ICOT Technical Memorandum: TM-0824

TM-0824

汎用並列マシン上での マルチタスクミックスによる 動的負荷分散法

神田陽治(富士通)

November, 1989

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

汎用並列マシン上でのマルチタスクミックスによる動的負荷分散法

神田 陽治
富士通(株) 国際情報社会科学研究所

1989 年 8 月 31 日

梗概

汎用並列マシンのための動的負荷分散の仕組みとして，マルチタスクミックスに基づき性能管理を行うオペレーティングシステム（OS）の枠組みを提案する．性能管理の目的は，汎用並列マシン上での対話処理技術の確立である．投入される仕事ごとに実行の性能記録を採りつつ，複数の仕事をじょうずにミックスし，並列マシンのたくさんのプロセッシングエレメント（PE）を活かし切ることを狙っている．仕事には，過去の性能記録から予想された準最適の分散戦略が与えられ，並列マシン上で実行に移される．並列マシンに余裕があるときに限り，準最適の分散戦略に加えて実験的な分散戦略も立案され，同時に実行される．実験的な分散戦略の目的は，準最適の分散戦略が改善できないかどうかを探るためである．

分散戦略はパス（いくつかの PE から構成される並列マシン上の仮想マシン）とプログラム分割戦術からなる．仕事内容を記述したプログラムと仕事を始めるプロセスが，最初にパスの始点に投入される．実行開始後は，PE の負荷が高まると，プログラムはパスの次の PE へプログラム分割戦術に基づいて転送される．プロセスは PE のローカルメモリに実行に必要なプログラムがないときに，パスに沿って転送される．結果的に，プロセスはプログラムを追いかけるように動的負荷分散される．

1 はじめに

本論文では，汎用並列マシン上での汎用オペレーティングシステム（以後 OS と略す）の新しい枠組みを提案する．汎用 OS 開発の設計目標は，マルチタスク下での動的負荷分散の問題の解決である．

汎用並列マシンが狙っているのは，プログラムの「手軽な」並列実行である．並列処理の恩恵を受けるために，プログラムに手をかなり加えなければならないというなら，ベクトル計算機あるいは専用並列マシン向きに手を加えた方が，高いコストパフォーマンスが得られる．汎用並列マシンが現在のワークステーション並みに成功するためは，存在意義を明らかにしなければならない．すなわち，

- ハードウェアシミュレーションエンジンなどの専用並列マシンや，スーパーコンピュータとも呼ばれるベクトル計算機など，高性能を目標に特別に設計された計算機に劣らない高速性を保証しつつ，使い勝手で有利なことを実証しなくてはならない．
- ワークステーションなど現在の逐次マシンと同等の使い勝手を保証しつつ，高速性で有利なことを実証する必要がある．

第一に，専用並列マシンやベクトル計算機はバッチ型で使われるから，それらに使い勝手で勝つためには対話処理で優れていることを示せばよい．そのためには，順不同で投入されてくる仕事をじょうずにスケジューリングしつつ並列に処理する技術（以後マルチタスクミックスと呼ぶ）が重要となるに違いない．第二に，逐次マシンと汎用並列マシンの最大の違いは，汎用並列マシンが多数のプロセッサエレメント（以後 PE と略す）を持っている点である．実

際の計算を進めるのはプロセッサのみであるから，逐次マシンに高速性で勝つためには，たくさんある PE を使い切るための負荷分散の技術が重要となるに違いない．以上の二点をまとめると，マルチタスク下での負荷分散が重要な技術課題であることがわかる．

さて本論に入るに先立って，使う用語を決めておく．投入される仕事をジョブ，分散実行されつつあるジョブをタスク，タスクを構成し PE への割り付け単位となる副仕事をプロセスと呼ぶ．どのように分散実行するかの方策を分散戦略と呼ぶと，タスクとは分散戦略付きのジョブのことだと言ってもよい．マルチタスクとは，複数のタスクを並列に計算機上で実行できる能力を指し，マルチタスクミックスとは，いかにじょうずにタスクを組み合わせて実行するかのスケジューリングの技術を指す．マルチプロセッシングとは，一つのタスクを構成するプロセスをいかにじょうずに PE に分散して実行するかのスケジューリングの技術を指す．

2 対話処理と並列処理

ここで対話処理と並列処理の相性は，潜在的には良いはずであることを述べる．並列処理は余分な逐次性をプログラムから排除することを目的とする．むやみに並列に実行させることが狙いなわけではない．逐次に処理すべき所のみプログラマは陽に指定すればよく，並列マシンも指示を守って逐次処理する．しかし指定されない箇所では，並列マシンが並列に実行しようと逐次に処理しようとまったく自由である．並列マシンには，自由度のある箇所をできる限り並列に実行することが期待されている．

一方，対話処理に望まれる特性は，全体処理時間の短縮だけでなく，最初の応答の速さである．解答がデータベース探索のように複数個あるときには，こちらが早く知りたい答えを優先的に検索して欲しいし，解答が CAD 等のグラフィックスで表現されるときはグラフィックス解の見たい箇所，たとえばマウスポインタの周り辺りから表示を開始してくれると嬉しい．並列プログラムでは，逐次に実行せよと指示された箇所以外はどんな順番で実行しようと構わないのだから，対話処理が要求する処理の優先度を上げて実行してやることができる．その結果，対話処理の応答性が確保できるはずである．

さて，プラグマ（pragma）方式あるいは注記（annotation）方式は，プログラムに優先度や分散の指示を直接埋め込む静的負荷分散法の一つである（例えば [5]）．しかし我々は以下に述べる理由で，プラグマ方式はマルチタスクミックスを必要とする対話処理には不十分であると主張する．

2.1 静的負荷分散方式としてのプラグマ方式の不十分性

プラグマ方式とは次のような静的負荷分散方式である．並列マシンと言えども，実装のレベルに近付けば逐次性が存在する．そこで，並列言語のプログラム上では並列にどんな順にでも実行してよいように書いてあっても，並列マシン上では特定の順番で実行した方が効率が良いという場合がある．このようなときに，並列言語プログラムの外見の機能には影響は与えず，並列マシンが性能を発揮できるよう，プログラムに直接書き込む指示がプラグマである．並列マシンごとに調整されたプラグマをうまくプログラムに付けられるなら，並列マシンの性能を最大限に発揮させることができる．プラグマにはプロセスの優先度指定の他，プロセスの PE への明示的な割り付け指示（どこへ，いつ），データ領域のローカルメモリへの明示的な割り付け指示（どこへ，いつ）などがありうる．

しかしプラグマは対話環境では以下に述べる三つの理由で能力不足である．第一に，プラグマ付けは煩雑な作業である．プラグマは並列マシンへの直接指示なので，プログラマが頭の中で抱いているプログラムの挙動とのギャップが大きく，プラグマ付けは簡単な作業ではない[1]．第二に，煩雑な作業を我慢するにしても，効果あるプラグマ付けは簡単ではない．経験によれば，ほとんど分散しないか分散し過ぎてしまうかであり，並列マシンに合った適度な分散を得るように付けるのは難しい [4]．ほとんど分散しなければ多数あるプロセッサの計算能力を活かすことができないし，分散し過ぎてしまえば通信コストのために性能は著しく劣化してしまう．第三に，効果的なプラグマ付けができたとしても，一つのタスク範囲を越えて，他のタスクとの干渉までを正しく予測してプラグマ付けすることは困

[1] プラグマを付けたメタインタプリタで部分評価する技法や，静的にプログラムを解析する方法 [6] もあるが，充分ではない．

難である．プログラムに直接書き込むプラグマは，基本的には静的な負荷分散手段であるので，いかなるマルチタスク状況下でも性能を発揮できるプラグマ付けは事実上不可能である．

以上の理由により，静的で低レベルな分散手段に過ぎないプラグマ方式は，マルチタスクを前提とした対話処理を行う汎用並列マシンには不適当である．プラグマに替わる（あるいは補完する）新しい動的負荷分散手法が必要である．

2.2 動的負荷分散方式としてのマルチタスクミックス / マルチプロセッシング

プラグマ方式に替わる，二段のスケジューリング機能を持つ，新しい並列 OS の枠組みを提案する．詳しくは次章で説明する．

マルチタスクミックス ジョブごとに毎回の実行の度ごとに性能の記録（性能記録）を取っていく．性能記録は，ジョブを実行するための分散戦略を選ぶ基準となる．分散戦略を与えられて，ジョブはタスクとして実行される．

マルチプロセッシング マルチタスクミックスにより投入されたタスクを，与えられた分散戦略の下で動的負荷分散実行する．複数のタスクの負荷が同一の PE にたまたま集中したときでも，負荷の集中が解消する方向に負荷の分散が自律的に解決される．

なぜ，二段のスケジューリング機能が必要かと言えば，第一に，マルチタスクミックス方式はジョブ投入時点で分散戦略を決めて与える静的方式なので，マルチプロセッシングのような動的な方式で補う必要がある．第二に，マルチプロセッシング方式は局所的に負荷分散する動的な方法なので，マルチタスクミックスのような大局的な方式で補う必要がある．

二段のスケジューリング機能を結び付けるのは，ジョブに与えられる分散戦略である．性能記録には，分散戦略が性能値とともにしまわれている．性能記録を管理し利用する OS を性能管理 OS と呼ぶ．

3 性能管理 OS

OS の基本的な役割は，計算機資源の効率的管理である．そして並列マシン上の OS が扱うべき最も重要な計算機資源は PE であり，その PE は計算を実際に進めるプロセッサと，プログラムやデータを格納するローカルメモリの二つの部分から成っている．

汎用並列マシン上の対話処理では，専用並列マシンやベクトル計算機のバッチ型の並列処理のときには深く考える必要の無かった点を考慮しなければならなくなる．バッチ型の並列処理の目的は大量高速計算なので，計算時間節約のためにプログラムは慎重に並列マシンの性能をできるだけ引き出すように作成されるはずである．その結果，プログラムの実行前のローカルメモリへの転送や実行後の消去の手間，最終結果のローカルメモリからの収集の手間は，特に問題にされないのが普通である．

しかし「手軽な」並列実行を狙う汎用並列マシンでの対話処理では，事情が違ってくる．プログラムは必ずしも慎重に作られないから，PE を使い切る分散戦略より，場合によっては一つの PE で実行した方が速いこともありうる．プログラムも本当に仕事を担当する PE のローカルメモリへのみ転送し，実行後はすみやかに消去するのが，後続の仕事の実行を妨げないためにも望ましい．すなわち，汎用並列マシンが活かしきれるか否かは，多数のプロセッサに負荷が均等になるようにじょうずに仕事を分散できるか否か，ローカルメモリが無駄なく使われるようにプログラムや計算結果をじょうずに出し入れできるか否か，の分散戦略の立て方に依っている．

3.1 マルチタスクミックス

ところで分散戦略の善し悪しは，仕事と並列マシンの相性によって微妙に変化する．従来の並列マシンでは個々の仕事の特性を考慮せずに，マシン側の都合で分散戦略を採用してきた．その結果，仕事と選んだ分散戦略の相性がたまたま合った場合にはうまく行くが，合わないと台数効果がうまく発揮されない結果に終わっていた．これは分散戦略の決定に仕事の性質，いいかえると仕事の性能をもっと考慮すべきことと強く示唆する．しかし並列プログラムの

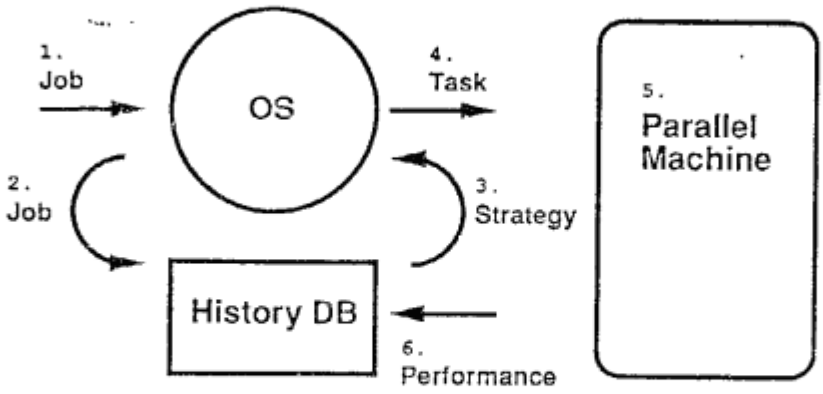

図 1: 性能管理 OS の構造

性能を形式的に捉えるのは難しいことであると思われるし，扱えたとしても現実に適用できるかどうかは別問題である．精密に計算される性能は必要ではなく，粗い近似で充分と思われる．

現実的な方法は，プログラムを実際に走らせて実験的に性能を測る方法である．OS は前回までの性能記録を見て，ジョブを効率良く並列マシン上で動かすための分散戦略を決定し，タスクに仕立てて投入する．OS はタスクが終了した後，今回の性能を記録し，次回からの分散戦略決定に役立てる．すなわち OS はジョブの性能を「学習」しつつ，その成果を再利用するわけである（図 1）

新しいジョブを新たに投入しようとしたとき，

- 並列マシンが中程度に混んでいるときは，現在の性能記録を参照して準最適の分散戦略を選び，その分散戦略のもとで実行する．実行後，性能記録を更新する．
- 並列マシンがたいして混みあっていないときは，現在の性能記録を参照して準最適の分散戦略を選ぶ他に，実験的な分散戦略も選び，ジョブを複数のタスクとして並列に実行する．もっとも早く答えを得たタスクがユーザに答えを返す．実行後，性能記録を更新する．残りのタスクは無駄な計算になるが，分散戦略の性能を学習するための実験の役割を果たしたことになる．
- 並列マシンが著しく混んでいるときは，ジョブの投入を延期する．もし，実験のために重複して行われているタスクがあれば止め，並列マシンの負荷を下げることを試みたのち，再度ジョブの投入を試みる．

並列マシン全体の忙しさをリアルタイムで知ることは一般に難しいが，マルチタスクミックスはジョブの投入時だけに実施され，かつジョブの投入はそんなにしばしばでないから，概算で忙しさを知れば充分である．

このフィードバックの仕組みがうまく働くと主張する根拠は次の通りである．第一に，対話環境では同じ（あるいは類似な）プログラムが何回も繰り返し実行されると期待できるから，記録した性能を後で使う機会は必ずある．デバッグ中のプログラムは少しずつ修正されながら何回も実行されるだろうし，コンパイラ等のユーティリティも何度も利用されるだろう．第二に，複数の分散戦略の比較のための実験が行われるときも，それまででもっとも成績がよかった分散戦略での実行も必ず行われる．並列マシンに余力があるうちは，その余力を性能測定に使うのであって，ジョブのすみやかな実行が実験のために無駄にされることはない．

問題は，他のタスクも並行して実行されるがゆえに性能を正しく測ることはできないのではないかという疑問である．性能を実際に測るのはマルチプロセッシングの段階であるが，それに先立ってマルチタスクミックスの段階があ

る．そしてマルチタスクミックスの段階では，並列マシンの事情が許す限り，なるべく実行環境が再現できるような分散戦略を割り付けようとする．それゆえにマルチタスクの環境でも安定して性能を測れると期待できる．

なお，実験タスクを止めるときは，最も最近起動された実験タスクを止めるとよい．スタック順で止めれば，せっかく長く走った実験タスクが無駄になるのが防げるからである．

3.2 性能とは何か

関心があるのはジョブが投入されたとき，どのような分散戦略のもとで実行すれば高い性能が得られるかを知ることである．性能記録は以下の三つ組で充分である．ジョブは少なくとも，プログラムと入力（初期値）を含む．

$$\text{性能記録} = [\text{ジョブ}, \text{分散戦略}, \text{性能}],$$

$$\text{ジョブ} = [\text{プログラム}, \text{入力}, \ldots].$$

OS は性能記録の管理者であり，性能記録をマルチタスク処理時に参照／更新する．性能記録は，準最適の分散戦略や実験用の分散戦略を選ぶときに参照される．ジョブをキーにして性能記録を検索すると，分散戦略の候補リストが得られる．性能記録から予測される「準最適の分散戦略」とは，候補リスト中，もっとも高い性能値を持った分散戦略である．性能記録から予測される「実験用の分散戦略」とは，候補リスト中，任意の分散戦略か，それを若干修正した分散戦略である．

性能を左右するのは仕事の内容を記述したプログラムばかりでなく，仕事の開始条件などを決める入力も影響する．性能記録を検索するジョブの定義の中に，入力が含まれているのはこのためである．

それでは性能とは何か．並列実行環境では実行が終了していなくとも，計算結果が求まっている場合があるので，仕事全体の実行の終了をもって性能を測ることはできない．そもそも何が求める答えかはユーザのみが知っている事柄である．さらに答えが求まったときに，満足できるほどすみやかに実行できたのか否かは，ユーザが判断する事柄である．

当面の解決策は，求める結果が求まったことの判断と，ユーザが見積もった性能値（以後，性能指標と呼ぶ）を返す義務を，ユーザに帰してしまうことである．性能を扱おうとするなら，ジョブの概念が拡張されねばならないのである．ユーザはジョブを定義するとき，性能指標を計算して返す性能測定法を，プログラムと入力とともに指定する．

$$\text{ジョブ} = [\text{プログラム}, \text{入力}, \text{性能測定法}].$$

3.3 マルチプロセッシング

ジョブは分散戦略を与えられてタスクとして並列マシンに投入される．分散戦略は，仮想マシンとしてのパスを含む．パスは並列マシン上に仮想的に構成される仮想マシンであり，いくつかの PE から構成され，その上でのみタスクを構成するプロセスは実行される．

$$\text{分散戦略} = [\text{パス}, \ldots].$$

ここで提案する新しい仕組みは，忙しい PE を避けるようにパスの上を負荷が自律的に移動する仕組みである．パス上の PE には順番が付いていて，PE は番号順に必要な個数だけパイプライン接続される．最初のプロセスがもっとも番号の若い PE に投入される．PE が仕事を開始し負荷が高まると，負荷はパスの次の PE へパイプライン経由で流され，負荷の分散が図られる．次の PE で仕事が再開実行され負荷が再び高まると，負荷はまたパスの次の PE へ流される．これが繰り返される．

パスの上を自律的に負荷が移動する仕組みは，負荷の集中を避けるために役立つ．並列マシンが多数の PE を持っていると言っても，投入タスクの数には上限がないので，マルチタスクミックスの段階でなるべくパス同志が重ならないように努力しても，タスクに与えられるパスが互いに一部でも重複せざるを得ない場合がある．重複してしまい，たまたま双方の仕事の負荷が流れて来て重なり合ってしまった時でも，自律的に双方のタスクは相手を避けるようにパスを流れ続け，負荷の集中が自律的に解消される（図 2）．

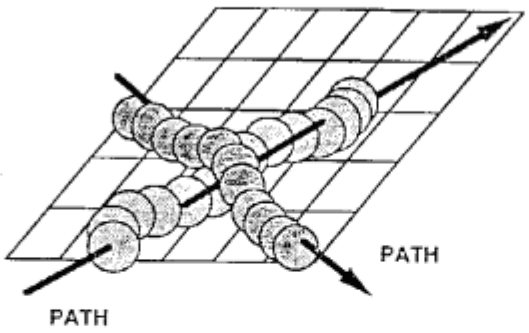

図 2: 交差したパスでの動的負荷分散

3.3.1 パス記述の方法

パスはラスターグラフィックスアルゴリズムを流用して記述する．アイデアの要点は，ラスター画面上のピクセルを並列マシンの PE に写像することにある．2 次元あるいは 3 次元のグラフィックス空間の格子点から，並列マシン上の PE へ 1:1 対応が付けば，基本的にはどんなマッピングでもよい．ただし，グラフィックス空間での遠近感覚が，並列マシン上でのネットワークを介した遠近感覚とあまり違わないことを要請する．グラフィックス空間で近い（遠い）点は，並列マシン上でも通信コストのかからない（かかる）距離にある二つの PE に写像されるべきである．

ラスターグラフィックスアルゴリズムは有用な特徴をいろいろ持っている．第一に複雑な線画でも図形でも容易に発生できること，第二にピクセルは逐次に生成されること，第三に図形を何度でも繰り返し生成できること，である．第一の性質により，複雑な仮想マシンも容易に設計できる．直線でも，円でも，多角形の塗り潰し第二の性質により，パスに属する PE には順番が自然に付けられる．この番号順を利用して，パス上の PE は順番にパイプライン接続される．第三の性質により，同じアルゴリズムをコピーして，各 PE が独立に他の PE の位置計算をしても，答えが食い違わないことが保証される．

例として，メッシュ状（二次元格子点上）に PE が並んだ並列マシンを想定する．並列マシンの縁は反対側の縁とつながり，メッシュは疑似的に無限平面になっているものとする．このときパス記述に 2 次ラスターグラフィックスアルゴリズムを用いると，ピクセルと PE には自然に 1:1 対応が付く（図 3）．パスとして直線を使う場合にしても考慮すべきことは多い．直線一本を描くにしてもただ真っ直ぐ引けば良いというものではなくて，始点終点を正確に通り，長さや角度によらず一様な明るさに光って見える直線を描く，高速なアルゴリズムが良いグラフィックスアルゴリズムとされている [3, page 21]．Bresenham アルゴリズム [3, page 25] はそんな優秀な直線描画アルゴリズムの一つである．そして良いグラフィックスアルゴリズムは，そのまま良いパス記述となる．パスを Bresenham アルゴリズムで記述すれば，始点を必ず通る性質は，最初にプロセスを投下する PE 位置が正確に決められることを保証し，長さや角度によらず一様な明るさに光る性質は，パス上の PE 分布に偏りがないことを保証する．

3.3.2 動的負荷分散法

パスを流れるのは，具体的にはタスクのプログラムとプロセスである．プロセスを流すことは仕事の負荷を分散することに他ならない．負荷は集中し過ぎれば，プロセッサの限界が効いて性能が劣化する．負荷は分散し過ぎても，プロセッサ間通信の限界が効いて性能が劣化する．ここで提案するのは，パスを利用した新しい動的負荷分散法である．

プログラムも流す対象としたのがここでの方法の特徴であり，これまでは暗黙になされていたプログラムのローカルメモリへの転送の問題も陽に解いている．各 PE がローカルメモリに持っているプログラムの種別は，他の PE

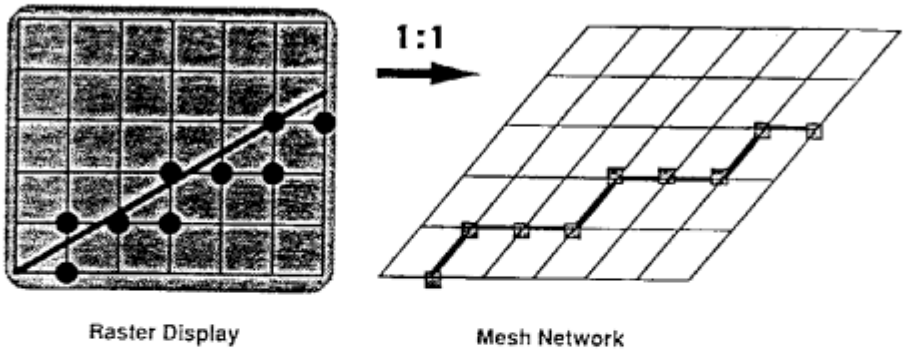

図 3: ラスター平面とメッシュネットワークとの 1:1 写像

にプログラムの一部を流したり，他の PE からプログラムが流れてくることによって，随時変化する．分散戦略はパスに加えて，プログラムの分割戦術を含む:

$$\text{分散戦略} = [\text{パス}, \text{プログラムの分割戦術}, \ldots].$$

PE がローカルメモリに持っているプログラムを，PE にローカルなプログラムと呼ぶ．最初にパスの先頭の PE にのみプログラムは転送される．ローカルなプログラムは，各 PE の判断で，尺取り虫のように延びたり縮んだりしながら，暇な PE を求めてパスに沿って転送されて行く．

タスクの実行を担当する PE 同志は動的にパイプライン接続される．原則としてローカルなプログラムを持つ PE が，パスに沿ってパイプライン接続される．PE は，ローカルなプログラムが送られて来た時，パイプラインに接続され，ローカルなプログラムが無くなったときに，パイプラインから外される．

パイプラインに参加している PE について，

- PE が負荷が高いと判断したときは，PE にローカルなプログラムの一部を分割し，パスに沿って流す．このルールの目的は，PE が忙しくなって来た時，プログラムを広く拡散することにある．尺取り虫が延びる動作を真似て，拡散はパス前方へ行う．
- PE が負荷が低いと判断したときは，パス後方の PE に対して，ローカルなプログラムを流してくれるよう指示する．要求された PE は，自分にローカルなプログラムの一部を分割し，パスに沿って流す．このルールの目的は，不必要に拡がってしまったプログラムをなるべく少ない個数の PE へ回収することにある．尺取り虫が縮む動作を真似て，回収はパス後方から前方へ向けて行う．

PE はそれぞれの判断でプロセスの実行を行う．PE でのプロセスの実行は，その際 PE にローカルなプログラムだけで試みられる．

- 実行に必要なプログラムがあるときは，同じ PE でふつうに実行される．
- 実行に必要なプログラムが無いときは，プロセスはパスに沿って自動転送される．

実行に失敗するのは，PE に必要なプログラムが揃っていないためなので，揃っている PE を求めて自動転送が行われる[2]．プログラムの広がりは暇な PE を求めてパスを転送されていくから，プロセスの広がりもまた暇な PE を求めて，プログラムの広がりを追いかけながらパスを転送されていく．

[2]パスの先に必要なプログラムが必ず保証されなければ，プロセスをパイプラインの先頭に戻す仕組みが必要となる．

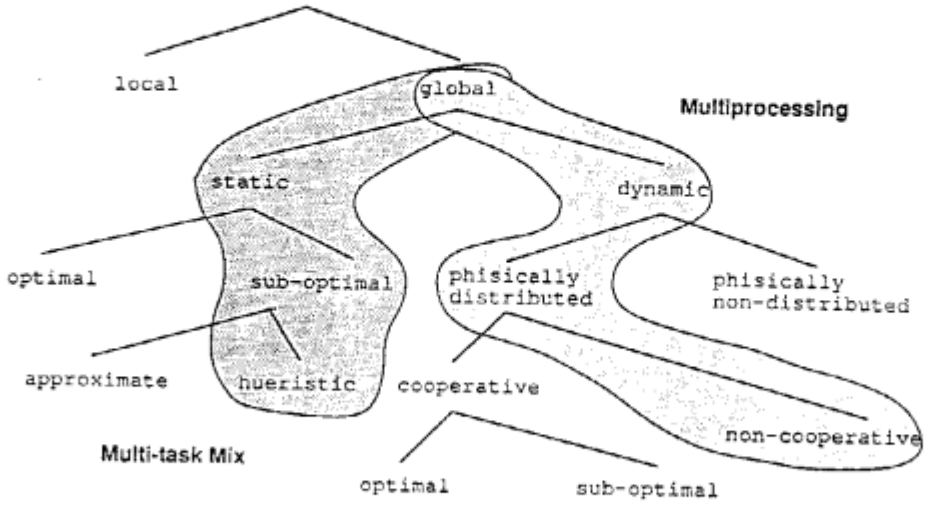

図 4: スケジューリング方式の階層分類

まとめれば本方式では，プログラムは PE の判断で直接転送され，プロセスはプログラムの流れを追うように自動転送される．プロセスを PE の判断で直接転送する方式に比べて，以下の利点がある．

1. プログラムを予めすべてのローカルメモリへ転送する必要はない．実行時にパスに沿って必要になった PE にのみ配られる．プログラムの消去もパスに沿ったローカルメモリを消去すれば充分である．
2. 必要なプログラムが揃っていないプロセスのみ転送されるから，仕事の負荷を選択的に分散できる．
3. 一般的にはプロセスは数多く生成されるから，転送する判断をプログラムに関して行った方がコストが小さい．

4 分類

負荷分散の問題はスケジューリングの問題に他ならない．文献 [1] は，汎用の分散システムのスケジューリング法の分類を試みている．図 4に示す彼らの分類法は，スケジューリング法を全体の中に位置付けるための階層分類 (Hierarchical Classification) と，個々のスケジューリング法の特徴を列挙する特徴分類 (Flat Classification Characteristics) の二つの部分からなる．ここでは彼らの分類法に我々の方法を位置づけてみる．

4.1 マルチタスクミックスの分類

性能管理 OS はスケジューリングを二段階に分けて行う．マルチタスクミックスの段階はジョブに分散戦略をスケジューリングする．分散戦略はパスを含む．

[階層分類] Global, static, sub-optimal, heuristic,

[特徴分類] Non-adaptive, load-balancing, no bidding, probabilistic, one-time assignment.

マルチタスクミックス時のスケジューリングは，実行直前に (static)，全 PE を対象にして (global)，パスがなるべく重ならないように (load-balancing)，過去の性能記録を参考に (sub-optimal, heuristic)，割り当られ

る (no bidding)．スケジューリング法自体は固定されて (non-adaptive) いるものの，余裕があるときは別のパスも実験的に試す (probabilistic)．パスは一度与えられると変更されない (one-time assignment)．

4.2 マルチプロセッシングの分類

マルチプロセッシングの段階はパス上でのプログラム分割の戦術をスケジューリングする．分散戦略はプログラムの分割戦術を含む．

[階層分類] Global, dynamic, physically disrtributed, non-cooperative,

[特徴分類] Non-adaptive, load-balancing, no bidding, probabilistic, dynamic assignment.

マルチプロセッシング時のスケジューリングは，実行時に (dynamic)，パス上の PE を対象 (global) にして，ゴールが特定の PE に集中しないように (load-balancing)，各 PE の独自の判断で (physically distributed, non-cooperative)，ローカルなプログラムの一部を転送したり併合を要求すること (dynamic assignment) で達成される．プログラムの分散に沿って，仕事の負荷の分散もパスに沿って自動的に行われる (no bidding)．スケジューリング法自体は固定され (non-adaptive) いるものの，余裕があるときは別のプログラムの分割戦術も実験的に試す (probabilistic)．

5 性能管理 OS のプロトタイプ

性能管理 OS は，性能記録を用いてマルチタスクミックスを行う．それゆえマルチタスクミックスがじょうずに働けるかどうかは，性能をうまく学習できる能力と，学習した性能値に基づいてじょうずに準最適な分散戦略を予測できる能力に依ってくる．ここで強調すべきは，「OS = 問題解決機」さらに言えば「性能管理 OS = 負荷分散の問題解決機」というパラダイムである．

「性能管理 OS = 問題解決機」というパラダイムを追求するために，並列論理型言語 GHC[7, 第1-4章] を用いてプロトタイプを作成した．GHC の簡潔な記述力はプロトタイプ作成に向いている [7, 第6-9章 GHC 応用プログラム]．しかしながら性能管理 OS の仕組みを備えたプロトタイプが動いたということと，現実のスケールで性能管理 OS が実際に役に立つということにはまだまだギャップがある．

我々が以下で展開するのは，性能管理 OS には，並列論理型言語が適しているという主張である．それは結果的に，性能管理 OS のプロトタイプを並列論理型言語で記述することに意義があると主張する．

5.1 並列言語型性能管理 OS

GHC の並列言語として利点を，現実のスケールで示しているのが，ICOT (新世代コンピュータ技術開発機構) が開発中の並列 OS, PIMOS[2] である．PIMOS は GHC をシステム記述向きに強化した (あるいは制限した) KL1 で書かれている．GHC では並列実行の仕組みと同期の仕組みが一体となっているおかげで，並列プログラミングではしばしば発見しづらいバグとなる同期ミスが，PIMOS 開発過程ではほとんど問題にならなかったと報告されている [2, page 250].

しかし同時に GHC は論理型言語でもある．ICOT が当初から論理型言語を中心にプロジェクトを押し進めた理由の一つは，論理型言語は問題解決など高度プログラミングに向いているという主張であった．「性能管理 OS = 負荷分散の問題解決機」というパラダイムは，GHC が論理型言語として問題解決向きであるという利点を，現実のスケールで証明するのにとても適した例題である．ところが PIMOS では負荷分散の問題は手つかずであり，ゴールの優先度と送り先 PE を指定できるプラグマ機能を用いて，ユーザの責任に帰しており，この点を追求する例題とは未だなっていない．従って性能管理 OS のプロトタイプを GHC で記述し，実験 (feasibility study) を行うことには意義があったと考える．

作成した並列論理型性能管理 OS のプロトタイプの詳細を説明する代わりに，並列論理型言語を記述言語に採用したことで，詳細化が進んだ点を説明する．

最初はジョブ指定法の実装についてである．並列論理型言語では，プログラムはクローズ（節）の集合で与えられ，頭部に同じ述語名を持ち同じ数の引数を持つクローズ群は，まとめてプロシージャと呼ばれる．プロセスに相当するのがゴールである．ゴールは PE によってリダクションされ，0 個以上の子ゴールへ置き代わる．性能測定法の指定は，性能指標に実行時に具体化される変数に置き換え，入力ゴールの中に性能指標を計算するゴールを含めることで実装できる．性能指標用の変数と性能指標を計算するゴールとの間の通信は共有変数の形で行う．よって，

$$\text{ジョブ} = [\text{プロシージャ群}, \text{入力ゴール}, \text{性能指標変数}].$$

次はゴールの自動転送方式についてである．PE でのゴールのリダクションは，その PE にローカルなプログラムだけでリダクションが試みられ，実行に必要なプログラムがあったどうかは，ゴールの成功／中断／失敗を調べればわかる．そこで，

- リダクションが成功 (success) したとき，子ゴールは（あれば）同じ PE へ戻される．
- リダクションが中断 (suspended) したときは，ゴールは同じ PE へ戻される．
- リダクションが失敗 (fail) したときは，ゴールはパスに沿って自動転送される．

5.2 考える OS

「OS = 問題解決機」というパラダイムを成功させるためには，標語的に述べれば，OS はただの OS から考える OS へと進化しなくてはならない．考える OS 実現のための課題について考察する．性能管理 OS はマルチタスクミックス時にジョブに分散戦略を割り付けるが，並列マシンがジョブをすみやかに達成できるかどうかは，分散戦略の正否に強く依っている．

5.2.1 パスの割り当て

性能管理 OS は性能記録に基づいて，ジョブに分散戦略を割り当てる．パスの決定は分散戦略の重要な決定事項である．

物理マシン上の PE の数は限られているから，パスがたくさん必要になれば一つ以上のパスに参加する PE が多く出てくるのは避けられない．マルチプロセッシングの動的負荷分散実行機能により，たとえ PE が重なっていても負荷の衝突は自動的に解消されるが，PE が重なることが少ないようにパスをジョブに分配できればそれに越したことはない．

パスのジョブへの割り当て問題は，パス同志の重なり（共有されている PE の個数）を充分低く押さえる一種の箱詰めパズルと見ることができる．箱詰め問題は，特にパスとして縦横斜めの直線を想定したとき，8-queen 問題 [7, page 93] に似ている．基本的な戦略は，

- 希望されたパスが，他で使用中のパスと重なりが小さければ，それを割り当ててよい．
- 希望されたパスが，他で使用中のパスと重なりが大きい場合には，希望したパスに類似のパスで他と重なりが小さいパスを生成して割り当てる．

注意すべきは，第一に，使用中のパスになるべく重ならないパスで，実績あるパスが優先的に割り当られるようになっている．これは準最適な分散戦略をなるべく活かしたいからである．第二に，パスはグラフィックスアルゴリズムで記述されるから，重なりの度合いを効率的に算出することが可能である．第三に，類似なパスを生成する場合にも，パスがグラフィックスアルゴリズムで記述されているために，生成パラメータの増加減だけで容易に行える．位置，大きさ，傾きなどを容易に変えられる．

5.2.2 プログラム分割戦術の割り当て

性能管理 OS は性能記録に基づいて，ジョブに分散戦略を割り当てる．プログラムの分割戦術の決定は分散戦略の重要な決定事項である．

PE にローカルなプログラムは PE の負荷状況に応じて，その一部が転送されたり併合されたりする．このとき，ローカルなプログラムの分割戦術について自由度が残っている．もっとも簡単な方法は，プログラムをいくつかのクローズの固まりに分割して，各 PE でローカルにキューで管理し，FIFO 順で転送する方法である．プログラムの一部を転送するときにはキューの先頭から送り，受け取ったときはキューの後尾に付ける．

プログラムの分割には注意が必要である．プロシージャを分割しない場合には安全である．しかしプロシージャをさらに分割した場合には，答えが求まらなくなる危険性がある．例えば二つのストリームを非決定的に併合する次に示した merge [7, page 70] の場合，値が来るまで中断 (suspended) 状態で待つ．例えば，第一クローズは第二引数のストリームに値が入って来ても中断状態のままである．ふつうなら第二クローズが反応してストリームを処理する．それゆえ第二クローズが次の PE に送られてしまうと，第二引数のストリームに値が入って来ても，第一クローズは中断したままで失敗しないので，ゴールの転送は行われないままになってしまう．

```
merge([X|Xs],Ys,Zs):-  % 第一ストリームを処理
   true | Zs=[X|Zs1], merge(Xs,Ys,Zs1).
merge(Xs,[Y|Ys],Zs):-  % 第二ストリームを処理
   true | Zs=[Y|Zs1], merge(Xs,Ys,Zs1).
```

6 議論

汎用並列マシンを現在のワークステーション並みに普及させることを目標に，対話処理に焦点を当てた性能管理 OS を提案した．これまで汎用並列マシンの研究というと，大量高速な実行のみが強調されて，対話処理という観点からの研究は乏しかった．プログラムをいつ PE に転送するのか，どの範囲の PE に送ればよいのか，そして，いつどこのプログラムを消去したらよいのか，といった問題は対話処理では重大である．対話環境は本質的にマルチタスク環境であり，あるプログラムによって不必要に PE のローカルメモリが専有されれば，他の仕事のタスク実行を不用意に遅らせることにつながる．

ここまでは汎用並列マシンについては，多数の PE がネットワークで互いにつながったものという以上の事は述べて来なかった．ジョブを構成するプロセスは，並列マシン上の仮想マシンであるパスに閉じ込められて実行されるから，マシン全体の PE 台数とは無関係でいられる．台数が少なくてパスが重なってしまい，タスクが互いに干渉しても，マルチプロセッシング機能により負荷は平均化される．台数があれば，多くのタスクを互いに干渉なく実行させることができ，性能予測も正確に出来るようになる．

パスには分割されたプログラムが流され，それを追うようにプロセスが流される．PE に余裕がある限りプログラムを流さなず，PE が忙しくなり流さなければならないときも，プログラム全体をなるべく広げないように流す．拡がっても負荷が下がれば，再びプログラムは集められる．このことはネットワークの通信コストが比較的大きい並列マシンでも，性能管理 OS が充分使えることを意味する．

最後にユーザから見た，汎用並列マシン上の性能管理 OS が与えるインパクトについて述べる．汎用並列マシンの普及を妨げているのは，並列プログラミングが難しいせいだと言われるが，与えられたマシンの性能を最大限引き出すプログラムを書くことは，並列マシンに限らず難しい．それより汎用並列マシンの普及を妨げている原因は，「手軽な」並列実行環境が提供されていなかったせいであると我々は考える．汎用並列マシンで走らせられるのは，よく考えられた並列アプリケーションばかりではない．デバッグ中のプログラムやその他，日常的なユーティリティも必要である．もしそれらをチューニングしないと良い性能が出ないならば，並列マシンを日常的に使う意義が出てこない．性能管理 OS を使えば，特別な手を加えることなくマルチタスクミックスがじょうずに達成され，「手軽な」並列実行環境が実現できる．

7 おわりに

汎用並列マシンのための動的負荷分散の方式について，性能管理を行う OS の枠組みを提案した．性能管理 OS 設計の狙いは，汎用並列マシンが対話処理の点で，専用並列マシンやベクトル計算機に優れていることを示すことを通して，汎用並列マシンが現在のワークステーション並みに広く普及しうる可能性を追求することにあった．バッチ指向の並列 OS が目標ではなく，チューニングが足りないプログラムもじょうずに取り混ぜて並列マシンの性能が引き出せる，マルチタスクで対話型の並列 OS の新しい枠組みの開発を目標とした．

性能管理 OS の核心は，ジョブに分散戦略を割り当てるマルチタスクミックス部分である．ジョブに分散戦略を与えて実行し，性能記録をとり，再びそれに基づいて改善された分散戦略を割り当てる．過去の事例からスケジューリングを行うこの方式は，Case-Based Reasoning[8] の一例でもある．残念ながら本論文では性能管理 OS 全体の枠組みを提案したに留まり，準最適分散戦略を見つけだす問題解決の方法までは議論できなかった．

謝辞

富士通国際研の第二研究部第二研究室の諸氏，原稿に有益なコメントを頂いた園藤進氏，戸田光彦氏に感謝いたします．GHC による性能管理 OS プロトタイプの作成には，富士通 SSL の太田祐紀子さんの補助を得ました．なお，本研究は第五世代コンピュータプロジェクト再委託研究の一環として行われました．

参考文献

[1] T. Casavant and J. Kuhl.: *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*, IEEE Trans. Softw. Eng., Vol. 14, No. 2, pp. 141-154 (1988).

[2] T. Chikayama.: *Overview of the Parallel Inference Machine Operating System (PIMOS)*, In Proc. FGCS'88, Vol. 1, pp. 230-251 (1988).

[3] W. Newman and R. Sproull.: *Principles of Interactive Computer Graphics* 2d ed., McGrwa-Hill (1979).

[4] 大原有理, 他: 核言語 *KL1* 分散実行方式 ソフトウェアシミュレータによる評価, 情報処理学会 35 回全国大会論文集 (1), pp. 637-638 (1987).

[5] E. Shapiro.: *Systolic Programming: A Paradigm of Parallel Processing*, In Proc. FGCS'85, pp. 458-470 (1985).

[6] E. Tick.: *Compile-Time Granularity Analysis for Parallel Logic Programming Languages*, In Proc. FGCS'88, Vol. 3, pp. 994-1000 (1988).

[7] 並列論理型言語 GHC とその応用, 知識情報処理シリーズ 6, 共立出版 (1987).

[8] Proceedings of Case-Based Reasonoing Workshop, DARPA (1888),