TM-0987

並列システムの進歩:
バッチ環境からマルチタスキング環境へ

神田　陽治、前田　宗則（富士通）

December, 1990

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

並列システムの進歩: バッチ環境からマルチタスキング環境へ

神田 陽治, 前田 宗則
富士通 (株) 国際情報社会科学研究所
{kohda,m-maeda}@iias.fujitsu.co.jp

マルチタスキング処理を行う並列システムについて述べる. 問題ごとにチューニングを施して高い性能を得るバッチ環境を越えて, 広い応用レンジでもある程度の性能が得られるマルチタスキング環境への進化を狙う. マルチタスキング技術の中核は, 性能管理と動的負荷分散である. ここでは性能管理と動的負荷分散の方法を説明し, 作成中のプロトタイプについて報告する.

1 はじめに

汎用マイクロプロセッサを用いて, 比較的簡単に複数 PE(Processing Element) を多数持つ並列計算機ハードウェアを組み立てられる時代になったのにかかわらず, 広く一般に使われるに至っていない. ずいぶんと普及したワークステーションのように, 一人に一台ずつ, 高並列ワークステーションが行き渡らないのはどうしてだろうか.

誰もが並列計算機には「高い性能」を期待する. しかしながら現在の並列計算機がその「高い性能」を発揮できるのは, 予め十全に準備がなされた場合に限るようである. すなわち, 並列性がある問題を選び取り, 並列計算機の特徴を捉えてプログラミングを工夫し, さらに実際に例題を何回も走らせてチューニングを行なった場合に限るようである. このような「箱庭的環境」で「意図された例題」を走らせて, 台数効果 (一つの例題を使用可能な PE 数を変えながら試行したとき, PE の台数が少ないうちは台数になるべく比例して性能が向上する効果) を見ることは, 残念ながらベンチマークテストの域を出ない. 我々は, バッチ処理的だった使用形態を, 逐次計算機の歴史に倣って, TSS あるいはワークステーション的な使用形態へと進化させるべきと考える. くだけて言えば, 「お手軽な」並列システムが欲しい. そのために, 並列計算機とその専用言語の作製という枠を破って, オペレーティングシステムを含めた, 並列システムへの研究へと進まねばならない. 特に, マルチタスキング処理を行う並列システムへの方向である.

2 性能を議論する

本章では, プログラムの実行「性能」について考察する. 性能の議論は一筋縄では行かない. 理想化された環境なら明快な議論もできようが, 出てくる結果は現実に適用できない弱い結果となるし, 具体的な環境を設定して議論すれば, 一般性がない結果に過ぎないと批判されよう. 適切な切り口を見い出すことによって, 性能を研究対象にするのが我々の最終的な目標である.

2.1 性能を改善する

ユーザから見たプログラムのマルチタスキング性能をあらわす指標として, スループット (throughput) とターンアラウンドタイム (turn around time) が使われる. スループットは, 単位時間当たりに (平均的に) こなせる作業の量で測られる. ターンアラウンドタイムは, 一つの仕事が開始されてから終了するまでの (平均の) 作業時間で測られる. スループットを上げるには, 仕事ごとの特性を (例えば統計的に) 滑らかに捉えて, 実行環境を全体的観点から

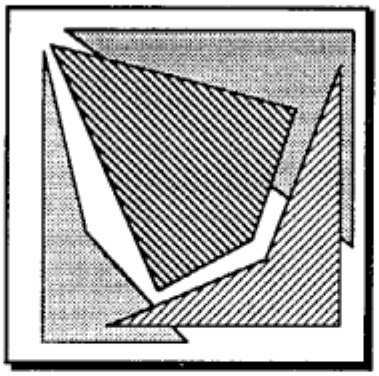

図 1: 性能改善の原理

調節する必要があり，一方，ターンアラウンドタイムを短くするには，仕事ごとの特性を詳細に捉えて，実行環境を特定の仕事に特化するように調節する必要がある．実行環境を調節するとは，具体的には仕事のスケジューリング戦略を適切に決めることである．

逐次システムの場合には，一つしかない実行環境を複数の仕事に合わせて同時に特化することはできない芸当なので，スループットとターンアラウンドタイムの二つを同時に改善することは難しい．ところが並列システムはたくさんの PE を持ち，たくさんの実行環境があるのだから，ターンアラウンドタイムとスループットを二つともに改善できる可能性が出てくる．各 PE はすべての仕事ではなく，割り付けられる仕事の分だけに合わせて調節 (チューニング) されるだけで済むからである．ここまでの考察で我々は，次の可能性を得た．

性能改善の原理

> 多数の PE を持つ並列システムにおいて，仕事ごとの作業の特性を勘案して，仕事どうしをうまく組み合わせて適切な PE に割りつけることによって，性能を改善できる．

調節機能とはスケジューリング戦略を作業の特性に合わせることであったが，並列計算機の場合には調節されるべきスケジューリング戦略は，動的負荷分散戦略を含む．負荷分散戦略とは，作業の単位をプロセスと呼ぶとき，実行しているプロセスを自分で引き続き実行するのか，それとも他の PE へ回送して (forwarding) 実行してもらうかの選択である．より詳しくは，いつプロセスを回送すべきかと，どこへプロセスを回送すべきかの二つの判断を行うことである．注意すべきは，これらの判断は，PE の状態と作業の大変さの兼ね合いで決定されねばならない．どちらか一方の単独の判断のみで負荷分散するのでは良い結果を生まないだろう．図 1は，多数の PE から成る計算平面上に，複数の仕事をうまく組み合わせることで，負荷が均等化し性能が改善されている様子を模式的に表したものである．

2.2 性能を管理する

次に，仕事の特性をどう捉えるかを議論する．消極的な (我々が採らない) 方法は，ジョブクラスという形での自己申告制を採用するやり方である．ジョブクラスは消費するだろう資源量を勘案して，ユーザが決めなければならない．もっと積極的に解決するために，これまでの実行性能を記録し，記録を根拠に将来の実行性能を予測する方式を採る．このやり方がとれるには，仕事は反復して行われるものだ，という前提がある．我々はこの前提には一理あると考えている．そもそも一回しか実行されないプログラムであったなら，多少実行性能が悪くとも答えが得られれば良い．チューニングの必要はない．何回も反復して実行されるプログラムの場合にこそ，高い性能が欲しいのである．すなわち，

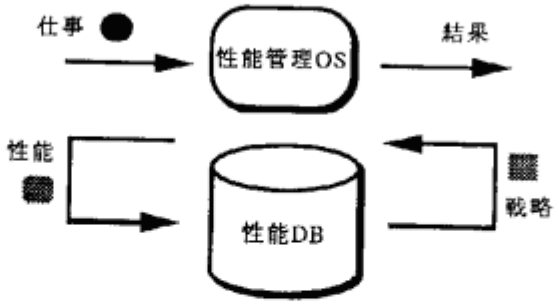

図 2: 性能管理の原理

性能管理の原理

> 仕事が繰り返し実行されるという前提のもとに，高い性能を引き出す負荷分散戦略は，過去の実行履歴から割り出せる．

図2に，性能管理の枠組みを示す．性能管理 OS は，投入される仕事を並列に実行し，求まったそばから答えを返す．その際，性能管理 OS は性能データベースを参照し，負荷分散戦略を決定して仕事の実行を開始する．仕事が無事終了したら，負荷分散戦略のもとでの実行性能を，性能データベースに登録する．このようなフィードバックループによって，苦労することなくほどほどの並列性が引き出して行ける実行環境が実現できるだろう，というのが我々の目論見である．

ここでの興味は，例題を固定し問題固有の性質を使って，どこまでチューニングできるかを示すことにはない．並列性を持った問題のプログラムならば，どんなものにでも一応通ずるような性能管理の枠組みが欲しいのである．それでは性能向上の根拠を，問題固有の性質に求めないならば別の何に求めるのか．我々は，次に述べる動的負荷分散機構に根拠を求める．

2.3 負荷を分散する

ここでは，負荷分散の方法について考察する．PE でプロセスを実行しようと思えば，その PE のメモリに必要なプログラム単位がなければならない．逐次計算機の場合には PE が一つしかないので，プログラムを PE へ持ってくる以外の選択はない．仮想記憶の実現がこれに該当する．

並列計算機では，必要なプログラム単位がなかったときに二つの方式選択がある．第一の方式は，必要なプログラム単位をどこからか持ってくることであり，第二の方式は，必要なプログラム単位を持つ PE にプロセスを回送することである．我々は第二の方式 — 必要なプログラム単位を持つ PE にプロセスを回送する — を採用する．この方式の利点は，プログラムの移動に連れて，プロセスが自然と分散してくれることである．図3が，この様子を表している．図における中央の PE が，プロセスを実行しようとして必要なプログラム単位が無かったことがわかったとする．もし，必要なプログラム単位を左右の (どちらかの)PE から中の PE へ動かせば，プロセスは中央の PE にとどまったままとなる．しかし，中央のプロセスを必要なプログラム単位を既に持っている PE へ動かせば，プロセスは自然と分散する．

負荷分散の原理

> プロセスの負荷分散の問題は，プログラム単位の再配置問題へと帰着できる．

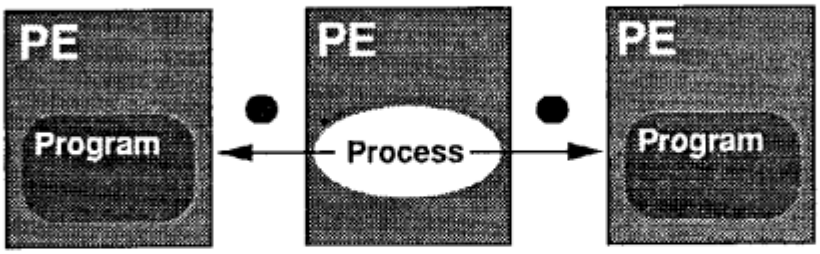

図 3: 負荷分散の原理

スケジューリングは，プログラム単位の再配置とプロセスの回送という，いわば二階建てのスケジューリングとなり，オペレーティングシステムはプログラム単位の積極的再配置を介して，プロセスの負荷分散に関与できる．そして最適再配置戦略を，実行の繰り返しの記録から選び出して行く仕組みを備えることが，性能向上の根拠である．

2.4 性能は科学できるか

三つの問題提起から始める．最初に，性能の記録を採ると一口で言ってきたが，そもそも記録すべき「性能値」には何を使えばよいのだろうか．第二に，性能を再現するために整えるべき条件 — 性能因子と呼ぼう — には何を採用したら良いのだろうか．第三に，扱うのに便利な性質を性能因子は備えているだろうか．特に，「構成的」な性質を備えているだろうか．残念ながら，我々はこれらの問題に満足行く答えを見い出してはいない．コメントを与えるにとどまる．

性能を何で測るか?

誰でも思い付くのが，実行時間を測ることであろう．実行時間が短いほど，高い性能である．実行時間は実行開始から，望む結果が求まるまでを測る．結果を出すものの止まらないプログラムもあるからである．

もう一つの方法は，論理マシン内の各 PE がどのくらい実質的に働いたか (稼働率) を測ることであろう．これは間接的に負荷分散 — 我々の場合にはプログラム単位の再配置 — がうまく実施されたかどうかを測っていることになる．仕事の量に比べて PE が十分に性能を持っていれば，むしろ再配置せずに実行すべきであるし，仕事の量が膨大ならば PE の負荷に見合う位までにプロセスを分散させるよう，プログラムの再配置を積極的に進めるべきである．

実行時間は，プロセッシング時間という資源に関する性能値であり，PE の稼働率は，メモリ空間という資源に関連した性能値であると言える．総合的な性能値は，これらを勘案して決定されるはずである．

性能を左右する因子は何か?

プログラム単位の再配置戦略が，性能因子の候補である．仕事の遂行するために必要なプログラムは複数のセグメントに分割され，オペレーティングシステムによって再配置の単位とされる．プログラム単位は必ずどれかのセグメントに入れられるが，同じプログラム単位のコピーが複数のセグメントに重複して入っていても構わない．

どのようにプログラムをセグメントに分割するかが，負荷分散戦略を特徴付ける．仮想記憶において，関連するプログラム単位がうまくページに納まっていると，参照の局所性が向上しページフォールトが改善される．同様に，うまくプログラム単位をセグメントに納めれば，やはり参照の局所性が向上して，実行性能が向上することが期待される．

複雑で巧妙な戦略が良いというものでもない．ここで採るべきアプローチは，できるだけ簡単であるが効果が著しい戦略を発見することであろう (参照 [5])．簡単なアプローチは実現のコストも実行時のオーバーヘッドも小さい有

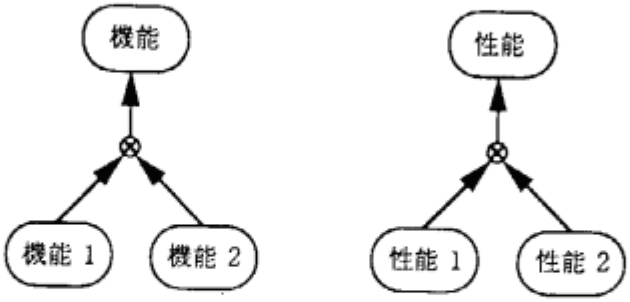

図 4: 機能設計と性能設計: 性能も機能と同様に構成的に設計できるか?

利さがある.

性能は「構成的」に扱えるか?

プログラムの性能について困るのは, それが「構成的」でない点である. プログラムの機能は, 全体仕様を部品仕様に分解し, 部品仕様を満たす部品を作って組み上げて全体を作りうるという点で, 「構成的」である. しかし, 個々の部品を単独で動かしたときに高い性能を示したからといっても, それを単純に組み合わせたプログラムが, 全体として良い性能を保てるかどうかはかなり怪しい. もし仮に, 性能を左右する性能因子のうちで「構成的」な性能因子を発見出来れば, 「性能設計」が可能になるだろう. 性能が落ちない合成の仕方というだけでも使い途がある. はたして「構成的」な性能因子は存在するだろうか?

例えば, 我々が期待する性能因子である, プログラムのセグメント分割はどうだろうか. 部品用の高い性能を達成できるセグメント分割が与えられたとしたとき, 合成品用のセグメント分割を「構成的」に作り出すことができたら嬉しい.

3 性能管理 OS の機能

プロセスの負荷分散問題はプログラム単位の再配置の問題なので, 負荷分散を達成するには, いつプログラム単位を再配置すべきかと, どこへ再配置すべきかの二つの判断を具体的にしなければならない.

まず, どこにプログラム単位を再配置するかに答えを与える. プロセスは, プログラム単位の再配置先を追うよう回送されるのが好ましい. いちいちどこにプログラム単位が再配置されたのかを探していたのでは, 手間がかかりすぎる. これを達成するには, 再配置先と回送先が常に一致するように, PE に順番が付いていればよい. この目的のために, 仕事ごとに, PE の部分集合から成る「論理マシン」を与える. 論理マシンを構成する PE には順があり, この順番に従ってプログラム単位は再配置され, 後を追ってプロセスは回送される.

次に, いつプログラム単位を回送するかに答えを与える. PE の持つ能力と比較して, 実行待ちのプロセスが PE にたまりすぎたと判定されたときに (PE の負荷が高くなり過ぎたときに), オペレーティングシステムはプログラム単位の一つを選んで, 論理マシンの次の PE へ転送する. このとき次の PE が混んでいるかどうかは調べないので, 空いている PE が見つかるまで, プログラム単位の転送が玉突きのように起こる可能性がある. 転送されるプログラム単位の選択の基準として, 流れて来た順番に外へ流す FIFO 順であるとか, もっとも最近参照されたものを外へ流す MRU 順 (Most Recently Used) などが良さそうである. さて, 性能管理オペレーティングシステムの機能の要点を簡単に説明する. 文献 [1] に詳しい.

3.1 性能を記録する

性能と性能因子がどうにか決まったとしよう．性能因子の具体値を決めることで，実行戦略が一つ具体的に決まる．性能データベースには，実行戦略と，そのもとで発揮された性能値をペアで記録していく．実行を行うときには，もっとも高い性能値をもたらしうる実行戦略を検索して，仕事の実行に適用する．

3.2 論理マシンを貸し出す

本実行でも実験実行でも，プログラムの再配置もプロセスの回送も，論理マシンに沿って行われる．仕事の実行に先立って，空いている論理マシンが決定され，仕事に割当られる．そもそも並列システムに，つぎつぎと仕事が投入されるという状況において，つねに特定のPEから投入する戦略よりも，ランダムにPEを選んで投入する戦略の方が，良い結果を生むと考えてよいだろう．さらに，ランダムに投入PEを選ぶよりも，空いているPEに投入した方がよい結果を生むに違いない．

空いているPEを正確に同定することには，並列計算機全体を見渡す必要があるので，コストがかかる．妥協案であるが，予め論理マシンを何通りか注意深く設計しておいて，これを仕事に貸し出すというやり方が考えられる．貸し出すときには，その時点で貸出中の論理マシンすべてとなるべく衝突しないような論理マシンを貸し出すようにする．この方式は，仕事の過剰投入(overcommit)を防ぐことにもつながる．

3.3 仕事を実行つつ，もっと良い戦略を探す

仕事が投入されると，まず，並列システムから空いている論理マシンが貸し出される．次に性能の記録から，最善と推測される実行戦略を割り出して，本実行を行う．加えて並列計算機にはたくさんのPEがあるので，貸し出せる論理マシンが残っている場合には，別の実行戦略を試してみる目的で，本実行とは並列に実験実行を行うこともできる．次の仕事が投入されたのに，空いている論理マシンがなく実行できないときには，実験実行があれば中断して余力を作り出すことができる．本実行と実験実行のうち，最初に答えが帰った方の答えを返す．本実行と(最後まで実行できたときには)実験実行の性能値の記録は，使用された実行戦略とともに性能データベースに追加記録されて，それ以降の仕事の実行に役立てられる．

4 シミュレータプロトタイプの作成

我々は，開発環境としてUnix上のKL1[3]言語処理系PDSSを用い，性能管理OSの試作を進めている．性能管理OS実現の全貌を紹介することは，別の機会に譲るとして，ここでは，その核を構成する負荷分散シミュレータの簡単な紹介を行なうことにする．

負荷分散シミュレータは，ユーザプログラム，ユーザプログラム実行器であるメタインタプリタ，及びメタインタプリタの挙動を観測し視覚のためのデータを取り出すメタメタシステムの3つを，手作業で部分評価(partial evaluation)し融合することで構成されている．最初，これらの三システムは融合されることなく，インタプリタの3段重ねで構成されていたが，予備実験の結果，実行効率が十分ではなく，実行速度も極めて遅いため，実用的な規模のタスクを投入して実験するのには不十分であった．そこで，部分評価を施し最適化を行なった．

ユーザプログラムの部分評価前後を，appendプログラムを例として取り上げてみる（下記参照）．現在手作業で行なっている，ユーザプログラムの部分評価は，専用のコンパイラを提供することで自動化することが可能である．部分評価後のプログラムは，オリジナルのプログラムに対して，実質上8つの引数を新たに加えたものになっている．（シンタクティックには，1つの引数しかないように見えるが，これはインプリメンテーション上の都合であり，本質ではない．）まず，この新しく付加された引数の説明を以下に示そう．ここで，変数の後に裂記された記号↓は、この引数が入力引数であること，また記号↑は，出力引数であることを陽に示すために導入された補助記号である．

(1)BB↓, (2)EB↑ これらは，ユーザプログラムの終了判定を行なうために付加された引数である．終了判定の技法としては，一般的なショートサーキットテクニックを用いている．BB に定数アトムを単一化し，ゴール実行を開始する．変数 EB に，始めに与えた定数アトムが具体化すること（ショートサーキット）は，全サブゴールの実行が終了したことを意味する．この方式は，終了判定処理のオーバーヘッドが極めて小さい．

(3)Can↓ プロセッサネットワーク上のある PE でユーザゴールをリダクションすることは，ユーザゴールにプログラムセグメント集合を与えて，自律的にリダクションすることに変換されている．このようなリダクション方式では，PE においてプログラムの分割と転送が実行された場合に，もし，データ入力待ちでサスペンドしているユーザゴールが存在するならば，これらのゴールを一旦回収して，新しいプログラム集合を与えて，再度実行する機構が必要になる．この第三引数 Can は，PE に接続しており，プログラム転送が行なわれた場合には，PE は，変数 Can を cancel に具体化することによって，このゴールのサスペンド状態を強制的に解消し，(4)Body 引数を通して回収することを行なう．

(4)Body↑ 回収されたユーザゴールや，そのユーザゴールを 1 回リダクションすることによって得られたサブゴールがリスト形式で具体化する．この引数は，PE に結合しており，PE は，こうして得られたゴールを繰り返し実行する．

(5)Key↓ PDSS は，実行時にプログラムセグメントの集合を付加したり，削除する機能を提供していない．そこで，負荷分散シミュレータがそういった機能を提供することになる．これを実現する素朴な方式は，メタインタプリタが，プログラムセグメントのいわゆるフローズンフォームを保持して，必要に応じてその一部をメルトして用いるものである．本シミュレータは，実行効率の問題から，これを採用していない．本方式では，プログラムセグメントの集合は通常どおり予め与えられており，番号づけされている．ユーザゴールは，プログラムセグメント番号の集合を引数に持つことで，選択されるべきセグメントの条件を与えている．より詳細には，現方式ではセグメントブロックの総数を 32 個に制限することで，プログラム集合を 32 ビット整数で表し，それを変数 Key に格納している．これによって，単純で高速な整数のビット演算によって，プログラムセグメント集合を分割したり，付加すること，また選択されるべきセグメントのチェックが容易に実行される．現方式のセグメント総数の制限は，本質ではなく，整数の配列を利用すれば，セグメント総数はいくらでも大きくすることが可能になる．

(6)GN↑ これは，(生成されたサブゴール数－1) であり，ゴールキューの増加分である．この値を PE のゴールキュー長に加えることで，現在のゴールキュー長が求められる．

(7)Done↑ あるプログラムセグメント（ここでは 1 ホーン節とする）がただちにコミット可能で，かつそのプログラムセグメントが存在しない場合に，そのゴールは，パス上の論理的に隣のプロセッサに投げられる．これは，ゴールを変数 Done から回収することで実現される．また，直ちにコミット可能で，プログラムセグメントが存在する場合は，実際コミットし，変数 Done には空リストが具体化する．

(8)Exe↑ ユーザゴールが，1 回リダクションされたとき，変数 Exe は，cancel,commited,thrown のいずれかに具体化される．cancel は，PE にプログラム分割が生じて，このゴールが回収された場合である．また，commited は，プログラムセグメントの 1 つが選択された場合であり，thrown は，対応するプログラムセグメントが存在しないので，他のプロセッサにゴールが投げられた場合である．これらの情報はプロセッサのスケジューリングにおいて利用される．

また，変換後のプログラムは，元の手続き（同じ述語名を持ち，引数の数も等しいホーン節の集合を“手続き”という．この場合 append/3 を構成している 2 つのホーン節が“手続き”である）にキャンセル処理とフォワーディング処理のための二つのホーン節を，前後に加えられたものになっている．他のホーン節は，オリジナルのホーン節に 1 対 1 対応している．

append/3 オリジナルプログラム

```
append([E|X],Y,Z):- true | Z = [E|Z1], append(X,Y,Z1).          %第1節
append([],Y,Z):- true | Z = Y.                                  %第2節
```

変換後のappend/1 プログラム

```
append({X,Y,Z,BB,EB,cancel,Body,Key,GN,Done,Exe}):- true |     %Cancel処理
        Exe = cancel, Done = [], GN = 0,
        Body = [goal({users3,append,1},
                     {{X,Y,Z,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                     {Can1,Body1,Key1,GN1,Done1,Exe1})].
alternatively.
append({X,Y,Z,BB,EB,Can,Body,Key,GN,Done,Exe}):- X = [E|X1] | %第1節
        ( Can = cancel -> Exe = cancel, Done = [], GN = 0,
          Body = [goal({users3,append,1},
                       {{X,Y,Z,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                       {Can1,Body1,Key1,GN1,Done1,Exe1})];
          alternatively;
          Key /\ 16#"00000001" =\= 0 -> Exe = commited, Done = [], GN = 0,
          Z = [E|Z1],
          Body = [goal({users3,append,1},
                       {{X1,Y,Z1,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                       {Can1,Body1,Key1,GN1,Done1,Exe1})];
          Key /\ 16#"00000001" =:= 0 -> Exe = thrown,
          Done = [forward_goal(goal({users3,append,1},
                  {{X,Y,Z,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                  {Can1,Body1,Key1,GN1,Done1,Exe1}))],
         GN = -1, Body = [] ).
append({X,Y,Z,BB,EB,Can,Body,Key,GN,Done,Exe}):- X = [] |       %第2節
        ( Can = cancel -> Exe = cancel, Done = [], GN = 0,
          Body = [goal({users3,append,1},
                       {{X,Y,Z,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                       {Can1,Body1,Key1,GN1,Done1,Exe1})];
          alternatively;
          Key /\ 16#"00000002" =\= 0 -> Exe = commited, Done = [], GN = -1,
          Z = Y, BB = EB, Body = [];
          Key /\ 16#"00000002" =:= 0 -> Exe = thrown,
          Done = [forward_goal(goal({users3,append,1},
                  {{X,Y,Z,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                  {Can1,Body1,Key1,GN1,Done1,Exe1}))],
          GN = -1, Body = [] ).
append({X,Y,Z,BB,EB,Can,Body,Key,GN,Done,Exe}):- Key /\ 16#"00000003" =:= 0 |
        Exe = thrown,  GN = -1, Body = [],                      %forwarding処理
        Done = [forward_goal(goal({users3,append,1},
                {{X,Y,Z,BB,EB,Can1,Body1,Key1,GN1,Done1,Exe1}},
                {Can1,Body1,Key1,GN1,Done1,Exe1}))].
```

ここで，視点をユーザプログラムレベルから，それを実行しているPEに移してみる．今まで単にPEと呼んでいた実行器は，1つの仕事に対して貸し出された仮想的なマシンであり，論理PEというべきものである．これら論理PEは，現実にあるPE（物理PE）に割り当てられることになるが，1台の物理PEに複数台の論理PEが割り当

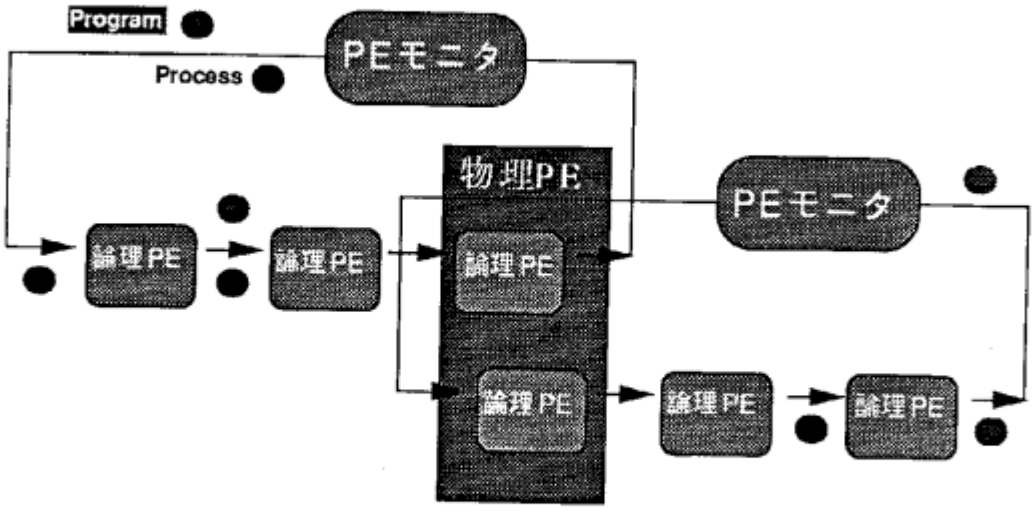

図 5: 物理 PE, PE モニタ, 論理 PE

てられることも許している.

負荷分散シミュレータは, 物理 PE に含まれる, 各論理 PE の負荷を測定し, それを合計することで物理 PE の負荷を求める. 求められた物理 PE の負荷が, 重すぎるか軽すぎる場合にのみそれぞれ, load(high), load(low) というメッセージが, その物理 PE にある, 全ての論理 PE にブロードキャストされる. ここで, "負荷" とは, 論理 PE で消費されたリダクション数によって計られる. 現在, PDSS が管理している唯一の資源がリダクション数であり, それを用いることにした.

load(high) を受信した論理 PE は, 自分の持つプログラムセグメントの 1 つを選び, 論理的に隣接する論理 PE に転送する. また, load(low) を受信した論理 PE は, 隣接する論理 PE にプログラムセグメント要求のメッセージを送信する. セグメント要求を受けた論理 PE は, メッセージ送信元に自分の持つプログラムセグメントの 1 つを同様に転送する.

最後に終了処理について簡単に触れる. 利用者の与えた仕事が終了したとき, 終了促進メッセージが, 仕事単位に与えられた論理 PE の管理者である, PE モニタに送信される. PE モニタは, 終了促進メッセージをある論理 PE に転送する. 論理 PE は, メッセージ受信と同時に隣接する論理 PE に同様に終了促進メッセージを送信すると自らも終了する. 最後の論理 PE は, PE モニタに終了促進メッセージを送信して終了する. こうして終了促進メッセージが一巡した時点で, この仕事に割り付けられた全ての論理 PE が終了する.

これら論理 PE, 物理 PE, PE モニタの構成を図 5に示す.

5 負荷の分散の様子

よく知られた例題 primes を動かしてみた. コンパイルの方法は append と同様である. プログラムの分割は, (簡単のため) すべて節単位とした. 図 6は, 30 までの素数を生成するタスクを投入したときの, プロセスの負荷分散の様子であり, 図 7は同じタスクを二つ同時に, 同じ論理マシンに (わざと) 投入したときの様子である. 図の横軸は時間の (相対) 経過であり, 縦軸はプロセス数である. グラフの輪郭をたどることで, プロセス総数の時間変化を知ることができる. さらに, プロセスの内訳 — 論理マシンのどの PE にどの位のプロセスがあるか — を知ることができる. プロセスの内訳の時間変化を追うことによって, プロセスの負荷の中心が, 番号の小さい PE から番号の大きい PE へと移動しているのが読み取れる. プロセス総数が二倍になっている図 7では, 図 6よりも, 遠方の PE にまで負荷が広がっている様子が示されている.

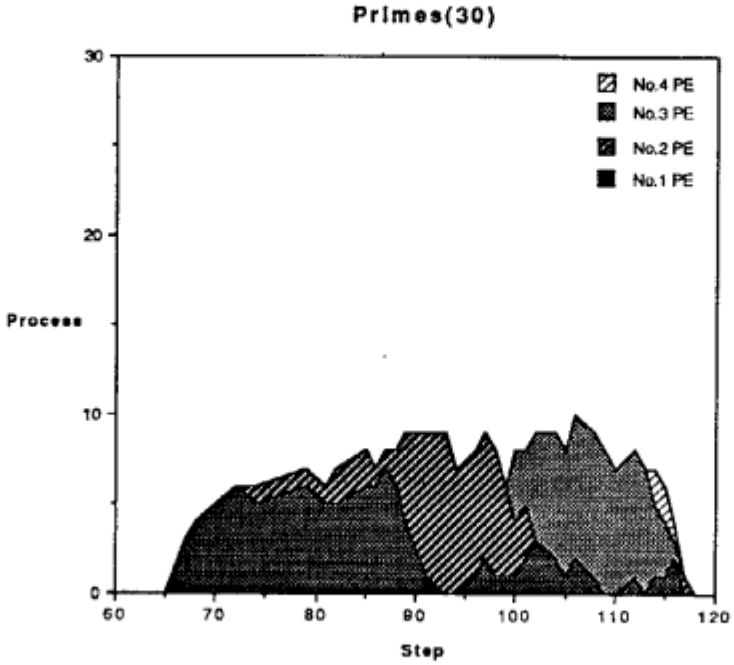

図 6: 論理マシン上の負荷分散 (1)

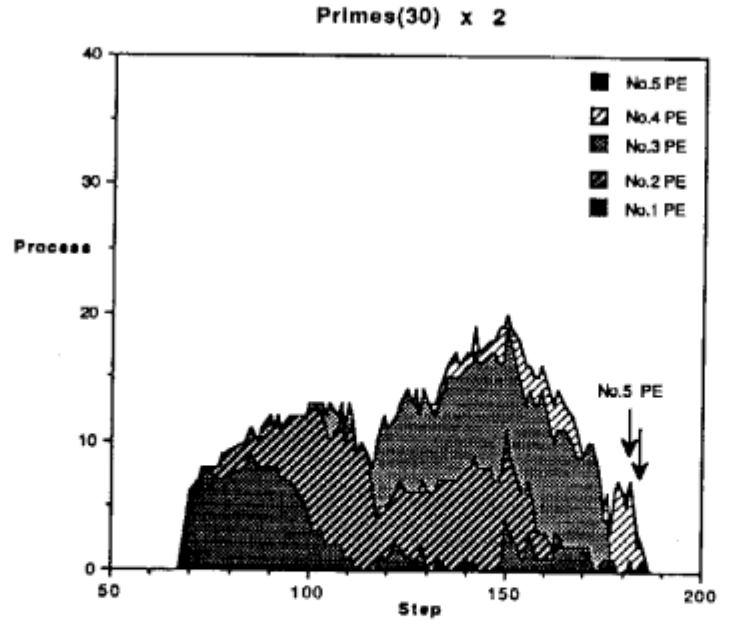

図 7: 論理マシン上の負荷分散 (2) – 二倍のプロセス数

6 おわりに

並列システム用に新しい枠組みを持ったオペレーティングシステムについて，特に性能管理について報告した．性能管理の要点は，第一に性能データベースを設け，本実行とともに実験実行を繰り返しながら，だんだんと最適な分散戦略を割り出すこと，第二にプロセスの負荷分散の問題を，プログラムの再配置の問題に還元させたことである．性能とはなかなか摑みどころのないものである．それがこれまで「機能設計」に見合った「性能設計」の技術が生まれてこなかった原因だと思われる．

急務なものとして，仕事が並列システム上でどのように実行されているのかを，うまく表現する技術が必要である．並列システムのなかでは数多くのことが起こっている．全部を詳しくみても，部分だけを詳しくみても，なかなか全体を摑むことは難しい．見たいのは，おおざっぱだが要を得た，ばらけて言えば「つぼを突いた」表示である．文献 [2] や [4] は，このような目論見を持った研究である．

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] 神田陽治: プログラムとプロセスの再配置による並列マシン用動的負荷分散方式, 情報処理学会論文誌, vol.31, no.12 (1990).

[2] 前田宗則: 色効果を利用した GHC デバッガ, 情報処理研究会報告, vol. 89, no. 71 (1989), pp. 21-29.

[3] 新世代コンピュータ技術開発機構第四研究室: KL1 プログラミング入門編 / 初級編 / 中級編 (1989).

[4] G. Roman and K. C. Cox: A Declarative Approach to Visualizing Concurrent Computations, *IEEE Computer*, (1989), pp. 25-36.

[5] D. L. Eager, E. D. Lazowska, and J. Zahorjan: Adaptive Load Sharing in Homogeneous Distributed Systems, *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 5 (1986), pp. 662-674.