ソフトウェア基礎論　23－8
(1987. 12. 10)

並列論理型計算機システムにおける負荷分散法について

How to Achieve Good Load-balancing on Parallel Logic Computer Systems

神田　陽治
Youji KOHDA

田中　二郎
Jiro TANAKA

富士通　国際情報社会科学研究所
International Institute for Advanced Study of Social Science, FUJITSU LIMITED

新世代コンピュータ技術開発機構
Institute for New Generation Computing

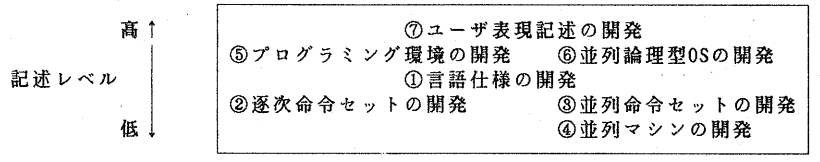
あらまし　並列論理型に関する課題を整理し、提出した課題に対する我々のアプローチを説明する。負荷分散の方式の開発は、並列処理技術の要である。負荷分散の問題を、タスクをじょうずに混ぜてスケジューリングする（マルチタスクミックスする）問題と捉える。問題の解決には、三つの技術が必要である。第一に、システム全体の負荷の様子を安く知る技術の開発。第二に、投入するタスクがシステムにかける負荷の予測。第三に、タスクの分散実行の制御方式の開発である。どれも完全に解くには難しい問題であるが、実用的な方法で解くことはできる。

Abstract　We summarize several subjects concerning parallel logic computer systems and answer some of them. It is particularly important to develop a technique achieving good load-balancing. We concentrate on a type of load-balancing problem on a parallel logic machine: "multi-task mixing". It is broken down into three sub-problems: first, how to know the load-balancing status of a whole system without delay, second, how to estimate the load of a task to be scheduled, third, how to control the distributive execution of the task. These problems are really difficult to be solved completely, but can be implemented practically.

1．はじめに

主な目的は二つある。第一は、並列論理型計算機に関する課題を整理することにより、将来の発展の方向を示すことであり、第二には、中でも重要な、並列論理型計算機システムにおける負荷分散法の問題について、我々のアプローチを述べることにある。なお本稿では、並列論理型というとき、ＡＮＤ型並列論理型に話を絞っているので、注意して読んで欲しい。

最初に簡単に、研究の流れを振り返っておきたい。なぜ、今このときに、「並列論理型計算機システム」を採り上げるかについて、その意義を明らかにしておきたいからである。ＡＮＤ型並列論理型言語を中心とした研究は、世界的にみて、主に三つのグループによって推められてきた。Wizemann Instituteの Shaprioらのグループ、Imperial Collegeの Clarkらのグループ、そしてICOTの計３グループである。研究の歩調が似通っているのは、彼等が互いに交流しつつ研究していることを考慮すれば驚くにあたらない。研究分野を、記述レベルに従って分類してみせたのが、次の図である。



①言語仕様の開発と、②効率良い逐次命令セットの開発は既に完了し、目下、次の段階へと研究は進んでいる。二つの方向が見える。下の方向には③並列命令セットと④並列マシンの開発、上の方向には⑤プログラミング環境の開発が位置する。こうやって見てくると、この次に進むべき方向が読めてくる。それは、並列マシンを充分活用するための、⑥並列論理型ＯＳの開発と、核言語の記述力を高めるための、⑦ユーザ表現記述の開発である。

さて、新しく開発するものには、独自の思想がなくてはならない。本稿の目的はまさに、並列論理型ＯＳや新規のユーザ表現記述の本格的な開発に先立って、解くべき問題や必要な概念を明らかにし、研究分野全体の概観を与えることを狙う。

以下は二部構成となっている。第Ｉ部は、並列論理型に関連する話題の整理と、それにまつわる課題の提示である。第Ⅱ部では、第Ｉ部で提起した課題に対する、我々のアプローチを説明する。

２．並列論理型言語

代表的な言語には、Concurrent Prolog 〔Shapiro83a〕、Parlog〔Clark86 〕、GHC 〔Ueda85〕がある。これら三つの言語は互いに良く似通っている〔Takeuchi86〕。実際のところKernel Parlog と呼ばれるParlogのサブセットと、Flat GHCと呼ばれるGHC のサブセットは、ほとんど同じものだといっても差し支えない。さらに同期の方法が違うことを除けば、Flat CP と呼ばれるConcurrent Prolog のサブセットもまた、ほとんど同じものとなる。

このように、言語設計の核となるアイデアはすでに固まったと言ってよい。核となる言語仕様が、並列言語として望ましい性質を備えていることを2.1 で説明し、続く2.2 で、核の仕様では規定されていないモジュール化について考察する。

２．１　並列言語としての、並列論理型言語

ＡＮＤ型並列論理型言語は、並列言語が持つべき三つの重要な能力を始めから持っている。よって有望株である。三つの能力とは、フロー生成能力、フロー選別能力、そして、フロー協調能力の三つで、フローとは制御の流れ（thread）である。並列言語として真に意味があり役に立つかどうかは、これら三つの能力が高く、かつ、バランスがとれているかどうかで決まる。

①　フロー生成能力

フロー生成能力は、並列言語には欠かせない能力である。逐次言語にはフローはただ一つである。一方、並列言語では複数個のフローが生成できねばならない。しかも、たくさんの数のフローを容易にである。この観点からすれば、逐次言語をベースにフローを陽に起動せねばならない言語では不十分である。並列論理型言語は、容易にフローを分岐し、複数のフローとして起動できる。

②　フロー選別能力

フローをたくさん容易に生み出せても、ただ生み出す一方では、必要な計算が不必要な計算に埋もれ、なにがなんだかわからなくなってしまう。不必要な計算を行うフローを抑え、必要な計算を行うフローを選択する仕組みが必要である。これは並列処理における同期の問題であり、並列論理型言語ではガードを用いる。ガードは並列に起動された一群のフローから、ただ一つのフローを選択し、残りのフローを強制消滅させる仕組みである。

③　フロー協調能力

フロー生成能力によって計算を行うフローを生成し、フロー選別能力によって必要な計算を行うフローを選択できても、必要な計算を行うフローが互いに全く無関係にしか動けないなら、全体としてまとまった仕事を果たすことはできない。フロー間で秩序だったデータ交換を行える仕組みが必要であり、これは並列処理における協調の問題である。実際のところ、三つの言語、Concurrent Prolog 、Parlog、GHC の違いは、フロー協調の手法の違いにある。

Concurrent Prolog では、変数に「読み出し専用」のマークを付けることができる。あるフローの中で、変数に読み出し専用のマークが付けられている場合には、その変数に値を代入できず、並列に動いている別のフローが変数（それには読み出し専用のマークが無い）に具体値を書き込むのを待たなくてはならない。Concurrent Prolog のフロー協調の仕組みは、このようにデータフローに沿ったもので、いろいろ凝った協調のやり方を実現できる強力な利点を持つ一方、コントロールフローとは別建てに考えてやらなくてはならない難しさも持っていた。

ParlogもGHC では、Concurrent Prolog と違い、フロー協調の達成に「モード宣言」を用いる。各変数は「読み出し専用」か否かが、コントロールフローに沿って決定される。ただしParlogではプロシージャ単位に陽にモード宣言を行うが、GHC では変数の出現場所によってクローズ単位にモードが暗黙に決まる。

２．２　ユーザ記述言語としての、並列論理型言語

並列論理型言語はまた、記述能力の点でも優れている。システム記述言語としての並列論理型言語は、３章に譲り、ここではユーザ記述言語としての並列論理型言語について述べる。なお、例題に用いる並列論理型言語は、Concurrent Prolog である。

①　オブジェクト指向型プログラミング

ストリームによるメッセージ通信を、フロー協調の仕組みとリスト構造で実現できる。ストリームを使えば、メッセージに順序を付けて送ることができる。メッセージが変数を持っていてもよいことを利用した、不完全メッセージという技法を使えば、一本のストリームで双方向の情報伝達が実現できる〔Shapiro 84〕。制限付バッファという技法を使えば、ストリームの伸長の限度を設定できる〔Takeuchi85〕。

ストリームをうまく使うと、言語の枠内でオブジェクト指向的なプログラミングができる。基本的なアイデアは〔Shapiro 84〕に尽きる。そのアイデアは簡単であって、次のクローズに示されている。

```
$( [ message | Msgs] ,State ) :-
        /* message の受入検査 */ |
        /* 新しい状態 NewState を、今の状態 Stateから算出 */,
        /* message の中の変数を介して、返値もできる */,
        $( Msgs?, NewState? ).
```

クローズの述語の名前は問題ではないので、とりあえず「＄」とした。クローズ一つで一種のメッセージを

処理するから、取り扱うメッセージの種類数だけのクローズが必要である。上のプログラムでは、第一引数が外からのメッセージストリーム、第二引数がオブジェクトの状態を保持する。オブジェクトの状態の一貫性を保ための排他制御は、ガード「｜」及び、再帰呼び出しと読み出し専用記法「？」によって達成されている。排他領域に入るには（セマフォのＰ命令と同じ効果を持たせるには）、ガードを使う。すなわち、各クローズは、自分が処理できるメッセージを選別できたなら、ガードによって他のクローズの実行を封じてしまう。また、排他領域から出るには（セマフォのＶ命令と同じ効果を持たせるには）、再帰呼び出しを使う。再帰呼び出しの引数が「？」によって読み出し専用となっているので、新しい状態が決まるまでは、次のメッセージの処理は始まらない。

第一引数のメッセージストリームが、オブジェクトポインタに当たる。ストリームには、書き込んだメッセージの順序が変わらないという利点がある一方、複数個所から書き込むには、複数のストリームを一つのストリームにマージする mergeを間に挟みこまねばならない。例えば、下に示すようにする。

```
producer(A), producer(B), merge(A?,B?,M), $(M?,initStatus).
```

②メタコール

言語の仕様が簡単なため、メタコールもそれ自身で記述できる。すなわち、プログラムとして書き下せる。これをメタインタプリタと言う〔Shapiro83b〕。以下のプログラムは、

```
call(Prog,Goals)
```

なるメタコールを実現するメタインタプリタである。Progはプログラムデータベースであり、Goals は、プログラムを起動するゴールの指定である。

```
call(Prog,true).                                          % halt
call(Prog,(A,B)) :- call(Prog,A?), call(Prog,B?).         % fork
call(Prog,A) :- system(A) | exec(A).                      % system
call(Prog,A) :- clauses(Prog,A,Cs) |                      % reduce
          resolve(Cs,Body), call(Prog,Body?).
resolve(A, [(Head:-Guard | Body) | Cs] ,Body) :-
          unify(A,Head), call(Guard) | true.
resolve(A, [(Head:-Body) | Cs] ,Body) :-
          unify(A,Head), Body≠(-|-) | true.
resolve(A, [ C | Clauses ] ,Body) :-
          resolve(A,Clauses,Body) | true.
```

ここで、system/1はすぐ実行できる述語かどうかを判定し、exec/1はそれを実行する。clauses/3 は、第一引数のプログラムデータベースを探し、第二引数のゴールとユニファイ可能なクローズを探して第三引数にリストで返す。unify/2 は二つの項をユニフケーションする。ただし、このユニフケーションは読み出し専用記法を扱えるように拡張されている。

③ユーザ表現記述

並列論理型言語のユーザ表現記述は、仕事向きの記述様式をユーザに提供することを目的とする。実際、Concurrent Prolog 、Parlog、GHC は、核の言語仕様でしかない。ユーザ表現記述の中心的枠組みは、モジュール化の仕組みである。ここで言うモジュールは｛プログラム、メタコール｝の二つ組、

```
{Prog, Call}
```

であり、そして、モジュールの起動時には、

```
Call(Prog,Goals)
```

なる形が作られ、直ちに実行される。ただし、Progはプログラムデータベース、Callはメタコール、Goals はプログラムを起動するゴールである。

モジュール化の、初期の試みとしてMandala 〔Furukawa84〕、Concurrent Prolog 用のモジュール化例として、Logix 〔Silverman86 〕、そして、Parlog用のモジュール化例としてPPS 〔Foster87〕がある。GHC については検討中ではあるが、〔田中87〕がある。

モジュールにゴールの実行を頼むことは、メタコールでプログラムを実行することである。しかし、これを正直にやったのでは、遅すぎて使いものにはならない。 Mandalaの開発がうまく行かなかった理由は、ここにあった。後続のLogix やPPS が曲がりなりにもうまく行っているのは、モジュールのコンパイル技法の開発に成功したからである。並列論理型言語は仕様が簡単であるので、プログラム変換が容易である。これを利用して、モジュールをコンパイルするのに、二つの方法が考えられる。

最初の方法は、まず、機能拡張したメタコールをメタインタプリタに書き下す。次にメタインタプリタとプログラムを組に部分評価を施して、機能拡張したプログラムを得て、これをマシンコードへとコンパイルする。

もう一つの方法では、メタコールをマシンコードへ直接コンパイルする。メタコールが充分効率よくコンパイルできるように、特別なマシンコードを用意する必要があるだろう。

前者の方法は、Concurrent Prolog 用のプログラミング環境、Logix で採用された方法であり〔Hirsh 86〕、後者の方法は、Parlog用のプログラミング環境、PPS の方法である〔Foster 87 〕。

モジュール化以外に、どんな仕組みがユーザ表現記述として役に立つかもまた、研究に値する話題である。例えば、Vulcan〔Kahn86〕の目的は、Concurrent Prolog への上に、オブジェクト指向の皮を被せることにあった。オブジェクトを作り出すとき、ストリームをいちいち張り巡らす記述は大変めんどうなので、この

自動化を狙う。具体的には、Vulcanは、Smalltalk のようなプログラムを、Concurrent Prolog へ変換するプリプロセッサである。

リフレクションは、プログラムの中から、動作環境をデータ化することを許すことにより、自在に動作環境を参照・変更できるようにするメタな仕組みである。GHC に組み込もうとする試みが〔田中87〕にある。システムコールがお仕着せの決まった機能を果たす「お決まり」であるのに対して、リフレクションは基本的な機構 (mechanism)のみを提供するのであって、いろいろなシステムコールをプログラマの責任で実現することができる「素材」である点が異なる。

3．並列論理型システムソフトウェア

システムソフトウェアの代表は、オペレーティングシステム（ＯＳ）である。並列マシンの上で動かすＯＳが、並列論理型ＯＳである。並列論理型ＯＳもまた、並列論理型言語で記述したい。これには少なくとも二つの意義がある。第一に並列論理型言語が並列マシン用のシステム記述言語として使えることを示すことであり、第二は論理型が持つとされる特徴、例えば演繹能力、をシステム記述に役立たせ得ることを示すことにある。

さて、ここで言う並列マシンとは、個々のプロセッサが自分専用のメモリを持ち、それらプロセッサが互いに（適当なトポロジを持った）ネットワークで結ばれたものとする。プロセッサは互いに対等であり、基本的にはゴールをどこのプロセッサに割りつけても良い。また、プロセッサ間通信コストが、遠くと通信しようとすればするほど高く付くようなものとする。これは現実的な仮定といえる。なお、本稿のなかでは、ネットワークの形状は２次元平面のメッシュであるとしている。

3．1　プログラミング環境

逐次マシンの上で動き、プログラムをいろいろ試せる並列論理型プログラミング環境が、既に試作されている。Shapiro グループのLogix 、Clark グループのPPS などがある。これらのプログラミング環境には、入出力（ＩＯ）管理、ファイル管理（プログラムデータベースなど）、タスク管理（割り込み、ステイタス報告）といった機能が含まれている。プログラミング環境構築のもっとも大きな道具立てが、メタコールあるいはメタインタプリタであり、これを用いてモジュールの階層化が達成されている。

①モジュールの階層化

2.2 のメタインタプリタを、とくとよく考えてみると、実質的な計算は system/1 によってピックアップされて、exec/1で実行されている。他の部分は、exec/1で実行できるまでに、ゴールを詳細化しているのだと言える。これをもう少し詳しく言えば、system/1は、引数のゴールが「この世界」で解ける述語であることを検出しているのであり、exec/1は「この世界」でゴールを実行するのである。そして「この世界」とは、活動が起こっているモジュールに他ならない。

この考えは容易に押し進められる。与えられたゴールがモジュールの世界では解けないときは、それを検出して、他のモジュール世界で解けばよい。それでは他のモジュール世界とは何か。

モジュール群は、メタインタプリタの呼び出しによる関係で結ばれ、全体は、一つの木構造を成しているのだと考える。ゴールが自分の世界で解けるものなら自分の世界で解く。解けなければ道は二つで、計算木の上の世界に戻して解くか、下の世界で解くかである。メタインタプリタの世界は階層を成す。そこで、上の世界とは、自分を呼び出したモジュールの世界であり、下の世界とは、メタインタプリタcallを使って作り出される新世界である。

メタインタプリタは３引数へ拡張される。新たに加えられた引数Env は、自分から上の、caller-callee 関係からなる、インタプリタの層(tower of interpreters) を指し、上の世界をアクセスするために使われる。残念ながら下に示すプログラムは不十分なものであるが、参考のために記す。一つの実現法であるが、モジュールの形でしかEnv を表現していない点に限界がある。

```
call(Prog,true,Env).                                          % halt
call(Prog,(A,B),Env) :- call(Prog,A?,Env), call(Prog,B?,Env).  % fork
call(Prog,A,Env) :- self(A)  | exec(A).                       % self
call(Prog,A,Env) :- super(A) |                                % super
          Env = [ {PROG,CALL } | ENV ]
          CALL?( PROG?, A, ENV?).
call(Prog,A,Env) :- A = call(PROG,CALL,GOALS)  |              % call
          CALL?( PROG?,GOALS, [ {Prog, call} | Env ] ).
call(Prog,A,Env) :- clauses(Prog,A?,Cs)  |                    % reduce
          resolve(Cs,Body), call(Prog,Body?,Env).
```

ここに、ガード述語self/1は、自分の世界で処理できるゴールを検出する。（実はsystem/1と同じ。）実際の実行は、exec/1が行う。ガード述語super/1 は、上の世界で解くべきゴールを検出する。環境引数Env を使って、呼び出しモジュールを見つけ出して、それに実行してもらう。Env が一部の情報しか覚えないので、呼び出し時の環境まで復帰することはできてはいない。ゴールがcall(PROG,CALL,GOALS) のときは、別のモジュール {PROG, CALL} を起動して、新しい世界を作り、そこでGOALS を解いている。なお、プログラム中で、CALL?(PROG?, A, ENV?)などは許されない書き方であるが、述語呼び出しの略記法として用いた。

② PPSプログラミング環境

PPS は、本質的には、①で述べたモジュールの階層化をそのまま使っている。ただし、環境を直接持つことをしないで、環境を作るための素材が提供される。具体的には、インタプリタと外界が交信するための二つのストリームが代わりに提供された。call/3〔Clark 84〕や call/5 〔Clark 87〕である。

call/3は、以下の形をしている。Goals が投入ゴール、Statusが実行時情報で、Control で実行状態を制御できる。

call(Goals,Status,Control)

call/5は、以下の形をしている。Program はプログラムデータベースの指定、Priorityは、計算優先度指定である。

call(Program,Priority,Goals,Status,Control)

なお、call/3はメタインタプリタで書くことができるが、call/5になると資源の管理を含むために、簡単には書けない。

③ Logixプログラミング環境

Logix のモジュール化機構は、オブジェクト指向である。モジュールはインスタンスを生み出すテンプレイトの役目を果たし、インスタンス同士はストリームを介してメッセージ通信を行い、計算が進む。オブジェクト指向の世界は、本質的には①で述べたモジュールの多層世界と同じである。もっとも上の層が、メッセージの収集分配を担当する。ただLogix では実行速度を確保するために、コンパイラがプログラム変換を行ってしまうので、メタインタプリタの階層が陽に表に出て来ない〔Hirsch86〕。

3．2　並列マシン上の並列論理型ＯＳ

残念ながら、逐次マシンの上のプログラミング環境を、そっくりそのまま並列マシンの上に持って来て、それで事足れりというわけにはいかない。解くべき重大な問題がまだ残っている。そもそも、並列マシンを大きいタスクを流すバッチ専用機として使うというのではもったいない。ふつうの計算機として、複数人が複数のタスクを動かせる、マルチユーザ・マルチタスクの計算機システムとして使いたい。並列マシンには、たくさんのプロセッサが載っている。負荷を各プロセッサにじょうずに振り分けて分散を図り、並列マシン全体で高スループットを達成したい。

並列論理型言語の実行は、解くべきゴールとプログラムとして登録されたクローズのヘッドとの一致を試し、ガードを試し、成功したらボディを次のゴールとして投入する作業の繰り返しである。投入されたゴールは展開され、数多くの副ゴールを生み出す。これら一群のゴール全体を「タスク」と呼ぶことにすると、複数のタスクを数多くのプロセッサにうまく割り当て、並列マシンの限界性能を引き出すことが並列論理型ＯＳに課せられる。これを「マルチタスクミックス」の問題と呼ぶことにする。マルチタスクミックスの機能は、並列型ＯＳには不可欠の機能である。それならば、どのようにすれば、じょうずにマルチタスクミックスを達成できるだろう。そのためには、三つの事が可能でなくてはならない。

第一に、各プロセッサの混み具合、すなわち計算機システム全体のロードマップ(load map)がわかること、第二に、これから走らせようとするタスクの「性能」を予測できること、第三に、タスクの分散実行を制御できること、である。「性能」を正確に定義することは難しいが、ここでは、タスクの実行がシステム全体のロードマップに与える影響の予測値と考えておけば充分である。並列型論理型ＯＳは、次のようにしてタスクをスケジューリングする。第一に、ロードマップを見て負荷の低い所を発見し、その個所にタスクを投入しようと決心する。第二に、性能予測値を使って、最適な投入になるよう戦略を決める。第三に、立てた戦略を実現するべく、タスクの分散実行を制御する。

マルチタスクミックスの問題を解決することは、プログラムのローディングについての問題も解決する。すべてのプロセッサの局所メモリに、プログラムを実行に先立ってローディングしてしまっては無駄が多すぎる。第一に、結局プロセッサにゴールが割り当てられず、せっかくロードしたプログラムが使われず、ローディングの操作自体が無駄になるかも知れない。第二に、マルチタスク下で次のタスクのプログラムをロードしようとしたとき、必要な分の空きメモリが他のタスクのプログラムのために専有されてしまっており、ローディングができないかも知れない。

マルチタスクミックスを行う過程で、タスクの性能予測を行うが、これより、タスクが割りつけられて実行されそうな範囲がわかる。まず、この範囲のプロセッサのメモリにのみ、実行コードをローディングしておく。範囲外のプロセッサにまで、タスクのゴールが割り付けられたときには、そのときローディングする。これによってメモリの浪費を未然に防ぐことが期待できる。

3．3　課題

続く第Ⅱ部においては、じょうずなマルチタスクミックスの達成、すなわち、並列論理型計算機システムにおける負荷分散法について、我々のアプローチを説明する。それには、上に述べた三つの課題に挑戦しなければならない。

４章において、タスクの分散実行を記述し、制御する方法を考察する。プラグマと呼ばれるプリミティブと、ポリシーと呼ぶプラグマのパッケージ化について議論する。

続く５章において、システム全体のロードマップを得る方法と、「性能」の取り扱いについて考察する。システム全体のロードマップを得るには、ハードウェアによる支援なしには手間がかかりすぎて、実用的でないだろう。5.1 で一つのアイデアを示す。また、タスクの「性能」を予測する問題についても5.2 で扱い、一つの解決策を示す。予測という操作は、推理する作業を含む。ならば、この問題の解決に、ＯＳが論理型であることが活かせるではないか、というのが我々の主張である。

4．プラグマとポリシー

マルチタスクミックスの問題を解くためには、タスクの負荷分散を制御する方式が必要であった。このためのプリミティブとして、プラグマが提案された。しかしプラグマは低レベルのプリミティブであって、使いづらい。同期のプリミティブであるセマフォが使いづらいとの似ている。同期のためのリモートプロシージャコールに相当するような、より高度な負荷分散のための処理技術が開発されなければならない。解決への一つのアプローチとして我々が押し進めているのが、プラグマのポリシー化である。

4．1　プラグマとは何か

本来プログラムは、行うべき仕事の機能仕様を記述するものであった。「これこれのときはああしろ」とか、「以下のことを繰り返せ」といった類である。しかしながら、機能だけが仕事のすべてではない。リアルタイムプログラミングでは、要求される時間内に応答したり、返答が無かったときにキャンセルするなど、時間制約の厳守もプログラムで記述すべき仕様となる。プログラミング言語は機能のみならず、他の必要事項も仕様記述できるように拡張強化されねばならない、というのが我々の主張である。

プラグマを次のように定義する。プラグマは、プログラミング言語の一部であり、プログラムの中で使われる。〔プラグマ（pragma）とはギリシャ語で、actionの意味がある。〕しかしそれは機能以外のものを規定し、それが付いていることによって、プログラムをうまく実行可能にするものである。言い換えれば、仮にプラグマをプログラムから全て削ったとしても、そのプログラムは果たすべき最低限のことは行えなくてはならない。しかし、プラグマ無くしては所定の性能がでないかも知れない。

高級言語でのプラグマは、処理系（コンパイラなど）への指示である。コメントが人間への注釈を果たすのに対して、プラグマは処理系への注釈の役目を果たす。処理系は、知らないプラグマは無視して構わない。また、ある処理系だけに特殊なプラグマもありうる。高級言語で仕様を記述し、それをコンパイラでコンパイルする方法の利点は、高級表現から低級表現へ変換する過程で、コンパイラの判断で「善きに計らう」最適化を行える余地が残っている点にある。例えば、コンパイルするマシンの資源に合わせて、最適なレジスタアロケーションをコンパイラは施せる。

いくつかの言語から例を拾うと、C言語では、レジスタ宣言によりコンパイラにレジスタアロケーションを示唆できるようになっている。Ｐａｓｃａｌ言語では、特殊なコメントでソースリスティングの出力を制御する。Ａｄａ言語では、プラグマは正式に採用された。変わったものとしては、動く対象マシンを指定するプラグマがある。

並列論理型言語に対しても、負荷分散の方法を指示するプラグマが提案された。基本的にはShapiro のアイデアであり、他はその変種である。Shapiro の方式はタートルグラフィックスを真似る〔Shapiro 84〕。各ゴールはいま自分がどこにいてどちらの方向を向いているかを記憶していて、「一歩前進」とか「右向け右」といったプラグマ指令に従い、別のプロセッサに渡され実行される。例えば、

```
p :- q@forward, r.
```

は、ゴールp が qとr を生成する点においては、

```
p :- q, r.
```

と同じであるが、異なるのは前者では qというゴールには、どのプロセッサに割りつけるべきかのプラグマ指定@forwardが付けられている点にある。@forwardは一歩前進、@rightならば右向け右の指示である。

Shapiro のプラグマ方式では、すべのゴールに、

（ プロセッサ座標、方向 ）

なるゴール属性が与えられている。子ゴールのq,r のゴール属性は、実行時に親の pのゴール属性から計算される。r のようにプラグマが付いていなければ、ゴール属性はコピーされるだけであるが、q のようにプラグマが付いているときは、新しいゴール属性はプラグマの指示するやり方で親ゴールの属性から計算される。すなわちプラグマは、一つのゴール属性を入力とし、一つのゴール属性を返す関数に他ならない。

それは次のメタインタプリタによく現れている。これは 2.2のメタインタプリタに、ゴール属性Pcb を加えて機能強化したメタインタプリタである。そして、locate/3は、プラグマが指定する方法で、ゴール属性を変換する関数である。

```
call(Prog,Pcb,true).                                          % halt
call(Prog,Pcb,(A,B)) :- call(Prog,Pcb,A?), call(Prog,Pcb,B?).  % fork
call(Prog,Pcb,A@P) :- locate(P?,Pcb,NPcb), call(Prog,NPcb?,A?). % pragma
call(Prog,Pcb,A) :- system(A) | exec(A).                      % system
call(Prog,Pcb,A) :- clauses(A,Cs) |                           % reduce
        resolve(A,Cs,Body), call(Prog,Pcb,Body?).
locate(forward,(X/Y,east),(NX/Y,east)) :- NX is X+1.
   ...
locate(right,(X/Y,east),(X/Y,south)).
   ...
```

さて、ICOTの方式〔Chikayama 86〕は、Shapiro の方式の変形である。Shapiro の方式ではプロセッサの平面座標は絶対的に固定されているが、近山の方式では平面座標がプロセッサの負荷状況に応じて変形を受ける。負荷が低い部分では座標間の間隔が狭まり、高い所では広がる。よって同じ一歩でも、混んでいると

ころでは遠くへの一歩となり、空いているところでは近くへの一歩となる。

Shapiro 式の方式には、しかし、欠点がある〔神田86〕。静的であるがため、コンパイル以前に負荷分散の方法を細かく指示するのは面倒である。さらに、マルチタスクの環境ではタスク同士の負荷がプロセッサ上で重なるのに、それを考慮したプラグマ付けはコンパイル以前では不可能である。タスク同士の干渉の問題の解決に、システムの状況を調べる述語を発明して、これをガード部に置いて、負荷分散の方法を動的に変更することも可能だが、コンパイル以前に充分考えてプログラムしなくてはならない点は解決されていない。しかしながら、負荷分散に役に立つシステム述語が発見できれば、充分価値がある。例えば〔Eishiewy 86〕では、リアルタイム記述に役立つシステム述語を提案している。

我々の考えでは、Shapiro 式の方式の欠点の根本的解決には、プラグマは制約と考え直し、ポリシーの考えを導入しなければならない。

通信し合う二つのゴールを割り付けるときには、通信の負荷を減らすために、なるべく近くに割り付けるべきである。Shapiro のやり方では、よく考えて各々のゴールにプラグマを施し、結果として近くに割り付けられるように、プラグマを付けなくてはならない。というのも、各々のゴールは独立の分散の仕方を指定されるからである。しかし、もっと直接に「この二つのゴールは、なるべく近くに配すること」とプラグマ指定できた方が、ずっと素直で手間もない。この話を一般化するならば、ゴールのプロセッサへの割り付け指定を、手続き的でなく宣言の形で行いたい、あるいは制約の形で行いたい、ということである。

4．2　ポリシーの導入

「戦略と機構(policy and mechanism)の分離」は歴史的に古い手法である〔Wulf74〕。システム側はプリミティブとプリミティブを組み合わせる方法を提供し、プログラマはそれらを使って、戦略（実現したい事柄）をプログラムする。この観点から見れば、Shapiro の方式は、負荷分散のために最小のプリミティブを提供したに過ぎなかった。導入されるべきは、戦略と機構の分離の考え方である。

プラグマはプログラムに直接書き込まれる。従って、コンパイル時には戦略が固定されてしまっていて、実行時に動的に変更することはできない。プログラマとしては、本当に指定したいプラグマだけを指定したい。プログラムの性能に本質的でない事柄まで考えたくはないというのが本音である。どうでもよいことは誰かが「善きに計らって」適当に決めてくれればよいのである。

①動的なプラグマ機構

Shapiro のプラグマは実は二つの指定を同時に行っている。いつ分散するかの指定と、どこに分散するかの指定である。ここでの方法は二つの指定を分離し、どこに分散するかの指定はまとめて外から与え、いつ分散するかの指定のみプログラムに書く。

どこに分散するかの指定を、プログラムとは独立に与えることを考える。この指定を「道」と呼ぶ。道を実行時に与えることにより、動的な分散実行制御が行える〔神田87〕。

Shapiro のプラグマ方式では、すべてのゴールに、

　　（ プロセッサ座標、方向 ）

なるゴール属性が与えられた。ここでは拡張して、

　　（ 道、プロセッサ座標、方向 ）

を使う。下のメタインタプリタにおいて、道が第四引数に与えられる。

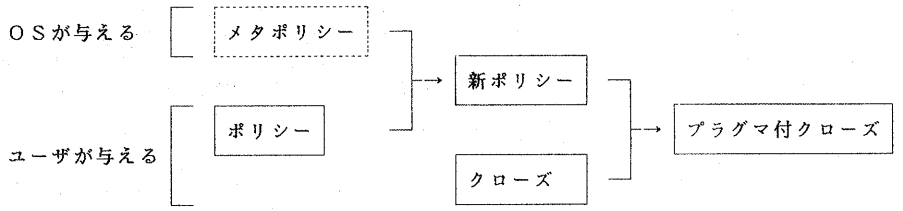
```
call(Prog,Pcb,true,Path).                                    % halt
call(Prog,Pcb,(A,B),Path) :-                                 % fork
        call(Prog,Pcb,A?,Path),
        call(Prog,Pcb,B?,Path).
call(Prog,Pcb,A@P,Path) :-                                   % pragma
        locate(P?,Pcb,NPcb,Path,NPath),
        call(Prog,NPcb?,A?,NPath?).
call(Prog,Pcb,A,Path) :- system(A) | exec(A).                % system
call(Prog,Pcb,A,Path) :- clauses(A,Cs) |                     % reduce
        resolve(A,Cs,Body), call(Prog,Pcb,Body?,Path).
locate(forward,((X/Y),east),((NX/Y),east))) :- NX is X+1.
   ...
locate(advance,(X/Y,Dir),(NX/NY,NDir),  [Pragma | Path] , Path) :-
          locate(Pragma,(X/Y,Dir),(NX/NY,NDir)).
   ...
```

②ポリシー化

いつ分散するかの指定は、プログラムに付ける。しかし直接付けることを止めて、ポリシーとして分離する。ポリシーとは、プログラムにプラグマを配することを指示する、一種のエディタスクリプトである。プラグマが付いていないプログラムをポリシーで編集すると、ポリシーで指定したやり方でプラグマの付いたプログラムが得られる。ここまでは基本的にShapiro の方式そのままである。ここでの工夫は、ポリシー自身を編集の対象とすることにある。与えられたポリシーをそのまま使うのではなく、いったん編集してからプラグマ無しのプログラムに適用する（次図参照）。ユーザはポリシーとプログラムを組で与えるが、ポリシーを系統的にいろいろ編集することで、系統的に多彩なプラグマ付プログラムが得られる〔神田86〕。

ポリシーの記述をどのように行うべきかは未定であるが、高レベルの記述であることが好ましい。既に述

べたように、制約の形で述べられていることが望ましい。制約の形であれば、ポリシーを適応できるプログラムの範囲が広がり、それだけいろいろな戦略をプログラムに与えることができるからである。



５．マルチタスクミックスの実現法

残る二つの課題に対する我々のアプローチを述べる。二つの課題とは、マルチタスク実行時のロードマップを安く手に入れる方法と、各タスクの性能を予測する方法の開発であった。

ここで想定している並列マシンのプロセッサの数は、可変である。とても大きな数かも知れない。そんなときでも、リアルタイムに安くロードマップを手に入れる方法が欲しい。プロセッサ一つ一つに渡る詳細なロードマップが必要ならば、プロセッサの数が増えたとき、転送されるべき情報が禁止的に増加してしまうだろう。しかし、プロセッサの数に弱くしか依存しないロードマップならば、うまく行く可能性がある。そして、マルチタスクミックスの問題はこのケースである。

タスクの「性能」を予測するとは、きわめて難しい問題に聞こえる。現状も、ジョブクラスという形で、タスクを投入する人の責任で「性能」を指定してるのが現実である。しかし、だからといって、「性能」を予測することなどできない、と、すぐ諦めることはない。完全に解くことはできなくても、一部を解くことはできるだろうし、それだけでも充分なことも多い。

５．１　ロードマップの入手

計算機システム全体の負荷の見取り図、すなわちシステム全体のロードマップをリアルタイムに安く作る方法が必要である。これはハードウェアの助け無しには難しい。ソフトウェア（分散アルゴリズム）で情報を集めているようでは、手間も時間もかかってしょうがない。というのも、たくさんの情報を誰かが大急ぎで集めなければならないからである。

マルチタスクミックスに必要なロードマップは、並列マシンのプロセッサの数に弱くしか依存しない情報である。プロセッサ平面は複数の区画に分けられているものとし、そのうち負荷がもっとも軽い区画の番号さえ得られれば良いとする。新しいタスクを、負荷がもっとも軽い区画に投入しようというわけである。

この方法は「負荷の連続性」を仮定している。ある区画を指定したとき、その中のプロセッサの個々の負荷はバラツキが少なく、大きく違わないという仮定である。従って、その区画の中のどのプロセッサにタスクを投入しようとも、大差が無いということになる。この仮定が、どの程度成り立つものなのかは明らかではないが、荒唐無稽のものではない。そもそもこの性質がなり立たなければ、タスクを負荷の軽いプロセッサに投入しようというマルチタスクミックスの問題を考えることに、凡そ意味が無くなるのであるから。

さて、この問題の解決に、次のようなハードウェアを提案しよう。ロードマップメモリを準備する。それは、本質的には（ビットマップディスプレイで使われているような）ビットマップメモリである。プロセッサ平面とロードマップメモリ平面は、適当に対応付けられているものとする。１プロセッサあたり１ビットでも、ｎプロセッサあたり１ビットでも、１プロセッサあたりｍビットでもよい。とにかく平面間の位相が保たれていれば良い。対応の単位となるプロセッサの集まりは、プロセッサ平面全体を区画分けする。

実行中に各プロセッサは、負荷情報を、適当な大きさのランダムな濃淡パタンの形で、ロードマップメモリの対応するビット位置を中心とする近傍へ「ビットＯＲ」で書き込む。この手間は大したことは無かろう。ビットマップディスプレイが普及した今日、矩形のビット演算を高速に行えるＶＬＳＩが開発されていると知った上での判断である。問題は多くのプロセッサからのアクセス競合であるが、ロードマップメモリをバンク分けして競合を無くす努力はもちろんのことであるが、バンク内でのアクセス競合は、ロードマップメモリのアクセスアービタがそのうち一つだけを成功させ、残りのアクセスは無視してしまう。というのも負荷の連続性の仮定があるから、どれが選ばれようと大差無いパタンが書き込まれるはずだからである。その一方、ロードマップメモリは常に、ランダムドットで構成されたマスクパタンを「ビットＡＮＤ」で書き込んでいる。結果として、プロセッサ平面の平均負荷パタンが、対応するロードマップメモリにパタン分布として残るという寸法である。基本的なアイデアは以上のようなものであるが、検討すべき余地は多い。例えば、アクセス競合の際の処置の影響を検討する必要があろう。

ロードマップメモリを利用すれば次のような事が可能になる。ロードマップメモリをそのままディスプレイに表示すれば、負荷の分布の様子がそのまま見てとれる。さらにロードマップの付加ハードウェアが常に繰り返し、その濃淡画像をスキャンして、近傍のパタン濃度が低いビットの番号を常に算出しているものとする。そしてこの番号は、すべてのプロセッサから、特別なレジスタ経由で、いつも見えているものとする。並列論理型マルチタスクＯＳはこの番号を参照して、負荷の低いプロセッサ平面上の区画を割り出すことができる。

プロセッサの数がとてつもなく増えたときでも、それに合わせてロードマップメモリを大きくする必要は

無いことに注意しよう。ｎ個のプロセッサを１ビットに対応させるときの、ｎを大きくして行けばよい。負荷の連続性がまがりなりにも成り立つとき、仮に百万個のプロセッサがあって、822,242 番目のプロセッサに割りつけるのと、822,543 番目のプロセッサに割りつけるのとで、どれほどの差があるだろうか。問題なのは区画の総数なのであって、区画の大きさ（プロセッサの数）ではないのである。

５．２　ヒストリ管理

マルチタスクミックスをじょうずに解くためには、システムのロードマップを手に入れる他に、投入しようとしているタスクの「性能」が予測できなくてはならない。現実問題としては、プログラムだけからの静的な解析だけで、実行時の挙動を詳細に予測することは、困難である。それではどうしたら良いか。

ここでの提案する方式の背景になるアイデアは、「歴史は繰り返す」という原理である。日頃経験することであるが、同じプログラムに同じデータで何回も実行することがよくある。デバッグ時とか、定型的なプログラムを走らせる場合である。そうであるならば、前の計算の経験を後の計算の活動の予測に活かすことはできないだろうか。以前の計算の活動を集めておいて、それを元に、未来の計算の挙動を推論しようというわけである。並列論理型でＯＳを書くという利点を、活かすことができるのではないか。

集めた過去の計算の記録のデータベースをヒストリと呼ぼう。2.2 ではモジュールを、

{Prog, Call}

の二つ組とした。ここでは新しい枠組みとして、モジュールを、

{Prog,Call,Hist}

の三つ組と考え直す。Histがヒストリである。Callはは、ヒストリ管理ができるように拡張されたメタコールである。

以下のメタインタプリタは、タスクに属するゴールが、プラグマによってプロセッサに割りつけられた時のプロセッサの座標を集めて報告する。もちろん、座標に限らず他の有用な情報を集めるように改造することは自由である。メタインタプリタの第５引数には、タスクの実行につれてゴールが割りつけられて行ったプロセッサの座標がすべて集められて来る。

```
call(Prog,Pcb,true,Path,  [] ).                                  % halt
call(Prog,Pcb,(A,B),Path,Hist) :-                                % fork
        call(Prog,Pcb,A?,Path,HistA),
        call(Prog,Pcb,B?,Path,HistB),
        merge(HistA?,HistB?,Hist).
call(Prog,Pcb,A@P,Path,Hist) :-                                  % pragma
        locate(P?,Pcb,NPcb,Path,NPath,Hist,NHist),
        call(Prog,NPcb?,A?,NPath?,NHist?).
call(Prog,Pcb,A,Path,Hist) :- system(A) | exec(A).               % system
call(Prog,Pcb,A,Path,Hist) :- clauses(A,Cs) |                    % reduce
        resolve(A,Cs,Body), call(Prog,Pcb,Body?,Path,Hist).
locate(forward,(X/Y,east),(NX/Y,east)) :- NX is X+1.
   ...
locate(advance,(X/Y,Dir),(NX/NY,NDir),
              [Pragma|Path] , Path,  [ X/Y|Hist] , Hist) :-
          locate(Pragma,(X/Y,Dir),(NX/NY,NDir)).
   ...
```

並列論理型ＯＳのヒストリ管理部分は、これから実行時のタスクの「性能規模」を割り出す。モジュールのヒストリ部に記録され、後にマルチタスクミックスを行うとき、タスク投入の戦略を決めるために参照される。

「性能規模」をどのように表したらよいかや、タスク投入の戦略の立て方は今後の課題として残すが、簡単な示唆を与えることはできる。ヒストリのレコードとして残すべきは、実行を再現するのに必要な情報すべてであるから、callへ与えた引数をすべて覚えていれば充分である。このうちProgはモジュールごとに決まってしまうから、とっておくべき情報は、

(Pcb,Goals,Path,Hist)

の四組である。「性能規模」を、このレコードから引き出す。PathとHistを比べて、二つの「相違」を見比べる。この二つが余りに違うようだと、このタスクは「行儀が悪い」タスクである。マルチタスクミックスの際は気を付けて戦略を決めねばならない。Histの形が丸とか三角とかに一定しているのは、「行儀が良い」タスクである。このようなタスクはマルチタスクミックスのときに、信頼して割りつけることができる。Pathによらず、Histの大きさが決まってしまうようなタスクも、同じように「行儀が良い」。

５．３　並列マシン用のメタコール／７

PPS で提案されたメタコールは次の形をしていた。

call(Program,Priority,Goals,Status,Control)

Program の上で、Priorityの優先度のもとで、Goals を、StatusとControl の制御化のもとに実行する。しかし、PPS は逐次マシン上のプログラミング環境を構築するものであるので、並列マシン上の並列論理型ＯＳを記述するには、いくつかの機能が抜けている。抜けている機能とは、

① 4.1で導入した、ゴール属性
② 4.2で導入した、道
③ 5.2で導入した、ヒストリ
であり、いずれも並列論理型ＯＳでのマルチタスク実現に必要な機能である。
Priorityは、ゴールの性質ゆえに、ゴール属性Pcb に吸収することができる。そして、道Pathと、ヒストリHistory を、PPS のメタコールに加えることによって、

call(Program,Pcb,Goals,Status,Control,Path,History)

の形の並列マシン用の７引数のメタコールができあがる。

６．まとめ

並列論理形計算機システムの構築に先立って、問題点を明確にし、それに幾分でも答えを与えることが本稿の目的であった。第Ｉ部では、並列論理型言語と並列論理型ＯＳについてまとめ、マルチタスクミックスの問題を提起した。マルチタスクミックスをじょうずに解くためには、システムのロードマップをリアルタイムに得る能力、実行しようとするタスクの影響力を知る技術、そして、実際に分散実行を制御する技術が重要であることを指摘した。第Ⅱ部では、第Ｉ部で提出した問題への我々のアプローチを示した。

まだまだ不十分な個所も多いが、そもそも詳細を決めるという事は、本稿の目的にはなかった。それよりも、個々の技術をつないで、全体としてのストーリを語って見せた点に、本稿の意義がある。

なお、本研究は、第５世代コンピュータ・プロジェクトの一環として実施しています。

参考文献

〔神田 86 〕 神田陽治: 並列論理形プログラミング言語におけるプラグマ機能について,
日本ソフトウェア科学会第３回大会(1986),D-1-2.

〔神田 87 〕 神田陽治: 並列論理形プログラムにおける新プラグマ方式とその応用,
情報処理学会第35回全国大会論文集(1987),4Q-5.

〔田中 87 〕 田中二郎, 太田祐紀子, 的野文夫、神田陽治:
GHC による仮想ハードウェアの構築とリフレクト機能について,
日本ソフトウェア科学会第４回大会(1987),B-5-3.

〔Chikayama 86〕 Chikayama, T.: Load Balancing in Very Large Multi-Processor Systems,
in Proc. of 4th Japanese-Swedish Workshop on FGCS,1986.

〔Clark 84〕 Clark, K. and Greogory, S.: Notes on Systems Programming in PARLOG,
in Proc. of Int. Conf. on FGCS, 1984, pp299-306.

〔Clark 86〕 Clark, K. and Greogory S.: PARLOG: Parallel Programming in Logic,
ACM Trans. Program. Lang. and Syst., Vol.8, No.1(1986),pp.1-49.

〔Clark 87〕 Clark, K. and Foster, I.: A Declarative Environment for Concurrent Logic
Programming, in Proc. of TAPSOFT'87(1987).

〔Eishiewy 83 〕 Eishiewy, N.: Extended PARLOG: Logic Programming of Real Time Systems,
in Proc. of 4th Japanese-Swedish Workshop on FGCS,1986.

〔Foster 87 〕 Foster, I.: The PARLOG Programming System (PPS), User Guide/Reference Manual
Version:0.4 ,Imperial College(1987).

〔Hirsch 86 〕 Hirsch, M., Silverman, W. and Shapiro, E.: Layers of Protection and Control
in the Logix System, CS86-19,Weizmann Instit.,1986.

〔Kahn 86 〕 Kahn, K. et al.: Vulcan: Logical Concurrent Objects,
Xerox Palo Alto Research Center,1986,pp.37.

〔Shapiro 83a 〕 Shapiro, E.: A Subset of Concurrent Prolog and Its Interpreter,
Tech. rep. TR-003,ICOT,1983.

〔Shapiro 83b 〕 Shapiro, E.: Systems Programming in Concurrent Prolog,
Tech. rep. TR-034,ICOT,1983.

〔Shapiro 83c 〕 Shapiro, E., Takeuchi, A.: Object Oriented Programming in Concurrent Prolog ,
New Generation Computing, Vol.1,No.1(1983),pp.25-48.

〔Shapiro 84〕 Shapiro, E.: Systolic Programming: A Paradigm of Parallel Processing,
in Proc. of Int. Conf. on FGCS, 1984, pp.458-470.

〔Silverman 86〕 Silverman, W., Hirsch, M., Houri, A., and Shapiro, E.:
The Logix User Manual, Version 1.21, Tech. rep. CS-21, Weizmann Inst.(1986).

〔Takeuchi 85 〕 Takeuchi, A. and Furukawa,K.: Bounded Buffer Communication in Concurrent Prolog,
New Generation Computing, Vol.3,No.3(1985),pp.145-155.

〔Takeuchi 86 〕 Takeuchi, A. and Furukawa,K.: Parallel Logic Programming Languages,
Tech. rep. TR-163,ICOT,1986.

〔Ueda 85 〕 Ueda, K, and Chikayama, T.: Concurrent Prolog Compiler on top of Prolog,
In Proc. of Symp. on Logic Prog.,1985,pp.119-126.

〔Wulf 74 〕 Wulf, W et al.: HYDRA: The Kernel of a Multi-processor Operating System,
Comm. ACM, Vol.17,No.6(1974),pp-337-345.