ICOT Technical Report: TR-373

TR-373

並列論理型言語KL1の実現方式と並列OSの記述

宮崎敏彦、内田俊一

May, 1988

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

あらまし

本稿ではICOTで研究開発中の並列論理型言語とその処理系及びPIMOSと呼ぶオペレーティング・システムの開発の現状に付いて報告する。開発中の言語はKL1と呼ばれストリーム通信を基本としたAND並列の論理型言語であり、Guarded Horn Clauses(GHC)を核として設計された。KL1の特徴は、荘園と呼ぶプログラムの実行制御機能と、実行優先度指定や負荷分散指定にある。KL1の処理系については、分散ユニフィケーションや荘園の実現方式,構造体の最適化及びガーベジ・コレクションに付いて述べる。PIMOSに関しては、フィルタの概念を使ったOSの保護方式を中心に説明する。また、並列処理システムの全体像の把握を可能とするために、これら言語やOSの土台となるマシンアーキテクチャの特徴に付いても簡単な説明を加えることとする。

1　はじめに

並列処理システムの研究開発には、並列マシンやそのマシン上のプログラミング言語の研究だけでなく、並列マシンを効率良く利用する為のオペレーティング・システム (OS) の開発や、応用ソフトウェアが必要とする並列アルゴリズムの開発なども含まれなければならない。しかしこれらの研究テーマをそれぞれ個別に進めるのでは、全体の見通しが悪くなり、例えばソフトウェアが要求する機能をハードウェア側が予測し開発するといったことは非常に難しくなる。そこで我々はKL1と呼ぶ並列プログラミング言語を並列処理システムの中核に置き、そこからソフトウェア及びハードウェアの研究を展開するというような方針を取っている。

ここで述べる並列プログラミング言語KL1に対して要求される性質としては、次のものが挙げられる。

- 同期のタイミングなど、言い換えればデータの依存関係と計算の実行順を極力ユーザに意識させない。
- 計算モデルや言語のセマンティックスが簡明で

あり、デバッグやプログラム変換等の見通しが良い。

- 並列性を内在しているアルゴリズムをその言語の仕様に従って自然に記述した場合に、作られたプログラムが十分な並列性を有する。
- システム・プログラムが記述できる、すなわちその言語自身で書かれたプログラムの実行を観測し制御できるようなメタ機能を有する。

KL1は、論理プログラミングの枠組を利用して設計されたGuarded Horn Clauses(以下GHC)[Ueda85][GHC]を基に、幾つかの機能拡張を施した並列論理型言語である。GHCに対して主に拡張された部分は、並列マシン上でのプログラムの実行制御機能であり、OSを記述する為のメタレベルの制御機構である荘園機能や実行優先順位指定,負荷分散指定の機能などがこれにあたる。

KL1のプログラムはKL1-B[Kimura87]と呼ばれる抽象機械命令にコンパイルされ実行される。KL1-BはWAM[Warren83]を基礎にしてKL1向けに種々の変更を加えた命令系であり、ICOTで開発中の幾つかの並列マシンは全てこのKL1-Bを中間言語ある

いは機械語として用いている。

図.1挿入

KL1を中核とする、並列OS(PIMOS)や並列マシンの関連を示すと図.1のようになる。ICOTで現在開発中の並列マシンにはマルチPSI[Taki86]とPIM(Parallel Inference Machine)[Goto86]がある。マルチPSIは、逐次型推論マシンPSI(Personal Sequential Inference Machine)[Taki84]のCPUを小型化し[Nakashima87]、これを最大64台までネットワークで結合したマルチ・プロセッサシステムであり、KL1の実装方式や並列ソフトウェアの研究の為のツールとすることを目的として早期に実現したマシンである。一方PIMは、並列プログラムの通信の局所性をより有効に利用する為に、8台程度のプロセッサを共有メモリで結合したクラスタとそのクラスタ間をネットワークで結合した階層構造のマシンで、ハードウェアによる高速化に重点を置いたマシンである。

PIMOSは並列マシンを円滑にかつ効率良く利用する為のOSであるが、単なる実験的なシステムでは

なく、その上でのアプリケーションの研究開発に十分耐えうる実用的なOSを目指している。また、PIMOSのような比較的規模の大きなプログラムをKL1で記述することによってKL1自身の記述力の評価を行うとともに、並列プログラミングのパラダイムを研究することも合わせて意図している。

以下、本論文ではKL1とPIMOSについて解説し、並列処理システムの研究において重要な研究課題とその実装例について紹介する。

2 並列論理型言語KL1

2.1 KL1の概要

KL1は、ストリーム通信を基礎としたAND並列の論理型言語GHCを核に、後で述べるOSの記述及びアプリケーションの記述のための幾つかの機能拡張を施して設計された言語であり、ユニフィケーションやゴールの書き換え規則, クローズの形式など基本的な部分はGHCとほとんど同じである。

KL1のプログラムはGHCと同様ガード付きホーン節の集まりであり、各節は次の形式で表される。

$$H \ :- \ G_1, ..., G_m \mid B_1, ..., B_n. \quad (m \geq 0, \ n \geq 0)$$

ここで、H 及び $G_1, \ldots, G_m$ はガード部、$B_1, \ldots, B_n$ はボディ部と呼ばれ、'|' はコミット演算子と呼ばれる。また H だけをヘッド部、G_i や B_j をゴールとも呼ぶ。ヘッドやゴールは原子論理式であり述語とも呼ばれる。

FGHC では節の中のゴールは並列に実行してよく、またゴールの実行 (リダクション) に使い得る各候補節 (同一の述語をヘッドに持つ節の集まり) は以下の規則の下に並列にリダクションを試みてよい。

(1) ガード部の実行では呼び出し側の変数を具体化してはならない。もしユニフィケーションの為に具体化が必要な場合には、そのユニフィケーションは中断 (suspend) する。しかし、他のゴールの実行によって行われたユニフィケーションで変数の具体化が行われれば (すなわち、このガード部におけるユニフィケーションでは具体化が必要なくなれば) 中断していたユニフィケーションは再開 (resume) される。

(2) ボディ部ではガード部の変数を具体化しても良いが、それはその節が選ばれた場合だけである。

(3) 節の選択は、候補節のうち最初にガード部の実

行に成功した節のみが選ばれる。

先にも述べたようにKL1ではこのような並列論理型言語の基本メカニズムの他にOS記述のための諸機能が付加されている。その主なものは、荘園と呼ばれる実行制御を行う為の機能とプラグマと呼ばれる優先順位及び負荷分散の指定機能である。

2.2 荘園

荘園とはKL1プログラムの実行管理や資源の管理を行う為の基本機能であり、以下で述べるコントロール・ストリームやレポート・ストリームを通じてその内部のゴール群の実行を制御/観測できる実行管理単位である。図.2に荘園の概念図を示す。荘園は次に示すような組込述語 execute によって生成される。

```
execute(Goal, Control, Report, Tag)
```

ここで Goal は荘園内で実行すべきゴールを意味する。(荘園内で実行されているゴールはその荘園に属すとも言う。また、あるゴールの実行によって新たに生成されたゴールは全て同じ荘園に属すことになる。)

図.2挿入

Control は生成した荘園を外部から制御する為のストリーム（コントロール・ストリーム）である。コントロール・ストリームを通じて制御できる項目には:

実行の制御: 実行の中断（stop）や再開（start）あるいは放棄（abort）が有り、どの命令もその荘園内の全てのゴールと子孫荘園にも伝えられる。

資源の割り当て: その荘園内で消費して良い資源量の最大値を与える。ここで言う資源量とは、その荘園内で消費した実行時間とメモリの量を適当に近似できるような単位量の和である。KL1ではこのような計算機資源の消費量は処理系レベルで管理される。

資源消費状況の問い合わせ: その荘園内で、その時点までに消費されている資源量を問い合わせる。

が有り、それぞれに対応したコマンドを、制御したい荘園のコントロール・ストリームに流す（ユニファ

イする）ことによってユーザはその記憶を制御することができる。

Report は実行中に生じた事象を外部に報告するためのストリーム（レポート・ストリーム）である。レポート・ストリームには以下に示すような情報が報告される。

- 実行の終了。
- 組込述語における入力データのタイプの誤りなど、組込述語で発生する例外。
- ユニフィケーションやリダクションの失敗[1]
- 積極的な例外発生機構を使った例外。（KL1 にはソフト的に例外を発生させる機構として組込述語 raise が用意されている。）

レポート・ストリームに報告される例外は、一般に次のような項の形をしている。

exception(例外の種別，例外を起こしたゴール，NewGoal)

[1] KL1 とは異なり、GHC ではこのような失敗は例外とはならず所謂実行の失敗として扱われ、連接関係 (and 関係) にある他の全てのゴールの実行は放棄される。

ここで NewGoal は例外処理用の未束縛変数であり、ユーザはこの変数に、発生した例外に対処する処理をユニファイすることによって、例外を起こした荘園内の実行を続けることができる。この機能を使うと例えば、通常KL1では失敗するようなユニフィケーションも NewGoal に適当なゴールを与えることによって成功させるといった、言語仕様の拡張を行うことも可能である。

荘園はその荘園内で再び上述の組込述語 execute を実行することにより何段にもネストして良い。（このように荘園がネストした場合、内側の荘園をその外の荘園に対して子孫荘園であると言う。）荘園がネストした場合、外の荘園に対する制御はその内側の全ての荘園に伝播される。荘園の実行は基本的には Goal で指定されたゴールの実行が終了することによって終了する。荘園がネストした場合も同様に、所属している全ての子孫荘園とゴールが終了することによって終了する。

Tag はこの荘園のレベルで処理する例外の種類を指定する為に用いられる。実行中に例外が発生すると、その例外の種類に応じて例外が発生した荘園から外側の荘園に向かって順に荘園のタグが調べら

れ、最初に適合した荘園のレポート・ストリームにその例外情報が報告される。

2.3 プラグマ

先に述べたように、KL1では現在のところプラグマとして記述できるものにゴールの実行優先順位と負荷分散の為の指定がある。

2.3.1 実行優先順位指定

荘園にはコントロール・ストリームを通じて、その荘園内で実行されるゴールの優先順位の上限と下限を与える。荘園内で生成されるゴールの優先順位は、この上限と下限の範囲内で相対的に指定することができるわけである。

ゴールの実行優先順位の指定方法には、所属荘園内割合指定と所属荘園内自己相対指定の2種が用意されている。前者は、所属している荘園の上限と下限の間のどのあたりで実行するかを割合で指定する方法であり、後者はその節を呼び出した親ゴールの持つ優先順位と所属している荘園の上限あるいは下限との差に対する割合で指定する方法である。このように相対値で指定することによって、優先度の幅

の異なるマシンや、上下限の異なる範囲内でもプログラムを変更すること無く実行することができる。

現在開発中のマルチPSIではこの優先度の幅を2^{12}程度用意しており、単にOSの記述だけでなくアプリケーション・プログラムを記述する上でも優先順位を使ったプログラミングが可能なように考慮されている。

2.3.2 負荷分散指定

マルチPSIやPIMのようなMIMD型の並列計算機では、その性能を最大限に引き出す為に、各プロセッサ(PE)の負荷の平均化は必須である。現在のKL1では、この為の最低限の機能として、実行して欲しいPEのPE番号をゴール単位に指定することが可能になっている。また、このPE番号の指定を実行時に与えることもできるので、適当な負荷分散の戦略をKL1自身で記述し実際の計算の進み具合を見ながら適度な分散指定を決めることも可能である。あるいは、実際の並列マシンの結合のトポロジーとは独立に、KL1レベルで仮想的なトポロジーを設定し実験するといったことも可能なようになっている。

負荷分散指定の方式としては、このような、直接プロセッサ番号を指定する方式のほかに、各ゴールが

それぞれ自分の進む方向を持ちトーラス状の空間内を動き回るタートル方式 [Shapiro84] やプログラムのローカリティを生かしつつ自動的な負荷分散を行うことを狙った平面分割方式[Chikayama86] などが提案されている。しかしこのような負荷分散の研究は、これまで本格的に行われたことのない分野であり現在は基本的な機能を実装するに留め、他の種々の指定方式はプリプロセサで現在のものに変換する方針を取っている。

3 KL1処理系の実現方式

KL1の処理系を実際の並列マシン上に実現するためには多くの研究課題を解決して行かなければならない。ここでは、現在検討を進めているマルチPSI上でのKL1の実現方式について、全体的な実行イメージと実現上の要点となる基本的なアイディアについて述べる。

3.1 対象とする並列マシン

ここで述べる実現方式が対象としている並列マシン、すなわちマルチPSIのアーキテクチャ上の性質

を示すと以下のようになる。（これらの性質は、PIMの各クラスタを一つのPEと見した場合にも当てまるものである。）

- 各プロセッシング・エレメント（PE）はローカルメモリを持つ。
- PE同士はネットワークで接続される。
- PE間では共有メモリを持たず、データ構造（もちろん論理変数もこれに含まれる）を共有する場合には別PE内部を指すポインタが生じる。
- PE内のアドレス系とPE間にまたがるアドレス系は異なる。即ち、PEを渡るポインタは、各PEの出入口でアドレス系の変換が施される。

別PEを指すポインタを外部参照ポインタ（exref）と呼ぶ。外部参照ポインタは参照先のPE番号と、そのPE内にあるPE間アドレスの変換テーブル（輸出表と呼ぶ）に対するオフセットで表現される。

3.2 実行イメージ

KL1にはゴール間の並列性や候補節間のリダクションにおける並列性，ユニフィケーションにおける並列

性など種々の並列性が内在されているが、我々にこのうちゴール間の並列性に着目した実現方式を採用している。このため実行の最小単位は基本的にゴールであり、個々のPEに割り付ける単位もゴール単位である。

図.3挿入

マルチPSIにおける個々のPE内の処理の概要を図.3に示す。各PE内にはゴール・スタックと呼ぶ優先順位付けされた複数本のスタックがあり実行可能状態に有るゴールがその優先順位に従って繋がれている。スケジューラはそのスタックの中からその時点で最も優先順位の高いゴールを取り出し順次実行して行く。

ゴールの実行は、まず候補節のガード部を順次実行し、いずれかの節が選択されたならば、その節のボディ部のゴールをゴール・スタックにプッシュする。もしこの時、ボディ部のゴールに他のPEに対する割り付け指定が有ったならばそのゴールは対応するPEに送られ、送られた先のゴール・スタックにプッシュ

される。どの候補節も選ばれなかった場合、もし中断した節が一つでも有ったならば呼び出し元のゴールの中断として扱われ、中断の原因となった変数の具体化を待つ為に、原因となった変数に付加されている中断スタックにプッシュされる。また全ての節が失敗であった場合には呼び出し元のゴールの失敗として扱われ、そのゴールが所属している述語のレポート・ストリームに例外として報告される。

中断スタックはガード部の同期メカニズム（すなわち変数の値が決まるまで待つメカニズム）を実現する為の機構であり、中断の原因となった未束縛変数に必要に応じて付加される。中断スタックを持つ変数がボディ部のユニフィケーションによって具体化されると、中断スタック中のゴールは全てゴール・スタックに移され、再びスケジュールされるのを待つ。

3.3 分散ユニフィケーション

KL1 には実現上ガード部とボディ部の２種類のユニフィケーションが存在する。ガード部のユニフィケーションは値の検査であり、ボディ部のそれは基本的に値の代入である。以下では分散ユニフィケーションの実現に特徴的な外部参照ポインタを扱うユニフィケー

ションの実現方式について説明する。

3.3.1 ガード部のユニフィケーション

ガード部で、ユニファイする相手が外部参照ポインタである場合には、参照先のPEに対して対応する変数の値の読み出し要求メッセージ(readメッセージ)を送り、値が読み出されるまでそのガード部のユニフィケーションは中断する。例えば図.4で、変数XがPE_jの変数Aに対する外部参照ポインタであった場合、ゴール p(X) の実行は中断され、PE_jに変数Aの値の読み出し要求を送る。この要求メッセージには、返信先として変数Xへの外部参照が付けられ、中断したゴール p(X) は要求の返答待ちの為、変数Xの中断スタックに積まれる。要求を受け取ったPE_jにおける処理は変数Aの状態によって次の2種に分けられる。すなわち、変数Aが既になんらかの値に束縛されていれば直ちに返信先にその値を送り、まだ未束縛であれば、束縛された際に返信するため、返信用の特殊なゴールをAの中断スタックに積む。

図.4挿入

値を返信する際に、その値がある程度大きな構造体であるような場合には、まずその構造体へのポインタを返信先PEに送る。これは、送った先のPEで結局その構造体が変われなかった場合に無駄な構造体のコピーができてしまうからであり、またPE間でのデータの送受のコストが高い為でもある。

3.3.2 ボディ部のユニフィケーション

ボディ部における外部参照ポインタとのユニフィケーションでは、ガード部のように値の読み出し要求を出すのではなく、参照先にユニフィケーションの依頼メッセージ（unify メッセージ）を送る。これは、他PEに対して負荷分散指定されたゴールを送ることとほとんど同じと考えて良い。

未束縛変数とのユニフィケーションは少し注意が必要である。例えば、2台のプロセッサ PE_i と PE_j で、PE_i に未束縛変数 X，PE_j に同じく未束縛変数 Y が有ったとしよう。さらに PE_i に PE_j 中の変数 Y を指す外部参照ポインタ Y_{exref} が存在し，PE_j には PE_i 中の変数 X を指す外部参照ポインタ X_{exref} が有るとする。この時、PE_i では X と Y_{exref} のユニフィケーションが実行され、PE_j では Y と X_{exref} のユニフィ

ケーションが実行されたとする。この場合、単純に外部参照ポインタを未束縛変数に代入してしまうと、PE_iとPE_jの間で外部参照ポインタを経由したループができてしまい問題となる。

そこで、外部参照ポインタを張る際に:

- 未束縛変数から他のPEを参照する場合には常にPE番号の小さい方から大きい方に参照ポインタを張る。

という方向付けの規則を定めることによってこの問題を解決している。先の例では、PE番号が $PE_i < PE_j$ の場合、PE_jのユニフィケーションはPE_iに送られ、PE_iのユニフィケーションはそのまま未束縛変数Xに外部参照ポインタを代入して成功する。

3.4 荘園の実現

荘園の主な機能としては、荘園内のゴールの実行制御と実行終了判定機能があるが、これを例に分散環境下での実現上の問題点と実現方式について述べる。

実行制御機能を実現する為に、各ゴールは、自身が所属している荘園の荘園レコードへのポインタを

持ち、荘園レコードにはその荘園の現在の実行状態を表すフラグが置かれる。また荘園はコントロール・ストリームを通じてstartやstop,abortなどの実行制御コマンドを受け取ると、自身の実行状態フラグをそれに応じて変更する。一方ゴールがスケジュールされるとそのゴールが所属する荘園の実行状態フラグを検査し、状態に応じて実行/中断/ 放棄する。

荘園内のゴールが複数のPEに分散された場合には、個々のPE でゴールがスケジュールされる度に荘園レコードの存在するPEに対して状態を問い合わせる必要が生じるが、これは通信のオーバヘッドを増加させ問題となる。そこで、それぞれのPEに荘園レコードのキャッシュ（これを里親と呼ぶ）を置きPE間の通信を減らす工夫をしている。（図.5参照）このため、荘園の状態が変更された場合には、里親のあるPEに対して状態の変更を通知するメッセージが送られる。

図.5挿入

終了判定機能は、基本的には荘園内で実行されているゴール及び子孫荘園の数を里親ごとに管理する

ことによって実現される。具体的には、その荘園に属す全てのゴールの実行が終了すると消滅し、荘園にその旨を伝えるメッセージを送る。荘園は基本的には全ての里親からの消滅メッセージを受け取ることによって終了する。しかし、マルチPSIのようにネットワークで結合された並列マシンの場合、メッセージの送信から到着までには有限だが一定ではない時間がかかるため、例えば負荷分散の為に投げ出されたゴールが目的のPEに到着するよりも早く里親の消滅メッセージが荘園のあるPEに到着するということも有り得る。言い換えれば、荘園は全ての里親から消滅メッセージを受信しても、ネットワーク上をゴールが飛んでいるかもしれないので、荘園内の全てのゴールが終了したと断定することはできない。

このような問題を解決する方法として、荘園や里親及びネットワークを流れるメッセージに重みを付けたWTC方式[Rokusawa88]を用いた荘園の終了判定機能を用いている。WTC方式とは重み付き参照カウント方式[Bevan87][Watson87]を応用した分散環境における終了判定法である。

WTC方式では、荘園に負の重みが与えられ、里親及びメッセージには正の重みが与えられる。そして、

全ての重みに次の関係が成り立つように管理される。

荘園の重み　＋ Σ 皇親の重み　＋ Σ メッセージの重み　＝ 0

例えばある皇親から他の荘にゴールを送る場合には、皇親は自身の重みから適当な量だけ分割しゴール送出メッセージに付けて送る。また重み付きのメッセージを受け取った皇親では、その重みを自身の重みに加える。皇親消滅メッセージには皇親が消滅する際に持っていた重みを全て付けて荘園に送る。消滅メッセージを受け取った荘園では皇親と同様、自身の重みに運ばれてきた重みを加える。加えたことによって荘園の重みがゼロに成れば、全ゴールの終了であり荘園は終了して良い。

3.5　参照カウントを用いた構造体の実装

KL1 のような論理型言語ではベクタなどの構造体を扱う場合、例えばあるベクタのｉ番目の要素だけを別の値に変えたようなベクタを得る為には、同じ大きさの別のベクタを用意してｉ番目の要素以外をコピーするといったことが必要になる。しかしこのような方法では、要素を変更する度に新たなベクタを作ることとなり問題がある。一方、KL1 で記述され

る実際のプログラムを考えた場合、要素が頻繁に変更されるようなベクタは参照数が1であることがほとんどである。そこで、更新前のベクタが明らかに必要でないような場合、すなわち参照数が1であるようなベクタに対する更新は、元のベクタ本体を直接書き換え、コピーをしなくてすむような最適化を行いたい。このために MRB(Multiple Reference Bit) 方式[Chikayama87] を考案し用いることとした。

MRB 方式は、基本的にはポインタ側に参照数が1か2以上かを区別する1ビットのフラグを設けた非常に簡略化された参照カウント方式である。 MRB 方式が他の通常の参照カウント方式と大きく異なる点は:

- 参照数が一度でも2以上になると参照数の管理は行わない。
- 基本的にポインタ側に参照カウンタを持っている為、変数とのユニフィケーションのようにポインタを渡すだけの操作ではデータ・オブジェクトにアクセスする必要がない。
- 未束縛変数を指すポインタの MRB の意味を、他のデータ・オブジェクトに対する場合と変えて

ある。これに、一般に未束縛変数が参照数２以上で意味があるからである。（参照数が１の場合、その変数が具体化されても値を見るパスが無い。）

である。MRB 方式はこのように参照数が１であるような構造体の更新操作を効率良くするだけでなく、リスト構造を使ったストリーム通信の最適化などにも利用されている。

3.6　ガーベジ・コレクション

マルチ PSI を対象とする KL1 処理系ではガーベジ・コレクション（GC）を PE 内 GC と PE 間 GC に分けて考えている。（PE 内と PE 間でアドレス系が異なるのは主にこの為である。）GC は実用的処理系実現のためには必須の機能であり、その効率化には多くの研究すべき課題がある。

KL1 のような言語は、その実行に際しメモリを多く消費する傾向が強く、 PE の処理性能が上がった場合には、当然のことながらメモリの消費速度も増大する。一方、一括 GC の処理はメモリ全体に対する処理であり、一般にメモリアクセスの局所性が少なくキャッシュ効率の悪化による性能低下が予想される。

このことから、PEの台数が多くなると、一括GCの処理時間がKL1の実行速度に及ぼす影響は大きくなり実時間GCの実装が必須となる。このため、PE内GCには、先のMRB方式を使った実時間GCとKL1の実行を停止して行う一括GCが実装されている。

PE間GCは、他のPEから参照されていることによってPE内GC後も残ってしまうゴミを回収するためのGCであり、PE内GCと同様、実時間GCと一括GCが考えられている。PE間の実時間GCは3.4章でも触れたWRC方式の応用であるWEC方式[Ichiyoshi88]を用いている。WEC方式では、輸出表エントリと外部参照ポインタに重みを付けた重み付き参照カウント方式である。輸出表エントリと外部参照ポインタの重みには次の関係が成り立つ。

輸出表エントリの重み ＝

Σ そのエントリを指す外部参照ポインタの重み

4　KL1 による OS の記述

4.1　PIMOS の概要

PIMOS はシングル・ユーザ，マルチ・タスクの機能を提供する集中単一型の　OS であり、複数の PE 上の独立した OS が協調しながら処理を進めるような分散 OS ではない。 PIMOS で言うタスクとは、資源管理や実行管理を行う単位であり、主に荘園の機能を使って実現される。以下では PIMOS に関する話題の内、 OS の保護の問題を取り上げ、その基本方針を中心に説明するとともに、 KL1 では言語レベルに特別な保護の為の機構を導入することなくこの問題を解決することができることを示す。

4.2　OS の保護

4.2.1　問題点

KL1 では、ユーザ・タスクと OS の間の通信は共有変数を介して行われる。しかし、単純にユーザ・タスクと OS が直接変数を共有すると幾つかの問題が発生する。

タスクは荘園機能を用いて実現されるが、次のプログラムは荘園を用いて一つのユーザ・タスクを生

示した例である。（ execute は述語生成の為の組込述語。）

```
:- pimos(Request, Control, Report),
   execute(user(Request), Control, Report, TAG).
```

このプログラムでは、ユーザは共有変数 Request を介してPIMOSと通信する。また、PIMOSはコントロール・ストリーム Control を通じてユーザ・タスクを制御し、レポート・ストリーム Report によってタスクの内部状態を知ることが出来る。

ここで、ユーザがPIMOSに対して時間の問い合わせをしたとしよう。時間を問い合わせる為には、通信経路である Request に問い合わせ用のコマンド what_time(Time) を流せば良いとする。

```
user(Request) :- true |
    Request = [what_time(Time)|NewReq],  % (1)
    user(NewReq).
```

一方、コマンドを受け取ったPIMOS側では時間を調べ[2]返信用変数 Time にその値をユニファイすることによってユーザに時間を知らせる。

[2]時間をどのようにして調べるかはここでは省略させて頂く。

```
pimos([what_time(Time)|NewRequest]) :-
    true |
        ....
    Time = '1時24分',                    % (2)
    pimos(NewRequest).
```

しかしこの共有変数を使った単純な通信方法には次のような2つの問題点がある。

問題点ー1: ユーザが、返信用の変数にOS側でユニファイするよりも早く別の値をユニファイしてしまった場合、OS側のユニフィケーションが失敗してしまう。例えば先のプログラムで、ユーザがPIMOS側の上記(2)のユニフィケーションよりも先に変数 Time に '1時24分' 以外の値をユニファイしてしまった場合、(2)のユニフィケーションは失敗してしまう。

問題点ー2: ユーザ側で具体化すべき値を具体化せずにOSに渡してしまった場合、OS側ではその値を待つことになるが、ユーザが結局具体化しなかった場合には、OS側にゴミのゴールが残ってしまう。例えば、(1)のユニフィケーションで:

```
Request = [_|NewReq]
```

としてしまうと

```
pimos([what_time(Time)|NewRequest])
```

の節は中断したままになってしまう。

上記 問題点－１ を解決する手段として KL1 に『ユニファイしようとしている変数が未束縛であることを確認した後に直ちにその節を選択し値を代入する』といった test and set 命令を導入し、ユニフィケーションが確実に成功することを保証した上で値を代入出来れば良い。しかし、このためには任意のプロセッサの任意のメモリ番地をロックする機能が必要になり、このような機構は並列マシンの場合、複数プロセッサからの同時ロックによるデッドロックの危険をはらんでいる。このことからできるだけ新たな機能を KL1 に導入せずに、現在のユニフィケーションのセマンティックスの中で OS の保護の問題を解決することが望ましく、以下のようなフィルタを用いた方式をとることとした。

4.2.2　フィルタによる OS の保護

問題点－１ では、ユーザが値を代入するかも知れ

たい変数にOS側でも値をユニファイしなければならなかったことが原因として考えられる。逆に言えば、ユーザと通信する為のOS内のユニフィケーションが必ず成功するように保証でき、失敗するとすればそれはユーザ・タスク内でのみ起こるようにできれば良いわけである。結論から述べれば、ユーザがOSのサービスを受ける為の通信規約が、各通信コマンドの全てのパラメタの入出力関係まで厳密に規定されているとすると[3]、次に示すようなフィルタ・ゴールをOSとの通信経路中に置き、しかもそのフィルタ・ゴールがユーザ・タスクに所属するようにしておくことによって、問題点ー1は解決することができる。

このような、OSの保護を目的としてユーザ・タスク内に置かれるフィルタを我々は保護フィルタと呼んでいる。

具体的な保護フィルタの仕事は、通信路中に流れる全てのコマンドに付いて次のことを行うことである。すなわち:

1. コマンド中で、ユーザ側が値を指定すべき部分は、その部分が規定の値に具体化されるまで待っ

[3]通常のOSでもエラー処理の為には厳密に規定されている。

てからOSに渡す。

2. OSからユーザに対して値を渡す部分に付いては、ユーザが返信用に指定した変数を直接OSに渡すのではなく、このフィルタだけが参照しているような新たな未束縛変数をOSに渡し、OSからの返答が決まってからユーザ側の変数とユニファイする。

3. コマンド中に、新たにOSとの通信路を開設するようなパラメタが含まれている場合には、その通信路に適合した保護フィルタを挿入する。

例えば、前出の例におけるフィルタは図.6のように定義することができる。図.6では、まずコマンド what_time(Time) に対して返信用の変数 Time の代わりに新しい変数 NewTime をOSに送り、NewTime が具体化されるのを待ってからその値を Time にユニファイしている。こうすることによって、ユーザが Time にいつ，どのような値を入れようとも失敗するのは（3）のユニフィケーションであり、ユーザ・タスクの中である。

実はこのフィルタを使った保護方式では先の 問題点－2 も同時に解決している。保護フィルタでは

ユーザが具体化すべきデータを全て待ってからOSに伝達するからである。もちろん、最悪的に具体化されなくてもユーザ・タスクがデッドロックすることになり、OS内には永久に具体化を待っているような無駄なゴールは生成されない。

図.6挿入

このようにKL1では、保護の為のフィルタを通信路に挿入するといったプログラミング・テクニックを用いることにより、言語に特別な特権命令を導入することなくOSの保護を実現できる。(荏苒もユーザが普通に使って良い言語機能である点に注意。)

4.3 OS資源の開放

OS資源とは、例えばウィンドウやファイル等のことである。PIMOSのユーザはOS資源を利用する場合まずその資源との通信路を開設し、その通信路に規定のコマンドを流すことによって処理を進めて行く。しかし、ユーザタスクがデッドロックに陥るなど

して、通信路が閉じられないまま残ることがある。通信路が閉じられなかった場合には先の保護の問題と同様、OS側にゴミのゴールが残ってしまう。このように、OSとの通信路が閉じられないまま残ってしまうのを防ぐ為に、PIMOSではバルブと呼ぶ機構を使ったOS資源の開放方式が採用されている。これは、ユーザタスクとOSとの通信路を水道管と見立て、送られるメッセージを水とした場合、バルブが外部から強制的に水の流れを止める機構として捉えられるところから名付けられた方式である。

ここで言うバルブも一種のフィルタであるが、ユーザタスク（ユーザの荘園）の外に置かれる点で保護フィルタと異なる。これはユーザタスクが強制終了された後に、その結果としてストリームを閉じる為である。また概念上保護フィルタが1入力1出力であるのに対しバルブは2入力1出力のフィルタである。すなわち、一方の入力は通信コマンドの流れるストリームであり、他方はバルブを閉じる為の命令が送られてくるストリームである。バルブを閉じる命令は、具体的には、タスクに対応する荘園をレポート・ストリームを通じて観測しているOS側のゴールによって発行される。PIMOSでは親子関係にあるタスクも許して

いるが、親タスクが何らかの理由で子タスクよりも先に強制終了させられた場合も、親タスクのパルプ閉じストリームを子タスクの使っている通信路のパルプに繋いで置くことによって自動的に閉じることができるようになっている。

5　おわりに

並列処理用のプログラミング言語やその処理系の実現方式に付いてはこれまで色々な提案がなされてきた。しかし、規模の大きな並列マシンを土台とし、実際に大規模な並列ソフトウェアの実験が可能な本格的な処理系や並列OSの実現を目指す研究開発は、ほとんど見るべきものが無かったと言えよう。本格的なものを目指すと従来個別に研究されてきた多くの研究課題が、一つの大きな枠組の中で相互に整合の取れた形で解決されなければならないという、より困難な問題が生じる。本論文では、ICOTで研究開発中の並列推論システムを例に取り、本格的な並列OSの実現を前提として、並列論理型言語KL1の機能と実現方式について解説し、幾つかの典型的な問題と現時点における解決法を述べた。実際にPIMOSが

動き始める頃にはさらに多くの興味ある問題が提起され、並列処理システムの研究も進展していくことであろう。

参考文献

[Bevan87] D. I. Bevan: *Distributed Garbage Collection Using Reference Counting*, Lecture Notes in Computer Science 259, Vol.2, Springer- Verlag, June 1987, pp176-187.

[Chikayama86] T. Chikayama: *Load balancing in a very large scale multi-processor system*, In Proceedings of Fourth Japanese-Swedish Workshop on Fifth Generation Computer Systems, SICS, 1986. Also in ICOT Technical Memo TM-276.

[Chikayama87] T. Chikayama, Y. Kimura: *Multiple Reference Management in Flat GHC*, Proc. of The Fourth International Conference on Logic Programming pp276-293, also as ICOT Technical Report TR-248, 1987.

[Goto86] Atsushi Goto, Shunichi Uchida: *Toward a High Performance Parallel Inference Machine - The Intermediate*

Stage Plan of PIM, Technical Report TR-201, ICOT 1986.

[Ichiyoshi88] 市吉伸行, 六沢一昭, 近山隆, 中島克人, 宮崎敏彦, 杉野栄二: 並列処理における重み付き参照カウントを用いた実時間 *GC*, 情報処理学会第 36 回 (昭和 63 年前期) 全国大会予稿 7E-3, 1988.

[Kimura87] Y.Kimura, T.Chikayama: *An Abstract KL1 Machine and Its Instruction Set*, Proc. of 1987 Symposium on Logic Programming, also as ICOT Technical Report TR-246, 1987.

[Miyazaki86] 宮崎敏彦, 瀧和男: *multi-PSI* における *FlatGHC* の実現方式, Technical Report TR-189, ICOT, 1986.

[Nakashima87] H. Nakashima, K. Nakajima: *Hardware Architecture of the Sequential Inference Machine: PSI-II*, Proc. of the 4th IEEE Symposium on Logic Programming, 1987.

[Rokusawa88] 六沢一昭，市吉伸行，渡和男，吉田あきる，若井基，中島浩： 並列処理における *PE* 間に渡るゴールの重み付き参照カウントを用いた管理方式，情報処理学会第 36 回（昭和 63 年前期）全国大会予稿 7E-2，1988．ICPP'88 に採録予定。

[Satou87] 佐藤裕幸，近山隆，杉野栄二，渡和男： *PIMOS* の概要 — 並列推論マシン用オペレーティング・システムの構築 —，情報処理学会第 34 回（昭和 62 年前期）全国大会予稿 2P-8，1987．

[Satou88] 佐藤裕幸，近山隆，杉野栄二，渡和男： 並列論理型 *OS--PIMOS(1)*— 資源管理方式 - 情報処理学会第 35 回（昭和 62 年後期）全国大会予稿 4D-3，1987．

[Shapiro84] E. Y. Shapiro: *SYSTORIC PROGRAMMING: A Paradigm of Parallel Processing*, Proc. of the International Conference on Fifth Generation Computer Systems '84, 1984, pp458-470.

[Taki84] K. Taki, M. Yokota, A. Yamamoto, H. Nishikawa, S. Uchida, H. Nakashima, A. Mitsuishi: *Hardware design and implementation of the personal sequential inference machine (PSI)*, In Proceedings of FGCS'84, ICOT, 1984. Also in ICOT Technical Report TR-075.

[Taki86] K. Taki: *The Parallel Software Research and Development Tool: Multi-PSI System*, Proc. of the France-Japan AI and Computer Science Symp. 86, Institute for New Generation Computer Technology, 1986.

[Ueda85] Kazunori Ueda: *Guarded Horn Clauses*, In Logic Programming '85, Wada, E., (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1986, pp. 168-179. A revised version is in Concurrent Prolog: Collected Papers, Shapiro, E.Y. (ed.), The

MIT Press, Cambridge, Mass, 1987,
Chapter 4.

[Warren83] D.H.D.Warren: *An Abstruct Prolog Instruction Set*, Technical Note 309, AI Center, SRI International, 1983.

[Watson87] P. Watson, I.Watson: *An Efficient Garbage Collection Scheme for Parallel Computer Architectures*, Lecture Notes in Computer Science 259, Vol.2, Springer-Verlag, June 1987, pp432-443.

[GHC] 並列論理型言語 *GHC* とその応用，淵一博監修，古川康一，溝口文雄共編，共立出版.

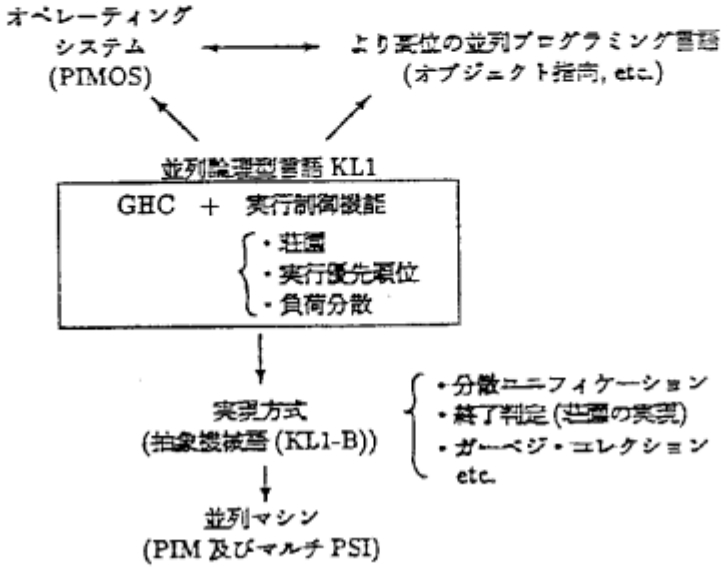

図 1: 並列推論システム開発のイメージ

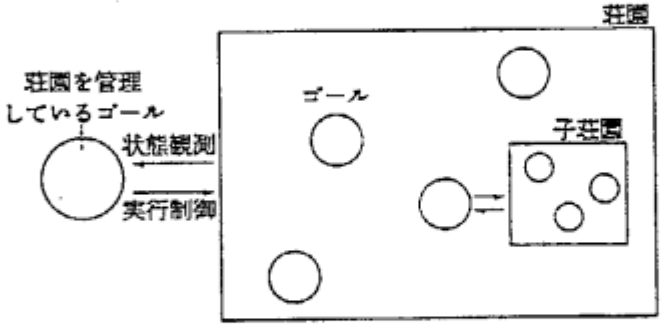

図 2: 荘園のイメージ

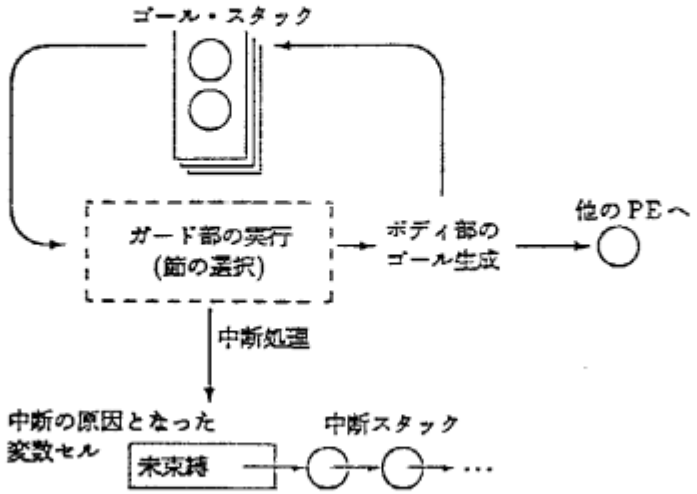

図 3: PE 内の処理概要

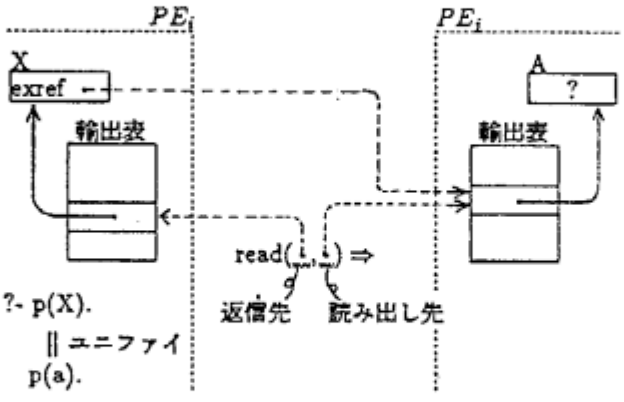

図 4: ガード部のユニフィケーション

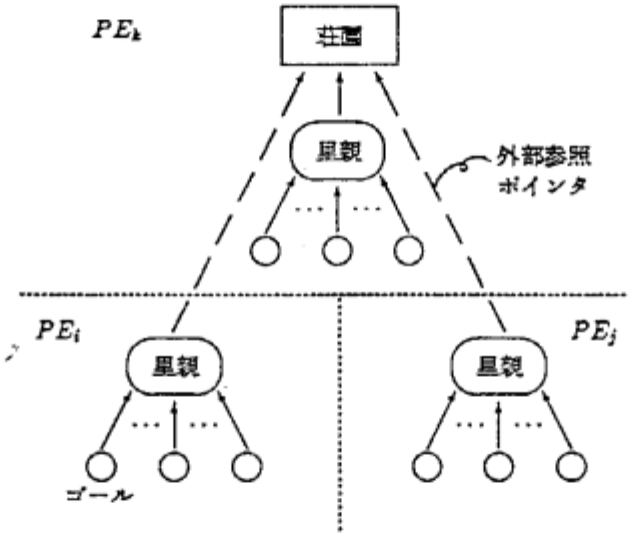

図 5: PE 間にまたがるゴール木

```
filter([what_time(Time)|USERreq], OSreq) :-
    true |
    OSreq=[what_time(NewTime)|NewOSreq],
    wait_and_unify(NewTime, Time),
    filter(USERreq, NewOSreq).
wait_and_unify(NewTime, Time) :-
    wait(NewTime) | Time=NewTime.     % (3)
```

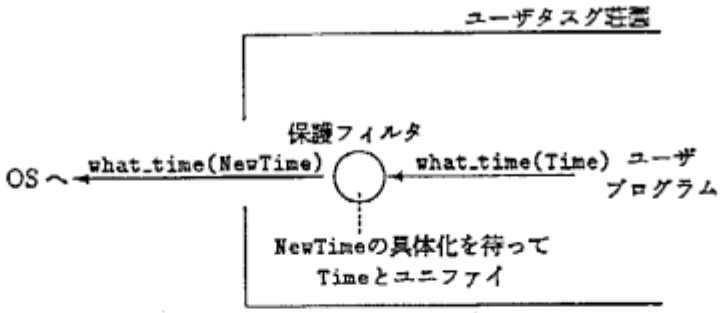

図 6: 保護フィルタの例