ICOT Technical Memorandom: TM-1237

TM-1237

PIM/p システム入門

今井　明

October, 1992

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

PIM/p システム入門

今井 明 (ICOT 第1研究室)

1992年10月27日

ようこそ、PIM/p の世界へ

本資料は、並列推論マシンモデルP(以下、PIM/p と略します) システム上でプログラム開発を行なう方々のために、留意すべき点を述べたものです。PIM/p は、最大構成で512台[1]の要素プロセッサ (PE) を持つKL1をターゲット言語とした並列計算機で、ようやくユーザの方に使って頂けるような環境を整えることができました。

既にICOTでは、先行して開発されたMulti-PSIや並列推論マシンモデルM(以下、PIM/m と略します) の PIMOS 上で応用プログラムを開発され、PIMOSの改良および KL1 のプログラミング手法の開発などが行なわれてきました。そのような開発に携わった人にとって、PIM/p システム上のプログラム開発環境が、従来と一部異なる点に違和感を感じられるかもしれません。もちろん、マシンの違いによる操作方法の違いなどない方が良いに決まっていることは承知の上で、できるだけ差異が生じないように設計して参りました。事実、PIMOS へのログインの方法、シェルの使い方などのユーザインタフェース部分に関して、マシンの違いを意識することはほとんどないと思います。しかしながら、PIM/p のハードウェア上の特長を生かすために、やむを得ず操作方法を変えてしまった部分が存在するのも事実です。本資料では、その変えてしまった部分に重点をおいて説明します。

なお、本資料は、Multi-PSI, PIM/m (以下、特に区別が必要な時以外は単に PIM/m と書いて両システムを代表させます) で PIMOS に親しんだ経験のある人を対象読者としています。PIMOS を初めて触る人は、まず [1, 2] を一読されたあとで本資料をお読みになることをお勧めします。

1 PIM/p と PIM/m のハードウェアの違い

1.1 概要

まず、PIM/p と PIM/m (Multi-PSI) のハードウェアの違いを列挙しますと、

- PIM/p には8台のPEが共有メモリ / 共有バスで接続されたクラスタと呼ばれる部分が存在しますが、PIM/m には共有メモリは一切存在しません。
- PIM/p の機械語は非常に単純な命令 (RISC ふう) であるが、PIM/m の機械語は抽象 KL1 機械語 (KL1 向き CISC) です。

となります。更に、PIM/p のハードウェアの詳細を知りたい方は、[3] を御覧になるといいでしょう。

[1] 当面 256 台で、1992年12月末に 512 台になる予定。

1.2 クラスタ構成

PIM/p は、要素プロセッサ(PE)をクラスタによって階層的に接続し、通信コストに局所性のある接続方式をとっています。

各クラスタは、共有バス / 共有メモリによって結合された 8 台の PE からなり、クラスタの内部では、メモリからの値の読み出し、メモリへの値の書き込みといった単純かつ高速な PE 間通信が実現されています。この特長を生かすため、クラスタ内の各 PE は、KL1 のゴールを単位として、自動的に負荷の分散を行なう機能(**自動負荷分散機能**)が KL1 処理系に実現されています。つまり、負荷分散プラグマによって陽に負荷分散を指示しなくても、アイドルになった PE は、クラスタ内の他の PE から、KL1 ゴールを取ってきて実行するようになっているのです。

要点 クラスタ内では指示しなくても自動的に負荷が分散されます

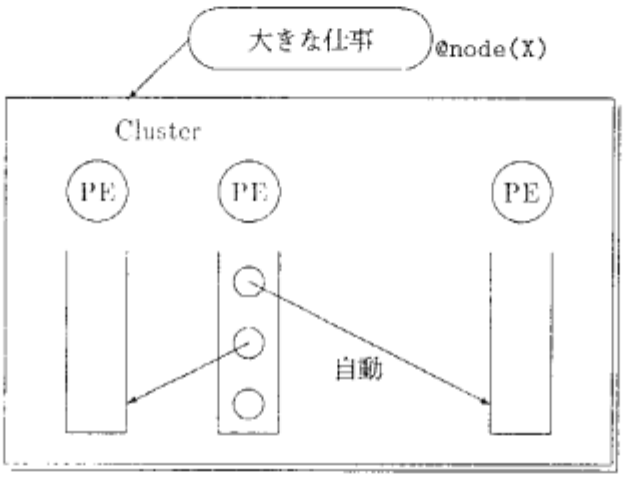

図 1: 自動負荷分散

一方、クラスタ間に渡る通信は、PIM/m と同様に、非同期のメッセージパケットを交換することで行ないます。このため、PIM/p におけるクラスタ間の通信のためのオーバヘッドは、PIM/m の PE 間通信に必要なオーバヘッドとほぼ同等であると言えます。このため、クラスタ間に渡る自動負荷分散機能は実装せず、PIM/m 同様にプログラマが `@node` プラグマによって指定したときのみ行ないます。

なお、クラスタの接続は、PIM/m のようなメッシュではなく、ハイパーキューブでの接続となります。ハイパーキューブ接続において、クラスタ間の距離は、クラスタ番号を 2 進数で表現した時の異なるビットの数(ハミング距離)ですが、実際の KL1 プログラミングでは、特にこの距離が問題になることはないと思います。

要点 PIM/p では、ノードは 8 台の PE を持ったクラスタになります

1.3 RISC ふう機械語命令セット

PIM/m では、KL1 を中間コード(KL1-B)にコンパイルし、それをマイクロプログラム(ファームウェア)によって解釈実行するという形態をとっています。ですから、日々 KL1 の処理系をチューニングしても、KL1-B の命令体系を変えない限り、KL1 プログラムの再コンパイルなしに実行ができます。

ところが、PIM/p の場合、プロセッサが実行するのは RISC ふうの命令セットです。KL1-B のある命令は、システム固有の領域に格納された KL1 処理系への単純な jump 命令になったり、数命令の組合せになったりします。ここで、KL1 処理系をアップデートすると、KL1 処理系への jump 命令の飛び先アドレスの変更が必要になります。PIM/p の場合、些細なアップデートで jump 命令の飛び先アドレスが変更になった程度であれば、KL1 プログラムの load 時に解決できるのですが、KL1 処理系への jump 命令だったものを複数命令の組合せに変更した場合などは、KL1 プログラムからコンパイルのし直しが必要になることもあります。

同じ PIM/p のコードでも、KL1 処理系[2]のバージョンが変わるとオブジェクトコンパチブルでなくなることがあります。できるだけコンパチビリティを保ったまま、KL1 処理系の更新を行なう予定ですが、再コンパイルが必要になった場合[3]も、高速実行のための改良だと思って温かい目で見守ってやって下さい。

余談

PIMOS(特にセルフコンパイラや例外処理) などの開発にあたり、モジュール操作組込述語の仕様変更がなされています。これは、PIM/p の機械語が RISC ふうであるため、共有メモリ中に置いた KL1 命令列にはタグを持たないことなどの理由によります。ただし、一般のユーザにはまず関係ないと思います。

2 PIM/p の CSP

(Pseudo) PIM/m では、マスター FEP 上に CSP (Console Processor) と呼ばれるウィンドウが表示され、ここから、マシンの立ち上げ、初期化、シャットダウンなどを行なっていました。この CSP プログラムは、PIMOS の一部のように見えていたかも知れませんが、実は PIMOS とは独立の、マシンのメンテナンスを行なうためのプログラムで、PIMOS の FEP 機能[4]とは全く別のものです。ですから、FEP の PSI 上で CSP が動いていたのは、実は「たまたま PIM/m ではそのように作るのが一番作りやすかった」という程度の意味しか持っていなかったわけです。

PIM/p の場合にも、そのような役割を担うプログラムは同様に CSP と呼ばれます。ただし、これは FEP の PSI 上で動いているのではなく、PIM/p に接続された SUN-4 ワークステーション上で動いています。たまたま PIM/p ではそのように作るのが一番作りやすかったからです。ですから、立ち上げの時に出る

```
***** LOCAL LOGIN SERVICE STARTS
```

というようなメッセージや、組込述語 display_console の出力などは全て SUN-4 上に表示されます。

PIM/p の立ち上げ方法、使用 PE 数の変更方法などは、PIM/m とはまるで異なる部分ですので、これは別の資料 [4] に説明されています。また、CSP の操作方法 (普通は知る必要はありませんが) は、[5] を御覧下さい。

---

[2] PIM/m では「ファーム」という言い方が一般的ですが、PIM/p の場合は「ファーム」と呼ぶことがふさわしくないので、しばしば「KL1 処理系」または単に「処理系」と呼ばれます。

[3] この場合、.sav ファイル (後述) をロードする時に、"unknown runtime routine" というエラーが出る場合が多い。

[4] 例えばリスナのウィンドウを表示したり、ParaGraph でグラフを表示したりなど。

3 PIMOS 機能使用上の注意

3.1 PIM/m とオブジェクトコンパチではない

Multi-PSI で動いていたコードは、そのまま PIM/m でも再コンパイルの必要なく、アンロードしたファイルをロードしさえすれば正常に動作していましたが、PIM/p の場合は機械語体系が完全に異なるため、KL1 プログラムをコンパイルし直さないといけません。コンパイルの方法については後述します。

弁解 SUN-3 と SUN-4 は、CPU が異なるので、オブジェクトコンパチではありません。

3.2 トレース・スパイには専用のコードがある

リスナ上で、プログラムの実行状況をトレースしたり、ある述語にスパイポイントを設定してから実行を行なうといったデバッグ機能は、PIM/p にも実装されています。ただし、これらの機能を使用するためには、この機能が使用できるように、**KL1 プログラムの再コンパイル**が必要になります。つまり、PIM/p の場合、

- トレース・スパイもできるが、オブジェクトサイズが大きく、トレース・スパイをしない時の実行が遅いコード
- トレース・スパイはできないが、オブジェクトサイズが小さく、その代わりに高速な実行ができるコード

の二種類が存在することになります。UNIX 上の C 言語で、シンボリックデバッガを使用したことのある人には、前者が (cc -g)、後者が (cc -O) でコンパイルされたコードだと思って頂いて結構です。なお、PIMOS のプログラムは、すべて後者でコンパイルされていますので、PIMOS ユーティリティを呼び出した場合 (例えばブールなど)、その中身はトレースできません。

なお、コンパイラでこの 2 種類のコードを出し分ける詳しい使用法は、3.3節にて説明します。

弁解 (Advanced Course)

この区別は、ユーザの方には面倒かもしれませんね。実は、PIM/m ではハードウェアのサポート (KL1-B 命令をフェッチする毎に割込が発生する) があってはじめて、トレース・スパイを行なわない時にも余分な実行時間がかからないような仕組みが実現されていたのです。しかし、PIM/p の場合には、KL1 の中間コードを解釈実行するのではないので、PIM/m のようなハードウェアサポートがあったとしても、それをうまく使えないのも事実です。これは、ユーザの KL1 コードも、処理系の中の (例えばメモリ管理の) コードも、同じ RISC ふう命令を用いているので、1 命令毎の割込の発生を、KL1 コード中を実行している時のみに限定する仕組みが必要です。それだけではなく、ユニフィケーションはコンパイルされたコード中では数命令で実現されたりしますから、KL1-B の抽象命令単位で、真に有効なトレース用の割込を数命令に 1 度しか発行させないような工夫も必要になってきます。このため、PIM/p ではトレース・スパイに割込を使用せず、専用のコードを用いているわけです。

PIM/p 上でスパイ・トレースを実現するに当たっては、前者のようなコードしか生成しないという案もありましたが、やはり少しでも速く動くコードを生成できた方がいいということになり、今の二重化コード方式を採用したのです。

3.3 コンパイラ

既に述べたように、機械語形式が PIM/m と異なるので、PIM/m のアンロードファイルを PIM/p にロードすることはできません。PIM/p 用のコンパイラを用いて、PIM/p 用の

コードを生成する必要があります。

3.3.1 セルフコンパイラ

compile コマンドは、バッチおよびインタラクティブな方法でコンパイル・リンク・ロードするコマンドで、使用方法は PIM/m と同様です。ただし、次のような拡張がなされています。

環境変数 compile:debug

on を指定した時、トレース・スパイのできるコード、off を指定した時、トレース・スパイのできないコードを生成します。省略時は、**on** となる。on の場合、トレース・スパイを行なわない場合に、off のコードよりも実行が遅いことに注意が必要です。

なお、on でロードしたプログラムをアンロードした場合、次回 load コマンドでロードした場合の環境変数 compile:debug が off であっても、on のコードでロードされます。つまり、環境変数 compile:debug が有効なのは、コンパイル時のみです。

3.3.2 クロスコンパイラ

(Pseudo) PIM/m 上で動く、PIM/p 用のコードを生成するクロスコンパイラも用意されています。

```
icpsi567::>sys>user>pimp-kc>kc.sav
```

をロードして[5]、

```
SHELL> kc("*.kl1", output=sav("all"), debug=on)
```

または、

```
SHELL> kc("foo.kl1", "bar.kl1", output=sav("all"), debug=off)
```

などとすれば、後は、PIM/p 上で load("all") を行なうだけです。

注意 1 icpsi567::>sys>user>pimp-kc>kc.sav は時々更新されます。常に最新のものを使わないと、PIM/p 上にロードできる保証はありません。

注意 2 出来上がったファイル (例では all.sav) は、PIM/m では実行できません。

3.3.3 運用上のお願い

実際に PIM/p 上のコードを生成するに当たっては、

1. 4.4節で説明しますが、KL1 の言語仕様に定められていない点の解釈方法が PIM/m のコンパイラと、PIM/p のコンパイラで違う点があり、その確認を予め行って欲しいこと
2. PIM/p システムは世界中に 1 セットしかなく、できるだけ PIM/p の予約利用時間をコンパイルで潰して欲しくないこと

の理由により、可能な限り予めクロスコンパイルを行なうことをお願いいたします。

[5]念のために申し上げると、これは (Pseudo) PIM/m 用のアンロードファイルです。

3.4 ランタイムモニタ

PIM/m にもある xrmonitor, xpmeter も PIM/p で動作しますが、表示形式が同じため、クラスタ構造が良くわかりません。PIM/p ではクラスタ構造をわかりやすく表示する、xpim の方が、ハードウェア構成に忠実であると言えましょう。

また、実際にアネックスマシン室[6]に PIM/p の筐体を見に来て下さい。PIM/p の本体には LED のパネルがついており、これはまさにリアルタイムに PIM/p の実行状況を鮮やかな色で表現します (図2)。ここで、各 PE あたり 4 個の LED の色は、図4のような意味を持ちます。

なお、1 筐体に 4 クラスタが格納されており、上から見た時のクラスタ番号は図3のようになっています (64 クラスタの接続時のクラスタ番号は予定)。

4 プログラミング上の注意

PIM/p 上で KL1 プログラムを動かすにあたり、

> 従来 PIM/m で正常に動いていたプログラムは、PIM/p でも一切変更なく正常に動作する

のが原則です。ただし、注意すべきことは、

> 従来 PIM/m で効率良く動いていたプログラムが、PIM/p でも一切変更なく効率良く動作するとは限らない

点です。特にクラスタ構成を持つことが、プログラムの実行に対して様々な (プログラマの意図しない) 挙動の変化をもたらすことは充分に考えられます。

4.1 ボディゴールは本当に並列に動作する

クラスタ内自動負荷分散機構により、

```
p(X) :- true | q(Y), r(Y), s(Y,X).
```

と書いたプログラムは、q/1, r/1, s/2 が本当に並列に動作します。もちろん、

```
q(X) :- true | X=a.
r(a) :- true | true.
```

のように、r が q の出力を待つような場合、q,r の順で逐次的に実行されることは、KL1 の言語仕様からしていうまでもありません。このような暗黙の同期機構のおかげで、本当に並列に動作したために現れるバグというものに出合うことはまず考えられません。実際に、PIMOS という KL1 で書かれた巨大なアプリケーションプログラムを PIM/p に移植するにあたって、本当に並列に動作したために出たバグというものに未だに遭遇したことがありません。

ただし、注意すべき点として、

```
p(X) :- true | q(Y), r(Z), s(X,Y,Z).
q(X) :- true | X=a.
r(X) :- true | X=b.

s(X,a,_) :- true | X=a.
alternatively.
s(X,_,b) :- true | X=b.
```

[6]将来、地下1階マシン室に移設される予定です。

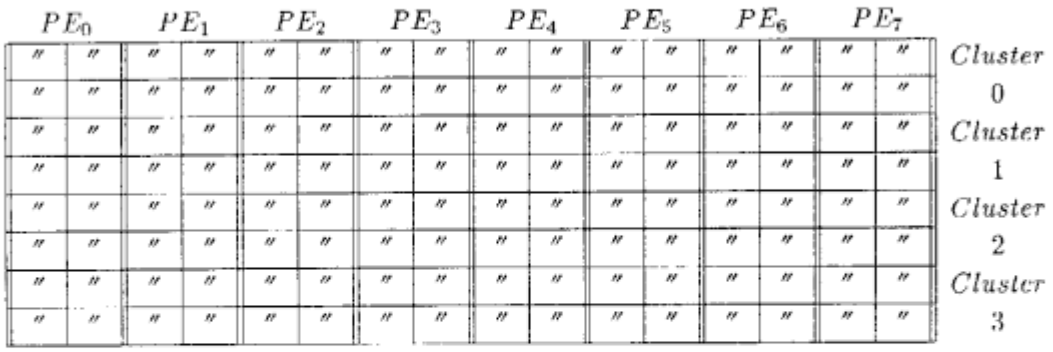

図 2: PIM/p の LED パネル

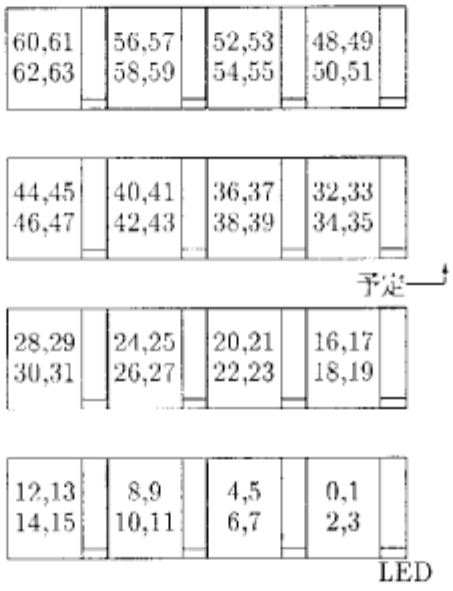

図 3: クラスタの配置

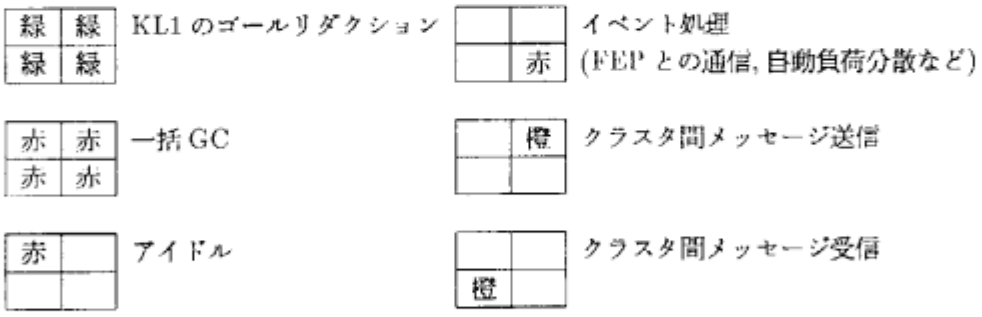

図 4: LED の色と意味

のようなプログラムで、:- p(X) の呼び出しの結果は、PIM/m では (たまたま q, r の順で逐次的に動くので) 必ず X=a がユニファイされますが、PIM/p の場合、q/1, r/1 のどちらが先に reduce されるかは実行する度に異なる可能性があるので、X=a となるか X=b となるかは「神のみぞ知る」ことになります[7]。

これに似た問題として、マージャの入力順に依存するようなコーディングが意図通り動かないことがあります。例えば、PIM/m では、

```
p :- true | merge({In1, In2}, Out), gen1(In1), gen2(In2), con(Out).
gen1(S) :- true | S=[1,2].
gen2(S) :- true | S=[3,4].
con([1,2,3,4]) :- true | true.
```

のようなプログラムは、(たまたま gen1/1, gen2/1 の順で逐次に実行されるので) 毎回成功しますが、PIM/p の場合には成功するとは限りません。これは、gen1/1, gen2/1 が並列に動作するので、マージャに対し 1, 3 のいずれの入力が先に来るか、これまた「神のみぞ知る」ことになります[8]。このようなプログラムは書くべきではないでしょう。

4.2 プライオリティが厳密に守られない

PIM/p にも実行効率を制御するためのプライオリティの概念は存在します。ただし、プライオリティ指定による実行効率の変わり方が PIM/m と PIM/p で異なるということはあり得ます。

現在の PIM/p 上の KL1 処理系では、各 PE 毎にプライオリティキューを持たせています。ですから、各 PE 内では、プライオリティの上下関係は厳密に守られます。ただし、PE 間でのプライオリティの均等化は (残念ながら) 図られていません[9]。ですから、先ほどの例で、

```
p :- true | merge({In1, In2}, Out),
     gen1(In1)@priority(4000), gen2(In2)@priority(2000), con(Out).
gen1(S) :- true | S=[1,2].
gen2(S) :- true | S=[3,4].
con([1,2,3,4]) :- true | true.
```

とやってみたところで、PIM/p 上で成功するとは限りません。プライオリティは、本来プログラムの意味を変えるものではなく、実行効率のみを変えるものであることを再認識して下さい。

また、プライオリティを用いて、クラスタ間自動負荷分散を行なう (例えば pentomino の「暇だ」ゴール) 方式が、必ずしも意図通り動きません。これは、クラスタの中に他の PE が実行可能なゴールを抱えているにもかかわらず、プライオリティの低い「暇だ」ゴールが動いてしまうからです。

4.3 クラスタに仕事がなさ過ぎはしないか?

PIM/m 向けに特にチューニングされたプログラムでは、ノード内に実行可能なゴールの数を極限まで減らして、サスペンションによるコンテキストスイッチが起きないような工夫がさ

[7] まず、このような書き方はしていないとは思いますが。

[8] 余談かも知れませんが、KL1 の仕様上、1 が 2 より、3 が 4 より先に con に到着することは保証されます。

[9] 今後、均等化する可能性は高いのですが…。

れていることがあります。このようなプログラムを、ノード(クラスタ)内に PE が 8 台もあるマシンに持ってきても、効率良く動きません。

また、(特に PIM/m 256 PE システムにチューンしたプログラムに多いのですが)、0 番ノードでユーザプログラムを実行しないようにしたプログラムも見受けられます。PIM/m では、FEP との I/O, KL1 処理系が発行するノード間メッセージ処理のなどのため、特に 0 番ノードに仕事を割り付けない方がいい場合もあるのですが、PIM/p の場合、8 台のプロセッサを遊ばせてしまうことになります[10]

もし、ランタイムモニタや LED パネルの色から稼働率が思わしくなかったり、台数効果が思わしくなかった場合には、クラスタの中に充分仕事があるかどうかをもう一度チェックしてみて下さい。

4.4 クローズの上下関係は意味を持たない

この節だけは、PIM/p がクラスタ構成を持つことに起因するのではないのですが、KL1 の言語仕様に、「クローズの上下関係は、otherwise, alternatively がない限り意味を持たない」というのがあります[11]。つまり、上に書いたクローズと、下に書いたクローズで、どちらを優先するかは、それを処理する側が適当に解釈してもよいということです。例えば、下のようなプログラム、

```
p(1,E) :- true | E=one.          %% (a)
p(2,E) :- true | E=two.          %% (b)
p(_,E) :- true | E=abnormal.     %% (c)
```

では、p(1,E). という呼び出しに対して、(a) を選択して、E=one としても良いし、(c) を選択して、E=abnormal としても良いとするのが KL1 の言語仕様です。

PIM/m のコンパイラでは、複数のクローズが選択可能な時には、(たまたま)上のクローズ優先というコードを生成していました。PIM/p のコンパイラは、(これまた、たまたま)コミット条件に包含関係がある場合($A \subset B$)、A が与えられなかったのと同じコードを生成します。すなわち、上述の例では

```
p(_,E) :- true | E=abnormal.
```

とだけ書かれた場合と全く同じコードを生成します。この例では、おそらく記述した人は、p(1,E). に対して、(a) を選択して、E=one となるのを期待されているでしょうから、その場合は KL1 の言語仕様に従って、正しく

```
p(1,E) :- true | E=one.
p(2,E) :- true | E=two.
otherwise.
p(_,E) :- true | E=abnormal.
```

のように、otherwise を挿入して下さい。

なお、不必要なクローズとみなしてそのコードを生成しなかった時に、PIM/p の KL1 コンパイラは警告メッセージを出します。例えば前述の例では、

```
WRN[IX]:4:p/2-2: This clause is redundant, so compiler delete it. (see #3).
WRN[IX]:3:p/2-1: This clause is redundant, so compiler delete it. (see #3).
```

[10] ちなみに、PIM/p の場合、ノード間メッセージ処理は、どの PE も master-slave の区別なく処理します。ですから、クラスタ内のある特定の PE のみがノード間メッセージ処理を行なうということは考えにくいのです。

[11] もちろん、Prolog では上下関係が意味を持ちます。

のように表示します。つまり、この例では、p/2 の第2クローズは不要であり、第3クローズに吸収合併されたよと言っています。第1クローズも同様です。逆に言えば、このような警告メッセージが表示された時は、その述語定義に otherwise の抜けがないかどうかを、実行前にチェックしてみた方が賢明です。

弁解

例えば C 言語でも、言語仕様上定義されていないのだけれど、たまたまあるマシン (処理系) ではユーザの意図したように動くということはあります。

```
main()
{
   int i=0;
   printf("%d %d %d\n",i++,i++,i++);
}
```

のような C 言語のプログラムの場合、オートインクリメント (++) がいつの時点で行なわれるかに関して定義されていないので、

ハードウェア (compiler)	実行結果
Symmetry (cc)	2 1 0
SUN-4 (cc)	1 0 2
SUN-4 (gcc)	0 1 2
ME (cc)	2 1 0
MIPS (cc)	0 0 0

のように、マシン (処理系) によって動作が異なることがあります。言語仕様に定められていないことは、実装しやすいように実装するということは、効率良い実装のための常套手段です。つまり、このような曖昧なプログラムを書くことは誤解の元です。

5 PIM/p を用いた実験上の注意

5.1 KL1 処理系は日々改良される

われわれ PIM/p 開発グループは、現在の PIM/p の絶対性能に決して満足していませんし、今後も満足することはないでしょう。何が性能のボトルネックになり、どこを変えればもっと速くなるという研究を今もなお続けています。そこで、PIM/p 処理系の改良は日々なされているものだという先入観を持って下さい[12]。特に台数効果の測定に当たっては、必ず同じ日に、あるいは KL1 処理系およびコンパイラのバージョンが同じであることを確認する[13]ことをお忘れなきようにお願いします。

5.2 PIM/p 向きのプログラミングパラダイムは見つかっていない

「PIM/p ではこのように書くと効率良く動作する」という方式は、まだ見つかっていません。それを見つけるのはあなたかも知れません。見つけると論文を書いて発表に行けるという特典が残されています。

[12] 実際に FGCS'92 が終ってから今までの間に、特定のプログラムでは 3 倍近く、平均でも 2 倍近く速くなっています。

[13] その方法は、[4] を参照して下さい。

また、現在実装されている自動負荷分散方式は、たまたま選んだ一方式に過ぎません。「どうも思ったように自動負荷分散がされていない」と思った時は、皆さんのプログラムに問題があるのではなく、我々にも問題があるのかもしれません。どうぞ遠慮なく PIM/p 開発チームにご相談下さい[14]。我々にとっても、自動負荷分散機能を改良するヒントになるかも知れません。

5.3 Multi-PSI, PIM/m に比べて倒れやすいかもしれない

Multi-PSI は約4年、それをベースにした PIM/m も約1年程の間、ユーザの方々に使って頂いています。その間多くのバグの指摘をして頂き、それをフィードバックすることで、日々 stable なものになってきたわけです。それに比べて、PIM/p は、文字通り FGCS'92 に滑り込みで間に合ったシステムで、やっと今、一般ユーザの方にお使い頂くレベルに到達したばかりです。それゆえ、まだまだ、お使い頂いているうちに KL1 処理系のバグにつき当たってしまい、それから先に進まないというような御迷惑をおかけすることも多いかと思います。そのような場合、今まで通り「PIMOS 照会書」を書いてください。報告して頂いたバグは、できるだけ早急に修正するつもりでおりますので、どうか温かい目で見守ってやって下さいますよう、くれぐれもお願い申し上げます。

参考文献

[1] 「PIMOS マニュアル (第3.0版) - 操作編 -」, ICOT PIMOS 開発グループ, 1991 年 11 月.

[2] 「PIMOS マニュアル (第3.0版)　プログラミング編　」, ICOT PIMOS 開発グループ, 1991 年 11 月.

[3] "Architecture and Implementation of PIM/p", K. Kumon, et. al, In Proceedings of FGCS'92, pp.414–424, 1992.

[4] 「PIM/p 起動・運用・終了方法の手引」西崎 (SSL), 1992 年 10 月.

[5] 「PIM/p CSP マニュアル」西崎 (SSL), 1992 年.

[14] PIMOS 照会書を書く、pimp@icot.or.jp にメイルを書く、icot.sys.pim あるいは pimnet.sys.pim に投稿するなど、いろいろな方法があります。