオープンとクローズは，後で述べる群スケジュールを実現するためのもので，ユーザから制御したり，観測したりすることは，基本的にできないことに注意．

クローズしている GS_i と GM とは V_i 以外に共有変数を持たないため，GS_i を実行することなく GM は空節に至ることが可能である．したがって，オープンしているゴール群のみをスケジュールすることにする．このとき，群スケジュールのアルゴリズムは次のようにすることができる．

1) 以下をメイン・ゴール群の Queue が空になるまで繰り返す．
1.1) メイン・ゴール群の Queue を n 段展開する．
1.2) システム・ゴール群の Queue のうち，E_i が空でないものを n 段展開する．

ただし，$n\geqq 1$ とする．また，Queue を n 段展開するとは，Queue に含まれるすべてのゴールを n 段展開することである．ゴール A を n 段展開するとは，A が展開されて，$B_1, \cdots, B_m$ となったとき，各 B_j $(j=1, \cdots, m)$ を $n-1$ 段展開することである．ただし，A が展開できない場合は何もしない．この Queue を n 段展開するアルゴリズムには様々なもの

が考えられるが，これは1つのゴール群のみに関する展開手続きであるため，従来の CCL 処理系で提案された方式がそのまま使える．したがって，本論文では省略する．

なお，本章で示した群スケジューリングは，既に我々が製作した GHC 処理系[10]を基に，実際に directory 述語を実現し，その動作を確認した[11]．

6. おわりに

本論文は，情報資源を CCL のプログラム節あるいはゴールの引数データとして保持するのではなく，ゴール群すなわちプロセスの集合として保持し管理するという新しい考え方に基づいている．ゴール群の管理機構を持つ処理系と，応用システムからゴール群にアクセスする手段としてプロセス・ディレクトリを提案し，仕様と実現法の概略を述べた．応用システム側で定義したゴール群（情報資源）はプロセス・ディレクトリを介して処理系に登録しておくことで，応用システム終了後にもそのゴール群に名前でアクセスできる．プロセス・ファイルの例からわかるように通常の情報資源あるいは入出力資源はゴール群として自然に保持することができる．また，ASP の例で示したプログラミング技法を用いれば特定のストリームへのメッセージの集中によるボトルネックを避けることができる．

以上の方式により，2.1 節で述べた5つの問題点が解決できた．すなわち，

(A1) directory 述語自身は非論理的な組み込み述語であるが，応用システムは完全に論理的に記述でき，プログラムの意味の複雑性を生じない．

(A2) システム・ゴール群として管理されているデータベースへストリームを介して検索メッセージを送ることにより，CCL の実行過程とデータベースの検索とが調和できる．

(B1) 応用システムの終了後にも情報資源がプロセス・ディレクトリに登録されたプロセス（システム・ゴール群）の形で存続する．

(B2) ASP で例示した技法により，資源を動的に分散させ，トランザクションの集中を避け得る．

(B3) プロセス・ディレクトリでゴール群を一括管理するので，分散した情報資源に対する一括した処理（バックアップや削除）が容易である．

本論文ではさらに，ゴール群管理の際に生じるスケジューリングの問題に対して，群スケジュール，ゴール群のオープンとクローズという概念を導入し，基本的な視点を確立した．本論文で提案した機構は既に GHC 上で実現され，簡単な OS 風プログラムが動いている．しかし，今後以下のような課題が残されている．

① エラー処理について検討すること．情報資源を保持しているゴール群はエラーにより破壊される可能性があるが，現状では，3.3 節で示した copy メッセージや frozen_goals メッセージにより，あらかじめゴール群のバックアップをとることが唯一の対応策である．

② 効率のよいゴールのスケジューリング技法を検討すること．5章で示した方法は，わかりやすさを第一に提案したものであるが，実用的なものにするにはさらに改良する必要がある．また，大規模の情報資源をゴールとして保持する場合，一部のゴールは二次記憶上にスケジュールすることを検討する必要がある．

③ 性能を評価すること．本方式は大規模で複雑な多数の情報資源を統一的かつ単純な方法で管理したいとき有効性が期待できる．特に，4.4 節で解析したように同一の資源に対して潜在的に多くのプロセスが高頻度でアクセスするような応用には極めて有効である．しかし，比較的単純な応用によっては逆オーバヘッドが大きい可能性がある．したがって，多くの例に応用し，本方式が有効な応用の範囲を明らかにする必要がある．（筆者らの試作したシステムは逐次処理系上のシステムであるため，物理的に並列処理を行ったときの性能評価はできなかった．）

以上のように，本方式はすべての面で従来方式より優れているわけでないし，またすべての問題点を解決したわけではない．しかし，特に複雑なソフトウェアを CCL で開発する際に生じると思われるいくつかの重要な問題点を解決し，その点では従来方式より優れていると考える．したがって，本方式は CCL による情報資源管理方式を決定する際に考慮されるべきいくつかの重要な選択枝の1つとなり得るものである．

謝辞 本研究に際し，有益なコメントをいただいた ICOT の上田和紀氏に深謝する．

参考文献

1) Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, Research

Report Doc 84/4, Dept. of Computing, Imperial College London (1984).
2) Shapiro, E. Y.: A Subset of Concurrent Prolog and Its Interpreter, TR-003, ICOT (1983).
3) Ueda, K.: Guarded Horn Clauses, *LNCS-221*, Springer-Verlag, Berlin, Heidelberg (1986).
4) Ueda, K.: Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, TR-208, ICOT (1986).
5) 上田和紀：GHC の基本，並列論理型言語 GHC とその応用，pp 33-66, 共立出版，東京 (1987).
6) 上田和紀：Prolog 上の GHC 処理系，並列論理型言語 GHC とその応用，pp. 217-270, 共立出版，東京 (1987).
7) Lloyd, J. W.: Foundation of Logic Programming, Springer-Verlag (1984). (佐藤雅彦，森下真一訳：論理プログラミングの基礎，産業図書，東京 (1987))
8) Shapiro, E. and Takeuchi, A.: Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, Vol. 1, No. 1, pp. 25-48 (1983).
9) 國友義久：オンラインネットワークの構造的設計，近代科学社，東京 (1978).
10) 藤村 考，栗原正仁，加地郁夫：LISP 上の GHC コンパイラ，情報処理学会論文誌，Vol. 28, No. 7, pp. 776-785 (1987).
11) 藤村 考，栗原正仁，加地郁夫：並列論理型言語に基づく資源管理の一方式，日本ソフトウェア科学会第 4 回大会論文集，No. B-5-2, pp. 451-454 (1987).

付 録

```
type (Is, Os) :-true|buffer (50, Is, Is1),
    reader (Is, Is1, S), writer (S, Os).
buffer (N, IH, IT) :-N>0|
    IH=[_|IH1], N1 :=N−1, buffer (N1, IH1, IT).
buffer (N, IH, IT) :-N= :=0|IT=IH.
reader ([X|IH], IT, Os) :-X≠end_of_file|
    Os=[X|Os1], IT=[_|IT1], reader (IH, IT1,
        Os1).
reader ([X|IH], IT, Os) :-X=end_of_file|
    Os=[ ], IT=[ ].
writer ([X|S], Os) :-true|
    Os=[write (X), nl|Os1], writer (S, Os1).
writer ([ ], Os) :−true|Os=[ ].
```

(昭和 63 年 2 月 5 月受付)
(昭和 63 年 9 月 5 日採録)

藤村 考 (正会員)

昭和 36 年生．昭和 59 年北海道大学工学部電気工学科卒業．昭和 61 年同大学院修士課程修了．現在，同情報工学科博士課程在学中．論理型プログラミングおよび協調計算アルゴリズムに興味を持つ．日本ソフトウェア科学会会員．

栗原 正仁 (正会員)

昭和 30 年生．昭和 55 年北海道大学大学院工学研究科情報工学専攻修士課程修了．同年同大学工学部助手．現在，同情報工学科助手．工学博士．システム工学，知識工学の研究に従事．電子情報通信学会，電気学会，日本 OR 学会，日本シミュレーション学会，日本ソフトウェア科学会，IEEE 各会員．

加地 郁夫 (正会員)

大正 15 年生．昭和 26 年北海道大学理学部数学科卒業．同年北海道大学応用電気研究所助手，40 年北海道大学工学部助教授，44 年同教授，現在，同情報工学科システム工学講座教授．システム工学，応用数学の研究に従事．日本 OR 学会，日本物理学会，電子情報通信学会，計測自動制御学会，日本原子力学会各会員．