

PIMOS の 資 源 管 理 方 式†

佐藤裕幸^{†*} 越村三幸^{††} 近山隆^{††}
 藤瀬哲朗^{†††} 松尾正浩^{†††} 和田久美子^{††††}

PIMOS は、ICOT で現在開発中の並列推論マシンの能力を最大限に引き出すために、高並列に動作するプログラムを効率的に制御するオペレーティング・システムである。本論文は、この PIMOS における資源管理方式について述べたものである。PIMOS では、その記述言語である KL1 の持つ実行制御/監視機能を用いて、ユーザが利用する資源をタスクと呼ばれる単位ごとに管理している。タスクは任意にネストすることができ、子孫タスクもそれを生成したタスクの中で使用されている資源として扱う。これらの資源は、タスクの階層に沿った木構造の機構によって、分散して管理され、ユーザはストリームを介して通信することにより、これらの資源にアクセスすることができる。また、ユーザ-PIMOS 間に保護のためのフィルタを置くことにより、システム全体を保護している。このフィルタは、ユーザの誤りが PIMOS に浸透することを防ぐだけでなく、ユーザの不正な侵入からもシステムを保護している。現在 PIMOS は、マルチ PSI 第2版上でその最初の版が稼働している。測定の結果、PIMOS の資源管理方式では、PIMOS を使用することによるオーバーヘッドが、ほとんど無視できる程度しかないと確認した。また、我々は、並列論理型言語 KL1 による OS の実現により、その有用性を実証できたと考えている。

1. はじめに

第5世代コンピュータ・プロジェクトでは、高度な知識情報処理を行うに当たって必要とされる計算能力を提供するために、並列推論マシン (PIM)^{1),2)} の研究を進めている。また、並列ソフトウェアの研究を促進させるためのシステムであるマルチ PSI^{3),4)} の開発も行った。これらの並列推論マシンの能力を最大限に引き出すためには、高並列に動作するプログラムを効率的に制御するオペレーティング・システム (OS) が必要不可欠である。

PIMOS (Parallel Inference Machine Operating System)⁵⁾⁻⁷⁾ は、この目的のために設計してきたものであり、PIM やマルチ PSI などの並列推論マシン用の OS である。この PIMOS は、以下のような特徴を持っている。

(1) KL1 で論理的に記述された OS

PIMOS は、並列論理型言語 KL1 (核言語第1版) を用いて、副作用などの論理的でない機能を用いずに記述されている。KL1 は、OS などを記述する際にこ

の超論理的な機能を用いなくて済むように設計されている。そのため、従来のシステムでは OS で実現しなければならなかった機能の幾つかのものを言語レベル以下で実現している。

(2) 単一 OS

PIMOS は並列処理マシン用の OS ではあるが、複数の OS の集合体ではない。プロセサごとに独立して動作する OS が協調して全体の機能を果たすのではなく、全体として一体であるシステムの並列動作可能な部分を並列に実行するものである。

(3) フロントエンド・プロセサを接続

PIMOS が扱う並列推論マシンには、フロントエンド・プロセサ (FEP) が接続されている。この FEP は、物理的な入出力操作を行うのであるが、それらの低レベルの操作が PIMOS から見えないように設計されている。マルチ PSI では、FEP として、PSI-II⁸⁾ を使用している。

(4) 本格的な OS として必要な機能を網羅

PIMOS は実験的なシステムではなく、本格的な OS として実用に耐えるだけの基本機能を網羅している。現在、工程の関係上、ウィンドウ・システムやファイル・システムなどの機能を、FEP 上の OS (SIMPOS) に依存しているが、これらも KL1 で記述できる仕様となっている。

(5) 使い勝手の良いシステム

ハードウェアや並列アルゴリズムの実験/評価が円滑に行えるように、使い勝手の良いシステムとなって

† Resource Management of PIMOS by HIROYUKI SATO, MIYUKI KOSHIMURA, TAKASHI CHIKAYAMA (ICOT Research Center), TETSURO FUJISE, MASAHIRO MATSUO (Mitsubishi Research Institute Incorporation) and KUMIRO WADA (Oki Electric Industry Corporation).

†† (財)新世代コンピュータ技術開発機構

††† (株)三菱総合研究所

†††† 沖電気工業(株)

* 現在 三菱電機(株)情報電子研究所

Mitsubishi Electric Corporation

いる。そのためには、ユーザが誤った使い方をした時に、システム全体を保護することも重要な要素となっている。

(6) 単一ユーザ/複数タスク

複数タスクを許すことはシステムの使い勝手の上から必須であるし、並列処理マシンの能力を最大限に引き出すためにも当然必要な機能である。複数ユーザを考えるとユーザ間の競合関係を調整する必要が生じる。これを並列環境の下で制御するのは、非常に困難な問題なので、当面は本格的には扱わないこととし、将来の課題として残している。

(7) クロスシステムの充実

実験的なハードウェアのための OS であるので、ハードウェアはいつでも利用可能とは限らない。したがって、ソフトウェアの研究開発を促進するために、PIMOS では KL1 による並列プログラムの開発環境を、PSI や汎用計算機などの、より利用しやすい計算機上に提供している。

計算機資源 (CPU 資源、メモリ資源、入出力機器) の管理を行い、ユーザの過ちからシステム全体を守ることは、OS の最も基本となる機能である。このためには、計算機資源をそのままユーザ (応用プログラム) に見せることはできないので、計算機資源に保護のための枠組を付加し、仮想化してから提供することが必要になる。こうした管理機能が PIMOS の中核となる部分である。

本論文では、この PIMOS の管理機構について、プロセスによる通信方式 (第2章)、言語処理系で提供する資源管理機構 (第3章)、資源木による資源の管理方式 (第4章)、ユーザの誤りからのシステムの保護方式 (第5章)、最後にこの資源管理方式の評価 (第6章) について報告する。

2. プロセス

ユーザ・プログラムが消費する CPU 時間、メモリ、入出力装置などの資源を、それぞれのプログラムが並列に動作する環境の下で管理するのは、容易なことではない。例えば、OS があるデータに対する処理を行っている最中に、そのデータの状態をユーザ・プログラムによって変更されることを防ぎたい場合がよくある。そのような場合、従来の逐次計算機上では、「OS がそのデータを処理している最中は、ユーザ・プログラムを動作させない」という実行順序によってデータの変更を防ぐことができた。しかし、PIMOS

が動作するような並列環境では、並列性を犠牲にせずに、そのような実行順序の制御はできない。つまり、OS がそのデータを処理している最中に、すべてのユーザ・プログラムの実行を停止させることになり、これは明らかに並列性を犠牲にすることになる。したがって、従来の逐次計算機と同様の方式を採用することはできない。

そこで PIMOS では、プロセスをオブジェクト指向言語におけるオブジェクトとして見なすようなプログラミング・スタイル⁹⁾を導入することにより、この問題を解決している。このプロセスは、管理しているデータと通信用の変数 (ストリーム) を保持している。プロセスの外からこのデータにアクセスする場合は、このストリームにメッセージを送るという形ではか行えない。したがって、そのストリームにアクセスすることができないプログラムは、このデータにアクセスすることはできない。つまり、データへのアクセス・パスを1つにすることで、複数のプログラムからの競合を防いでいる。

PIMOS が管理するすべての資源は、このプロセスを用いて表現し、ストリームを介して通信することによって管理されている。つまり、このシステムでの資源へのアクセスは、従来のシステムのスーパーバイザ・コールのような、資源を集中管理しているモジュールへの通信によって行われるのではなく、それぞれの資源を管理しているプロセスへストリームを介して通信することによって行われる。したがって、PIMOS では、これらのストリームを管理することが、資源を管理することに相当している。

ストリームを介したメッセージの送受信は、具体的には、共有変数の具体化によって行う。例えば、ユーザが、ある入力装置から文字列を読み込む場合は、以下のようなプログラムになる (KL1 言語については、その基礎となった GHC についての解説を参照されたい¹⁰⁾)。

```
?- pimos(Req), user(Req).
user(Req) :-
    Req=[getb(N, String) | ReqT], ...
pimos([getb(N, String) | ReqT]) :-
    readFromKbd(N, KBDString),
    KBDString=String,
    pimos(ReqT).
```

ユーザと PIMOS とは、共有変数 Req をストリームとして利用することで通信を行う。ユーザは、そこ

に N 文字読み込みたいという要求 `getb/2` を送る。PIMOS は、そのメッセージを受け取ると、`read-FromKbd/2` により必要な N 文字分の文字列を読み込み、String を読み込まれた文字列に具体化することによりユーザに返す。そして、`ReqT` により次の通信を行う。

この `pimos/1` のようなプログラミング・スタイルをプロセスと呼んでおり、メッセージによってのみ、自分の管理している資源（この例ではキーボード）への操作を許している。

ユーザと PIMOS との通信は、最初に与えられた唯一の共有変数（ストリーム）`Req` を基に、共有変数を増やしていくことで進んでいく。例えば、ユーザが新しいウィンドウを生成し、そこに対して入出力を行う場合は、以下ようになる。

```
?- pimos(Req), user(Req).
user(Req):-
    Req=[create_window(Window)|ReqT],
    Window=[getb(N,String)|WinT],
    .....
```

新たに生成されたウィンドウに対する入出力操作は、PIMOS から与えられた通信変数 `Window` およびそこから派生した変数を介してのみ行える。つまり、これらの変数を持っていない他のプログラムは、このウィンドウにアクセスすることはできない。したがって、PIMOS ではこれらの通信用の変数をユーザに与えることが、資源にアクセスすることのできる権利 (capability) を与えることに相当しており、この機構により、ユーザの不正なシステムへの介入を防いでいる。

3. 荘園 (資源管理, 例外処理) 機能

PIMOS の記述言語である KL1 は Flat GHC^[11] を基にしているため、すべてのゴールは AND 関係になっている。したがって、このままでは、ユーザ・プログラムが (上記の例の `user(Req)` から呼ばれたサブ・ゴールのうちの一つでも) 失敗した場合は、PIMOS も失敗し、システムダウンを起こしてしまう。これは、PIMOS とユーザ・プログラムが同じレベルになっているからである。しかし、ユーザ・プログラムと OS とは、オブジェクト・レベルのプログラムと、そのオブジェクト・プログラムを監視/制御するメタ・レベルのプログラムの関係になるべきである^[12]。

メタ・プログラミングの機能を実現する単純な方法には、インタプリタがあるが、実行効率面で大きな損

失が生じてしまう。そのため、KL1 では、このメタ・プログラミングの機能を言語レベルで組み込んでおり、これを「荘園」と呼んでいる。荘園は、実行制御/監視機能、資源管理機能、例外処理機能を持っている。

図 3.1 実行/資源管理機能

荘園は以下のような組込述語を用いて生成する。

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

ここで、各引数は以下のような意味を持つ。

(1) Goal

荘園の中で実行すべきゴールを指定するための変数である。

(2) ExpMask

この荘園で、どのような例外を処理するかを決めるためのマスク・パターンである (詳しくは省略する)。

(3) Control

荘園内の実行を制御したり、資源の割り当てを指定するためのストリームである。ここには、以下のような機能を持ったメッセージを流すことができる。

- 荘園内の実行の開始/中断/再開/放棄
- 荘園内で消費できる計算資源の許容量の問い合わせ/追加
- 荘園内でこれまでに消費された計算資源量の問い合わせ

(4) Report

荘園内の実行状態が報告されるストリームである。ここには、以下のような 2 種類の報告が行われる。

(a) 事象の報告

荘園の実行にともなって生じた事象が報告され、これには以下のようなものがある。

- 荘園内の実行の終了
- 荘園内で消費できる計算資源の不足
- 荘園内で発生した例外事象 (オーバフロー、ゴールの失敗など)

(b) 受け取りの確認

制御ストリームからのメッセージを受け取ったことを報告する。荘園の制御/監視は、制御用と報告用の 2 つのストリームによって行うので、この確認の報告がないと、それらのストリームに流れるメッセージ間の同期をとることができない。例えば、制御ストリームから計算資源の許容量を追加した後、報告ストリームから計算資源の不足が報告された場合、

- 資源を追加したがそれでも不足してしまったので、さらに追加する必要がある
- 資源追加のメッセージが届く前に不足を検出したので、その後資源は追加され、今は追加する必要がない

のどちらか分からない。しかし、報告ストリームから資源追加メッセージを受け取ったという報告がなされれば、それと資源不足の報告との前後関係によって、上記のどちらであるかを知ることができる。

荘園内のゴール群は、荘園外とは独立した AND 関係を成している。すなわち、荘園内での失敗は荘園内に閉じたものであり、荘園外のゴールを巻き添えにすることはない。したがって、概念上、荘園はインタプリタを機械語レベルで実現したものであると考えて良い。

また、荘園の中でさらに荘園を作ることも可能である。この場合、外側の荘園に実行の中断を指令すれば、内側の荘園の実行も中断される。外側の荘園の実行を再開すれば、内側の荘園の実行も再開される。また、内側の荘園が使用できる資源は、外側の荘園から分け与えられるものである。つまり、内側の荘園が消費した資源は、外側の荘園も消費したものとして扱われる。

このような管理方式をとることにより、メタ・レベルはオブジェクト・レベル以下にどのようにメタ/オブジェクトの階層構造があるかを意識することなく、全体として管理することが可能となっている。

この荘園の機能を使った、ユーザと PIMOS との通信は以下ようになる。

```
?- pimos(Req, Cnt, Rep),
   execute(user(Req), -1, Cnt, Rep).
```

PIMOS は、変数 Cnt によりユーザ・プログラムの実行を制御し、変数 Rep により実行の監視および資源の管理を行う。このように荘園は、制御ストリームと報告ストリームをインタフェースとした、言語レベルで提供されているプロセスと考えることができる。

4. 資源木による資源の管理

OS が管理すべき資源には、荘園で管理されるような計算時間やメモリのほかに、入出力装置などがある。前者のような資源を「言語定義資源」と呼び、後者のような資源を「OS 定義資源」と呼んでいる。このように、PIMOS がユーザの消費するすべての資源を管理するためには、荘園の機能だけでは不十分であ

り、入出力装置などの OS 定義資源も管理する機構を導入しなければならない。

4.1 タスク

これらの言語定義資源や OS 定義資源の管理を行うために、PIMOS では、「タスク」と呼ばれる資源管理の単位を導入している。このタスクは、荘園の機能を用いて実現しており、荘園と同様に任意にネストすることができる。このタスク内で生成した子タスクも、親タスク内で使用している資源の1つとして管理する。つまり、PIMOS では、あるタスク内でどのような荘園を生成したかは管理していないが、どのような子タスクを生成したかは管理している。

ユーザがタスクを生成すると、タスク制御ストリームとタスク報告ストリームが返される。ユーザは、タスク制御ストリームにメッセージを送ることにより、タスク内の実行を制御したり、タスク内で使用されている資源に対して問い合わせることができる。また、タスク報告ストリームから知らされる情報により、タスクの実行状態を監視したり、例外処理を行うことができる。

4.2 資源木

PIMOS では、タスクの中で使用されているすべての資源（言語定義資源および OS 定義資源）を、「資源木」と呼ばれるストリームでお互いに結合されたプロセスの木構造によって、タスクの階層ごとに管理している。

タスクの階層構造および資源木の構造の例を図1に示す。この例では、3つのタスクが存在しており、それぞれのタスクは以下のような OS 定義資源を使用している。

- 親タスク：1つの入出力資源と「子タスク1」と「子タスク2」
- 子タスク1：2つの入出力資源
- 子タスク2：1つの入出力資源

この図の右側の部分を資源木と呼び、ユーザ・タスクとは別の領域に置かれる。資源木の各ノードは、タスクの階層にそって木構造になっており、各タスク内の資源をそれぞれ環構造で結んでいる。以下に、資源木を構成する各要素について説明する。

(1) タスク・ハンドラ

タスク内の資源を管理しているプロセスであり、ユーザがそれらの資源を操作する時に、ここにメッセージが流れてくる。また、祖先タスクからもここで管理している資源に対するメッセージがモニタ(後述)

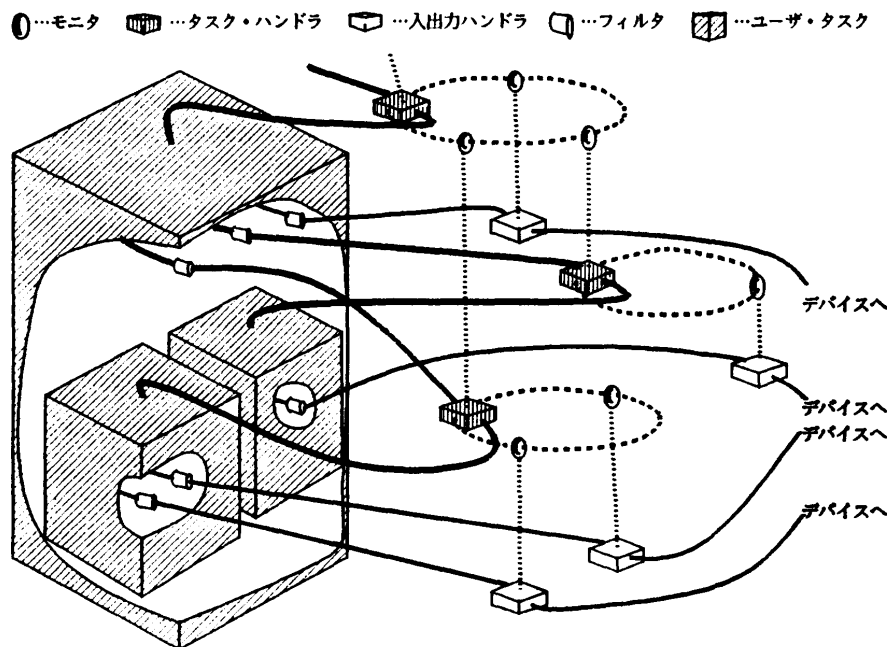


図1 資源木とユーザ・タスクの構造
Fig. 1 Structure of resource tree and user tasks.

を經由して流れてくることもある。ここでは、タスクを実現している荘園の制御ストリームと報告ストリーム(図ではタスクの上につながる太実線)を保持することで、タスクの実行を制御し、実行状態を管理している。

また、このタスク内で使用されている入出力装置や子タスクなどの OS 定義資源を、それぞれ対応するモニタを環状につなげること(これを資源ループと呼ぶ)で管理している。これらの資源に対する操作メッセージは、モニタを通して子資源のハンドラに送られる。

(2) 入出力ハンドラ

入出力装置を資源としてユーザに見せるためのプロセスであり、一般の入出力要求に応じる機能と、この資源に対するメタな問い合わせ(例えば、どのような種類の資源なのかなど)に答える機能を持っている。前者のようなメッセージはユーザからフィルタを通して送られ、さらにデバイスに近いプロセス(デバイス・ドライバ)に再送される。また、後者のようなメッセージはモニタを通して送られ、このプロセスで処理される。

(3) モニタ

タスク・ハンドラからのメッセージを自分が保持しているハンドラに送るかどうかを決める。そのメカニズムについては、後述する。

(4) フィルタ

ユーザの誤りからシステム全体を保護するためのプロセスである。ユーザと PIMOS が通信する場合には、必ずこの種の保護フィルタが付けられる。この保護フィルタはユーザ・タスク上で実行されるが、プログラム・コードとしては PIMOS が提供するものである。保護フィルタの詳細については、5章で記述する。

4.3 OS 資源の管理方式

PIMOS が管理するすべての OS 定義資源には、各タスク内でユニークな ID (資源 ID) が付けられており、資源 ID 列によって特定の資源を指定することができる。例えば、[3, 4, 2] であれば、タスク内の 3 番目の資源内の 4 番目の資源内の 2 番目の資源を表している。

ユーザは、タスク制御ストリームを通して、メッセージを送ることで OS 定義資源に対する制御を行える。この時、資源 ID をメッセージに付けるとその資源に対する操作となる。モニタは、それぞれ自分が管理している OS 定義資源の ID を保持している。そして、自分のところに流れてきたメッセージに付いている ID と自分の ID を比べ、一致すればそのメッセージを(資源 ID 列の残りを付けて)自分のハンドラに送り、一致しなければ隣のモニタに再送する。

タスク・ハンドラは、親モニタからメッセージを受

け取ると、それが自分宛であれば自分で処理し、子孫資源宛であれば自分の資源ループに再送する。資源ループに送ったメッセージがもう一方のストリームからそのまま返ってきた場合は、指定した ID に相当する資源が存在しなかったことになり、エラーとして扱う。

タスク・ハンドラは、タスクを実現している荘園の報告ストリームを監視している。そこから実行の終了または放棄が報告されると、そのタスク内で使用していた未解放のすべての資源（資源ループ）に対して、タスク・ハンドラが解放要求を送ることにより、資源を自動的に解放する。

OS 定義資源を解放する時には、モニタが自分の兄からのストリームと弟へのストリームをユニファイすることにより、環の中から消滅する。ただし、資源木内のそれぞれのプロセスが並列に動作しているので、資源の解放には注意を要するが、その詳細については省略する。

4.4 従来 OS との比較

従来 OS では、タスク（システムによってはプロセスと呼ぶこともある）を、1つの平坦なテーブルで集中的に管理している場合が多い。また、入出力装置への操作も、例えば、「タスク内の入出力資源テーブルの N 番目に登録されている出力装置に出力する」という形で行うことが多い。したがって、これらの資源に対する操作を行うたびに管理テーブルへアクセスすることになる。

従来逐次計算機システムの場合は、この管理テーブルへのアクセス集中がそれほど問題になっていなかった。一方、並列計算機の場合は、独立した資源に対する操作を並列に行うことができるが、メモリ共有型の並列計算機の場合には、資源へのアクセス頻度などを考えると、集中管理によるネックがそれほど問題とされていない。

しかし、PIMOS が対象としているネットワーク型でしかも並列度の高い並列計算機システムの場合は、集中管理によりプロセッサ間の通信量が増えてしまうので、並列性を犠牲にすることになる。そのため PIMOS では、資源のアクセスが、「資源へのストリームに直接メッセージを送る」という形になっており、独立した資源への操作は、それぞれ並列に行えるようになっている。

また、「タスク内で使用しているすべての資源の状態を返す」といった処理の場合、タスク・ハンドラ

は、資源ループへその旨のメッセージを送るだけなので、すぐにユーザからの次のメッセージに対する処理を行える。またモニタは、そのメッセージを受け取ると、資源ハンドラにメッセージを送るとともに、次のモニタに同じメッセージを送る。このように、メッセージが各資源ハンドラに分散されるため、それぞれ独立して各資源の状態を返すことができる。

5. 通信の保護機構

5.1 問題点

ユーザ・プログラムの実行をメタなレベルから監視する機構は、KL1 の荘園機能により行える。しかし、PIMOS-ユーザ間の通信では、KL1 に個々のユニフィケーションを制御/監視する機能がないため、単純に通信を行ってしまうと、システムダウンに陥ることがある。

ユーザが指定した長さ（文字数）の文字列をキーボードから読み込む場合を考えてみる。

```
user(Req):- true |
    Req=[getb(N, String) | ReqT], ...
pimos([getb(N, String) | ReqT], Cnt, Rep):-
    true |
    readFromKbd(N, KBDString),
    KBDString=String, ... (a)
```

ユーザは、通信用ストリーム Req に入力要求メッセージ getb/2 を送る。PIMOS はメッセージ getb/2 を受け取ると、readFromKbd/2 により必要な N 文字分の文字列を読み込む。この時、readFromKbd/2 は、変数 N が具体化されるのを待つ。そして、変数 String を読み込まれた文字列 KBDString に具体化し、ユーザに返す。しかしこのプログラムには、以下のような 2 種類の問題点が含まれている。

(1) 問題点 1

ユーザが以下のように（誤って）文字列が入るべき変数をアトムに具体化してしまったとする。

```
String=foo (b)
```

並列環境の下では、効率を落とさずにゴール (a) と (b) の実行順序を規定することはできない。そのため、(a) が先に実行されれば (b) が失敗するし、(b) が先に実行されれば (a) が失敗する。前者の場合であればユーザ・タスク内で失敗が起きるだけであり、問題はないが、後者の場合だと PIMOS 側で失敗が起きるので、システムダウンに陥ってしまう。KL1 はユニフィケーションが成功するかどうかをチェックす

るような(テスト・アンド・ユニファイ)機能を持っていない(もし、KL1がテスト・アンド・ユニファイの機能を持った Concurrent Prolog¹³⁾を基にしているのであれば、問題とはならない)。例えば、上記の String が未定義変数かどうかのチェックができたとしても、そのチェックとユニフィケーションとの間に、String が他の値に具体化されてしまう可能性があるからである。

(2) 問題点 2

タスク側が文字数設定用共有変数 N に値を設定するのを忘れたか、または設定する前にタスクの実行が放棄された場合、PIMOS 側の N の具体化を待つプロセス (readFromKbd/2) が永遠に待ち続け、デッドロックしてしまう(これは、たとえテスト・アンド・ユニファイの機能があっても問題となる)。

5.2 保護フィルタ

これらの問題点を解決するために、PIMOS-ユーザ・タスク間のストリームに保護のためのフィルタ・プロセス(保護フィルタ)を設け、そのプロセスをユーザ・タスク内で実行することにした(図2)。

タスクは PIMOS とつながっているストリームに直接メッセージを送らずに、保護フィルタへのストリーム(図2の(1))へメッセージを送る。このフィルタはユーザのメッセージを絶対に失敗しない形式に変換し、また、PIMOS 側で待ち続けられないことが保証されるまで待つから、PIMOS へのストリーム(図2の(2))へ変換されたメッセージを送信する。

このように PIMOS 側のストリームをユーザに直接見せないため、ユーザの誤りが PIMOS 側へ波及するのを防ぐだけでなく、ユーザからの悪意を持った PIMOS への侵入を保護フィルタが監視して防ぐことができる。つまり、保護フィルタは、ストリームの capability を制限していることになる。

この保護フィルタの具体的な仕事は以下のようなことである。

- ユーザが値を設定すべきデータ部分は、その値が設定されたことを確認してから PIMOS 側へ送る(問題点2の解決)。これにより、PIMOS にメッ

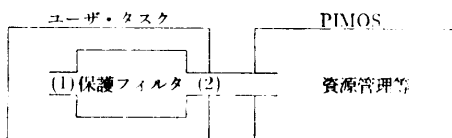


図2 保護フィルタ付き通信

Fig. 2 Communication with protection filter.

セージが送られた時には、それらの値は確定していることが分かる。ただし、ストリームの場合だけは、そのヘッド(第1要素)の具体化しか待たない。そうしないと、そのストリームを閉じるまで、すべての要求が PIMOS 側に流れないことになるからである。PIMOS では、すべてのストリームを資源として管理しており、タスクの実行が放棄された時に資源の解放処理を行うので、ストリームの場合は問題点2を気にする必要はない。

- PIMOS 側から返されるデータ部分については、必ず未定義状態であるような別の変数を送る(問題点1の解決)。そして、PIMOS から値が返された後で、ユーザが指定した変数を返ってきた値とユニファイする。これにより、PIMOS 内での値を返すユニフィケーションは、必ず未定義変数とのユニフィケーションとなり、失敗することはない。例えば、上記のキーボードの例に対する保護フィルタ・プログラムは以下ようになる。

```
pfilter([getb(N, String) | T], OS) :- wait(N) |
    OS=[getb(N, OSString) | OST],
    waitAndUnify(OSString, String),
    pfilter(T, OST).
waitAndUnify(OSV, USERV) :- wait(OSV) |
    OSV=USERV.
```

保護フィルタ (pfilter/2) は、読み込む文字数 N に値が設定されるまで待つから PIMOS へ要求を送信する。また、読み込まれた文字列が設定される変数 String の代わりに、未定義変数 OSString を PIMOS に送る。そして、その変数の値が設定されるのを待つから、ユーザの変数 String へ文字列が設定される。したがって、ユーザが変数 String を誤った値に具体化したとしても、ユーザ・タスク内で実行されるフィルタ (OSV=USERV) が失敗するだけで PIMOS には影響はない。

このフィルタは、ユーザ・プログラムを起動する時に、PIMOS により以下のように挿入される。

```
?- pimos(OSReq, Cnt, Rep),
    execute((user(Req),
            pfilter(Req, OSReq)),
            -1, Cnt, Rep).
```

また、ストリーム Req から新たに派生した通信変数のためのフィルタは、pfilter/2 の中で生成される。例えば、ウィンドウを生成するメッセージのための保護フィルタのプログラムは、以下ようになる。

```
pfilter([create_win(Name, Win)|T], OS):-
    wait(Name)|
    OS=[create_win(Name, OSWin)|OST],
    pfilter_win(Win, OSWin),
    pfilter(T, OST).
```

ユーザは新たに生成されたウィンドウに対するメッセージを Win に送るが、それは PIMOS と直接つながれておらず、pfilter_win/2 により保護される。

この保護フィルタのプログラムは、OS-ユーザ間の通信プロトコルを定義すれば、機械的に生成することができる。PIMOS では、プロトコル定義言語を設け、プロトコル・コンパイラを用いて、保護フィルタを自動生成している。

6. 評価

PIMOS で採用しているプロセスをストリームで結合した資源管理方式で、保護フィルタや資源ハンドラなどの各プロセスがどの程度のオーバーヘッドになっているかを評価した。

一般に入出力は、1文字ずつ行うのではなく、何文字かをバッファリングして行う方が、スループットが良くなる。PIMOS が扱うような並列計算機の場合も逐次計算機と同様にこのバッファリングの効果があり、PIMOS がバッファリングを行うユーティリティを用意している。そこで今回は、ある固定のサイズの文字列をバッファリングのサイズを変えて入出力した時の実行時間の変化を

- (1) オリジナルの PIMOS (保護フィルタ, 入出力ハンドラ付き)
- (2) 保護フィルタを取った場合
- (3) 保護フィルタおよび入出力ハンドラを取った場合

の3種類について、測定した。

これらの測定を行う場合、実際の入出力装置 (例えばファイル) を使うと、それ自体の遅さが目立ってしまうので、今回はダミーの入出力装置を KL 1 で作成した。PIMOS での入出力装置は、プロセスとして見えるので、KL 1 でそれを模擬するものを作成するのは容易である。このダミー装置は、あるサイズの文字列を出力したり、指定したサイズの文字列を入力する機能を持っている。

このダミー装置は測定プログラムとは別プロセッサで実行させるのが実際の入出力装置に近い動きになる。しかし、これでは、通信オーバーヘッドのために、PIMOS プロセスのオーバーヘッドが隠れてしまう可能性がある。そこで、ダミー装置プロセスが測定プログラムと同一プロセッサに存在する場合についても測定した。

1メガ文字 (1文字は2バイト) の文字列を読み込む場合の、バッファ・サイズを変化させた実行時間の推移を図3に示す。このグラフから以下のようなことが言える。

- ダミー装置が同一プロセッサにある場合には、バッファ・サイズ1キロ文字くらいまで保護フィルタや入出力ハンドラのオーバーヘッドが出ているのに対し、別プロセッサの場合は出していない。これは、バッファ・サイズが1キロ文字くらいまではプロセス間の通信オーバーヘッドが PIMOS プロセスのオーバーヘッドを打ち消してしまうほど大きいからである。
- バッファ・サイズが1キロ文字より大きい場合は、ダミー装置プロセスが別プロセッサにあると同一プロセッサにあると、PIMOS プロセスのオーバーヘッドが出ないし、両者の実行時間がほぼ等しくなっている。これは、バッファ・サイズが1キロ文字より大きい場合には、ダミー装置の仕事量が PIMOS プロセスのオーバーヘッドを打ち消してしまうほど大きいためである。
- ダミー装置プロセスが別のプロセッサにある場合、

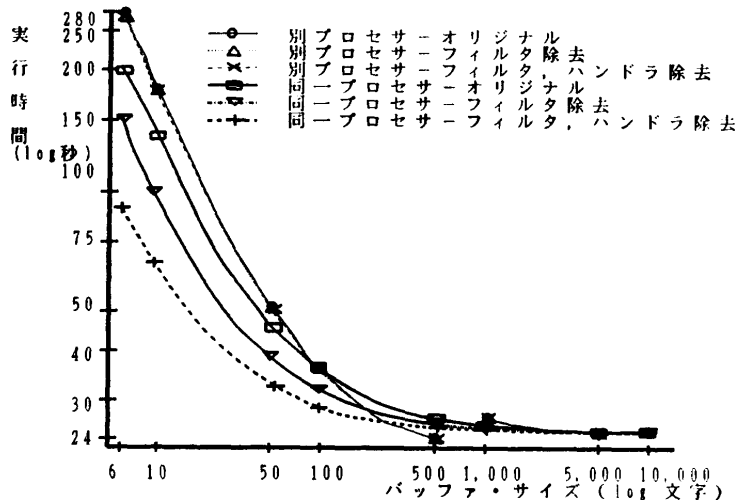


図3 各バッファ・サイズの実行時間と PIMOS プロセスのオーバーヘッド
Fig. 3 The execution time of each buffer size and overhead of PIMOS processes.

バッファ・サイズが500文字近辺の処理時間が最小になっており、それを越えると(1,000文字近辺で)急激に処理時間が上がっている。今回の測定で使用したKL1処理系では、プロセッサ間通信をパケットにより行っており、1度にプロセス間で通信できるデータのサイズが、500文字あたりになっている。それを越えると1つのバッファでも複数のパケット(マルチ・パケット)に分割されるため、ここで実行時間が不連続になっている。

以上の測定の結果、本来はPIMOSプロセスがオーバーヘッドになっているが(同一プロセッサによる測定より)、実際の入出力においてはどのようなバッファ・サイズをとっても、PIMOSプロセスのオーバーヘッドは無視できる程度であると言える。また、バッファリングのサイズは、マルチ・パケットになる寸前である500文字近辺が最適であることが分かった。このことから、現在バッファリング・ユーティリティでは、バッファ・サイズの既定値をこのあたりに設定している。

7. おわりに

我々は、本論文で述べたプロセスによる資源の分散管理方式を用いて、マルチ PSI 第2版上でPIMOSの開発を行っており、現在、その最初の版が稼働している。

現在までの開発で得られた成果として、並列論理型言語KL1でOSを「記述できる」という点だけではなく、「記述しやすい」という点が挙げられよう。従来の手続き型の言語でOSを記述する際に最も注意を払う点は、同期の問題である。この同期は、データに対して行われる(例えば、必要なデータがそろうまで待つ)場合が多いが、それを実行順序を制御することで(つまり、実行順序の同期を用いて)行われてきた。したがって、ここがOSの中で最もバグが出やすくデバッグしにくい部分である。一方KL1の場合は、言語の基本機能として、データに対する同期機構が備わっているため、OSのようなプログラムを楽に記述できるし、バグも少なくなっている。

現在のPIMOSは、入出力機能をFEP上のOSに依存している部分が多いが、今後、ファイル・システムなどを並列推論マシンの本体上に移行していく予定である。こういった部分に関しても、並列実行による高速化およびKL1による記述の評価が行えると考えている。

並列マシンのOSにとって最も重要な課題は、負荷の分散である。現在のPIMOSでは、負荷分散をすべてユーザに任せており、OSはユーザに近いプロセッサで仕事をしている。この負荷分散を何らかの形で自動化する必要があるが、完全自動にするのは、かなり困難な研究課題である。

謝辞 本研究に関して有益な助言を頂いたICOT第4研究室、協力会社の方々および沖電気工業の宮崎敏彦氏に深く感謝する。評価のための測定には、ICOT第4研究室の星田氏に協力して頂いた。また、本研究の機会を与えてくださったICOT第4研究室の内田室長および測所長に感謝する。

参考文献

- 1) Goto, A. and Uchida, S.: Toward a High Performance Parallel Inference Machine—The Intermediate Stage Plan of PIM—, Technical Report TR-201, ICOT (1986).
- 2) 後藤ほか: 並列推論マシン PIM—中期構想—, 第33回情報処理学会全国大会論文集, 3B-5 (1986).
- 3) Taki, K.: The Parallel Software Research and Development Tool: Multi-PSI System, Technical Report TR-237, ICOT (1986).
- 4) 瀧ほか: Multi-PSI システムの概要, 第32回情報処理学会全国大会論文集, 3Q-8 (1986).
- 5) Chikayama, T. et al.: Overview of the Parallel Inference Machine Operating System (PIMOS), *Proc. of FGCS '88*, Vol. 1, pp. 230-251 (1988).
- 6) 佐藤ほか: 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—, 第34回情報処理学会全国大会論文集, 2P-8 (1987).
- 7) 佐藤ほか: 並列論理型 OS-PIMOS (1)—資源管理方式—, 第35回情報処理学会全国大会論文集, 4D-3 (1987).
- 8) Nakashima, H. and Nakajima, K.: Hardware Architecture of the Sequential Inference Machine: PSI-II, Technical Report TR-265, ICOT (1987).
- 9) Shapiro, E. and Takeuchi, A.: Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, Vol. 1, No. 1, pp. 25-48 (1983).
- 10) Ueda, K.: Introduction to Guarded Horn Clauses, Technical Report TR-209, ICOT (1986).
- 11) Ueda, K.: Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, Technical Report TR-208, ICOT (1986).

- 12) Clark, K. and Gregory, S.: Notes on Systems Programming in PARLOG, *Proc. of FGCS '84*, pp. 299-306 (1984).
- 13) Shapiro, E. and Takeuchi, A.: A Subset of Concurrent Prolog and Its Interpreter, Technical Report TR-003, ICOT (1983).

(平成元年 5 月 31 日受付)

(平成元年 9 月 12 日採録)



佐藤 裕幸 (正会員)

昭和 34 年 11 月 2 日生。昭和 57 年筑波大学第 3 学群情報学類卒業。同年三菱電機(株)に入社。昭和 60 年 6 月～平成元年 4 月、(財)新世代コンピュータ技術開発機構に出向し、プログラミング環境、並列オペレーティング・システムの開発に従事。現在、三菱電機情報電子研究所にて、並列ソフトウェアの研究に従事。著書「Prolog 入門」(共著、啓学出版、昭和 59 年)。



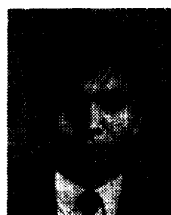
越村 三幸 (正会員)

1961 年生。1984 年筑波大学自然科学類卒業。1986 年同大学院修士課程修了。同年日本ビジネスオートメーション(株)入社。同年 11 月より(財)新世代コンピュータ技術開発機構に出向中。並列処理、オペレーティングシステムの研究開発に従事。



近山 隆 (正会員)

昭和 28 年生。昭和 52 年東京大学工学部計数工学科卒業。57 年同工学系大学院情報工学専門課程博士課程修了。工学博士。同年富士通(株)入社、(財)新世代コンピュータ技術開発機構に出向、現在に至る。この間、手続き型言語、関数型言語、論理型言語、オブジェクト指向言語とこれらの逐次および並列処理系、プログラミング環境、論理型言語専用計算機とそのオペレーティングシステム研究開発に従事。



藤瀬 哲朗 (正会員)

1959 年生。1984 年電気通信大学大学院修士課程修了(情報数理工学専攻)。同年(株)三菱総合研究所に入社。現在、同社情報技術開発部に所属。数式処理算法および処理系の研究、並列処理に関する研究・開発に従事。日本ソフトウェア科学会会員。



松尾 正浩 (正会員)

昭和 35 年生。昭和 59 年成蹊大学工学部電気工学科卒業。昭和 61 年成蹊大学大学院工学研究科電気工学専攻修了。工学修士。同年(株)三菱総合研究所に入社。現在、同社情報技術開発部に所属。並列プログラミング、ソフトウェア工学等に関する研究・開発に従事。



和田久美子 (正会員)

1962 年生。1984 年東京理科大学理学部数学科卒業。1986 年同大学大学院理学研究科修士課程数学専攻修了。同年沖電気工業(株)入社。現在、同社総合システム研究所知識情報処理研究部所属。並列論理型 OS 等の並列ソフトウェアの研究開発に従事。