ICOT Technical Memorandum: TM-0573

TM-0573

並列制御実行のためのユーザ表現記述

神田陽治(富士通)

July, 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

並列制御実行のためのユーザ表現記述

Descriptive Specification for Controlled Parallel Execution

神田　陽治
Youji KOHDA

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

あらまし　汎用並列マシン上のマルチタスク下での動的負荷分散の問題について考察を行う。我々の方式の要点は、仕事の内容を記述するモジュール仕様に性能仕様の項目を設け、仕事ごとの特殊性をオペレーティングシステムの分散戦略決定に反映すると同時に、実際の実行結果から性能仕様を改訂していく、フィードバックの仕組みにある。部分評価の技術を、性能仕様をプログラムに系統的に組み込むために用いる。リフレクションの技術を性能仕様の動的改訂に用いる。

Abstract　We consider how to achieve dynamic load-balancing under multi-tasking on general purpose parallel computers. A method is developed; first adding a performance specification section to ordinary module specification, second composing a strategic load-balancing mechanism in operating systems. The new strategic mechanism refers to the performance specification of modules to make up load-balancing strategy. The performance specification is revised again and again on the success or the failure of the strategy. Partial evaluation technique is used to compile performance specification systematically into programs. Reflection technique is used to revise the performance specification.

はじめに

第Ⅰ部は一般論である。汎用並列マシン、マルチタスクミックス、ユーザ表現記述、動的負荷分散のためのシナリオ、の順に話を進める。第Ⅱ部では、基礎となる二つの技術　―部分評価の技術とリフレクション―　について解説を行う。第Ⅲ部は、第Ⅰ部の議論を並列論理型の場合で具体化する。

第Ⅰ部　一般論

1．汎用並列マシン

我々の研究の目的は、（汎用）並列マシンのプロセッサ資源を活かす方法を新たに開発することにある。あるプロセッサはひどく忙しいのに別のプロセッサは暇である、というような利用率の偏りは避けるに越したことはない。投入されるひとかたまりの仕事をタスクと呼ぶと、タスクはさらに複数の仕事に分割されてプロセッサに割り付けられて実行される。プロセッサに割り付けられて実行される仕事の単位をプロセスと呼ぶ。タスクをどのようにプロセスに分割し、プロセスをいかにプロセッサに割り付けて実行させるかが、並列マシンでタスクを効率よく実行できるかの要であり、これが並列マシン上の負荷分散の問題である。

タスクの性質によっては、負荷分散の問題をじょうずに解ける場合がある。タスクの構造が時間不変的かつデータフロー的であるがために、比較的容易にタスクをプロセスに分け、プロセスをプロセッサに規則的に割り付けられる場合である。画像処理などがこの例にあたる。これを見て、「並列マシンを活かすには、専用の並列アルゴリズムの開発から始めなければならない」という主張も生まれるわけである。しかし現実には並列アルゴリズムの開発は難しい。安くなったマイクロプロセッサを多数つなげてとにかく「並列マシン」と称するものを作ってしまい、あとは並列アルゴリズムの責任にしてしまうのでは、何も解いていないのと同じである。単に目先の問題（並列マシン上の負荷分散の問題）を別の難しい問題（並列アルゴリズムの開発）にすり替えたに過ぎない。

与えられたどんな問題をも汎用の並列マシンを使って首尾よく解くこと、これが我々の考える課題である。（もちろん問題の特殊性を利用した専用マシンを否定するわけではない。ここでの問題意識は、日常使用に耐える実用＂ホーム＂並列マシンはあり得るのかということに尽きる。）

2．マルチタスクミックス

逐次マシンの限界を指す言葉に、フォンノイマンボトルネックという言葉がある。これはプロセッサとメモリを結ぶ経路が、逐次マシンの性能を制約しているのだ、という考察を表す。しかし並列マシンにも、プロセッサとプロセッサの間に通信という名の経路が存在する。それゆえ、分割されたタスクのプロセスをめいっぱい、並列マシン上のプロセッサにバラマクと極端に性能が落ちる。一つのプロセッサで解いた方が実は格段に速いという現象も生じる。並列マシンだから速くなるというのは宣伝文句でしかない。速くなる場合があるというのが正しい。難しいのは速くなるのはどんな場合かを発見することである。問題ごとに適切な分割というものがあるはずであり、問題によっては、多数のプロセッサが遊んでいようとも、一つのプロセッサだけで解くという決心さえしなければならない。

将来は100 万個規模のプロセッサを持つ並列マルチプロセッサマシンも登場するだろうが、バッチマシン（一度に一つのタスクを流して実行する）としての使い方に固執している限り、既に説明した理由から大多数のプロセッサは遊んでいるということになりかねない。

このことより汎用並列マシンの性能を発揮させるためには、マルチタスク（複数のタスクを同時に実行すること）の機能が不可欠であると知れる。並列マシンを卓上で（一歩下がって机の横で）パーソナルに使おうとするときこそ、マルチタスクの能力は一層不可欠である。じょうずにタスクを混ぜ、並列マシン全体の性能の向上を目指す技術を、マルチタスクミックスと我々は呼ぶ。マルチタスクミックスには二つの局面が含まれている。第一に、タスクをじょうずにプロセスに分割する段階である。第二に、プロセスをプロセッサに割り当てじょうずに並列実行する段階である〔神田86〕。

さて問題は、二つの仕事を誰がいつ行うかである。両極端には、両方ともユーザが実行前までに決めておく方式と、両方と

も並列マシンが実行時に動的に決める方式がある。先に述べた並列処理がうまく行く場合は往々にして前者であるし、コンピュータアーキテクチャの研究者が描く理想は後者である。

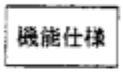
3．ユーザ表現記述

現在までに並列マシンは数々作られて来たし、一部は商用化されている。しかし並列マシンが実際に役立つかどうかという点は、「たまたま相性の良い例題にうまく当たれば」うまくいくという程度に見える。台数効果と称して、プロセッサの数を増やしたとき応じて速くなるのも、「都合のよい例題を固定したときに」という前提があってのことである。並列マシンを作る前のハードウェアシミュレーションにしても、小さい特定の例題、お馴染8クイーンばかりを使って評価していたら、出来上がる並列マシンは8クイーン専用マシンである。はなはだ心もとないと言わねばなるまい。一体どこがまずいのか。

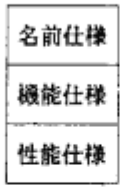
仕事ごとの特質を見ずに、並列マシンが自分の都合だけで負荷分散を図ろうとする所に、限界の原因があるように思える。例えば、負荷分散の戦略として、「負荷の重いプロセッサは、隣合うプロセッサから負荷が一番軽いプロセッサを選んで、仕事の一部を相手に任す」といった戦略である。

限界を打ち破る方策の一つは、仕事ごとの特質を考慮に入れて負荷分散を行うことである。例えば、「負荷の重いプロセッサは、自分の仕事から将来負荷となりそうな仕事を選んで、他の仕事との連係上最適と思われる他のプロセッサに任す」といった戦略である。ここで「将来負荷となりそうな」とか、「他の仕事との連係上最適と思われる」といった仕事の特質を知ることは、そうと明示されていなければ判るものではない。この種の決定に役立つ仕事の特質に関する情報のことを、ここでは仕事の「性能」と呼ぶ。それでは性能はいかに記述されるのだろうか。

並列マシンを有効に働かせるための仕様言語が、並列言語である。ソフトウェア工学の教える所ではソフトウェアは、モジュールに切り分けて記述されねばならない。モジュールとはある程度まとまった仕事の単位である。従来モジュール仕様とは、主に仕事の機能を記したものであった（図(a)）。しかし、それだけに止める理由はない。我々はモジュールの概念を拡張し、性能仕様を加える。さらに並列マシンの負荷分散の問題とは直接は関係ないが、モジュールの名前を拡張定義する名前仕様を加える。この拡張されたモジュール仕様を我々は「ユーザ表現記述（Descriptive Specification for Users）」と呼ぶ（図(b)）。



図(a)　従来の記述



図(b)　ユーザ表現記述

A．名前仕様

従来、モジュールの名前は単なる文字列であり、一致／不一致のみが重大であった。名前それ自身が自然言語の単語として持つ意味が、モジュールを読む人間にとって重大であるのにかかわらずである。モジュールの名前仕様は、モジュールの名前に自然言語の単語としての意味を与え、自然言語が持つ潜在パワーを利用しようとする試みである〔神田84〕。

B．機能仕様

モジュールがどんな仕事を果たせるか、いかに成し遂げるかを記述する。抽象データ型などは、この部分に含まれる。

C．性能仕様

モジュールの性能を記述する。例えば、オペレーティングシステムがマルチタスクミックスを行う際に、どのようにタスクを混合したらよいかを決める根拠に使われる。従来で言えば、タスクのジョブクラスの指定であるとか優先度の指定に相当する。名前仕様や機能仕様と違い、次章に説明するように、この部分は実行マシンによって実行時にもどんどん改訂されて行くから、実行マシンごとに異なった記述となる。

4．動的負荷分散のためのシナリオ

しかしすぐさま思いつく疑問は、果たしてモジュールの性能を正確に決めることは可能だろうか? という問である。「この仕事はあの仕事より大変である」と比較できないことはないが、「この仕事の大変さは正確に2242である」とはちょっと言えそうもない。性能というのは本来相対的なものであり、仕事単独では性能は決められないのである。プロセスに優先度を割り付けようとするときに、それが良くわかる。このプロセスはあのプロセスより優先して実行すべき仕事であるとは言える。しかし、プロセスの優先度は絶対に2543にしなければならぬとは、無意味な言明である。

とはいえ、性能を割り出すことは難しいから、プログラマに「性能に関する示唆」をして欲しい。「この仕事は多量の出力を必ず伴う」といった示唆である。あるいは「この仕事とあの仕事は互いに強く関係している」といった示唆である。性能示唆のための言語要素をプラグマという。蛇足ながら、プラグマという術語は、同様な意味あいで、Ada 言語にも登場する。

さて（この相対）性能は、仕事の実行を実際に担当するマシンにも依って変わるし、将来投入されるであろう新規の仕事にも影響されて変わる。モジュールの性能仕様は、計算機システムごとに別個に記述されねばならないし、新しいモジュールの開発などに伴ってどんどん改訂していかねばならない性質のものである。とすれば、性能の管理はオペレーティングシステムが負うべきものである。モジュール仕様のうち、名前仕様や機能仕様の管理はプログラマが行うが、性能仕様の管理はオペレーティングシステムがハードウェアの助けを借りて行うのである。それゆえに性能仕様は実行マシンごとに違う。

オペレーティングシステムは、性能仕様をもとにマルチタスクミックスの戦略を立てて計算機システムを能率よく動かす。と同時に、各々の仕事の他の仕事に対する相対性能を計測し、各仕事の性能仕様を改訂し、次回の戦略決定がより精密になるように備える。フィードバックがここには在り、これが性能仕様管理の基本的なシナリオである。始めのうちは性能仕様の精度が粗く戦略決定も失敗するかもしれないが、だんだん経験を積むに従って、オペレーティングシステムは計算機システムをうまく運営できるようになるだろう。途中で追加される仕事により発生する、仕事間の相対性能の調節の仕事もオペレーティングシステムにより行われる。

事情は仮想メモリを実現するためのページングアルゴリズムに似ている。OPT 戦略は最適であるが、それが未来の絶対の予測の上に立っているために、決して実現することはできない。しかし LRU戦略などを使うと、過去の履歴を参照して準最適な戦略を取ることができるのであった。同じように性能仕様の場合にも、過去の行動の蓄積として集め推測し、それを同時に未来の予測のために用いることができる。

第Ⅱ部　基礎技術

5．部分評価〔例えば Hirsch86 参照〕

部分評価はプログラム変換の技術の一つであり、変換を行う目的は最適化である。最適化であるから出力プログラムは元のプログラムと同じ仕事をする。しかし期待されているのは「より良く」仕事をしてくれることである。いいかえれば部分評価はプログラムの「性能」を改善する技術なのであり、我々が基礎技術として着目する理由でもある。良いということの判断記述は、実行時間の短縮であるとか必要資源量の削減であったりするのだが、その改善の度合は部分評価プログラムが用いる最

適化アルゴリズムの強さに依っている。

部分評価を形式的に示す。

PE^L (P_L (Known,Unknown), C) = P_L'(Unknown)

ここで P_L'(Unknown) ≡ P_L (C, Unknown)

PE^L は言語L用の部分評価プログラムである。P_L (Known, Unknown)は言語Lで書かれたプログラムであり、P_L の引数は、具体値が分かっている部分Known と分からない部分Unknown に分けられているものとする。Cは引数Known に与える具体値である。上式の意味を述べると、部分評価プログラムPE^L は、言語Lで書かれたプログラムP_L と引数の具体値Cをとって、言語Lで書かれたプログラムP_L' にプログラム変換する。そしてP_L' は元のプログラムP_L と同じ仕事をする。

興味深いのは、P_L が、ある言語のインタプリタ、Cがその言語で書かれたプログラムである場合である。言語Lで書かれた言語MのインタプリタをI_L^M、言語Mで書かれたプログラムをP_M としたとき、I_L^M とP_M を部分評価プログラムにかけると、

PE^L (I_L^M (Prog,Input), P_M) = P_L (Input)

ここで P_L (Input) = I_L^M (P_M ,Input)

ところでプログラムP_M をインタプリタI_L^M 上で間接に実行したときと、直接実行したときの効果が同じという事実は、

I_L^M (P_M ,Input) ≡ P_M (Input)

と書けるから、

P_L (Input) ≡ P_M (Input)

が得られる。これは、言語Mで書かれたプログラムから、部分評価を使って言語Lで書かれたプログラムで意味的に等価なものが得られたことを意味し、言語Mから言語Lへのコンパイルに等しい。

言語Mが言語Lに同じ場合にも意味がある。この時のI_L^M をメタインタプリタという。部分評価はメタインタプリタの介在によるオーバヘッドを解消する働きをする。さらにメタインタプリタが機能拡張されていれば、拡張機能をプログラムに組み入れる効果もある。（この事実を8章で利用する。）具体例として言語Mとしに、並列論理型言語Concurrent Prolog (Shapiro 83) を採ったとしよう。callをConcurrent Prolog の基本的なメタインタプリタとする。P_M の例として次のappendを使う。

```
append( [ X | Xs] ,Ys, [ X | Zs] ) :-
          append( Xs?, Ys?, Zs).
append( [ ] ,Ys, Ys ).
```

このとき、PE^L (call(Prog,append(X,Y,Z)), P_M) を計算して、次のプログラムP_L が得られる。

```
call(P_M ,  append( [ X | Xs] ,Ys, [ X | Zs] )) :-
          call( P_M , append( Xs?, Ys?, Zs)).
call(P_M ,  append( [ ] ,Ys, Ys )).
```

さてappend2(A,B,C)= call(P_M ,append(A,B,C)) と置くと

```
append2( [ X | Xs] ,Ys, [ X | Zs] ) :-
          append2( Xs?, Ys?, Zs).
append2( [ ] ,Ys, Ys ).
```

となり、メタインタプリタのオーバヘッドが除去された。

6. リフレクション〔例えば、田中87参照〕

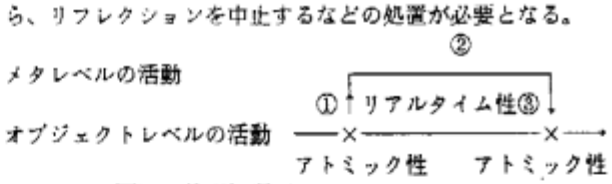
体力的に最大100kg までしか荷物を持ち上げられないクレーンがあるとする。「そこの120kg の荷物を持ち上げよ」と指示を受けた、ごく普通のクレーンの場合には、指示を忠実に守ろうとした結果、重みに耐えかねて壊れてしまうに違いない。ちょっと賢いクレーンなら、「自分の最大限度は100kg 、だから 120kgは無理だ」と判断して、指示を無視するだろう。しかし120kg の荷物を90kgだと偽って持たされたときは壊れてしまう。どうすればよいかと言えば、真に賢いクレーンは常に自分の過負荷の状態を監視していて、無理だと判断したら指示の遂行を中断してしまうようなものでなくてはならない。クレーンがかりに老朽化していれば、90kgの荷物でも実行を中断しなければならないのである。しかも実行中断の断行はクレーンが壊れる前にすみやかに（リアルタイムに）行わなければ、せっかくの判断が無駄になってしまう。

ここでの本質は、①現在の自分の状況を客観的に知り、②（その理解を基に）検討し、③（その結果を基に）自分の行動の戦略を変更する能力である。この一連の操作をリフレクションという。

これを専門用語に翻訳すると、メタとオブジェクトの二つのレベルがあって、メタなレベルはオブジェクトのレベルを言及するに十分な記述力を持っていなくてはならず、二つのレベルが因果的関係 (causality)で結ばれていること、となる。①の自分の状況が属しているのがオブジェクトのレベルであり、②で客観的な立場で自分の状況を調べ、検討するのがメタなレベルである。リフレクションについての考察を進めよう。以下の諸点は従来、指摘されたことはなかった。

並列処理におけるリフレクションにおいては、因果的関係を保証するために、①と③の操作においてアトミック性が、②の操作にはリアルタイム性が保証されねばならない。なぜなら、メタなレベルとオブジェクトのレベルの計算は並列に行われてもよく、メタなレベルの活動中にもオブジェクトのレベルの活動は進んで行くからである（図(c)）。

①でのオブジェクトレベルの状況の読み取り、あるいは③での書換えは、一貫性あること (consistensy)が要求される。これがアトミック性である。これゆえにメタとオブジェクトの二つのレベルが区別されねばならなかったのである。もし一貫性が必要なければ、二つを区別するには及ばなかったであろう。さらに①で得たオブジェクトレベルの状況に基づく②の検討 (reasoning)の根拠が、③のアクションを起こそうとしたときには既に失われてしまっている可能性があるので、リアルタイム性とは、この根拠が失われないうちに③のアクションは完了しなくてはならないという要請を指す。もし根拠が失われたなら、リフレクションを中止するなどの処置が必要となる。



図(c) 並列処理下でのリフレクション

リフレクションの具体例は、現代の粋を凝らしたマイクロプロセッサに見ることができる。保護機能を備えたＶＬＳＩである。システムレジスタはマイクロプロセッサの動作を決定する重要なレジスタ群であり、特権命令でしかアクセスすることは許されない。割り込みやシステムコールによって特権命令が使える保護レベルに移り（①）システムレジスタやプロセスコントロールブロックを参照し（②）、必要に応じて変更を加えたのち非特権レベルに帰る（③）、これら一連の操作はまさしくリフレクションである。この間、非特権レベルは中断しているので、必要なアトミック性やリアルタイム性は自明である。

このハードウェアによるリフレクションと、モジュールの性能仕様を組み合わせよというのが、我々の提案である。マイクロプロセッサの性能を決めるパラメータを固定してしまわずに、性能レジスタとして操作できるように公開する。個々のモジュールの性能仕様は、そのモジュールの機能を充分に発揮させ得るような性能レジスタ値を記憶している。モジュール間コールのときには、性能レジスタ値を旧モジュールの性能レジスタ値から新モジュールのそれへと交換する。モジュールの実行が行われている間に、暗黙に（割り込みの場合）、あるいは陽に（システムコールの場合）リフレクションが引き起こされる。リフレクションの検討の段階を経て、性能レジスタの値が最適に書き直される。改訂された性能レジスタの値は、モジュールが切り替わるときに書き戻され性能仕様に反映される。

第Ⅲ部　並列制御実行のためのユーザ表現記述

7．並列マシンとプラグマ〔神田87b〕

並列論理型言語Concurrent Prolog〔ShapiroR3〕を用い、第Ⅰ部のシナリオの具体化を試みる。その前に、並列マシンに仮定している能力とプラグマについて、若干のことを述べておきたい。

並列マシンは、プロセッサ間距離に応じて、プロセッサ間通信のコストが段々と増すものを想定している。例えばメッシュ状のネットワークで、メッシュの各交点に一つずつプロセッサがぶら下がったようなタイプである。しかしプログラムの側から見ると、どのプロセッサにどのプロセスが割りつけられているかを意識せずともよいとする。すなわち必要なメッセージの転送は、すべてハードウェアにより自動的に行われる。

以上の条件は次のことを意味する。いかにバラバラにプロセッサにプロセスを分散配置しようとも、ともかく実行できる。しかし配置のうまいへたは実行性能を大いに左右する。この観点からは、マルチタスクミックスとは、各タスクの性能を案配してプロセスの配置を戦略的に決め、並列マシン全体の性能を確保しようという技術に他ならない。

ここでのプラグマは、タスクを構成するプロセスをプロセッサにどのように割り付けるかを指示する仕組みである。Concurrent Prolog に対しては簡単なものが、Shapiro によって提案されている〔Shapiro 84〕。節の形は、

H :- G1,G2,...,Gm | B1@P1,B2@P2,...,Bn@Pn.

であり、ここに Hはヘッドで、Giはガード、Bjはボディのゴールである。従来に比べると、ボディのゴールにはプラグマPjが、@ を挟む形で付加されている。

ゴールは、Concurrent Prolog でのタスクのプロセッサへの最小割付け単位、すなわちプロセスに当たる。Shapiro のプラグマは「前へ行け」とか「右向け右」といった直接指示であった。プログラマはデータ構造を設計すると同じ感覚で、プロセスの配置を予め設計し、プラグマを使ってプログラムに直接書き入れる方式である。「負荷分散はプログラミングの問題に還元できる。」これがShapiro 方式の本質である。しかし、直接指示であるために、マルチタスク環境に対処できない。

我々の考えによれば、Shapiro のプラグマは記述のレベルが低すぎる。もっと高レベルで制約的なものでなくてはならない。例えば、「あるゴールが投げられたと同じプロセッサに、このゴールも投げて欲しい」とか「ゴール内の変数の値が確定した後で、このゴールを投げて欲しい」といった、必要以上にグローバルな判断を要求しない、そういう記述レベルである。

我々は、Shapiro のプラグマ方式が直接指定であったのを、より柔軟な形へと拡張する。いわばプラグマに構造を持たせる。第一に、ヘッドにもプラグマを付して、

H@P0 :- G1,G2,...,Gm | B1@P1,B2@P2,...,Bn@Pn.

を節の形とする。こうすることによって通常のヘッドユニフィケーションを通して、プラグマの値を伝達できる。

第二に、プラグマにはPCB（プロセスコントロールブロック）情報を含ませる。PCB 情報には、ゴールの載っているプロセッサのアドレスであるとか、優先度といった情報が含まれている。プラグマ間の共有変数を利用することによって、「あのゴールと同じプロセッサに、このゴールを割り付ける」といったことを記述できる。

最後に注意を一つ述べておく。プラグマを使って並列マシン上でのマルチタスクミックスが可能であるという前提には、「負荷の連続性」の仮定が暗黙に成立していなければならない。ある時刻を固定したとき、プロセッサ平面上の負荷分布はなだらかに変化し、一つのプロセッサを固定したとき、その負荷の経時変化もなめからでなくてはならない。そうでなければ、プラグマでゴールを近傍のプロセッサに放って負荷の分散をしようという発想自体に意味がなくなる。この仮定が現実に成立するものであるかどうか、ぜひ調査の必要がある。

8．部分評価による性能のコンパイル

プラグマ付けをすべてプログラマに押しつけるShapiro のアプローチはあまりに手抜きである。ユーザは真に特定したい分散戦略のみ制約的に指定すれば良いとし、残りは適当に誰かが付けてくれた方が都合よく楽である。我々の方法は、メタインタプリタの部分評価を利用する。この方法の利点は、プラグマを半ば自動的に系統的にプログラムに付け得る点である。

もっと一般的には、基本的なメタインタプリタに種々の分散戦略を組み込んだものを用いる。アルゴリズムとして組める限り、メタインタプリタの部分評価によりプログラムに分散戦略を組み込む（コンパイルする）ことができる。

どのような分散戦略を盛り込むかは、メタインタプリタをどうプログラミングするかに尽きるので、分散戦略を数え上げることには意味がない。代わりに、ここでは役に立ちそうな分散戦略をいくつか組み込む具体的なメタインタプリタを示すことにする。まず、基本的なメタインタプリタを示す。

```
call(Prog,true).                                          (1)
call(Prog,(A,B)) :- call(Prog,A?), call(Prog,B?).        (2)
call(Prog,A) :- system(A) | exec(A).                      (3)
call(Prog,A) :- clauses(Prog,A,Cs) |                      (4)
          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.
```

callがConcurrent Prolog メタインタプリタである。節(1)は解くべきゴールがtrueなら仕事完了であり、節(2)は解くべきゴールが複数の仕事に分けられるなら、それぞれを個別に解けばよいということを、節(3)はシステムコールはシステムに任せて実行してもらえばよいことを言っている。節(4)はその他の場合であり、解くべきゴールを副ゴール群に分解し、それを解けばよい。ゴールの分解は、プログラムデータベースProgをclausesによって探すことで行う。resolve はゴールの副ゴール群への分割が複数通りあるなかから適当なもの（ガード部ゴールを全て成功裏に完了したもの）一つを選び出す。unify はユニフィケーションを行うが、Concurrent Prolog の「読みだし専用記法 ?」を扱えるように拡張されている。

さて、これを次のように拡張してみた。

```
call(Prog,true).
call(Prog,A@P) :- call(Prog,A?)@P?.                       (p)
call(Prog,(A,B)) :- call(Prog,A?), call(Prog,B?).
call(Prog,A) :- system(A) | exec(A).
call(Prog,A) :- clauses(Prog,A,Cs),                       (q)
          get-info(Prog,A,Info) |
          clause-sort(Info,Cs?,Cs2),
          resolve(Cs2?,Body),
          goal-sort(Info,Body?,Body1),
          attach-pragma(Info,A,Body1?,Body2),
          call(Prog,Body2?).
```

部分計算は、計算できるところはできる限り計算を進める手法であった。プラグマ付のゴールを扱うために加えられた節(p)においても、プラグマP の実行を遅延して、call(Prog,A?) の部分評価を続けることができる。プラグマ間に分配則や簡約則など演算則が整備されていれば、遅延したプラグマを分配則を使って展開し簡約則で除去するなどして、部分評価結果をさらに簡単化しうる。（これができるのは、プロセッサ間通信を透明にするハードウェアが暗黙に下にあるからである。）また、各々のプロセッサのローカルメモリへのプログラムローディン

グの問題に寄与することもできる。節(p)で、プラグマを遅延させる時に、プログラムデータベースをローカルに管理するゴールを（まだ作られていなければ）相手先プロセッサ上に作るようにすればよい。

A．統計情報を利用しての、ゴールや節の順の並び換え

メタインタプリタに組み込まれている分散戦略の最初のものは、節内のゴール並び換えと、同じ名前のヘッドを持つ節同志の並び換えである。並び換えは各種の統計情報に基づく。統計情報はモジュールの性能仕様にしまわれているものとする。

並列論理型言語では、節内のゴールの順とか、節の順番には意味がないはずであるが、並列マシンの上で実行すると言えども、どこかに逐次性が残っていて必ず順番というものが性能に影響してくる。そこでゴールあるいは節を並び換えて、並列マシンにもあるに違いない順番への敏感さを利用しようというのである。並び換えを行うのは、ゴールについては節(q)内のgoal-sort 、後者はclause-sort である。並び換えの根拠となる統計情報はオペレーティングシステムが常に現実を反映するように管理する。統計情報とは例えば、発火トライ回数、成功回数、実行時負荷などである。ここではかりに節(q)内のget-infoが、プログラムデータベース内のモジュールの性能仕様から、並び換えなどに必要な情報を取り出してくるプリミティブとしている。

並び換える戦略によって、組み入れる分散戦略が決まる。例えば、節内のゴールの並び換えでは、「読みだし専用記法」によって明示されている変数の入出力関係を見て、値の生産者に当たるゴールを値の消費者に当たるゴールより前に置き、等位にあるもの同志では実行時負荷の重い順に並べる、などである。また、同じ名前のヘッドを持つ節間の並び換えでは、過去に選択された確率の高い順に並べたり、同程度の場合にはラウンドロビンで順繰りに回して並べたりする。

B．部分評価によるプラグマの系統付け

二番目の分散戦略は、プラグマの系統付けである。節(q)内のattach-pragma が並び換えの終わったゴール列に、プラグマを付与する。このとき、プログラムデータベースから引き出された情報InfoとヘッドゴールAの情報を利用してよい。

この場合も、どのようにプラグマを付けるかで組み入れる分散戦略が決まる。例えば、Infoの統計情報に基づき、負荷の重い順にゴールいくつかを選び、「他の適当なプロセッサへ投げよ」などのプラグマを付与する。

部分評価をする前にプログラマによって、既にプラグマが付けられているかも知れない。それをそのまま残したり、必要なら、より低位のプラグマにコンパイルしてもよい。

9．負荷分散方式について

並列マシン上で、分散戦略を組み込んだプログラムを実行することを考察する。負荷分散の方式についての総括的議論から始める。負荷分散の方式は動的と静的に分けられる。動的負荷分散は実行時に行う。静的負荷分散はコンパイル時に行う。

動的負荷分散は、データフローマシンなどに見ることができる。プラグマで分散を指示されたゴールに出会うたびに、その時点でもっとも暇な隣接プロセッサへゴールを投げる。あるいは暇なプロセッサが自発的に負荷を請け負う方法もある。こういった方式は、確かにその時点では最善の策と言えるが、全体を見渡したとき最善であるとは限らない。なんとなれば、ゴールが拡がり過ぎれば、ユニフィケーションのための変数アクセスの総コストが増えてしまいかねないからである。一方、この方法はマルチタスクミックスにはうまく対処できる。

一方、Shapiro に代表される方式は、静的負荷分散方式である。コンパイル時にプラグマを完全に固定してしまう。動的分散の余地を残さない。シングルタスク下では、変数セルの配置を予め注意深く設計できるので、ユニフィケーションのための変数アクセスコストをうまく押さえ得るが、一方、マルチタスクミックスの問題にはまったく対処できない。わざわざ忙しいプロセッサにゴールを投げてしまい、実行時間が延びるおそれが大である。

A．短期、中期、長期の三つのスケジューリング方式

正しい解答は、二つの両極端の中間にあるに違いない。動的負荷分散方式は、その場限りの判断で負荷分散を行う、いわば「短期スケジューリング方式」である。静的負荷分散方式は、現状を省みず、予めの判断だけで負荷分散を行う、いわば「長期スケジューリング方式」である。だとすれば、ある程度の展望を持ちつつ、その場の状況も加味して負荷分散を行う、「中期スケジューリング方式」に相当するものが必要である。

まず、短期、長期のスケジューリング方式は並存できる。動的負荷分散（短期スケジューリング）で利用するのは、静的負荷分散（長期スケジューリング）から漏れ出た自由度である。例えば、特定のプロセッサを名指しで指定するプラグマの場合、投げるプロセッサについては自由度は残っていない。しかし、どこでもいいから放れというプラグマなら、放るプロセッサをどれにするかの自由度が残っている。オペレーティングシステムはこの残る自由度を活用して、マルチタスクミックスを達成する。

さてそれでは、残った自由度をどのように利用するか。我々の提案は、短いスケジューリングパスを与えるという方式である。スケジューリングパスは実行時にときどき見直され、マルチタスクミックスの状況に合うように修正される。このスケジューリングパスを利用する方法が、中期スケジューリング方式である。スケジューリングパスは、プラグマにスケジューリングの自由度があるときに、その自由度を決める助言となる。別の言いかたをすれば、プラグマの解釈を決める。例えば、放るプロセッサに自由度があるプラグマに対して、放るプロセッサを特定する情報をスケジューリングパスで補う。このスケジューリングパスを〔神田87a 〕では、「黄金の道 (Golden Path)」と呼んだ。

10．リフレクションによる中期スケジューリング方式の実装

6章のリフレクションの話の最後で、マイクロプロセッサに性能レジスタを設ける話をした。性能レジスタの値をスケジューリングパスとすれば、中期スケジューリングはリフレクションによって実装できる。すなわち、リフレクションの検討の過程を経て、スケジューリングパスの値は現在の状況を反映するよう随時、変更されて行く。

A．並列論理型言語でのリフレクション

リフレクションというアイデアが、並列論理型言語の枠組みのなかでどのように実現されるべきかについて、ここで考察する。一般的な注意として、リフレクションとは固定された仕組みを指すのではなく、言語が持つ特徴を最大限に利用して実現されるべきものである。並列論理型言語を特徴付けるのはストリームで交信するゴール群のパラダイムである。よってリフレクションは、二つの行き交うストリームによって連結された、二つのゴールによって実現されるのが無理がない。要点はストリームを通して交わすべきメッセージ通信のプロトコルである。リフレクションを実現するに十分な情報がオブジェクトレベルから一貫性を持って取り出せ、また時間内に戻せなくてはならない。

繰り返せば、リフレクションを並列論理型言語ではっきり定義するとは、プロトコルの仕様をはっきり定めることである。少なくとも次のような機能が要求される。第一は、オブジェクトレベルの計算を止めたり（suspend)、再開したり（resume)、中止したり（abort)するものである。すでに提案されている

call/3〔Clark 86〕と同じでよい。その他、ブレイクポイントを指定するトラップ命令（trap）とか、トレイス実行用のステップ実行命令（step）などもあればよい。第二は、オブジェクトレベルの計算状態を読むもの、書き戻すものである。

厳密にアトミック性とリアルタイム性を満たすためには、まずオブジェクトレベルの計算を止めた後、計算状態を読み、検討し、計算状態を書き戻し、計算を再開すればよい。しかし、アトミック性あるいはリアルタイム性の基準が甘くて良い場合には、オブジェクトレベルの計算を止めておく時間をもっと小さくすることができる。アトミック性とリアルタイム性への要求は、性能を扱うリフレクションの場合、厳しいものではない。性能データの一貫性が崩れても性能予測の多寡に響くことはないし、性能データ更新が遅れても影響はたいして無いからである。

B．保護と効率の問題

一般的に言えば、なるべく多くの計算状態を読み書きできた方が、強力なリフレクションを実現できる。しかし保護の問題とも絡んでくる。我々が提案するのは、メタレベルがオブジェクトレベルの状態を読むときに、始めに「読み書きしたい計算状態の仕様」を送ると、オブジェクトレベルが「相手の権利を調べて可否を決定する」方式である。適切な言語仕様（読み書きしたいデータとかアクセス手続きを定義するインターフェス仕様）を決めれば、コンパイル時に検査することも可能であろう。この方式は保護の問題と同時に、リフレクションの実現効率の問題をも解決する。リフレクションの要求の度ごとに、すべての計算状態を読み書きできるように操作するとなると、ひどく性能が熟化するのは避けられそうにないが、必要なだけの情報を読み書きするなら、それが避けられうる。

11．並列制御実行の枠組み

さていよいよ、マルチタスクミックスを達成する、並列制御実行の枠組みを説明する。たくさんのタスクが既に実行されている並列マシンに、新たなタスクが投入されたとする。

最初の決定は、どのプロセッサにタスクの初期ゴールを投入するかである。局所的な判断ではだめで、マシン全体の負荷分布を知って行わねばならない。そのための技術については既に別のところで説明した〔神田87b〕。

さて投入プロセッサが無事決まったなら、タスクを起動する。タスクを実際に実行するexecute、プログラムデータベースを管理するdb、中期スケジューリングを担当するmeta、を同時に走らせる。（本来は、オペレーティングシステム上位に繋がるストリームなども必要だが、一切省いている。）

```
start(Prog,Goal) :-
        execute(Goal,DI?,DO,MI?,MO),
        db(Prog,DI,DO?),
        meta(MI,MO?).
```

execute は初期ゴールGoalをとり、スケジューリングパス（9章）やマシン全体の負荷分布に基づき、プラグマ付ゴール（7章）を分散する。すなわちexecute は短期スケジューラである。dbがとる引数は、部分評価により性能が組み込まれたプログラム（8章）である。部分計算を行う過程で長期スケジューリングが実施される。metaはリフレクション通信プロトコルを用い、中期スケジュールを実施する（10章）。

三つは互いに並列制御実行の仕事を分担している。execute は基本的なインタプリタの拡張であり、clauses やunify といったプロセッサ間通信を引き起こすような重たい操作を含む（8章）。execute とdbは二本のストリームによって連結されている。execute はclauses を通してdbにプログラム探索を頼む。またexecute は、unify やclauses が報告する実行時の負荷を累積して、ゴールの実行時負荷としてdbに報告し、dbは統計情報としてモジュールの性能仕様に反映する（8章）。

さらにexecute とmetaも二本のストリームにより連結されている。ここにリフレクションが使われている（10章）。metaがメタレベルを、execute がオブジェクトレベルを受け持っている。プロセッサのハードウェアには性能レジスタが組み込みになっている（6章）。性能レジスタの中身はスケジューリングパスである（10章）。execute は、その値を用いてプラグマ付きゴールを短期スケジューリングする。metaはリフレクションの通信プロトコルを使って、スケジューリングパスを適宜改訂する。すなわち中期スケジューリングする。metaはまた、マシン全体のプロセッサの負荷分布の情報も管理し、execute に伝える。

まとめ

長かった話の根幹は「性能を扱う」ことにある。これは従来には無かった新しい視点である。要点を述べる。

第一に、性能の記述をモジュール仕様に性能仕様として追加し、部分評価によって性能仕様をプログラムに系統的に組み込む。第二に、動的負荷分散と静的負荷分散の欠点を補い合う方法として、中期スケジューリングが必要と考え、スケジューリングパスという考えを新たに導入した。第三に、スケジューリングパスを利用した中期スケジューリング方式の実現に、リフレクション機構を用いる。

最後に、一つの質問に答えておこう。マイクロプロセッサに性能レジスタを設け、スケジューリングパスの値をしまい、それを扱う命令をハードウェアで実現したとしても、オペレーティングシステムで性能までも扱おうとすることは、オーバヘッドが大き過ぎないかという問である。これには次のように答える。現存の大型機のオペレーティングシステムは資源管理のために 40%ものCPU 時間を消費していると言われている。しかし人手の資源管理はオペレーティングシステムの資源管理に到底かなうものではない。よって、性能管理のオーバヘッドは問題であると言うは説得力に欠ける。

末尾ながら、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。

〔神田 84〕　神田陽治，他：名前付けの統括管理システム NameMasterの構成，日本ソフトウェア科学会第１回大会(1984)，1C-3.

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

〔神田 87a〕　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.

〔神田 87b〕　神田陽治：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，電子情報通信学会，技術研究報告，COMP87-62(1987).

〔田中 87〕　田中二郎，他：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.

〔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.

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

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

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