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†

神田 陽治††

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 (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マルチタスク処理技術の確立である．プロセスを実行するプログラムがプロセッシングエレメント (PE) にないとき，プログラムを持つ PE へプロセスを自動的に送りだす．OS は PE 上のプログラムを動的に再配置することを通して，プロセスの負荷分散を間接に制御できることになる．仮想マシンの仕組みにより，プログラムとプロセスを送り出す先を一致させることができる．プログラムの再配置戦略と性能結果の組が，戦略データベースに記録されている．このデータベースを参考に，仕事は常に，その時点でわかる最善の戦略で実行される．このとき並列マシンに余裕があれば同時に，もっと良い戦略を求めて実験的戦略を立案し試験実行を行い，探索的に最善の戦略の改良を試みる．

1. はじめに

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）の新しい枠組みを提案する．本 OS の開発目標は，“手軽な”並列実行を達成する枠組みの開発にある．なお，ここで言う汎用並列マシンは，プロセッシングエレメント（以後 PE）が互いに独立に働き，並列処理の粒度が細かくとれるタイプを想定している．

専用並列マシンやベクトル計算機では特別の手をかけてチューニングしないと高い性能が得られないし，適する応用分野の幅も狭い（**図 1**（a））．専用並列マシンやベクトル計算機は主にバッチ形態で使われる．バッチ形態だからこそ，手間暇かけたチューニングが許されたとも言え，手軽な並列実行環境にはほど遠い．例えば全解探索型の問題はバッチ向きであり，例題で十分チューニングしておいて，実際規模の問題に挑戦するやり方に意味がある．しかしバッチ形態にそぐわない仕事もある．

汎用並列マシンが，専用並列マシンやベクトル計算機に対抗して存在意義があることを示すには，「手軽な並列実行に適性があることを示せればよい」と，我々は考えた．狭く特殊な領域での高いチューニング可能性を犠牲にしても，広い応用の範囲で，プログラムにできるだけ手を入れることなしにそこそこの実行性能が得られるならば，汎用並列マシンの普及に大きな力となる（図 1（b））．対話処理では様々な段階のチューニングレベルのプログラムが，順不同に実行されるわけであるから，手軽な並列実行環境が望まれる．

本論文の構成を述べる．第 2 章では，新しい負荷分散の方法を提案する．第 3 章では，性能管理 OS の枠組みを説明する．第 4 章では，性能管理 OS の実証に向けての課題をまとめる．

2. 新しい負荷分散法

負荷分散法について論ずる．現在の並列マシンが抱える欠点と，それに対する我々の解決策を述べる．

話を，逐次マシンと並列マシンの進化を比較することから始める．逐次マシンは最初，バッチ形態で仕事を一度に一つずつ，投入された順に処理していた．プログラム全体がメモリに転送された後，実行が開始され，それが終了するまで後続の仕事は待たされた．逐次マシンは次に，バッチ形態から TSS (Time Sharing System) 形態に進化した．TSS の技術の導入によって，マルチタスク処理が可能になり，対話処理が普及した．TSS を実現する上で重要な発明は，仮想メモリ技術と仮想プロセッサ技術である．仮想メモリは，メモリの断片化を起こさずに複数の仕事のプログラムを，同時にメモリに載せておくように見せかける技術である．仮想プロセッサは，時分割で一つの実プロセッサを代わり番で仕事に割り当て，一つの仕事に一つのプロセッサがあるように見せかける技術である．

実のところ，現在の並列マシンの実行形態は，バッチ形態の頃の逐次マシンと同じである．基本的には，全並列マシンを構成する PE のメモリに，プログラム全体が転送されねばならない．一つの仕事がマシン全

† A Dynamic Load-Balancing Strategy on Parallel Machines through Remapping of Programs and Processes by YOUJI KOHDA (International Institute for Advanced Study of Social Information Science, FUJITSU LIMITED).

†† 富士通(株)国際情報社会科学研究所

体を専有し，後続の仕事は待たされる．例えば，新世代コンピュータ技術開発機構で汎用並列マシン PIM のために開発中の PIMOS[1] は，この型の並列 OS である．

汎用並列マシンでも手軽な並列実行を可能にするために，TSS の技術導入が望まれる．仮想メモリの技術は，プログラムをどのように各 PE のメモリへ転送するかの「プログラムの転送問題」であり，仮想プロセッサの技術は，たくさんの仕事を複数の PE にどのように配分するかの「プロセスの負荷分散問題」である．

2.1 バッチ形態での負荷分散法の限界

バッチ形態で開発されてきた技術は，TSS 形態では不十分であることを明らかにする．最初にプログラムの転送問題に関しては，現在の並列マシンは，基本的にはすべてのプログラムがメモリに転送されていることが暗黙の前提になっている．すべての PE へプログラムをあらかじめ転送すれば，プロセスがすべての PE へ行き渡らない場合には，無駄にメモリを専有したことになる．メモリを無駄に浪費したばかりでなく，空きメモリがなくなり他のプログラムが格納できず，後続の仕事の実行が不必要に遅らされてしまう．一方，必要とわかった時点で転送する方式では，転送終了を待つ必要が生ずる．

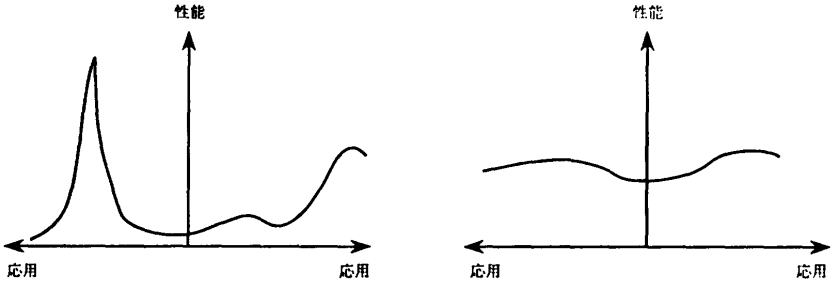
一方，プロセスの負荷分散に関しては，バッチ形態下で，ただ一つの仕事が全マシンの資源を専有する場合に関心があった．そこではチューニングを前提とした静的負荷分散法が主に用いられる．例えば，注記 (annotation) 方式は，プログラムに分散の指示や優先度を直接埋め込む方法である (例えば文献 3), 8))．しかし一つの仕事範囲を越えて，他の仕事との干渉までを正しく予測して注記付けすることは不可能であるから，一つの仕事内での負荷分散に対処できるだけという限界を持つ．プログラムを解析して注記をヒューリスティックに付ける試み[6] もあるが，一つの仕事の枠を越えられない．あるいは仮想マシン層を導入して，注記付プログラムを動的負荷分散実行させる試みもある[5] が，これも一つの仕事の枠を越えての負荷分散まではできない．

まとめれば，プログラムの転送問題は未解決であり，プロセスの負荷分散法はマルチタスク処理には考え直す必要があることがわかった．我々の提案は，プログラムの転送問題に新しい方式を導入することによって，同時にプロセスの負荷分散の問題を解いてしまおうというものである．

2.2 プロセスをプログラムのある所へと送り出す負荷分散法

逐次マシンの仮想メモリ技術では，プロセスがプログラムを要求したとき，必要な箇所を含んだプログラム単位を，ディスクからメモリへ転送する．仮想メモリの本質を，プロセスとプログラム単位を出会わせることにあると考えると，プロセスをプログラム単位のある所へ移動させても良いはずである．逐次マシンでは実プロセッサは一つしかなく，そこにプロセスがある必要があるので，プログラムのあるディスクへプロセスを移すのは意味がないが，並列マシンでは PE が複数個あり，プログラム単位がすでに送られている PE へプロセスを送りつけるやり方も現実味がある．

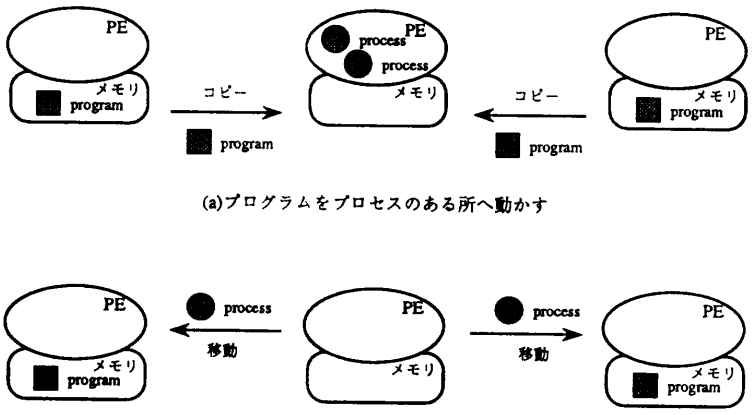
二つの方式を比較してみる．プログラム単位を動かす方式では負荷が集中する結果を招く (**図 2**(a)) が，プロセスを動かす方式には，負荷を自然に分散させる効果があること (図 2(b)) がわかる．このことはプログラム単位を動的に，マシン上の PE に最適配置してやれば，プロセスの負荷分散が自律的に行われることを示唆する．プロセスを分散させ，PE を使い切れば良いというものではない．通信コストを考えれば，なるべく分散させない方が得策とも言える．他方，他の



(a) 従来の並列マシンの応用対性能曲線　(b) 性能管理OSが狙う応用対性能曲線

図 1 性能管理 OS の目標
Fig. 1 Target of OS with performance management capability.

タスクとの衝突が生じた場合には，干渉を避けるために分散した方が良い．必要なのは「周囲の状況に応じて，負荷が高まれば分散し，負荷が低くなれば集中できる」動的負荷分散法である．プロセスが数多く発生し，PE の負荷を分散した方が良い場合には，プログラム単位を分散することによって間接にプロセスの分散を行える．逆にプロセスの数が減少し，PE に余裕が出て来た場合には，プログラム単位を再び集中させることによって間接にプロセスの再集中を行える．



(a)プログラムをプロセスのある所へ動かす

(b)プロセスをプログラムのある所へ動かす

図 2 二つの動的負荷分散法の比較
Fig. 2 Comparison between two dynamic load-balancing schemes.

結局 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できるので，プロセスの負荷分散に OS が直接介入しなくてよくなる．この方式の意義は，プロセスの動的負荷分散の問題が，OS によるプログラム単位の最適配置というメタ問題に還元されたところにある．

2.3 性能管理

性能管理という枠組みを OS に導入する．逐次マシンの TSS 形態においても，ジョブクラスという形での利用者による陽な形での粗い性能管理は存在していた．ジョブクラスを参考に OS は仕事に必要な資源量を見積もることができ，限られた資源を有効に配分する戦略を建てることができた．利用者にとっても，あまり資源を喰わないクラスの仕事は優先されるというメリットがあった．

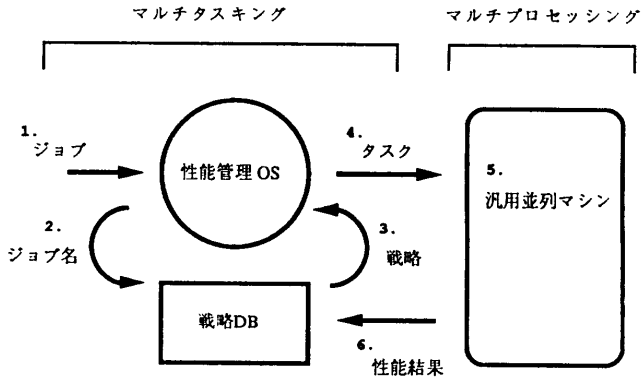


図 3 性能管理 OS の枠組み
Fig. 3 Structure of OS with performance management capability.

新しい負荷分散法では，プロセス負荷分散の戦略とは，プログラムの再配置の戦略にほかならない．性能管理とは，この再配置戦略のデータベースを管理することである．戦略データベースは，その時点までに登録された戦略の良否を示す実験値を参考に，その時点で最善と思われる戦略と，より良い戦略を探すための実験的な戦略を立案し提供する．OS は，その時点で最善の戦略で仕事を実行するとともに，並列マシンに余力がある場合には実験的な戦略でも仕事を実行する．並列マシンが込み合ってきたら，実験的戦略による実行をいつでも廃棄することができるので，実験的実行が本来の実行に影響することは抑えられる．仕事の実行が終わった段階で，各戦略の良否が戦略データベースに追加登録され，次回からの最善の戦略決定に使われるようになる．性能管理 OS はこのような，フィードバックの仕組み，あるいは試行錯誤の仕組みを備える．

機能設計については部品を完成し，それを結合して全体が完成するという階層性が利用できた．性能設計については，そのような階層設計が難しいと思われる．例えば，うまく注記付けができた二つのタスクを結合し，新たなタスクを一つ作ったとしても，子タスク間の連絡がうまくいくとは限らないので，結合されたタスクが良い性能を示すとは限らない．性能管理 OS が備えるような試行錯誤による戦略の決定手順は，性能設計に対する一つの弱い解答を与える．

使う用語を整理しておく．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．ジョブは互いに独立であり，タスクとして順不同にマシンに投入される．**図3**は性能管理を行うOS全体の枠組みを示している．性能管理OSは，二つのレベル，マルチタスキングとマルチプロセッシングからなる．マルチタスキングとは，独立なタスクを並列マシン上に展開し，いかにじょうずに分散実行させるかのスケジューリングの技術を指す．具体的には，ジョブの実行開始時に一度実施され，プログラムの再配置戦略を決定する．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．具体的には，タスクの実行中に実施され，マルチタスキングで与えられたプログラムの再配置戦略に従って，プログラムを動的再配置し，プロセスをプログラムのあるPEへ自動転送することでプロセスの負荷分散を自律的に達成する．

3. 性能管理 OS

性能管理OSの全体構成を明らかにする．最初に性能管理OSが用いる情報構造について述べ，次いで機能の概要をマルチタスキングとマルチプロセッシングの二つのレベルに分けて示す．本章の目的は，性能管理OSの枠組みを実際の実現に依らない形で示すことにある．実証に向けての技術課題についての議論は，次章で行う．

3.1 配置戦略と仮想マシン記述

配置戦略はプログラムをPEにばらまく際のヒントを与えるものである．具体的には，プログラムをグループ分けして切り分けた，分割リストの形を取る．ただし分割といっても，同じ部分が複数のグループに重複して入っていても構わない．分割リストの要素はばらまくときの単位とされるので，プログラム単位と呼ぶことにする．プログラムを送る必要が生じたときには，メモリに格納されたプログラム単位の一つが選ばれ送られる．

配置戦略はプログラムの分割の仕方を示すに過ぎず，具体的にどのPEに割り付けるかまでは指定しない．プログラムが実際に転送されるPEを指定するのに，仮想マシン記述を用いる．タスクを投入するのに，いつも同じPEから投入し，決まったルートでプログラムを流していたのでは負荷の均一化はできない．仮想マシンという名前が暗示するように，仮想マシン記述は実マシンのPEのいくつかを仮想のネットワークで結んだもので，プログラム単位やプロセスは，この仮想的ネットワークに沿って流される．OSは，ジョブの投入時に，比較的空いていると思われるPEを選んで並べて，仮想マシン記述としてジョブに提供する．仮想マシンどうしが重なっていても構わない．マルチプロセッシングの仕組みによって，重なっているPEでの負荷の集中は自律的に回避されるからである．

3.2 マルチタスキングの構造

図3と**図4**にマルチタスキングの構造を模式的に示す．投入されたジョブには配置戦略と仮想マシン記述が与えられる．配置戦略は戦略データベースを探索して，仮想マシン記述は並列マシンの現在の状態を参照して求められる．そして［ジョブ，配置戦略，仮想マシン］の三つ組からなるタスクが起動される．タスクの実行終了後，戦略と性能結果が組で戦略データベースに追加登録され，次回からの戦略立案に利用される．その時点での最善戦略で実行されるタスクは本タスクと呼ばれ，必ず一つ起動される（本タスク起動手順）．仮想マシンの仕組みがあるので，並列マシンに余裕を作るために実験タスクの実行を中断する際には，仮想マシンに沿ってプロセスとプログラムを回収していけば良く，並列マシン全体を見る必要はない．

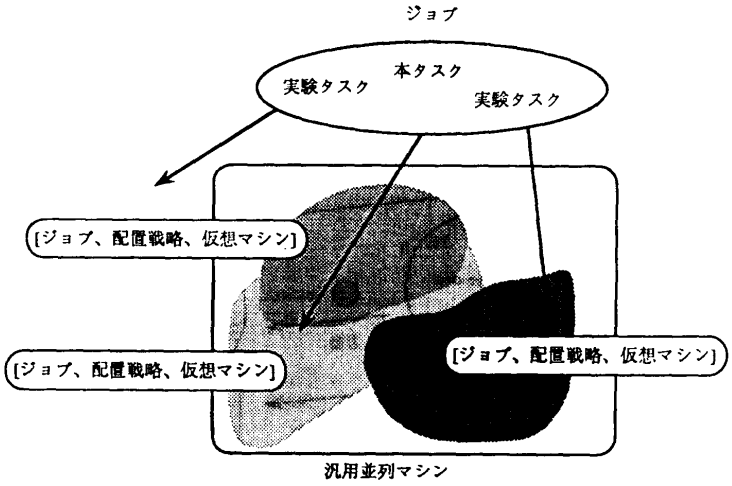


図4 マルチタスキングの構造
Fig. 4 Multi-tasking level.

実験戦略で試される実験タスクは，並列マシンに余裕がある限り，本タスクと並列に，いくつでも起動してよい（実験タスク起動手順）．

- 本タスク起動手順＝
  if 並列マシンに余裕がないならば
  {実験タスクのいくつかを棄てる}
  戦略データベースを参考に，その時点での最善の配置戦略を立案し，並列マシンの現在の状態を参考に，空いている仮想マシンを選び，[ジョブ，配置戦略，仮想マシン]を，本タスクとして起動する．実行終了後，最善戦略の性能結果を戦略データベースに登録する．
- 実験タスク起動手順＝
  if 並列マシンに余裕があるならば
  {戦略データベースを参考に，実験的配置戦略を立案し，並列マシンの現在の状態を参考に，空いている仮想マシンを選び，[ジョブ，配置戦略，仮想マシン]を，実験タスクとして起動する．実行終了後，実験戦略の性能結果を戦略データベースに登録する}

3.3 マルチプロセッシングの構造

図5にマルチプロセッシングの構造を模式的に示す．[ジョブ，配置戦略（プログラム分割リスト），仮想マシン]の三つ組として投入されたタスクを実際に実行する．具体的には，仮想マシンの最初の PE にプログラム分割リストが転送された後，初期プロセスが投入されることで始まる．PE はプロセスを実行するが，必要なプログラムがない場合には仮想マシンの次の PE へプロセスを自動転送する（プロセス実行手順）．実行中にはプロセス実行の合間をぬって，時に応じて PE の負荷を調べ，負荷が高いならプログラムの一部を仮想マシンの次の PE に送ってプロセスが減るのを待ち，負荷が低いなら，仮想マシンの前の PE からプログラムの一部を引き取り，プロセスが自動転送されてくるのを待つ（プログラム転送手順）．この仕組みによって，「周囲の状況に応じて，負荷が高まれば分散し，負荷が低くなれば集中できる」動的負荷分散が行える．

- プロセス実行手順＝
  if PE に必要なプログラムがあるなら，
  {プログラムを使って実行を進める}
  if PE に必要なプログラムがないなら，
  {プロセスを次の PE† へ転送する}
  † プロセスが流れ続ける事態の防止に，最後尾の PE への戻し（図5参照）を行う．
- プログラム転送手順＝
  if PE の負荷が高くなったら
  {次の PE へプログラム単位を一つ選び送る}
  if PE の負荷が低くなったら
  {前の PE へプログラム単位を要求する}
  if プログラム転送要求を受けたら
  {要求元にプログラム単位を一つ選び送る}

3.4 例　題

性能管理 OS のプロトタイプを，逐次マシン上の並列言語実行系を用いて試験的に作成した．使用したのは並列論理型言語 KL 1[8] である．並列論理型言語（Concurrent Logic Programming Languages）を使うと，メタプログラミングが容易に行えることは既に良く知られている[4]．メタプログラミングの技法を用いて，プログラムとプロセスを陽に引数に持った PE のシミュレータをネットワーク接続したプロトタイプを作成した．ただし，逐次マシン上での疑似並列を用いる限り，実行時間などに意味がなく，確かめられることには限界がある．

並列論理型言語ではプログラムはクローズの集まりなので，配置戦略はクローズの集合のリストの形で表

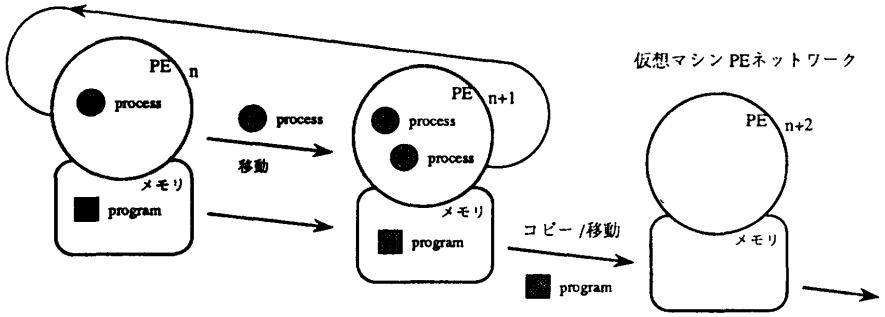


図5 マルチプロセッシングの構造
Fig. 5 Multi-processing level.

現できる．またゴールのリダクションがプロセスに相当するので，ゴールをPEのプログラムでリダクションできなかったとき（fail したとき）は，ゴールは仮想マシンの次のPEへ自動転送される．

並列論理型言語の特徴は，ストリームを利用した並列プログラミングが，同期を取ることにわずらわされずに容易に書けることである．ストリームによるプログラミングは，プロセスネットワークを作りあげるフェイズと，でき上がったプロセスネットワークを動かして実際に問題を解くフェイズから成る．例えば，図6の素数生成のプログラム prime は，gen, sift, filter の三つの手続きからなるが，このうちネットワーク構築に関与しているのは sift で，gen と filter は問題解決を担当する．sift は二つのクローズから成るが，第二クローズは終了処理をするだけなので，積極的にネットワーク構築に関与しているのは，sift の第一クローズのみである．

性能が良いと思われる配置戦略を一つ，手で設計してみる．ネットワーク構築ではプロセスを動的に作り出すので，このフェイズを担うプログラム単位を，空いているPEに積極的に移動させることで，問題解決プロセスの負荷分散が行える．問題解決を担うプログラム単位は，問題解決プロセスを持つPEすべてで必要となるので，コピーをして次のPEへ送るのが良い．例題の場合であれば，sift の第一クローズのみを移動により転送を行うプログラム単位とし，残りはコピーによる転送を行うプログラム単位にすれば良い．この二つのプログラム単位が配置戦略であり，これに適当な仮想マシンを割り当てて，本タスクとして起動する．

この戦略の下でのタスクの実行状況のスナップショットを図7に示す．見てわかるようにgen と filter か

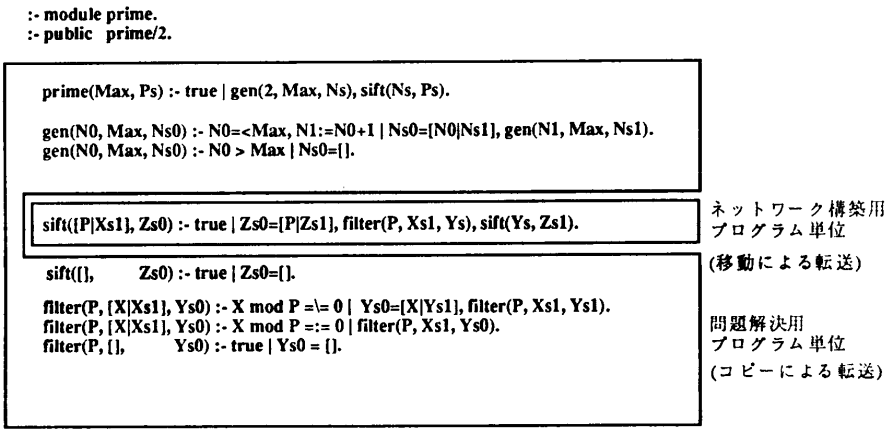


図6 prime プログラムの再配置戦略
Fig. 6 A replacement strategy of prime program.

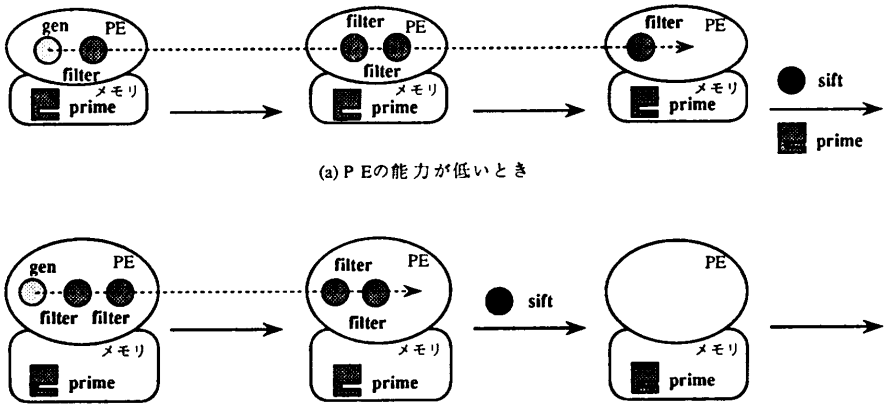


図7 prime プログラムの実行スナップショット
Fig. 7 Two execution snapshots of prime program.

ら成るプロセスネットワークが，仮想マシンの PE ネットワークに沿って形成されていることがわかる．注意すべきは，ネットワークの形成は同じ配置戦略を使っても，PE の混雑状況に応じて臨機応変になされうる点である．例えば PE の能力の違いに応ずることができる．低い能力の PE を持った並列マシンの場合には，一つの PE で担当できる filter の数は少なく，filter ネットワークの長さは長くなる（図7（a））．高い能力の PE を持った並列マシンの場合には，一つの PE で担当できる filter の数は多く，filter ネットワークの長さは短くて済む（図7（b））．別の例では，二つのタスクの仮想マシンが PE を共有している場合に，共有 PE では負荷の衝突が起こりうるが，そのような場合にも負荷の集中は回避される．このような適応性は，注記を用いた静的負荷分散法では一般的には得られない．

4. 性能管理 OS の課題

前章に述べた性能管理の枠組みは，抽象的に記述されていた．性能管理の方法を実証していくには具体化していかねばならないが，いろいろな具体化の方策がとりうる．本章では，実証に向けての課題を整理するとともに，プロトタイプ作成で採用した具体策を述べ，性能管理 OS の実現可能性（feasibility）を示す．

4.1 判定基準

性能管理の方法には三つの判定基準が含まれている．第一に，並列マシン全体に新しいタスクを走らせる余裕があるかどうかの判定．判定結果は，実験タスクを走らせる余裕があるかどうか決めるのに必要である．第二に，タスクの実行後，使用した戦略の良し悪しを決める判定．判定結果である戦略の良し悪しは，戦略データベースに記録されて，後の戦略立案に利用される．第三に，タスクの実行時に，PE が込んでいるか空いているかを決める判定基準．判定結果は，PE の込みぐあいに応じてプログラム単位を送る手順に必要である．

プロトタイプでは次のような代替基準を採用した．並列マシンの余裕の判定には，仮想マシンの供給状況を見る．仮想マシン記述はあらかじめ複数作成してリスト化しておき，使用中の仮想マシン記述の多い少ないで判定する．すでに多く供給されていれば余裕がないと判定する．仮想マシンを構成する PE をどう構成するかについては制約はないが，通信コストを考えるとなるべく隣接した PE を結ぶ形状の方が好ましい．

戦略の良し悪しの判定には，望む答えが求まるまでの時間を計って比較する．本実行と試験実行は同じ仕事を実行して比較するので，仮想マシンが他のタスクと干渉しないという条件のもとで，時間を比較する意味がある．何が求まれば計算が終わったと言えるかは，利用者に陽に指定してもらう．PE の込みぐあいの判定には，PE のプロセスキューの長さを見る．特定のプロセスに着目するなら，キューが長いと次にサービスを受けるまでの時間が伸びるので，PE は込んだ状態にあると言えるからである．

4.2 探索手続き

性能管理の方法には二つの探索手続きが含まれている．第一に，戦略データベースを参考に，プログラムの再配置戦略を立案する探索手続き．その時点での最善の戦略と実験戦略を立案する，二種の探索手続きが必要である．第二に，現在の並列マシン状況から，空いている仮想マシン記述を見つけ出す探索手続き．既に使用中の仮想マシンと重なってもよいが，なるべく重ならない方が好ましい．

プロトタイプでは次のような手続きを採用した．戦略データベースには，［ジョブ名，プログラム分割リスト，仮想マシン記述，性能結果］を記録する．ジョブ名はプログラムを同定するためのキーの役割を果たす．最善の戦略を立案するときには，ジョブ名をキーとして最も高い性能結果を持ったエントリを探す．再配置戦略にはエントリのプログラム分割リストをそのまま用い，仮想マシンはエントリの仮想マシン記述が使用中でないなら採用する．使用中の場合には，なるべく規模や形状が近い仮想マシン記述で使用中でないものを選ぶ．

実験戦略の立案については，ヒューリスティックス（その一つを例題の節で紹介した）で分割したプログラムを初期値として，分割を少しずつ改変しては試す方法をとる．改変が大局的には改善方向を向いていることを保証するために，遺伝アルゴリズム（genetic algorithm）の手法（例えば文献 2））が使えると考えている．すなわち，よい性能を示した二つの分割リストを選び，それらを親として，プログラム単位を部分的に交換／改変することで子孫となる分割リストを作る．分割リストが階層的な構造を持っていないことが，遺伝アルゴリズムが適用できる根拠である．

仮想マシンの記述にはラスタグラフィックスアルゴリズムを流用した．プロトタイプでは実マシンとして PE が平面メッシュネットワークで結ばれた構成を仮

定し，ネットワーク平面をフレームバッファに見立てて，図形を描く要領で PE の選択を行っていく．ドットを描く順番で PE に順番を付けることで，プログラムやプロセスの送り先を一意に指定できる．グラフィックスアルゴリズムで記述しておくと，仮想マシンの形状は直線や円など線図形に限らず，閉領域の塗り潰しの形でも指定可能である．複雑な形状の仮想マシンも比較的少数のパラメータで指定ができ，しかもパラメータの系統的変更で，大きさを変えたり位置を変えたりが簡単にできる．また，仮想マシン同士がどの PE で交わりを持っているかも計算で割り出せる．並列マシンの余裕を判定するときに，使用中のすべての仮想マシンの重なり合いの程度を計算して，余裕の度合いを判定する方法なども一つの利用法である．

4.3 選択手続き

性能管理の方法には一つの選択手続きが含まれている．プログラム単位転送時に，プログラム単位が複数あったとき，どのプログラム単位を送るべきかを決める選択手続きである．逐次マシンの仮想メモリ技術において，必要なページをメモリに転送した代わりに，どのページをディスクに追い出すかの置き換え規則 (replacement rule) は，仮想メモリの性能に大きく影響することが調べられてきた（例えば文献 7)）．現在では LRU (Least Recently Used) と呼ばれる手法に固まっている．

考えられる方法は，プログラム単位が送られた来たと同じ順序で送り出す FIFO 式，最近送られたものほどはやく送り出す LIFO 式，最近もっとも頻繁に使われたプログラム単位を送り出す方式などである．プロトタイプは簡単な FIFO 式である．最後の方式は LRU の逆で，MRU (Most Recently Used) とでも呼べる手法である．MRU が有望なのは，最近もっとも頻繁に利用されたプログラム単位を送り出すから，そのプログラムを利用しなければならないプロセスが頻繁に転送され，PE の負荷がはやく低くなると期待できるからである．

プログラムを実行時に転送する手間は，他の効果に隠れて実効的に小さくなる．仕掛けは，逐次マシンのページ転送の場合も，性能管理 OS のプログラム転送の場合も同じである．参照の局所性（必要なプログラム部分を含んだプログラム単位には近い将来必要となる部分が含まれている可能性が高いこと）は，ページ転送／プログラム転送の回数を減らす効果がある．ページ転送／プログラム転送のコスト自体も，他の実行可能なプロセスの実行にスイッチしてしまうことで，見えなくなってしまう．

5. おわりに

本論文の狙いは，性能管理 OS という新しい枠組みの実現可能性 (feasibility) を示すことにあった．

性能管理 OS はプログラムの動的再配置を通して，間接にプロセスの負荷分散を達成する．従来の「プログラムをプロセスのある PE へ転送する」という発想を逆転し，「プログラムのある PE へプロセスを転送する」ことがアイデアの要である．プログラムの再配置とプロセスの負荷分散という，いわば二階建てのスケジューリング法を採用したことで，プロセスの負荷分散に OS が直接介入する必要がなくなった．加えて性能管理 OS は，プログラムの再配置戦略を記録し，後の実行に役立てるフィードバックの仕組みを持つ．その時点での最善戦略による実行を行うとともに，OS は並列マシンに余裕があるときには，実験戦略を使っての試験実行も実施する．各戦略のもとでの性能結果を記録し，後の戦略決定に利用する．

逐次マシン上での疑似並列実行によるプロトタイプ版による具体化例を作成したが，本格的な汎用並列マシン上での実証が課題として残っている．しかしチューニングを前提としたバッチ形態の並列実行環境を，手軽な並列実行ができるマルチタスクの環境へと，質的に変化させえることを示した．

謝辞 性能管理 OS プロトタイプの作成には，富士通 SSL の太田祐紀子氏と富士通国際研の前田宗則氏の助力を得ました．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) Chikayama, T., Sato, H. and Miyazaki, T.: Overview of the Parallel Inference Machine Operating System (PIMOS), *Proc. FGCS '88*, Vol. 1, pp. 230-251 (1988).
2) Holland, J.: Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems, *Machine Learning: An Artificial Intelligence Approach*, Michalski, R. S. et al. eds., Vol. II, Morgan Kaufmann (1986).
3) Shapiro, E.: Systolic Programming: A Paradigm of Parallel Processing, *Proc. FGCS '84*, pp. 458-470 (1984).
4) Shapiro, E.: The Family of Concurrent Logic Programming Languages, *ACM Comput. Surv.*,

Vol. 21, No. 3, pp. 413-510 (1989).
5) Takeda, Y., Nakashima, H., Masuda, K., Chikayama, T. and Taki, K.: A Load Balancing Mechanism for Large Scale Multiprocessor Systems and Its Implementation, *Proc. FGCS '88*, Vol. 3, pp. 978-986 (1988).
6) Tick, E.: Compile-Time Granularity Analysis for Parallel Logic Programming Languages, *Proc. FGCS '88*, Vol. 3, pp. 994-1000 (1988).
7) Tsichritzis, D. and Bernstein, P.: *Operating Systems*, Academic Press (1974).
8) 新世代コンピュータ技術開発機構第四研究室: KL 1 プログラミング入門編／初級編／中級編 (1989).

(平成元年8月31日受付)
(平成2年9月11日採録)

神田 陽治 (正会員)

1959年生. 1981年東京大学理学部情報科学科卒業. 1986年同大学院情報工学博士課程修了. 工学博士. 同年富士通(株)国際情報社会科学研究所入社. 並列OS, 新しいユーザインタフェースの設計原理, グループウェア等に興味を持つ. 電子情報通信学会, 日本ソフトウェア科学会各会員.