システムソフトウェアと
オペレーティング・システム 76－1
（１９９７．８．２１）

AP/Linux上での並列プロセススケジューリングの設計

須崎 有康　田沼 均　一杉裕志

suzaki@etl.go.jp

電子技術総合研究所

時分割と空間分割を融合した並列プロセススケジューリングを並列計算機AP1000+上に実装するための設計方針を示す。並列計算機OS AP/Linuxをベースとし、AP/Linux本来のGang Schedulingにプロセス割り当てとwake-up同期を行なうデーモン(sliced)を加えることで可能とした。ここでは、時分割は個々のプロセッサ上でのローカルなスケジューリングを利用し、slicedが空間共有できる並列プロセスを一定間隔で一斉にwake-up割り込み起こす。性能評価には複数のプロセスを実際に走らせ、時分割と空間分割を融合したスケジューリングが効率的であることを確認した。

A design of parallel process scheduling on AP/Linux

Kuniyasu SUZAKI, Hitoshi TANUMA, Yuuji ICHISUGI

Electrotechnical Laboratory

1-1-4 Umezono,Tsukuba-city,Ibaraki,305,Japan

We report a design to implement a scheduling which combines time sharing and space sharing for parallel processes on a real parallel computer AP1000+. The design is based on a parallel operating system AP/Linux. We used some functions of the gang scheduling of AP/Linux. We added a daemon (sliced) which managed process allocation and wake-up synchronization. To confirm the performance of the new scheduling, we ran some processes. We found that the new scheduling had small overhead and achieved good performance.

1 はじめに

並列計算機を多くのユーザに効率良く提供するために様々な研究がなされている。シングルユーザモードでアプリケーションを高速に処理するのみだと多くのユーザを獲得できない。また、バッチ処理的に単一プロセスに並列計算機資源すべてを与えては、使用されないプロセッサを生じて効率的でない。複数のユーザが同時に使用でき、各アプリケーションが応答性良く、しかも並列計算機を効率良く稼働するスケジューリングが必要である。

我々は並列プロセスを効率良く並列計算機上で実行するため、時分割と空間分割を融合したスケジューリング方式を提案してきた [1]。この方式はプロセッサ利用率を高めると共に、個々のプロセスの応答性も良くすることをシミュレーションによって確認した。

本論文では、我々が提案した方式を実際の並列計算機に実装する際の一つの設計例とその性能について述べる。ここでは並列計算機 AP1000+ を対象にし、並列オペレーティングシステム AP/Linux[2] の gang scheduling を改良して実装する設計を示す。

AP/Linux は、オーストラリア国立大の CAP グループにより開発された AP1000+ 用の並列オペレーティングシステムである。AP/Linux は通常の UNIX 環境と共に並列環境も提供する。並列プログラミング環境としては、MPI ライブラリ [3] と AP1000+ が独自に提供してるライブラリ等を利用できる。また、並列実行環境としては簡単な gang scheduling を提供するが、これはある特定のプロセッサに負荷を集中させてしまう欠点を持つ。この欠点を克服するため、我々のスケジューリング方式を実行できるように改良を行なった。新しい機能はプロセス割り当てと wake-up 割り込みをを行なうデーモンによって実現した。本論文では、この設計方針と現在までの実装状況、その性能について述べる。

以下 2章で我々が提案した時分割と空間分割を融合したスケジューリング方式を紹介する。3章では対象とする並列計算機 AP1000+ を概観し、4章で並列オペレーティングシステム AP/Linux の実装を紹介する。ここでは特に並列実行環境に絞り、gang scheduling を中心に説明する。5章で AP/Linux 上での時分割と空間分割を融合したスケジューリングの設計を述べ、6章で現在の実装状況とそのパフォーマンスについて述べる。最後に 7章で結論を述べる。

2 時分割と空間分割の融合

我々は並列プロセスを効率的に実行するために時分割と空間分割（パーティショニングアルゴリズム）を融合したスケジューリングを提案した。パーティショニングアルゴリズムは投入するプロセスが必要とするプロセッサ数や結合形状に合わせて、FCFS (First-Come-First-Serve) でプロセスを物理プロセッサに効率良く割り当てるアルゴリズムである。このアルゴリズムによって複数のプロセスを同時に実行できるようになり、全体のプロセッサのうち稼働しているプロセッサの比率も高めることができる。しかし、パーティショニングアルゴリズムは FCFS であるため、あるプロセスが並列計算機の大部分を占有するとその前後のプロセスを組み合わせることができない閉塞状態を起こすことがある。この状態を回避するため、我々はパーティショニングアルゴリズムを用いた時分割処理を提案した [1]。時分割処理のために実並列計算機と同一結合形状の仮想並列計算機を用意し、空間分割処理は各仮想並列計算機上でパーティショニングアルゴリズムを用いて行なう。この仮想並列計算機をスライスと呼ぶ。一つのスライスに割り当てきれなかったプロセスは別のスライスを新たに用意し、そこに割り当てられる。時分割処理では、これらのスライスに実並列計算機をラウンドロビンで一定時間ごとに割り当てる。この時分割処理化によって、プロセッサ群全体を利用するようなプロセスには一つのスライスを用意し、他のあまりプロセッサを要求しないプロセスは別のスライス上に割り当て、プロセッサ空間を効率的利用できるようにした。

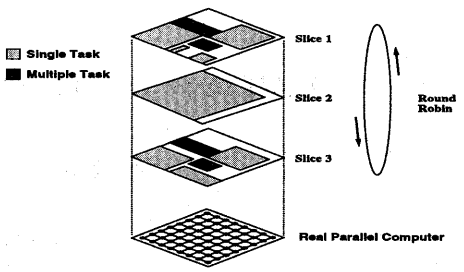

図 1: 時分割と空間分割の融合

時分割処理のスライスの様子を図 1に示す。各スライス上の長方形がプロセス割り当てを表している。一つのスライスにおいてはパーティショニングアルゴリズムによってプロセッサ利用率を上げることができる。更に複数のスライス間で割り当て可能なプロセスは、複数のスライスに存在しプロセッサ利用率を上げることができる。一つのスライスにのみ存在するプロセスを single task、他のスライスに存在することができるプロセスを multiple task と呼ぶ。図 1で薄い灰色の長方形が single task を表し、濃い灰色の長方形が multiple task を表す。また、応答性を良くするために single task が存在しないスライスは削除される。

3 AP1000+ の概観

AP1000+ は富士通から市販される並列計算機である。この計算機ではプロセッシングエレメントの単位を cell と呼び、各セルは 50MHz で動作する SuperSparc を有する。AP1000+ ではリモートメモリアクセスが可能であり、キャッシュはデータの一貫性を保証するため、write

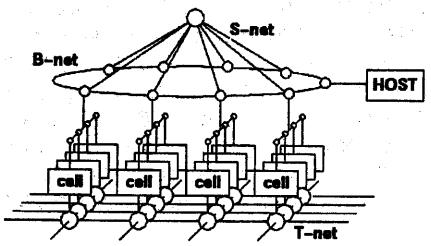

図 2: AP1000+

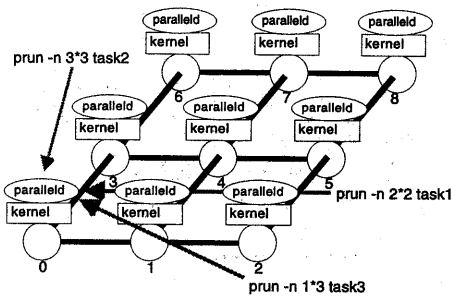

図 3: prun コマンドによる並列プロセス実行

through モードで動作する。

図 2に AP1000+ の概観を示す。この並列計算機は外部とのインターフェースとして HOST 計算機 (Sparc Station) と S-BUS で接続する。また cell 間あるいはホスト計算機と繋ぐネットワークには、T-net, B-net, S-net を有する。T-net は二次元トーラスのネットワークであり、隣接する cell 間を繋ぐ。T-net は cell 間を繋ぐ一つのリンクがバンド幅 25MB/s であり、worm hole routing を有する。T-net を使う各 cell 間の通信方法としてはメッセージ転送 (send,receive) とリモートメモリアクセス (put,get) を提供する。B-net は 50MB/s のブロードキャストネットワークであり、全 cell とホスト計算機と接続する。S-net は同期ネットワークであり、全 cell とホスト計算機と接続する。

AP1000+ は独自のオペレーティングシステム CellOS を提供している。これは single user single program であり、プログラム実行時にプログラムと一緒にロードされ、cell 上に常駐することがない。このため複数のユーザが AP/1000+ を同時に共有利用できない。

4 AP/Linux

AP/Linux[2] は、CellOS の欠点である single usr を克服し、かつ unix プログラム環境を提供するためオーストラリア国立大学の CAP グループで開発された[1]。AP/Linux は Sparc/Linux をベースにし、UNIX プログラミング環境を提供する。特に SUN-OS のシステムコールをエミュレートしているため、SUN-OS のアプリケーションプログラムと binary 互換である。

AP/Linux はホスト計算機から B-net を通して各 cell に kernel がロードされる。ファイルシステムはホスト計算機上のディスクを NFS マウントする。各 cell 上では inetd デーモンが起動され、外部の計算機からの login が可能である。cell に login すると通常の linux 環境が提供される。また、B-net 上で TCP/IP が使えるため、通常の分散環境としても利用できる。並列実行環境としては、AP/Linux 独自で通信ライブラリと gang scheduling を提供している。

[1]Linux on the Fujitsu AP1000+
http://cap.anu.edu.au/cap/projects/linux/

4.1 並列プログラミング環境

現在の AP/Linux では、並列プログラミングのために MPI ライブラリ [3] と AP1000+ が独自に提供するメッセージ通信ライブラリを提供している。並列プログラムはこれらの提供された関数を使い、T-net を通して cell 間のメッセージ転送やリモートメモリアクセスを行なう。この cell 間の通信にはユーザ空間のみで実現され、システムコールを必要としない。このため cell 間で高速なメッセージのやり取りを可能にしている。また、受け取り側のメッセージの検出方法に polling 方式、signal 方式、polling/signal 方式を提供し、プロセスの負荷状況ごとの効率が確かめられている。

4.2 Gang Scheduling

AP/Linux は並列プロセスを実行するために簡単な Gang Scheduling を提供している。この Gang Scheduling は、ユーザが並列実行を要求する *prun* コマンドと並列プロセスの生成／消滅管理を行なう *paralleld* デーモンと *kernel* の並列スケジューリングによって実現されている。

4.2.1 並列プロセスの生成／消滅

図 3に prun コマンドを用いて並列プロセスを実行する例を示す。図中の丸は cell を表し、各 cell には kernel と paralleld が存在する。prun コマンドは -n オプションで必要とする cell 群と実行する並列プロセスを引数とする。purn コマンドはどの cell からも実行することができる。図中では、cell 5 からは task1 処理を 2×2 の cell 群で実行することを要求している。cell 3 からは task2 処理を 3×3 の cell 群で、cell 2 からは task3 処理を 1×3 の cell 群で実行することを要求している。

これら prun コマンドは、cell0 上の paralleld に並列プロセスのための cell 確保とその生成を依頼する。paralleld はそれぞれの prun コマンドが要求した cell 群を cell0 から始まる cell 領域に割り当てる。図 4に図 3の並列プロセスの割り当てを示す。cell0 の paralleld は割り当てられた cell 上の paralleld に並列プロセスの生成を依頼する。

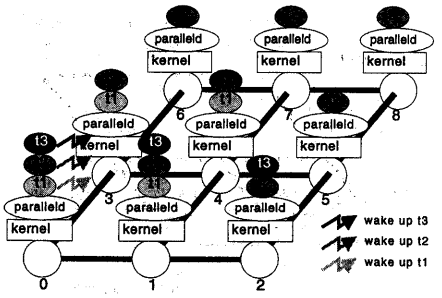

図 4: Gang scheduling によるプロセス配置

各 cell 上で paralleld が並列プロセスを生成するには、POSIX の *clone* 関数の改良版を使用する。この clone 関数により、kernel の外部からプロセス ID(PID) とタスク ID(TID) を統一することができる。並列プロセスの PID は PID の上位半分を利用し、逐次プロセスは下位半分を利用する。各 cell 上で生成された並列プロセスはそれぞれのローカルキューに入れられる。

また、paralleld は prun と並列プロセス間の標準入出力を中継する。prun が実行されている端末からの入力は paralleld を通して送られ、逆の並列プロセスからの出力も paralleld を通して端末に送られる。

並列プロセスが終了するとそのプロセスは終了のメッセージを paralleld に送る。これを受けとった paralleld は prun と並列プロセスの標準入出力セッションを切り、prun に並列プロセス終了のメッセージを送る。prun はこのメッセージを受けとることでその役割りを終える。

4.2.2 並列プロセスの同期

各 cell 上で処理される並列プロセスは、他の cell 上の並列プロセスと通信を行なう際に同期を取った方が効率が良い。AP/Linux の通信ライブラリのメッセージ交換の効率については論文 [2],[3] に詳しい。この中で、並列プロセスの多重環境では、並列プロセス間のメッセージ待ちの時間を解消するため cell 間に分散した並列プロセスを同期して起こす (wake-up 同期する) 必要性が述べられている。

AP/Linux の gang scheduling では並列プロセスの割り当てが必ず cell0 から始まるため、cell0 上のローカルキューがすべての並列プロセスを持つ。並列プロセスの wake-up 同期はこの cell0 上のローカルキューに依存する。cell 0 上のローカルキュー中の並列プロセスが次の実行対象となった際に、kernel は全 cell にこの並列プロセスが起動されることを知らせる。この様子を図 4の例で示す。cell0 上の並列プロセスが起動される直前に wake-up 同期がかけられる。wake-up 同期では、対象となる並列プロセスの TID を全 cell に割り込みで渡し、各 cell 上で該当するプロセスがあればプロセス切り替えを行なう。

各 cell 上で wake-up 割り込みを受けた際、その割り込み処理は緩やかに行なわれる。Sparc/Linux では例外処理に対して generic handler と fast handler が用意されている。generic handler は割り込みが発生した際にすぐにその処理を行なう。fast handler は割り込みが発生した際にその原因だけ突き止め、処理は現在処理中のプロセスを切り替えた後に行なう。プロセス切り替え後に行なう処理はボトムハーフと呼ばれる。AP/Linux での並列プロセスの wake-up 同期は fast handler で実現している。

cell0 から wake-up 同期割り込み受けた cell はローカルキューに対象となる並列プロセスが存在するか fast handler で調べる。存在する場合には現在処理中のプロセスの切り替え後、ボトムハーフで次の実行時間の割り当てはその並列プロセスにするようにスケジューリングする。つまり AP/Linux の wake-up 同期はハードウェアで割り込みが発生した後すぐにプロセスが切り替えられるのではなく、現在処理中のプロセスの終了を待って行なわれる比較的緩やかなものである。

また通常の並列計算機では、プロセス切り替えを起こした際にネットワーク上のメッセージの切り替えが問題になる。CM5[4] や RWC で作成されている myrinet を使ったクラスタマシン [5] では、ネットワークがプロセッサ上のプロセス切り替えと同期し、ネットワーク上のメッセージもスイッチにキューイングされて切り替えられる。しかし AP/Linux の場合、ネットワーク上のメッセージは cell 上のプロセス切り替えの影響を受けない。なぜならメッセージのヘッダに並列プロセスの TID が付けられ、転送先のプロセスが切り替わっても宛先が特定できるためである。メッセージはユーザ空間にバッファリングされる。このメモリ空間がスワップアウトしていた場合、そのメモリ空間を MMU により実メモリにマッピングし、メッセージを書き込む。

さらに AP/Linux の gang scheduling では、wake-up 同期が起動されなくても各 cell で負荷が低くて並列プロセスがローカルスケジューリング可能ならば、並列プロセスは個々の cell で処理を進めることができる。例えば図 4では、cell0,1,2 で task3 が同期処理されている間に、cell2,4 で task1 か task2 が処理可能である。しかし task1 も task2 も同期処理されていないため、どちらのプロセスが実行されるかはローカルスケジューリングに依存する。

AP/Linux の gang scheduling では、cell0 がすべての並列プロセスを保持するため、cell0 に割り当てられた並列プロセスの処理が通常もっとも遅い。AP/Linux では、この最も緩やかに進む cell0 上のスケジューリングに従って wake-up 同期が取られる設計になっている。

5 時分割と空間分割の設計

AP/Linux の gang scheduling は、我々の方式を実装するのに幾つかの欠点がある。まず、並列プロセスの割り当てを cell0 から行ない、空間分割が不可能なことである。これでは並列計算機のプロセッサ空間を有効に利用できず、何の処理も行なわない無駄なプロセッサを生じ

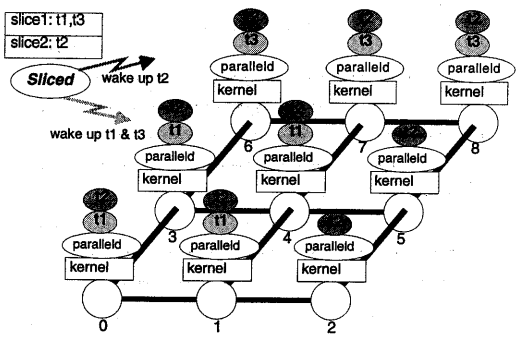

図 5: sliced によるプロセス配置

てしまう。さらに cell0 近辺に並列プロセスが集中し、負荷が偏ってしまう。

また、並列プロセスの wake-up 同期も cell0 のプロセスキューに依存しているが、これでは空間分割して並列プロセスを割り当てることができない。wake-up 同期は個々のプロセスキューから独立して行なう機構が必要である。

我々の設計では、AP/Linux の gang scheduling の機能を活かしつつ、 時分割と空間分割を融合したスケジューリングを実現するようにした。このため、新たにプロセスの割り当てと wake-up 同期を管理するデーモン (*sliced*) を加える。ここでは、paralleld からプロセス割り当ての機能を、kernel から wake-up 同期を行なうタイミングの機能をそれぞれ sliced に移す。sliced は並列計算機全体で一つ存在すればよく、どの cell に置かれても構わない。

5.1 プロセス割り当て

並列プロセスに割り当てる cell 空間は sliced によって管理される。cell0 の paralleld が prun コマンドから並列プロセス生成の要求を受けた際には、sliced に割り当て可能な cell 領域探索を依頼する。sliced は仮想並列計算機であるスライスを考慮したパーティショニングアルゴリズムにより、割り当て可能な領域を探す。ここでは、我々が提案したスケジューリングと同様にスライス間に跨る multiple task も存在する。sliced で割り当て領域を決定した後、結果を cell0 の paralleld に返す。並列プロセス生成は gang scheduling と同一の方法で各 cell 上の paralleld が行なう。

図 5に図 3と同じ prun コマンドを受けた際の割り当て例を示す。ここではスライスが二つ生成され、一つのスライスには task1 と task3 が割り当てられ、もう一つのスライスには task2 が割り当てられる。

並列プロセス終了時に並列プロセスから paralleld に終了メッセージが送られる。このメッセージを silced にも転送する。sliced では並列プロセスの終了を知らされるとスライスからそのプロセスを削除する。また、プロセスを削除したスライスが必要であるか、multiple task になれる並列プロセスがあるかなど、スライス内で cell をを効率的に利用できるプロセスの組み合わせを再計算をする。

5.2 wake-up 同期

並列プロセスの wake-up 同期は cell0 のプロセスキューから独立させ、sliced から一定時間毎に割り込みをかけるようにした。sliced のスライスのデータにより、並列プロセスがどこに割り当てられ、どの並列プロセスが複数同時に起動できるかが分かっている。同時に起動できる並列プロセスは一つのスライスに存在しているプロセス群である。図 5の例では task1 と task3 が同時に起動可能である。

一つのスライスにどれくらいの間隔で wake-up 同期をかけるかは実装に依存する。また、wake-up 同期は kernel 内で提供していたものを流用する。この wake-up 同期を呼び出すためのシステムコール (mpp_schedule) を一つ付加した。

このシステムコールでは同時に複数の並列プロセスが呼び起こすため、kernel 内の wake-up 同期割り込みを若干変更した。gang scheduling では一回の割り込みで一つの TID しか渡せなかったが、変更後では一回の割り込みで複数 PID を渡せるようにした。従来の割り込みでは TID を渡していたが、kernel 外部からは PID による制御の方が簡単なので PID を渡すように変更した。この wake-up 同期では、各スライス上にある並列プロセスの PID すべてを全 cell に割り込みで渡し、該当するプロセスがあればプロセス切り替えを行なう。

6 現在の実装

現在、AP1000+ の 16 台 cell で AP/Linux が稼働している。sliced はどの cell 上にでも配置可能だが、現在の実装では cell0 上の paralleld と通信し合うため、cell0 に配置した。この sliced はまだ二次元のパーティショニングアルゴリズムを実装していない。現在の gang scheduling でも prun での二次元の cell 形状要求は cell 数のみに変換されている。現在の sliced は一次元の First Fit のパーティショニングを有する。sliced では一次元のパーテイショニングに基づいて multiple task の機能も有する。

各 cell 上のプロセスの tick time は 16.7(1/60)ms とし、sliced が wake-up 同期を行なうのが 100ms 間隔とした。つまり逐次プロセスやプロセス切り替えのオーバーヘッドを考慮しなければ、約 6 回 tick time が経過するうちに 1 回の割合で同期がかかる。

6.1 性能評価

実装したスケジューリングを評価するため、複数のプロセスを投入し、実行状況を確かめた。投入するプロセスは必要とする CPU 処理時間を同一とした。

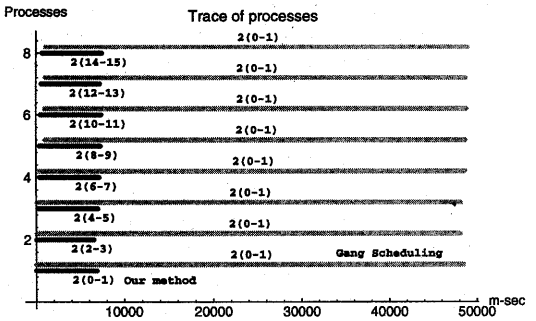

図 6: プロセスが処理される過程：プロセス処理開始時間から処理終了までを示す。

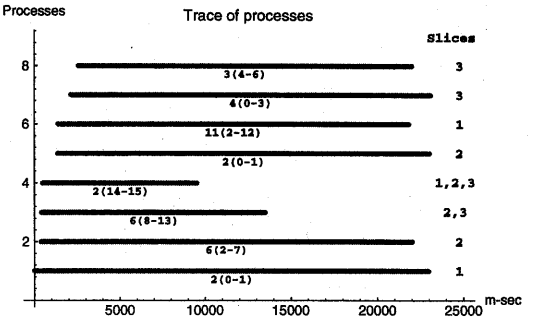

図 7: プロセスが処理される過程：cell 数をランダムに変化させた場合

図6では、2つの cell を要求するプロセスを8個投入し、その処理時間を調べた結果を示す。このグラフでは gang scheduling と我々の方法で、プロセスの処理を開始した時間と終了した時間（経過時間）を示す。グラフ内の数字は要求された cell 数と括弧内に配置 cell 領域を示す。この場合の cell 要求パターンは、我々の方法では各プロセスが全 cell に分散し、空間的に最も効率良く cell を利用できる例である。

結果のグラフより、我々の方法では各 cell に1個の並列プロセスが配置され、gang scheduling では cell0,1 に8個のプロセスが配置されていたことがわかる。このため、paralleld, sliced 等のシングルプロセスのオーバーヘッドを考慮しなければ、我々の方式は gang scheduling で処理されるプロセスの経過時間の 1/8 で終了できる計算である。実際に個々のプロセスの経過時間を調べると平均で 1/7.1 となっており、この処理ではオーバーベッドより空間分割の効果が大きいことがわかった。

またプロセス開始時間を比べると、我々の方式では空間的な負荷分散を実現できたため、スケジューリングを行なう cell0 に負荷が集中せず、プロセス開始時間が早くなることがわかった。負荷が軽い初期は gang scheduling の方が実行開始時間が早いが、複数をプロセスを実行し負荷が重くなるとこれらの処理に時間が取られ、開始時間が sliced による割り当てよりも遅くなっていた。

図7では、cell 数をランダムに要求するプロセスを8個投入し、その経過時間を調べた。このグラフでは我々の方法でのプロセスの経過時間のみを示す。投入したプロセス群を処理するために3つのスライスが作成され、プロセス3と4が multiple task になったことがわかった。この二つのプロセスより、multiple task の効果があったことがが確かめられた。プロセス3は3つのスライスのうち2つのスライス上に存在し、その経過時間が single task の約 2/3 であったことがわかった。また、プロセス4は3つのスライスのすべてに存在でき、経過時間が single task の約 1/3 であったことがわかった。

7 おわりに

本論文では、時分割と空間分割を融合したスケジューリングを並列計算機 AP/1000+ 上で実装する設計例の一つを示した。この設計では AP/Linux を用い、ここで実現されている gang scheduling の機能を利用した。我々の設計では、並列プロセスを空間的に割り当てるパーティショニングと wake-up 同期を行なうデーモン (sliced) を提案し、時分割と空間分割を可能にした。また、実際の性能を幾つかのプロセスを走らせることで確かめた。現在の解析はまだ不十分なものであるが、sliced により空間的に複数のプロセスが割り当てられ、シミュレーションで得ていた結果とほぼ同じであった。スケジューリングのオーバーヘッドも従来の gang scheduling と比較しても大きくなかった。むしろ空間的な負荷分散を実現できたため、スケジューリングを行なう cell に負荷が集中せず、高速な割り当てが可能であった。今後 wake-up 同期のオーバーヘッドや sliced での領域探索の時間などの詳細な性能測定をすると共にアプリケーションによる wake-up 同期の最適頻度の調節を行なう予定である。

参考文献

[1] 須崎, 田沼, 平野, 一杉, 塚本. メッシュ結合並列計算機用パーティショニングアルゴリズムの時分割処理化. 情報処理学会論文誌, 37(7):1332–1343, 1996.

[2] A. Tridgell, P. Mackerras, D. Sitsky, and D. Walsh. AP/Linux a modern os for the ap1000+. *The 6th Parallel Computing Workshop*, pages P2C1–P2C9, 1996.

[3] D. Sitsky, P. Mackerras, A. Tridgell, and D. Walsh. Implementing MPI under AP/Linux. *Second MPI Developers Conference*, pages 32–39, 1996.

[4] *Connection Machine CM-5 Technical Summary.* Thinking Machines, 1992.

[5] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of gang-scheduling on workstation cluster. *Lecture Notes on Computer Science*, pages 173–182, 1996.