オペレーティング・システム　58－3
（１９９３．３．１９）

超並列システム用オペレーティングシステム「超流動 OS」の構想

平野 聡　田沼 均　須崎 有康　浜崎 陽一　塚本 享治

電子技術総合研究所

hirano@etl.go.jp

超並列システムのためのオペレーティングシステムである超流動 OS の研究を開始した。現在、高並列システム用の OS が実用化されているが、並列度に対する柔軟性と PE の使用率が低い問題がある。超流動 OS(Fluid) は 100 万個程度の PE 群をコントロールするために時間と空間の分割共有を行ない、単一プログラムの実行時間の短縮と多数のプログラムのスループットの向上の両立を図る。超流動 OS はアプリケーション・プログラムを動的に実行環境やデータに適合させるアルゴリズム・アダプテーションやメモリの負荷分散を行なう大域的仮想仮想記憶などを備える。

Research Plan for Fluid, A Massively Parallel Operating System

HIRANO Satoshi　TANUMA Hitoshi　SUZAKI Kuniyasu
HAMAZAKI Yoichi　TUKAMOTO Michiharu

Electrotechnical Laboratory, Japan

hirano@etl.go.jp

We present a research plan for Fluid, a massively parallel operating system. Some parallel operating systems have been developed and used. These operating systems, however, limits the parallelism of programs and attain low utilization of processing elements. In order to manage millions of PEs efficiently, Fluid aims at both high performance of a program and high though-put of many programs by sharing PE space and time, that will be archived by giving fluidness to objects in the massively parallel system.

1 はじめに

永年に亘る並列処理技術の研究の成果もようやく花開き、ワークステーションからスーパーコンピュータまで広い範囲で並列システムを使用することが可能となってきた。しかし、性能向上への期待は留まるところを知らず、より強力で、しかも使い易い並列システムの登場が心待ちにされている [1, 25]。

そこで我々は、百万個程度の PE を有する将来の汎用超並列システムにおけるオペレーティングシステムのあるべき姿として、多数のプログラムが時間と PE 空間を分割共有し、単一プログラムの実行時間の短縮とシステム全体のスループットの向上の両立を目指す、「超流動 OS(Fluid)」の研究を開始した。

以下本論文では、次章で現在の並列システムソフトウェアの現状について簡単に触れた後、3章で超流動 OS の設計指針について、4章で超流動 OS の構成要素について述べる。5章と 6章では超流動 OS の構成要素のうち「アルゴリズム・アダプテーション」と「大域的仮想仮想記憶」の二つを取り出してその概念と予備評価の結果を簡単に紹介する。

2 並列 OS の現状

プログラムの実行環境としての OS について、その構成法を分類すると以下のようになる。各種の構成法が混在しているのが現状である。

1. 初期の並列システムの多くは単一の並列言語で動作する専用機であったため OS はなく、言語に組み込まれたランタイムライブラリによって管理された。次に言語からの独立のために、複数言語から使用する事が可能で、アーキテクチャに比較的依存しない並列プリミティブ・ライブラリが使用されている。言語としては ABCL[26] など多くの実験的言語があり、ライブラリとしては PARMAX[2], P-4(Argonne macros)[3] 等がある。また、SIMD 動作をするために、シーケンサから PE の状態を完全に管理する CM-2[22] や MasPar MP-1[16] のようなアプローチもある。

2. Micro Kernel+ フロントエンド・プロセッサ上の Unix サーバの形態。商用機では Unix への互換性が必要になる。そこで、各 PE にマイクロカーネルと Unix システムコールの API ライブラリを載せ、フロントエンド・プロセッサ上の Unix サーバに RPC を行なう。このような構成は、並列 OS を必要としないためシステムの製作が容易であるが、フロントエンド・プロセッサがボトルネックとなる。商用機では Parsytec/GC[17] 等がある。

3. スタンドアローンで Micro Kernel + パーティション毎の Unix サーバの形態。ボトルネックの原因となるフロントエンド・プロセッサを排し、一部の PE を Unix サーバとして割り当てる構成法である。しかし、独立したフロントエンドが果たしていた役割を同一筐体内の PE が分担しただけで、フロントエンドとバックエンドから構成されていることには変わりなく、管理負荷の動的な再配分は難しい。PARAGON[10] と CM-5[19] に代表される。CM-5 のノードは複数のパーティションに分割され、それぞれのパーティションにひとつのパーティション・マネージャ (PM) が配される。PM は Unix (SunOS) であり、他のノードにはマイクロカーネルが載っている。PM 内のノードのスケジューリング、メモリ管理、I/O などは全て PM が行なう。パーティション内のスケジューリングは同期しており、同一の時間にはひとつのプログラムしか動作しない。パーティションの大きさはシステム管理者によって決められる。

4. スタンドアローンで、計算用 PE と OS 用 PE(Unix サーバ) の区別がない形態。共有メモリマシンでは、32PE までのシンメトリック・マルチプロセッシング環境を提供する SVR4/MP、キャッシュのプリフェッチをサポートする Dash[12] 等がある。また、PIMOS は単一の言語 KL1 の処理系の上にメタプログラムとして実装されており、タスクの管理などは PE に跨る木構造を基に分散して実行される [5]。

2 と 3 の大規模な並列システムの多くは、システム内の PE をパーティションに区切り、パーティション毎に異なるプログラムを動作させる。ユーザはパーティションをあたかもひとつの独立したシステムとして扱うことができ、データパラレルのプログラムを高速に実行することが可能であるが、反面、パーティションの大きさはプロセスに固定であるため、複数のシステムを同じ筐体に納めているだけに等しい。プログラムが有する並列性や規則性は処理すべきデータや時間軸に沿って大きく変化して行くため、プロセスに固定された大きさのパーティションは、プロセスの並列度がパーティションの大きさより大きい場合は並列性を制限し、また、プロセスの並列度が小さい場合はパーティション内の多くのプロセッサを無駄にする (図 1)。

超並列システムは高価であるため、可能な限りの無駄は排されるべきである。従って、今後追求しなければならないのは、同時に実行されるプログラムを各々の性質に応じた最適な並列度で実行し、かつ資源を適正に分割共有する事である。そのため、静的・動的な PE (パーティション)

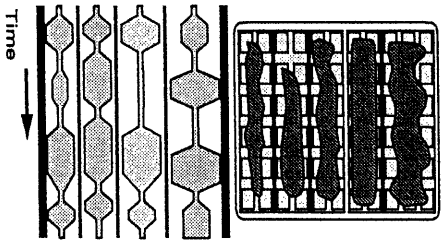

図 1: パーティションによる並列性の制限

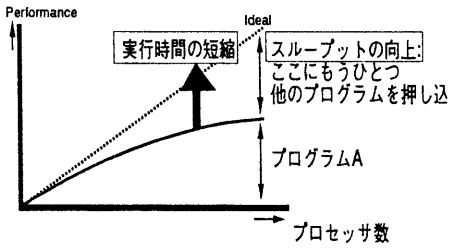

図 2: スループットをどこから得るか

のアロケーション、マイグレーションを含むリロケーション、スケジューリング等、多くの研究が行なわれている [4, 6, 7, 8, 11, 13, 21]。しかし現時点において、これらはモデルあるいはシミュレーションを用いた研究であり、超並列 OS として形を現してはいない。

その原因として考えられるのは、まず第一に単一プログラムの性能向上と複数プログラムのスループット向上の間のジレンマである。最大の性能を追求してスケジュールされたプログラムはその「形」が固く、スループット向上のために PE の再配置を行なうと性能が極端に低下したり、他のプログラムと混合して実行すると隠蔽したはずの通信・同期のオーバーヘッドが露呈してしまうことがある。

第二にハードウェアとソフトウェアの役割分担の問題がある。効率のよい実行のために、ハードウェア資源をできる限り「生」で扱い、OS 介入のオーバーヘッドを減らしたいという要求がある。例えば、通信の機能がプロセッサアーキテクチャに組み込まれている場合、より細粒度の並列処理が可能となる [18] が、ここに OS による管理を必要とする複数プログラムの同時実行を入れることは更なる研究を必要とする。

3 超流動 OS の設計指針

超流動 OS は、これまでの研究を踏まえ、百万台規模の PE を有する超並列計算機上で様々なパラダイムに基づいたプログラムを複数同時に動作させ、個々のプログラムが高速に実行されると共に、システムの資源を有効に共有しシステム全体が高いスループットのもとに動作する実行環境の実現を目指す。対象とするハードウェアは我々が参加している RWC 計画において試作する各種の超並列マシン及び市販の汎用超並列マシンであるが、更に本研究により、将来のアーキテクチャに影響を及ぼすことができれば幸いとしたい。

単一プログラムの性能向上のためには、解くべき問題が含有する並列度をできる限り制限しないことが重要である。即ち、プログラムが使用する PE 数は台数効果が期待できる限り大きく確保すべきである。そのためには、固定したパーティションを排し、時には 100 万個全ての PE を使用することをも可能とする柔軟な PE 管理や負荷分散を行なう必要がある。また、PE 以外の資源も負荷分散の対象とする。例えば、あるプロセスに対してそのプロセスが使用するメモリをシステム全体のプロセスのメモリ使用量の比で割り当てることも有効であろう。

複数プログラムのスループット向上のためには、使用していない資源を有効に利用する必要がある。これは、

- プログラムは変化する並列度に合わせて最適な量の PE を使用し、使用していない PE を無駄に確保したままにしない。
- 同期・通信のアイドル時間を他プログラムに開放する。

といった些細な努力の積み重ねである。後者を図 2に示す。あるプログラム A が図のような台数効果曲線を有するならば、プログラム A が無駄にしていた資源、即ち、台数効果曲線と理想曲線の間隙を他のプログラムで使用することにより、スループットの向上を図る。

汎用の超並列システム上で動作するプログラムは、大規模数値計算などで用いられるデータパラレルや、並列オブジェクト指向のようなコントロールパラレルなど様々な形態をとると考えられる。超流動 OS ではその形態を以下の諸性質により特徴づけ、プロセスアトリビュートと呼ぶ。

粒度 Granularity プログラム中の並列処理の単位（以下、アクティビティ）の大きさ。例えば、データパラレルのプログラムでは並列処理の単位は配列要素のひとつひとつなので、一般に粒度は小さい。コントロールパラレルのプログラムではそれよりも大きいであろう。

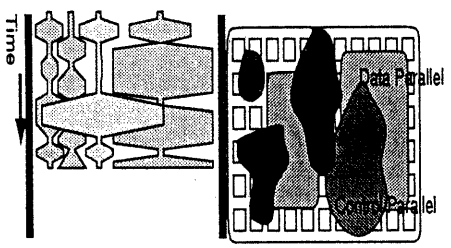

図 3: 超流動 OS の概念

規則性 Regularity プログラムが必要とするネットワーク上のトポロジ、アクティビティ間の時間依存性、及び並列度の変化の大きさ。データパラレルでは規則性は大きく、コントロールパラレルでは小さいであろう。

大きさ Size プログラムとデータの大きさ。また、並列度の大きさ。

純度 puritiness あるアクティビティがどれだけメモリあるいは二次記憶中のデータとかかわっているか。副作用のない関数の計算だけならば最高の純度を有するとする。

生存時間 Lifetime アクティビティの生存時間、また生成消滅の頻度。

超流動 OS ではプログラムが有するこれらの特徴を積極的に利用することにより、プログラム各々に適した効率的な管理を行なう。PE のアロケーション、リロケーション、スケジューリング等のポリシーに内蔵された評価関数がプロセスアトリビュートを利用可能なよう、各種の特徴を抽象化・数値化する方法を考案したい。

超流動 OS は図 3のように、ネットワークトポロジへの依存性が高く規則的な動作をするデータパラレルや、規則性が低いコントロールパラレルといった複数のパラダイムのプログラムを混在して実行した場合にも、変化する並列度や粒度に合わせて情報 (プログラム、データ等) を流動させることにより、非常に多数の PE やアクティビティを効率良く管理することを目指す。

4 超流動 OS の構成要素

本章では超流動 OS の主要な構成要素の構想について述べる。

超流動 OS は C++ を基にしたオブジェクト指向言語を用いて記述するものの、オブジェクトを OS の管理の基盤とはしない。これは High Performance Fortran[9] を始めとする様々なパラダイムの言語に対応するためには、全てをオブジェクトとして扱うことは大きなオーバーヘッドを伴うと考えられるためである。

OS はマイクロカーネルと PE に跨るアクティビティ管理部分 (各種ポリシー) から構成される。アプリケーション・プログラムはアルゴリズム・アダプテーションにより変態し、OS と相互作用する。

以下、超流動 OS の構成要素を列挙する。

アルゴリズム・アダプテーション アプリケーション・プログラムがシステムの構成や使用可能な PE 数などの環境に合わせて自律的にプログラムの形態を変えて適応してゆく。これをアルゴリズム・アダプテーションと呼ぶ。変態はニューラルネットワークを用いてプログラムが使用するアルゴリズムを環境に合わせて動的に変更・調整して行くことにより実現する。また、並列実行の粒度の調節もアルゴリズム・アダプテーションの目的のひとつである。プロセスに与えられたデータの量と性質及び使用可能な PE 数から動的に最適な粒度を推論し適応する [23]。逆に、プログラムが PE 数、ネットワーク・トポロジ等を OS に要求・折衝し、OS が環境を変更して行くことをコンフィグレーション・アダプテーションと呼ぶ。リフレクション [14, 20] は実行中に言語レベルから実行環境の操作を可能にする手法であるが、あくまでポリシーを伴わないメカニズムであるため、プログラマが最適化の判断基準を別に用意する必要がある。アルゴリズム・アダプテーションはリフレクションよりも積極的に動的な最適化を行なう。これについては 5章でより詳しく述べる。

アクティビティのアロケーション・リロケーション 新たに生成されたアクティビティを PE に割り付ける際に、そのアクティビティが属するプロセスが高速に実行されるよう負荷の集中を避けて割り当てること、及び、システム全体の負荷を分散し他のプロセスの実行を妨げないことを両立したい。また、プロセスの粒度、規則性、大きさなどのパラメータは時間と共に変化してゆくため、一度割り付けた PE の場所や数が不適当になって行く。そこで、アクティビティの移動を行ない、各パラメータの最適性を確保する。

PE の割り付けは、メッシュネットワークに対する割り付けアルゴリズム [13, 21]、ハイパーキューブネットワークに対する割り付けアルゴリズム [4, 6] 等を参考に、他のプロセスの移動や、プロセス・アトリビュートの利用によるプロセスに適した形状の割り付けを行なう。例えば、データパラレルでネットワークトポロジに対する規則性が高いプロセスはそのトポロジを確保するように、逆に規則性が低いプロセスは規則性が高いプロセス間の隙間を埋めるように

PEを割り付ける。

これらはアルゴリズム／コンフィグレーション・アダプテーションと密接な相互作用をしながら動作する。

2レベルスケジューリング データパラレルのプログラムのような規則性の高いプログラムを効率良く実行するためには、空間的に分散するアクティビティを同期して実行する必要がある。そこで、スケジューラはPE間での同期を加味したグローバル・スケジューラとPE内のローカル・スケジューラの二つにより構成する。

グローバル・スケジューラ グローバルスケジューラはプロセスアトリビュートに基づき同期性の高いプロセスがPE空間内で同一時刻のタイムスライスに割り当てられるようスケジューリングを行なう。これはギャングスケジューリングと呼ばれる [8]。各タイムスライスに割り当てられた1つのプロセスをそのタイムスライスにおけるプライマリ・プロセスと呼ぶことにする。

ローカル・スケジューラ プライマリプロセスは同期や通信により待ち状態になることがある。このアイドル時間を利用するのがローカル・スケジューラである。[8] では粒度が小さくなるとビジーウェイトをするギャングスケジューリングの方がプロセスをブロッキング(ブロック中は他のプロセスが動作)し各PEで独立にスケジューリングを行なうよりも効率が高くなることを示しているが、ギャングスケージューリングとブロッキングの組み合わせも考えられる。ローカル・スケジューラには空間的な同期性の不要なアクティビティが登録され、プライマリ・プロセスが待ち状態になったらプロセッサが割り当てられる。プライマリ・プロセスの待ち状態が解消後、直ちにプライマリ・プロセスのアクティビティが実行を再開する。プロセスがビジーウエイトをするかブロッキングをするかは動的に決定したい。

大域的仮想仮想記憶 (GVVM) 超並列システムにおいては逐次システムと比較して PE 数に対する I/O バンド幅が非常に小さくなる。従って仮想記憶(ページング)の性能が低くなる。また、特定 PE でメモリが不足する Fatspot 現象が発生し、その PE が全体の実行の律速段階となる。そこで、各 PE で個別に仮想記憶を行なうのではなく、システム全体で使用頻度の低いページをページアウトすること(大域的仮想記憶)、及び、他 PE のより使用頻度の低いメモリをスワップアウト領域として使用すること(仮想仮想記憶)により、仮想記憶の性能の大幅な向上を図る。

これは、プロセスが使用するメモリ(実記憶)をPEの割り付けとは独立に確保することを意味する。6章でGVVMの概要について述べる。

メタ情報の共有 – メタシェア 上記の管理ポリシーはいずれもが大域的な負荷情報やアクティビティの割当状況等のメタ情報を使用する。メタ情報は大部分がポリシー間で共通であるため、各ポリシーが独立にメタ情報を収集するよりも、メタ情報を共有するメカニズムを作りそこで収集をする方が効率がよい。このメカニズムをメタシェアと呼ぶ。メタシェアは各PEに存在するエージェントから構成され、そのエージェントがデータパラレル的にメタ情報を収集し、各PE内のクライアント(ポリシー)に供給するメタ情報はPEの近隣は詳細に、遠方は全体像が容易に把握できるよう抽象化されて提供される。

様々な管理は必ずオーバーヘッドを伴うため、効率を向上するはずの管理が必ずしも向上にはつながらない。オーバーヘッドの少ないアルゴリズムと実装の開発と共に、OSが介入を行なうか行なわないかを動的に決定し、最適化して行くメカニズムが必要である。

5 アルゴリズム・アダプテーション(適応的アルゴリズム選択法による動的最適化)

本章ではアルゴリズムアダプテーションのひとつとして、プログラムの動的なチューニング法である AASM (Adaptive Algorithm Selection Method) について紹介する。

ソートや文字列検索のようにひとつの問題に対して複数のアルゴリズムが存在し、データの特徴によって最適アルゴリズムが異なる場合がしばしばあるが、予め最適なアルゴリズムを選択することは理論的計算量の考察からだけでは難しい。また、データの性質ばかりではなく、実行するマシンの構成や実行環境によっても最適なアルゴリズムは異なる。図4では、データのサイズが小さい場合にはアルゴリズムA、中くらいの場合にはB、大きい場合にはCが最適である。この場合、データのサイズに応じて自動的にそれぞれのアルゴリズムの選択ができれば広い範囲のデータに対して効率的な処理が可能となる。

並列計算機が一般に使用されるようになった現在ではアーキテクチャとアルゴリズムの適合性の問題が拡大してきている。実行する計算機アーキテクチャに適合しないアルゴリズムを使ったソフトウェアは並列化による恩恵をこうむることができない。またプログラムのポータビリティにも支障をきたす。

これらの問題点を解決するため、プログラムの実行時にデータやアーキテクチャの性質に適合するアルゴリズムを動的に選択し、実行するための一般的方式として AASM (Adaptive Algorithm Selection Method) を提案した [24]。

本方式ではライブラリ中に、同一の機能を実現する複

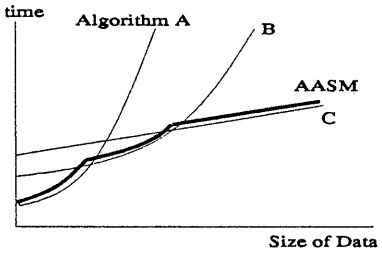

図 4: AASM の概念

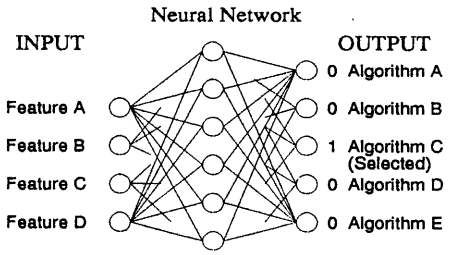

図 5: AASM のニューラルネットワーク

数のアルゴリズムをインプリメントしておく。そのライブラリが呼び出されるたびに AASM のランタイムルーチンは引数であるデータからデータ量等の特徴量を抽出し、そのデータと実行環境に対して最適アルゴリズムの判別を行い、実行する。ライブラリをインターフェースとするのは、並列計算機上でのプログラミングは難しいため、予め用意されたライブラリを用いてプログラムを作成することが多くなるであろうと予想することがひとつの理由である。

本方式は各データの特徴量に対する最適アルゴリズムの選択基準を記憶する学習フェーズと、選択基準を基にして実行時に最適アルゴリズムを選ぶ実行フェーズからなる。

学習フェーズでは、まず、データの特徴量と最適アルゴリズムの相関関係を学習するために適したデータで性能測定を行なう。学習にはニューラルネットワークを用いる (図 5。測定の結果に基づいて特徴量から最適アルゴリズムを判別する選択基準を獲得する。性能測定の段階で計算機アーキテクチャに適合しないアルゴリズムは悪い結果を出すので、アーキテクチャに適するアルゴリズムが選択基準に反映される。

実行フェーズにはライブラリが呼び出されるたびに引数として渡されるデータから特徴量を抽出し、それを入力とするニューラルネットワークが学習フェーズで得られた選択基準を基に最適なアルゴリズムを判別し、そのアルゴリズムで処理を行なう。

5.1 複数文字列検索を対象とした予備実験

本方式の有効性を確かめるために、複数文字列検索の予備実験を行なった。複数文字列検索アルゴリズムは次の 5 つを候補とした [27]。また、データとして UNIX のオンラインマニュアルを用いた。

- Quadratic アルゴリズム (QD)
- Knuth-Morris-Pratt アルゴリズム (KMP)
- Boyer-Moore アルゴリズム (BM)
- Aho-Corasick アルゴリズム (AC)
- 拡張 Boyer-Moore アルゴリズム (EBM)

QD	KMP	BM	AC	EBM	AASM
4018.8	4179.9	1133.9	4870.1	1515.3	1200.6 (1101.5)

表 1: 各アルゴリズムと AASM の総計算時間 (sec)

これらのアルゴリズムの計算時間に影響する要素には、解析的結果 [27] より検索ファイルのサイズ、検索する文字列の数、文字列の長さであることわかっている。さらに拡張 Boyer-Moore アルゴリズム では、検索文字列の長さの差が影響することがわかっている。そこでファイルのサイズ、検索文字列の数、検索文字列長の平均、検索文字列長の分散を特徴量として採用した。対象計算機としては、Sparc Station 2 を使用した。

上記の条件で検証データ 2065 パターンについて各アルゴリズム及び AASM 総計算時間を表 1 に示す。AASM の括弧内の数字は、純粋に問題解決のアルゴリズムで使われた時間である (ニューラルネットのオーバヘッドを含まない)。

表から BM を実行することが最適な戦略であるが、AASM は二番目によい結果を示すことがわかる。その差は 5% ほどである。この結果はニューラルネットワークの計算時間というコストを支払っても、ほぼ最適なアルゴリズムを選択したことを意味している。並列計算機のように計算機アーキテクチャが複雑となり、アルゴリズムの適合性が問題となる場合にはその効果を示すと思われる。

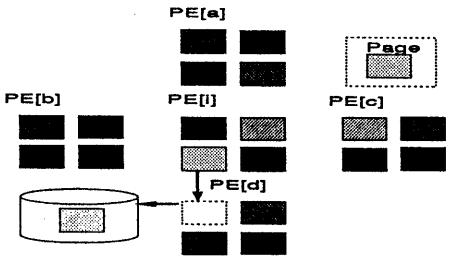

図 6: 大域的仮想仮想記憶の概念

6 大域的仮想仮想記憶による記憶管理

仮想記憶の目的はプログラムに実記憶よりも大きな仮想空間を提供し、実記憶の大きさを超える問題を解くことである。超並列システムにおいては逐次システムと比較してPE 数に対する I/O バンド幅が非常に小さくなる。従って仮想記憶 (ページング) の性能が低くなる。また、いくつかの PE でメモリが不足すると、システム全体ではメモリが余っていてもそれらの PE が全体の実行の律則段階となる。

超流動 OS は複数のプログラムを混在して動作させる環境を提供するため、メモリアクセスに関して PE 内での局所性と共に PE 空間内での局所性や偏りも期待できる。そこで、本論文で提案する大域的仮想仮想記憶は以下の二つの基本概念を基にして上記の問題の解決を図る。

大域的仮想記憶 Global Virtual Memory 大域的仮想記憶はページアウトの対象となるページをシステム全体の PE 上のメモリを対象にして決定する。即ち、全 PE 中で最も過去にアクセスしたページをページアウトする。

仮想仮想記憶 Virtual Virtual Memory 二次記憶への転送バンド幅が小さいため、全てのページングを二次記憶に対して行なうことはシステムの性能低下を引き起こす。そこで、他 PE 上に未使用のページがある場合、または、自 PE のページアウト可能なページよりも使用頻度の低いページが他 PE 上に存在する場合、そのページをスワップ・スペースとして用いる [15]。ネットワークの転送速度は磁気ディスクの転送速度を大幅に上回るため、高速性が期待できる。

GVVM は上記の仮想仮想記憶において、他 PE 上のスワップ領域として使用可能なページを選択する際、大域的仮想記憶の概念を用いる (図 6、色の濃さはページの使用頻度を表す)。これは、あるプロセスが使用するメモリを、そのプログラムに割り当てられた PE 群とは独立に、システム全体のプロセスのメモリ使用量の比でメモリを割り当てることに相当する。即ち、メモリの負荷分散が実現される。

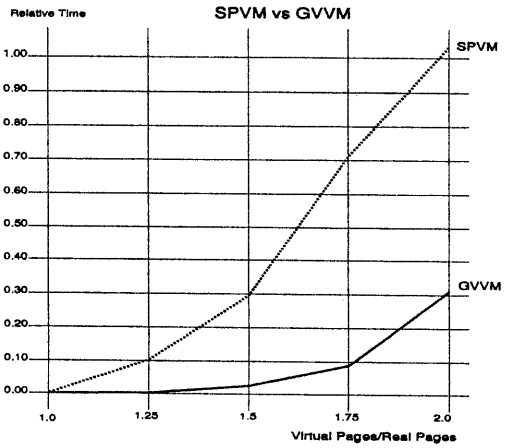

図 7: SPVM 対 GVVM の実行時間

6.1 シミュレーションによる予備評価

プログラムがアクセスする仮想記憶空間の大きさが実記憶を超えた場合でも、GVVM を用いると単一 PE 仮想記憶 (SPVM) を用いた場合と比較してシステムの性能低下がより少ないことを検証するため、シミュレータを作成し予備的な評価を行なった。簡単のため、プログラムがアクセスするページのワーキングセットは仮想空間内で正規分布に従うものとし、PE 空間内でも正規分布に従ってワーキングセットの分散が変化するものとする。それぞれ二次記憶装置を有する 64 台の PE が、ストアアンドフォワード転送を行なう 2 次元トーラスのネットワークで結合されており、ネットワークの転送速度は二次記憶の転送速度に対し 50:1 の比を有するとする。結果を図 7に示す。

横軸は仮想空間と実空間のページ数の比、縦軸は各 PE が一定数の仮想空間をアクセスするのに要した相対時間である。本図から、使用する仮想記憶空間が実記憶空間の大きさの 1.5 倍の場合、GVVM は SPVM の約 11 倍の性能を示し、2 倍の場合には約 3 倍の性能を示すことがわかる。

7 終りに

本研究はリアルワールドコンピューティング計画の前期 5 年に行なわれ、現在は 1 年目である。現在は AASM 及び GVVM の詳細化、アクティビティ管理の基本設計等を行

なっている。本文中で挙げた様々な管理ポリシーを有機的に結合して行き、実行効率の良い超並列OSを目指したい。

本研究は一部はリアルワールドコンピューティング計画の一環として「超並列システムアーキテクチャに関する研究」で行なわれたものである。関係各位に感謝する。

参考文献

[1] G Bell. ULTRACOMPUTERS A Teraflop Before Its Time. *Communications of the ACM*, Vol. 35, No. 8, pp. 27–47, 1992.

[2] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[3] R. Butler and E. Lusk. *User's Guide to p4 Programming System.*

[4] G. Chen and T. Lai. Scheduling Independent Jobs on Partitionable Hypercubes. *J. of Parallel and Distributed Computing*, pp. 74–78, 12 1991.

[5] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). *Proc. of FGCS*, Nov. 1988.

[6] P. Chuang and N. Tzeng. A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers. *IEEE Trans. of Computers*, Vol. 41, No. 4, pp. 467–479, 1992.

[7] F. Dehne. Computing the Largest Empty Rectangle on One- and Two-Dimensional Processor Arrays. *J. of Parallel and Distributed Computing*, pp. 63–68, 9 1990.

[8] D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *J. of Parallel and Distributed Computing*, pp. 306–318, 16 1992.

[9] High Performance Fortran Forum. High Performance Fortran Language Specification version 1.0 Draft. Technical Report CRPC-TR 92225, Rice Univ., Jan. 1993.

[10] Intel Corp. *Paragon XP/S Product Overview*, 1991.

[11] R. Krishnamurti and E. Ma. An approximation Algorithm for Scheduling Tasks on Varying Partition Sizes in Partitionable Multiprocessor Systems. *IEEE Trans. of Computers*, Vol. 41, No. 12, pp. 1572–1579, 1992.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, and J Hennessy. The Stanford Dash Multiprocessor. *IEEE COMPUTER*, pp. 63–79, Mar. 1992.

[13] K. Li and K. Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *J. of Parallel and Distributed Computing*, pp. 79–83, 12 1991.

[14] P. Maes. Concepts and experiments in conputational reflection. *OOPSLA'87*, pp. 147–155, 1987.

[15] M. Malkawi, M. Abaza, and D. Knox. Process Migration in Virtual Memory Multicomputer Systems. *Proc. of HICSS-26*, pp. 90–98, 1993.

[16] J. Nickolls. The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer. *Proc. of IEEE COMPCON*, pp. 25–28, 1990.

[17] Parsytec GmbH. *Persytec GC - Technical Summary*, 1991.

[18] S. Sakai, Y. Kodama, and Y. Yamaguchi. Architectural Design of a Parallel SuperComputer EM-5. *JSPP'91*, pp. 149–156, 1991.

[19] Thinking Machines Corp. *Connection Machine CM-5 Technical Summary*, Nov. 1992.

[20] A. Yonezawa and B. Smith. Refelction and Meta-Level Architecture. *Proc. of New Models for Software Architecture '92*, 1992.

[21] Y. Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *J. of Parallel and Distributed Computing*, pp. 328–337, 16 1992.

[22] ヒリス, 喜連川. コネクションマシン. パーソナルメディア, 1990.

[23] 須崎, 栗田, 田沼, 平野. 超並列システム用OS「超流動OS」におけるアプリケーションソフトウェアの動的最適化. 情報処理学会第*46*回全国大会 発表予定, 1993.

[24] 須崎, 栗田, 田沼, 平野. 適応的アルゴリズム選択法による動的最適化. 第 *34* 回 プログラミングシンポジウム報告集, pp. 177–184, 1993.

[25] 馬場. 超並列マシンへの道. 情報処理学会誌, Vol. 32, No. 4, pp. 348, 364 1991.

[26] 八杉, 松岡, 米澤. ABCL/onEM-4: データ駆動計算機上の並列オブジェクト指向計算システムの高性能実装. *JSPP'92*, pp. 171–178, 1992.

[27] 尹, 高木, 牛島. 5種類のパターンマッチ手法をC言語の関数で実現する. 日経バイト, pp. 175–191, 7 1987.