ICOT Technical Memorandum: TM-0823

TM-0823

マルチタスクミックスOSの
プロトタイプとその評価

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

マルチタスクミックス OS のプロトタイプとその評価

神田 陽治

富士通(株) 国際情報社会科学研究所

あらまし

汎用並列マシンのための動的負荷分散の仕組みとして，性能管理を行うオペレーティングシステムの枠組みを提案する．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する．仕事には，過去の性能記録から予想される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．

Multi-task Mix OS and its Evaluation

Youji KOHDA

International Institute for Advanced Study of Social Information Science, FUJITSU LIMITED

17-25, Shinkamata 1-Chome, Ota-ku, Tokyo 144, Japan

Abstract

A new dynamic load balancing scheme is proposed, and the structure of *Performance Management* operating system is described. It has two-storied scheduling mechanism: *Multi-task Mix* and *Multiprocessing*. The Multi-task Mix picks up a "best-guessed" distribution strategy based on performance histories for each submitted job, mixing the job with the tasks currently running on the machine. The Multiprocessing executes the job as a task under the distribution strategy. An "experimental" distribution strategy is also picked up and executed along with the "best-guessed" one, if the machine can afford it. The purpose of the experiment is to make the guess of "best-guessed" strategy better.

1 はじめに

汎用並列マシン上での汎用並列オペレーティングシステム (OS) の新しい枠組みを提案する．本 OS の狙いは，マルチタスクを使っての動的負荷分散の枠組みの導入である．本論文の目的は，作成したプロトタイプを用いて，枠組みが達成すべき効果について予備的調査を行うことにある．

2 三つのなぜ

研究の背景を明確にするために，ここでは三つのなぜ，「なぜ汎用並列マシンか」，「なぜマルチタスクか」，「なぜ動的負荷分散か」に答える．

2.1 なぜ汎用並列マシンか

汎用並列マシンの目的はいうまでもなく，プログラムの高速処理にある．汎用並列マシンの重要な資源はプロセッサであり，複数のプロセッサにうまく仕事を配分して高速化を達成しようとする．一方，高速処理という意味では，他にも逐次処理ながらパイプラインで高速化するベクトル計算機や，シストリックアレイのように特定の応用に目標を絞って設計された並列処理専用マシンがある．もしかりに，汎用並列マシンで計算する利点が，ベクトル計算機あるいは並列処理専用マシンに比較して充分現れて来ないならば，労力をかけて汎用並列マシンを作る意味が出てこない．**汎用並列マシンが，ベクトル計算機や並列処理専用マシンに比べて，真に優れる応用分野を発見しなければならない**．これは挑戦に値する重要な問いである．その分野とは何であるか．

ベクトル計算機は高速大容量のバッチ処理向きであり，並列処理専用マシンは専門的な仕事に特化して作られる．そこで速度の点で汎用並列マシンが他に優れると証明するのは難しいから，速度向上以外の観点から利点を見出すことが必要と我々は考える．

2.2 なぜマルチタスクか

我々は，**対話処理型のアプリケーションにこそ，汎用並列マシン上での並列処理は向く**と主張したい．OS 自身もユーザとの対話を主体とした，対話式の重要なアプリケーションの一つであることに注意．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，応答の速さである．たとえば解答が複数個あるときは，最初に欲しい解から返して欲しい．解答がグラフィックス表示等のときにはグラフィックス解の見たい箇所，マウスポインタの周り辺りなどから表示を開始して欲しい．前者の例にはデータベースでの検索があり，後者の例には CAD でのグラフィックス解表示がある．

対話処理においては，ユーザの都合で仕事はいつ投入されるかわからない．すなわちマルチタスクの環境が必然的である．汎用並列マシンには数多くのプロセッサがある．素早い応答につながるタスク，プロセスを優先してプロセッサに割り当てることにより，要求に素早く答えることが原理的に可能である．

2.3 なぜ動的負荷分散か

対話処理型のアプリケーションに汎用並列マシンが向くと主張する根拠は次の通りである．解が求まってくる順番を，実行中にまで遅れて指定できることが，並列処理が逐次処理に優れている点である．並列処理の本質は，余分な逐次性の排除である．本当に逐次に処理すべきもののみユーザは逐次と指定すればよい．逐次と指定されなかった箇所を逐次にやるか並列にやるかは，並列マシンに一任される．すべて並列にやっても，すべて逐次にやっても，並列と逐次を混ぜてやっても構わない．そこで，どんな順番で実行しようと構わないというのだから，必要な応答を返す計算を優先できるわけである．そして，この実現には動的負荷分散が不可欠である．

動的負荷分散では分散戦略を予め決定しておくのではなく，実行時の状況に応じて臨機応変に戦略を変え，より良い結果を得ようとする．この能力は対話処理の環境には不可欠である．なぜなら投入される個々の仕事は独立であり，予め仕事間の干渉を予測して戦略を立てることはできないからである．

3　二つの課題

これまで関心を引くことが少なかったように思われる課題を二つ提出する．

3.1　プログラムの転送方式

対話処理を中心とした動的負荷分散を行うからには，多数あるプロセッサのローカルメモリへのプログラムの転送(load)の問題を避けては通れない．すべてのプロセッサへプログラムを予め転送してしまう方式は，必ずしもプロセスがすべてのプロセッサへ配分されるとは限らないので，無駄が出る．ローカルメモリを浪費するばかりでなく，空きメモリが無くなり他のプログラムが格納できず，後続のタスクの実行が不必要に遅らされる無駄も無視できないからである．といって，必要とわかった時点で転送を始めるのでは，プロセスの粒度が細かい並列マシンでは転送終了を待つ遅れが無視できない．両者の中庸に位置するプログラム転送方式が必要である．

3.2　性能管理 OS

汎用並列マシン上での動的負荷分散の問題を解くにはどうしたらよいだろうか．いろいろな方法が提案されて来たが，「たまたま相性が良い問題に当たれば」うまく行く程度に見える．**限界の原因は，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計ろうとするところにある**ように思える．そして限界を打ち破る一つの方策は，仕事ごとの特質を考慮に入れて負荷分散を行うことだと我々は考える．仕事ごとの特質のことを，以下では仕事の性能と名付ける．

従来，モジュールは仕事の機能を記述したが，仕事ごとの特質を表す性能という重要な面の記述が抜けていた．機能はモジュールが行う計算のセマンティックスに関係するが，性能はモジュールを有効に利用するためのプラグマティックスの領域に関係する．並列環境を活かすには性能を積極的に扱う方法が必要である．

ところでユーザに性能の管理を任すことはできない．なぜなら性能は OS のスケジューリング戦略を決め，ひいてはシステム全体の資源戦略に影響するからである．そこで性能管理は OS の仕事でなくてはならない．実行履歴を管理し，後のタスク実行に役立つように編集しておくのは，OS の役目である．次の仕事の実行のときには，これまでの実行履歴の解析に基づいて分散戦略を決定し使用する．このようなフィードバックの仕組み，あるいは学習の仕組みを性能管理 OS は備える．

性能を取っておいて後のスケジューリングに役立てるという仕組みが有効に働くためには，同じ仕事あるいは類似の仕事が繰り返し投入されるという前提が必要である．ここで狙っているのは対話型の OS であり，この仮定が成立する．ただし作成中のプログラムでは，だんだんと性能が変化していくので，この変化に追随できる融通性がフィードバックの仕組みの中に必要である．

4　性能管理を行う OS の枠組み

さていよいよ，汎用並列マシンのための性能管理 OS の枠組みを述べる．始めに説明の便宜のために，いくつかの言葉を取り決める．

汎用並列マシンは完備された OS を持ち，順不同に投入される仕事を並列に処理して行くマシンである．これら独立の仕事の一つ一つをジョブと呼ぶ．OS はジョブに適切な分散戦略を割り付けて，並列マシンに投入する．この分散戦略が与えられたジョブをタスクと呼ぶ．さらに各々のタスクは内部でも並列に実行される．タスク内で並列に処理される単位をプロセスと呼ぶ．

汎用並列マシン上の OS の役目は，独立のタスクをうまく組み合わせ並列マシン全体にうまく負荷を分散させることと，各タスクを構成するプロセスを適切にプロセッサに分散配置することにある．前者の段階をマルチタスクミックス，後者の段階をマルチプロセッシングと呼ぶ．

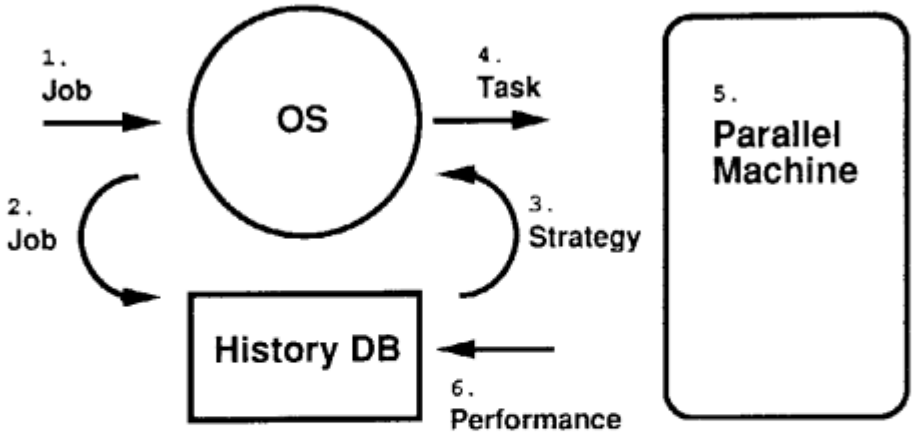

図 1: 性能履歴を管理する OS の枠組み

4.1 性能管理を行う OS の仕組み

図 1は性能管理を行う OS 全体の枠組みを示している．説明中の数字は，図中の番号に対応する．

マルチタスクミックス 投入されたジョブに分散戦略を割り当て，タスクとしてマルチプロセッシングに引き渡す．マルチタスクミックスの際，性能履歴データベースを参照する．

1. 投入されたジョブを受け付ける (1).
2. ジョブのこれまでの性能記録を参考に (2)，準最適[1]の分散戦略を選ぶ (3)．並列マシンに余力があれば，性能を学習するための実験用分散戦略も選ぶ．
3. タスクをマルチプロセッシングにより実行し始める (4)．準最適の分散戦略タスクと（あれば）実験用分散戦略タスクを起動する．
4. タスクの終了を待ち，答えを返したタスクの性能値を記録する (6)．終了したタスクが使用したローカルメモリを空にする．

マルチプロセッシング 与えられた分散戦略の下でタスクを動的負荷分散実行する (5)．準最適な分散戦略による本実行と，実験用分散戦略による実験実行の二種がある．

1. 分散戦略には**パス**記述が含まれる．パスは並列マシン上に構成される仮想マシンであり，いくつかの PE を通信路で順番に結んだものである．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される．4.2節で詳しく述べる．
2. 分散戦略には**プログラム分割リスト**が含まれる．プログラムが PE の負荷に応じてパスに沿って流される．プロセスは PE に実行に必要なプログラムが無いときに，パスに沿って流される．結果的にプロセスの雲が，混んでいない PE を探しつつパスに沿って流れて行く．この仕組みにより，たとえパスが他のパスと重なっていても負荷の衝突は自律的に解消される．4.3節で説明する．

マルチタスクミックスは仕事が投入されたときに一度行われ，マルチプロセッシングは仕事実行中に自律的に働く．マルチタスクミックスでは二種の分散戦略を選ぶ．本実行用の準最適戦略と実験実行用の実験用分散戦略の二種である．マルチプロセッシングでは仕事を与えられた分散戦略のもとで実行する．実験用分散戦略は並列マシンに余力がある場合のみ立案されるから，並列マシンに余力がない場合には本実行に影響を与えることは防がれていることに注意する．

[1]準最適という言葉を，ここでは過去の事例から推測できる限りにおいて最良という意味で使用している．

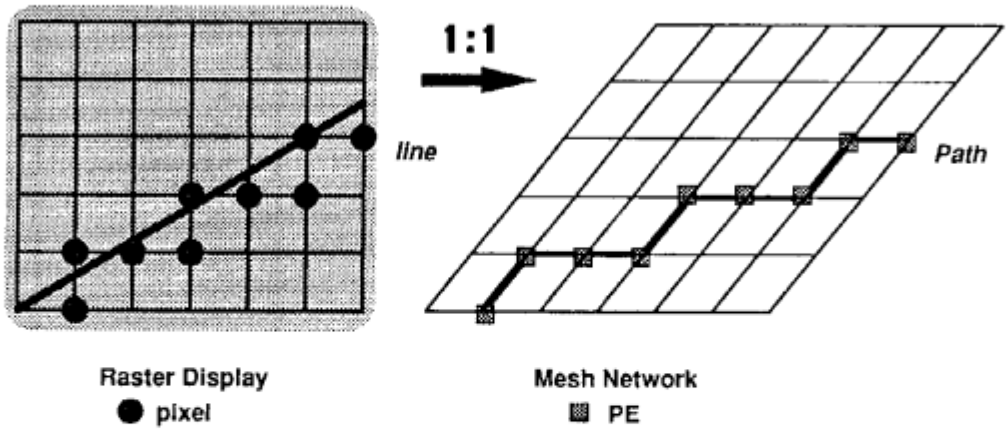

図 2: ラスター平面とメッシュネットワークとの 1:1 対応

4.2 パス

パスは，並列マシン上に構成される仮想マシンである (図 2)．投入されたジョブにはパスが一つ割り当てられ，ジョブを構成するプロセスはパス上のプロセッサによってのみ実行される．ここではパスの記述にグラフィックスアルゴリズムを流用することを提案する．

話を簡単にするために，汎用並列マシンのプロセッサ間結合を平面メッシュ構成に限定して話をすすめる．ここでのアイデアは，メッシュ平面をフレームバッファに見立てて，コンピュータグラフィックスの技術を利用するというものである．（二次元）ラスターグラフィックスアルゴリズムは，フレームバッファに図形を書き込む [3]．フレームバッファはピクセル（あるいはドット）を格子状に並べたものであり，アルゴリズムは，ドットをオン・オフして図形をフレームバッファ上に描く．次の三つの点が重大である．第一に，ラスターグラフィックスの分野は既に広く研究が行われていて，複雑な図形がいろいろ描ける．第二に，アルゴリズムは図形を構成するドットを順番に 1 ドットずつ，再現性あるように発生できる．第三に，（この性質はコンピュータグラフィックスではふつう利用されないが）アルゴリズムの計算をわざとゆっくり進めることによって，図形の描画速度を変えることができる，という点である．

アイデアの要点は，フレームバッファメモリをメッシュ平面に対応させることである (図 2)．すなわち，フレームバッファの 1 ドットに，メッシュ平面上のプロセッサー台を 1 対 1 に対応させる．するとラスターグラフィックスアルゴリズムは，メッシュ平面上のプロセッサを，第一に総体的にみるとある図形になるように，第二に描く過程を見ると順番に一台一台，第三に任意の速度で指定して行くアルゴリズムを与えている．すなわち，新しいプロセッサ割り当て法を与えている．パス記述にグラフィックスアルゴリズムを使う利点は，コンパクトな表現でありながら高度で再現性あるプロセッサ割り当て法を与えている点である．

4.3 プログラム分割

プログラムが格納されねばならないのは，パス上のプロセッサのローカルメモリにだけである．しかしそれでも，パス上のすべてのプロセッサにプログラムを予め全部コピーしておくのは得策ではない．パスのどの辺りまでゴールが投げられるかは，実行してみないとわからないから（プロセッサ数は有限個，それゆえパス上のプロセッサは実際には有限個と言っても），予め格納しておいても使われないコピーが出てくるからである．そこで状況に応じてプログラムをプロセッサに動的に転送する方式が欲しくなる．

動的に転送するならばプログラムを小さい単位に分割しておくと便利である．ジョブのプログラムを予めいくつかの部分に分割し，転送の単位とする．ただし分割といっても，同一内容のプログラム部分が複数のプログラム単位に重複して入っていても構わない．

4.4 マルチタスクミックス

資源管理は資源の効率的な分配を目標とする．マルチタスクミックス時のパスの分配戦略とプログラムの分割戦略の部分に，従来から研究されてきた問題解決の技術 [2] が使えると期待できる．

実行履歴データベースは [ジョブ，分散戦略，性能値] から成る三つ組の集まりである．各組は，指定された分散戦略の下で指定されたジョブを実行したときの，測定された性能値を記録している．ただし性能値とは何かという問題に真正面から答えることを避けるために，性能値を計算して返す責務をジョブの側に帰してしまうことにする．ジョブのプログラムには，本来の計算に加えて，望む答えがどの程度うまく求まったかの判断を数値に還元して返す計算を含めて行うことが期待される．

4.4.1 基本アルゴリズム

ジョブに分散戦略を割り当てる基本アルゴリズムを以下に示す．探索空間が広すぎて，適当な分散戦略が発見できるまでに時間がかかりすぎるのが難点である．

- 過去に良い性能値を示した分散戦略が現在割り当て可能ならば，そして実験実行でない場合ならば，その分散戦略を割り当てる．
- 過去に良い性能値を示した分散戦略が現在割り当て不能ならば，あるいは実験実行の場合には，ランダムに生成した分散戦略で現在割り付けられていない分散戦略を割り当てる．

新しい分散戦略が，ジョブがタスクとして実行される度ごとに現在の並列マシン全体の状況を考慮して付与される．分散戦略はパスとプログラム分割から成っている．パスの場合には，並列マシンの平面メッシュ上で，他のタスクのパスとなるべく重ならない条件のときに割り当て可能である．他となるべく重ならないパスを見つけだす基本的な方法はしらみつぶし法である．プログラム分割の場合には，常に割り当て可能である．

パスの付与は具体的には，グラフィックス手続きを一つ与えることでなされる．グラフィックス手続きをパス記述に用いる利点は，パラメータ変更で系統的にパスの形や大きさを変更でき，また，交点の計算などのアルゴリズムがしっかりしていることなどである．

プログラム分割は，プログラムを小さい単位に分割することで与える．一つ一つの単位をプログラム単位と呼ぶ約束とする．各プロセッサはローカルメモリにプログラム単位を一時的に蓄えるプログラムキューを持っている．プログラムキューに入っているプログラム単位をすべて併せただけが，そのプロセッサから見えるプログラムである．プログラムキューは次のように使う．プログラムを転送する要求が起こった場合には，プログラムキューの先頭のプログラム単位を一つパスに沿って転送し，送られた先ではそれを自分にローカルなプログラムキューの後尾に付ける．

5 マルチプロセッシング

パスとプログラム分割を用いることにより，プログラムの転送方式を解決し，プロセスの動的負荷分散も行える．アイデアの要は，プロセスの実行に必要なプログラムは予め転送しておかねばならない，という従来の発想を逆転して，**プログラムが転送されていなければプロセスをプログラムのある所へ転送する**と考え直すところにある[2]．

- プロセスは，プロセッサのローカルメモリに格納されているプログラムを使って実行に失敗したとき，パスをたどって一つ先のプロセッサに投げられる．そして投げられた先のプロセッサのローカルメモリに格納されているプログラムで再び実行が試みられる．

すなわち，プロセスは必要なプログラムが転送されているプロセッサへ自動的に転送される．そこで，プログラム単位を動的にパス上で配置転換することによりプロセスの自動分散が行えることになるのである．この

[2]ちょっと見にはインヘリタンスの発想に似ている．

ときパスは，プログラム単位を転送する先とプロセスを転送する先を一つに決定するために使われている．OSは，**プログラム単位の再配置という手段を介して，制御された方式でプロセスの負荷分散を達成する**のである．この方式の意義は，プロセスの動的負荷分散の問題がプログラム単位のパス上への分配の問題に還元されたところにある．プロセスの実行管理の問題を，メタレベルのプログラム分割管理の問題に置き換え得たことになっている．

初期プロセスが投入された直後では，プログラムのすべてはパスの起点のプロセッサ上に転送される．仕事の実行が開始された直後からは，プログラムはプログラム単位でパスに沿って流される．あるプロセッサの負荷を減らしたいとき，そのプロセッサのローカルメモリに載っているプログラム単位を，パスに沿って次のプロセッサに転送する．あるプロセスの実行に必要なプログラム単位を移動させてしまえば，そのプロセスとそれから将来生まれるプロセスは，上記の動的負荷分散の仕組みによりパスに沿って移動させられる．

プログラム単位のパス上での配置は，プロセスが実行されようとしているプロセッサから，パスの最先端までの部分パス上に，必要なプログラムのコピーが最低一つある条件が守られれば，いろいろ工夫して行ってもよい．（ただしパスの先に必要なプログラムが必ずしも保証されなければ，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ろう．）例えば，プログラムのコピーをいくつも作りパス上に分散格納すれば，タスクは広い範囲で実行されることになる．逆に，コピーを減らし限定配置すれば，タスクは限定された範囲でのみ処理されることになる．また，コピーが存在する領域をパス上で移動して行けば，タスクの実行範囲も連れて移動して行く．

6 評価

何を持って評価が得られたとするかは答えるに難しい問題である．とはいえ，枠組みとしての評価と，枠組みを実現する際に行った決定の評価は峻別すべきと考える．ここでは枠組みとしての評価を，スケジューリング法の観点から分類してみせることによって行い，枠組み実現のために行った決定の評価はプロトタイプ作成によって行う．

6.1 枠組みの分類学的評価

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．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．

6.1.1 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)．

6.1.2 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）．

6.2 枠組み実現のプロトタイプによる評価

並列論理型言語 GHC をプロトタイプ記述言語として用いた．並列論理型言語の場合には，プログラムは節（クローズ）の集まりとなり，ゴールがプロセスに相当する．性能管理を行う OS の枠組みは，特に並列論理型言語の場合には魅力的なシナリオである [7]．なぜなら，論理型言語は推論を主体とした問題解決に向いていると言われて来たからである．

プロトタイプは GHC 実験システム ExReps [5] 上に作成した．ExReps は，PSI-II の GHC システムの上に実現されている，ソフトウェアによる汎用のマルチプロセッサシミュレータである．仮想的につくり出された GHC プロセッサを任意のネットワークでつないだ実行環境を提供する．ゴールをプロセッサへ割り付けるための注記である，プラグマ [4] をサポートしている．作成したプロトタイプの規模は 200 節規模程度のものである．しかし，問題解決の部分のアルゴリズムは未熟であり，まだまだ改訂の余地がある．

6.2.1 マルチタスクミックス

作成したプロトタイプでは先に述べた基本アルゴリズムをそのまま使っている．しらみつぶし法であり，実行履歴は良くなかったとわかったものを排除するためにのみ使っている．そこで現時点で評価するには時期尚早である．

かわりに，問題解決の手法を考慮した，より高度なアルゴリズムのアウトラインを示そうと思う．最初にパスの決定について議論する．パスの形を固定（例えば直線のみに）してしまえば，パスの決定はパラメータ値の決定の問題になる．ジョブごとに性能を左右する主要な（dominant）パラメータ項があるとき，それを過去の実行履歴から同定できれば，有望なパスを優先的に生成しやすくなる．つまり，性能にあまり関係がないパラメータを主に変更することで，性能を保ちつつパスのヴァリエーションを手に入れることができる．

次にプログラム分割について議論する．呼び合う関係の節同志はなるべく同じか，引き続くプログラム単位に入ることが望ましいと思える．仮想記憶におけるワーキングセットのような概念が，ここでもまた有効であると期待できる．このようなまとまりを，ヒューリスティックスを使って見つけだす試みが報告されている [6]．

しかしここで目指すのは，過去の実行履歴に基づいて理想的な分割を自発的に見つけてくれるような仕組みの導入である．遺伝アルゴリズム（genetic algorithm）の手法を使って，良い性能を示したプログラム分割リスト同志を掛け合わせ，有望なプログラム分割リスト候補を求める方法はどうだろうか [2]．プログラム分割リストには全体を見渡すような構造がないので，性能が良かったプログラム分割リスト同志を交差した結果のプログラム分割リストは，ふたたび良い性能を示す可能性が高いと高い確率を持って期待できる．

6.2.2 マルチプロセッシング

プロトタイプは以下に述べるマルチプロセッシングの仕組みを備えている．ゴールはパスの上でのみ実行され，ゴールの雲はパス上を流れて行く．

- プロセッサのローカルメモリに必要な節がなく，ゴールのリダクションが失敗した（fail）したときには，ゴールをパスに沿って流す．リダクションが成功（success）や中断（suspend）のときには転送しない．
- 仕事を担当しているプロセッサの負荷が処理能力を越え始めると，そのプロセッサはローカルメモリにあるプログラムキューの先頭からプログラム単位（実際には節の集まり）を取り出してパスに沿って流す．副作用でリダクションに失敗したゴールが転送されるようになるので，プロセッサの負荷が下がると期待できる．
- 逆にプロセッサの負荷がかなり落ちて来たときは，プロセッサはパスの上流のプロセッサへプログラム単位を流してくれるように依頼する．依頼されたプロセッサは，プログラム単位を一つパスに沿って流す．結果的に負荷が落ちたプロセッサにゴールが再び集まることが期待できる．

以上のようにマルチプロセッシングは自律的動作する．この自律的動作の意図は，一つの仕事の中での負荷量の変動を吸収することと，複数の仕事がパス上で衝突したときに衝突を回避する，の二つである．マルチプロセッシングの最初の仕組みによって，負荷が高まったときにはゴールの雲は自律的に広がることになる．二番目の仕組みによって，散らばり過ぎたプログラム単位を集めて再び負荷を少数のゴールへ集中させることができる．

この様子を ExReps 上で見たのが図 3である．左の図は二つのタスクを並行するパスで走らせた場合であり，右の図は交わるパスで走らせた場合である．右の図のパスの交点のプロセッサでは，二つのタスクの負荷が重なり，ゴールの雲の移動スピードが速くなっていることがわかる．図中，各プロセッサに表示されている数字は，最初のゴールが到達してからプロセッサの手に負えなくなるまでのゴールが増大するまでの時間である．

7 おわりに

マルチタスクミックス OS は資源管理の問題を二つの部分，マルチタスクミックスとマルチプロセッシングに分けて解決しようとする．

マルチタスクミックスの目的は，ジョブに準最適な分散戦略を見つけだすことにある．すなわち，問題解決の様相を持っている．言い換えると「考える OS」がキャッチフレーズである．準最適な分散戦略を与えるための我々の方策は，性能管理という新しい次元を開くことになった．

マルチプロセッシングの目的は自律的に，仕事の負荷が衝突した場合や，一つの仕事の中で負荷が高まったときに負荷の自動分散を行うことと，逆に再び負荷が低くなったときに負荷の集中を再び果たすことにある．マルチプロセッシング方式は，同時に二つの技術的課題を解いていることに注意して欲しい．プロセスの動的負荷分散の方式と，プログラムの転送方式に同時に解を与えている．キーになるアイデアは，プロセスの実行に必要なプログラムを予め転送しておくという従来の考え方から，プロセスの実行に必要なプログラムがなければプログラムがあるところまでプロセスを転送する，と発想を逆転した所にある．

もともとの狙いは，汎用並列マシンが真に役立つものとするために対話処理の技術を開発することにあった．すなわち，マルチタスクミックス OS の最終的な目的は，汎用並列マシン上の対話処理支援である．この目的が果たされているか否かを確かめるには，いっそう注意深い評価が必要である．

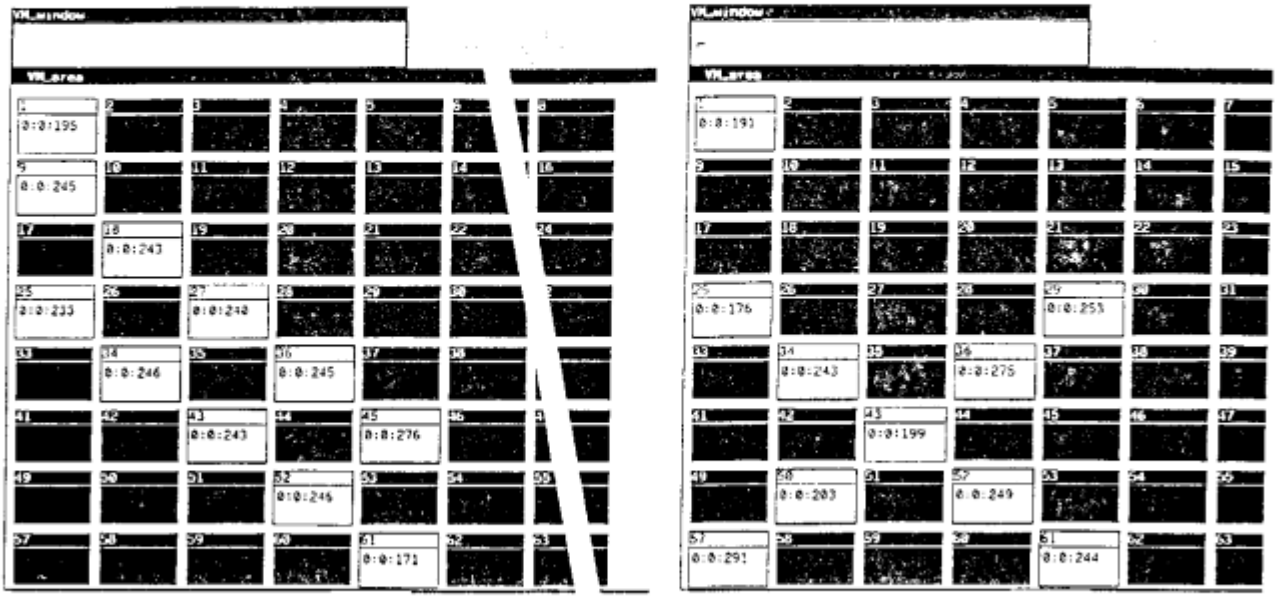

図 3: バス上での負荷衝突回避

謝辞

富士通国際情報社会科学研究所の諸氏，特に同じ研究グループに属す，田中二郎，村上昌己，香野博靖，前田宗則の各氏感謝いたします．また GHC による性能管理 OS プロトタイプの作成には富士通 SSL の太田祐紀子氏，同じ研究グループの前田宗則氏の助力を得ました．最後に，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ています．

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] R. Michalski, et al.: *Machine Learning Vol. II*, Tioga Publishing Company (1986).

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

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

[5] J. Tanaka, Y. Ohta, and F. Matono.: *An Overview of ExReps System*, Fujitsu Scientific & Technical Journal, Vol. 26, No. 1 (to appear) (1990).

[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).