システム評価 1−5
（2001．6．15）

並列プログラムの並列度を自律的に制御するアクティブスケジューラの実装と評価

田頭 茂明　　Deng Lei　　伊東 靖英　　藤田 聡
広島大学大学院 工学研究科

我々は，様々なユーザが並列プログラムを実行する並列分散システムにおいて，他のジョブの実行を妨げず，多くの計算機資源を利用してジョブを実行するアクティブスケジューラの構築を目指している．アクティブスケジューラは，他のジョブの実行を妨げず効率良くジョブを実行するために，実行環境に応じてジョブの優先度の自律的な制御と，計算機の負荷分散を行う．ジョブの優先度制御は，そのジョブの並列度を制御することによって実現している．本稿では，提案するアクティブスケジューラのプロトタイプシステムを実現し，様々な条件のもとで，異なるユーザのジョブを同時に実行した場合の性能を評価した．結果から，アクティブスケジューラを用いることで，オリジナルの実行時間の 15%のオーバヘッドで並列プログラムの実行を制御できることを示した．

Implementation and Evaluation of an Active Scheduler for Controlling Concurrency of Parallel Programs

Shigeaki Tagashira, Lei Deng, Yasuhide Ito, and Satoshi Fujita
Graduate School of Engineering, Hiroshima University

We propose a new scheduling method that can simultaneously achieve two main goals of task scheduling in distributed parallel systems; i.e., to minimize the execution time of a parallel job without disturbing the execution of other jobs. We challenge to achieve those goals by introducing a new scheduler, called active scheduler, that controls the priority of parallel programs dynamically and balances the workload of computers, depending on the current status of runtime environment. Priority of parallel programs is controlled by controlling the concurrency of the programs. In this paper, we describe the implementation of a prototype system and evaluate the effectiveness of active scheduler under various conditions. The results of experiments imply that the overhead of introducing active scheduler is bounded by 15% of the original execution time, and it is in fact effective to adjust the execution of parallel programs to an actual distributed parallel processing environment in which many users execute their jobs at the same time.

1　はじめに

近年，多くのアプリケーション分野において，高速ネットワークで複数の計算機を接続する分散システムの重要性が増している．分散システムでは， 数人のユーザによって様々な計算機資源を利用するジョブが発行される．例えば， 大規模な計算機資源を必要とする科学計算プログラム，図面を作成するための設計支援ツール，科学技術論文を書くための文書処理プログラムなどがある．このような分散環境における重要な課題は，通信ネットワーク上で分散される種々雑多な計算機資源を効率良く利用するタスクスケジューラの構築である．スケジューラの主な役割は，分散システムを構築する計算機の負荷が均一となるように，計算機にタスクを割り当てることである．

過去 30 年間，分散システムにおける負荷分散に関する様々な手法が提案されている [1, 2]．しかしながら，単一のユーザで単一のジョブを実行する専用の並列計算機環境を想定しているために，異なる組織の多くのユーザによって共有される一般的な分散システムへ，多くの手法を直接適応することができない．

大規模な分散システムにおいては，異なる組織のユーザにより計算機が保持／管理される．このような環境では，他のユーザのジョブ（*Guest* ジョブ）より高い優先度で計算機を所持するユーザのジョブ（*Native* ジョブ）を実行するスケジューラが要求されている．このため，ここ数年間では，*Native* ジョブの実行を妨げず，*Guest* ジョブを実行するスケジューラの研究が広範囲にわたり行われている [3, 4, 5, 6]．例えば，文献 [7] において Tandiary は遠隔の計算機で *Guest* ジョブを実行する Batrun システムを提案している．このシステムでは，*Guest* ジョブが動作する計算機のキーボードやマウスなどの入力デバイスを監視する．入力デバイスからの入力を感知したとき，その計算機から他の計算機へジョブを移動することで，*Native* ジョブの優先実行を実現している．これらの多くのシステムでは，計算機の完全な個人利用を実現するために，入力デバイスの動きを感知すると *Guest* ジョブをすぐに他の計算機へ移送する．しかしながら，低優先度の *Guest* ジョブの実行を許容できるユーザの存在と，ジョブ移送のための高いオーバヘッドを考えると，*Guest* ジョブの優先度を低く設定し，その実行を継続する方がより効率的に *Guest* ジョブを実行できる．

このため本研究では，並列プログラムを対象とした

アクティブスケジューラを開発している [8]．アクティブスケジューラは，*Native* ジョブの実行を妨げることなく，出来る限り多くの計算機資源を利用することで，*Guest* ジョブのスループットの向上を図る．アクティブスケジューラは，ジョブの優先度として並列度を利用し，実行環境に応じて並列度を自律的に変更する．これにより *Guest* ジョブは，*Native* ジョブの実行を妨げず，低い優先度での実行を継続する．また，性能が高く負荷が軽い計算機へ，*Guest* ジョブを重点的に割り当てることで，より多くの計算機資源を利用する．

本稿では，提案するアクティブスケジューラの実装と評価について示す．我々は，実際の分散システム上でアクティブスケジューラのプロトタイプシステムを実現し，様々な条件のもとで，*Native* ジョブと *Guest* ジョブを同時に実行した場合について評価する．評価から，*Native* ジョブと *Guest* ジョブに対する影響を確認し，アクティブスケジューラの有効性を示す．

2 アクティブスケジューラ

本章では，提案するアクティブスケジューラの概要について述べる．

2.1 並列度を用いた優先度制御

単一プロセッサ上において，タスクの並列度を増やすことで実行時間が削減できる．マルチタスク環境では，実行キューで待つタスクは，CPU 時間をほとんど平等に割り当てられる．これから，あるジョブに割り当てられたタスク数が $m1$ から $m2$ に増え，キューにある他のジョブのタスクが x なら，そのジョブに割り当てられる CPU 時間の割合は，$\frac{1}{1+x/m1}$ から $\frac{1}{1+x/m2}$ に増える．タスク数が増えるとジョブは明らかに多くの CPU 時間を割り当てられ，実行時間を短縮できる [8]．以上のことから，並列度を増加させることでジョブの優先度を高く設定でき，並列度を減少させることでジョブの優先度を低く設定できる．アクティブスケジューラでは，この並列度を用いた優先度制御方式を採用している．

しかし，あるジョブの並列度が増加すると，その他のジョブの実行時間も増加する．その他のジョブの実行を妨げないことを目的とするアクティブスケジューラでは，ジョブの並列度を現在の状況に応じて適切に制御する機構が必要である．この機構を実現する重要な項目は，現在のシステムの負荷状況を表す適切な *Workload Descriptor* を見つけることである．

2.2 System Workload Descriptor

UNIX システムでは，*Workload Descriptor*(WD) として利用できるいくつかの統計情報を利用できる．例えば，OS 内のユーザタスク数，物理メモリの空き容量，コンテクスト切替の頻度，システムコールの頻度，1 分間の CPU 利用率や，CPU アイドル時間などである．我々は，適切な WD を見つけるために，これらの統計情報とジョブの実行時間との関係を実験により調べた [8]．結果から，実行キュー内にあるタスクの長さ (LRQ) と，物理メモリの空き容量のサイズ (SAM) が実行時間に大きく影響し，システムの負荷を適切に表現する重要な情報であることがわかった．LRQ は，多くのタスクがキュー内に存在すると，あるタスクに割り当てられる CPU 時間が少なくなり，応答時間やスループットが悪化する．SAM が閾値より低くなると，ディスクスワップの頻度が多くなり，実行時間が増加する．

以上のことから，提案システムでは，Primary WD として LRQ を，Secondary WD として SAM を採用している．

2.3 システム構成

提案システムにおいて，我々は共有ファイルシステムを持つクライアント・サーバモデルを採用している．また，利用する並列プログラムは，デットロックを避けるために，並列プロセス間で順序制約がないと仮定している．我々は，アクティブスケジューラをサーバホスト上で稼動する Global Scheduler と，個々のクライアント上で稼動する Local Scheduler で構成している．

Local Scheduler は，システムの現在の状態に応じて並列プログラムのタスクの並列度を制御する．これは，タスク状態を変更することで実現する．我々は，OS のタスク状態とは異なり，Local Scheduler 内で以下のタスク状態を定義している．

実行可能タスク (RT) 実行できる準備が整った状態のタスクのことである．実際には実行されていない．

実行タスク (AT) 現在，実行されている状態のタスクである．

待機タスク (ST) 実行タスクが一時中断した状態のタスクである．

また，Local Scheduler は，これらのタスクを管理するために，RT を管理する実行可能キュー (RQ)，AT を管理する実行キュー (AQ)，ST を管理する待機キュー (SQ) を持つ．

並列度の制御手順は，WD の値を，あらかじめ決定された閾値を超えないように，AT の数を自律的に制御する．具体的なアルゴリズムについて説明する．ホスト v の LRQ 値と SAM 値を，LRQ_v と SAM_v とする．$T_L(v)$ と $T_S(v)$ は，ホスト v における LRQ と SAM の閾値とする．閾値については 2.4 節で説明する．これらの値を使って，Local Scheduler は以下のように振舞う．

以下の操作を一定間隔で繰り返す．

Case 1 $LRQ_v < T_L(v)$ *and* $SAM_v > T_S(v)$ の条件を充たすなら以下の操作をする．

(1)SQ が空でないなら，一つの ST を選択し，AT へ状態を変更する．

(2)SQ が空ならは，RQ から一つの RT を選択し，AT へ状態を変更する．

Case 2 $LRQ_v > T_L(v)$ の条件を充たすならば，スケジューラは $|LRQ_v - T_L(v)|$ の任意の AT を選択し ST へ状態を変更する．

Global Scheduler の役割は，サーバ上の RT をクライアントへ AT として公平に分配することである．提案システムでは，クライアントの性能差を考慮するために，各ホストの性能を正規化した Abstarct Performance Index を導入している．この Index は，タスクを移動する際に適切なホストを見つけるために利用される．ここでの適切なホストとしては，低い性能の計算機より，高い性能の計算機としている．Global Scheduling の手順は，以下のようになる．

前処理 $L = (l_1, l_2, , , l_n)$ は，各ホストの性能を表すリストである．リストは，システムを構成するときに計算され，ファイルに保存されている．

クライアント クライアント v の WD が，あらかじめ設定された閾値を下回った場合，すなわち $LRQ_v < T_L(v)$ *and* $SAM_v > T_S(v)$ の条件を充たすとき，クライアントはサーバへ通知する．

サーバ サーバ u の WD が閾値を超えた場合は，すなわち，$LRQ_u > T_L(u)$ *or* $SAM_u < T_S(u)$ を充たし，RQ が空の場合では $LRQ_v < T_L(v)$ *and* $SAM_v > T_S(v)$ を充たすクライアント v を L のリストから探す．そのようなホストがない場合，タスクの移動せずに Global Scheduler は終了する．

その他は，$|LRQ_u| - T_L(u)$ のタスクをクライアント v に移動する．

2.4 閾値の動的な決定方式

アクティブスケジューラの性能は，閾値の値が大きく影響する．本節では，WD の閾値について議論する．

SAM の閾値 (T_S) については，実験の結果 [8] より物理メモリの 10%と設定した．LRQ の閾値 (T_L) では，周期的に CPU の利用率を計測して動的に更新するアプローチをとった．この理由としては，閾値 T_L はユーザタスクによる CPU の利用時間と CPU のアイドル時間が影響し，動的に変化するからである．例えば，並列プログラムによる CPU 利用率が 10%の場合を考える．もし CPU アイドル時間が 50%の場合は，CPU 利用率を 30%にするために閾値を増やす必要がある．しかしながら，CPU アイドル時間が 1%の場合は，同じホスト上で沢山の他のタスクがあることを示している．このため，CPU の利用率を 5%に制限するために閾値を減らす必要がある．

具体的な決定方式について説明する．アクティブスケジューラで実行される並列プログラムの CPU 使用率を U_p とする．U_f は，CPU のアイドル時間とする．スケジューラにより，U_p と U_f は，定期的に計測される．β_p と β_f は，U_p と U_f の予め設定された下限値である．そして，α_p は，予め設定した U_p の上限値である．例えば，$\alpha_p = 90\%, \beta_p = 10\%,$ and $\beta_f = 20\%$のように設定する．加えて，我々は，T_L の無制限な増加を防ぐために，T_L の上限値 T_u を設定する．これらの値を用いて，T_L の値を以下の用に制御する．

Case 1 $\beta_f < U_f$ *and* $T_L < T_u$ の条件を m 回連続して充たすか，$U_f < \beta_f$ *and* $\beta_p < Up < \alpha_p$ *and* $T_L < T_u$ の条件を m 回連続して充たすなら，T_L を一つ増やす．

Case 2 $U_f < \beta_f$ *and* $\alpha_p < U_p$ の条件を n 回連続して充たしたなら，$T_L = 2$ にする．

ここで，m と n は $m > n$ を満足する．

3 アクティブスケジューラの実装

本章では，アクティブスケジューラのプロトタイプシステムの実装について示す．プロトタイプシステムにおいては，Local Schedule として稼動するデーモンプロセス (d-process) と，Global Schedule として他の計算機と通信する通信プロセス (c-process) から構成される．図 3 はプロトタイプシステムの概要を示す．我々は，プラットフォームを，100M Ethernet で接続されたファイルシステムを共有する計算機（OS は FreeBSD3.4）で構成している．また，一つのタスクは，一つのプロセスとして実行する．

3.1 Local Scheduling by d -Processes

d-process は自身のホストの WD を定期的に計測し，その計測した情報をホスト上に蓄積する．また d-porcess は，signal システムコールを使って，ユーザ定義タスクの状態を制御する．各ユーザ定義タスクは，実行前に sleep シグナルハンドラが設定される．d-process から sleep シグナルを受け取ると，ユーザ定義タスクは sleep ハンドラが実行される．ハンドラ内では，次の wakeup のために wakeup シグナルハンドラを設定し，pause サブルーチンを呼ぶことによって，ST の状態へ遷移する．その後，wakeup シグナルを受け取ると，そのユーザ定義タスクは手続きを中断した地点から実行を再開する．

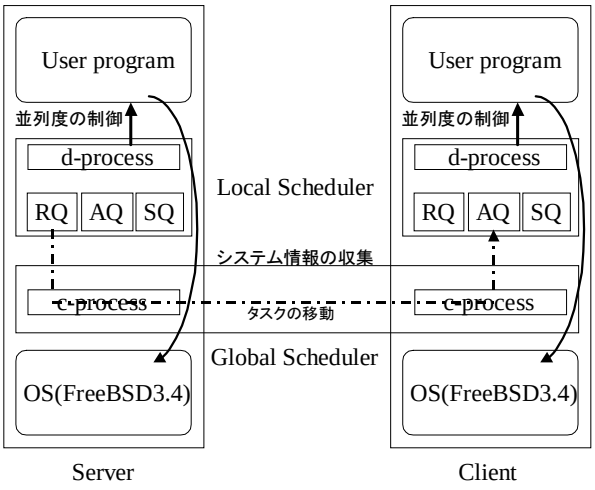


図 1: プロトタイプシステムの概要

3.2 Global Scheduling by c -Processes

c-process の役割は，他の計算機とのコネクションの確立とそれを管理することである．c-process は，他のホストの WD 情報を使って適切な遠隔ホストを選択する．d-process は，新しいユーザ定義タスクを作るために，決定した遠隔ホストに要求をする．呼び出しの受入れについては，遠隔ホストの d-process によって WD を考慮し決定される．

具体的には，サーバと各クライアントの間にコネクションが確立したとき，各ホスト上の c-process は次のように振舞う．

システム情報の収集 クライアントの c-process は，定期的に WD を Performace Notice メッセージによってサーバへ通知する．このメッセージを受け取った後，サーバはホスト情報を更新する．

遠隔タスク呼び出し クライアント上にタスクを作成したい場合，サーバは CreateProcessRequire メッセージを indexOfUDP パラメータと一緒にクライアントへ送信する．indexOfUDP は，タスクの番号を示す．クライアント上の d-process はそのメッセージを受け取った後，その呼び出しを受け付けるかどうかを決定する．もし，受け付けるなら，対応するタスクの実行を開始し，CreateProcessConfirm メッセージをサーバへ返す．もし，受け付けを拒否するなら，CreateProcessReject メッセージをサーバへ返す．本プロトタイプにおいて，RQ をすべてのホストで共有オブジェクトとして実現している．従って，タスクの移送は，ユーザ定義手続きの indexOfUDP を送信するだけ良い．

すべてのユーザ定義タスクが完了したとき，サーバは DisconnectRequest メッセージを各クライアントへ送る．このメッセージを受け取ったあと，クライアントは DisconnectionConfirm メッセージをサーバへ返し，実行を終了する．

表 1: 評価で用いた計算機の一覧

ホスト名	CPU	メモリ
Host 1	Pentium II 233 MHz	196 MB
Host 2	Pentium II 233 MHz	64 MB
Host 3	Pentium 200 MHz	64 MB

表 2: SORT プログラムの実行時間の結果

Case	実行時間 (sec)
A	221.680
B	187.414
C	233.500

4 評価

我々は，プロトタイプシステム上でいくつかの実験によって，提案方式の有効性を評価する．本実験で用いたホストを表 1 に示す．

閾値 T_L を決定するために利用するパラメータを $\alpha_p = 90\%$, $\beta_p = 10\%$, $\beta_f = 10\%$, $m = 10$, $n = 2$ とした．また各ホストの Abstract Performance の値を，Host1 と Host2 は 6 に設定し，Host 3 は 5 に設定した．

第一の実験として，サーバとして Host 1 を，クライアントとして Host 2 を利用する環境で Local Scheduler の性能を計測した．実験環境としては，3 つの異なる条件で 100 の並列度を持つ並列のソートプログラム（SORT）を実行し，SORT の実行時間を計測する．CASE A では，SORT プログラムの 100 の並列度を逐次に実行する．CASE B と CASE C では，SORT プログラムをアクティブスケジューラ上で並列に実行する．また，CASE B では 2 つのホスト上にユーザはログインしていない．CASE C では，他のユーザがログインしており，WD は動的に変化する．表 1 に結果を示す．CASE B では，SORT プログラムの実行時間を CASE A と比較して短縮できている．これは，アクティブスケジューラが SORT プログラムの並列度を自律的に制御し，並列に SORT プログラムを実行したためである．CASE C では，SORT プログラムの実行時間が CASE B と比べて増加している．これは，ログインしているユーザを妨げないように，アクティブスケジューラが動作しているからである．

次の実験として，アクティブスケジューラを導入することによる *Native* ジョブへの影響を調査した．アク

表 3: コンパイルプログラムの実行時間の結果

Machine	ORG (sec)	AS (sec)	slow-down rate
Host 1	2.264	2.616	15.5%
Host 2	2.171	2.304	6.1%
Host 3	2.543	2.746	8.0%

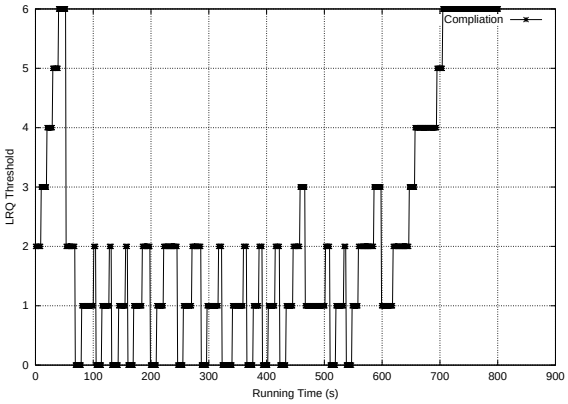
表 4: SORT プログラムの実行時間の結果

アプリケーション	ORG (sec)	+AS (sec)	AS (sec)
エンコーディング	341.22	351.51	782.09
コンパイル	302.68	409.49	716.30

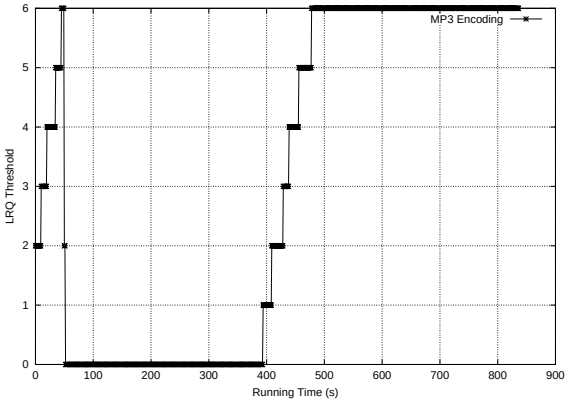
ティブスケジューラを利用した場合と利用しない場合の 2 つの条件で，コンパイルプログラム（*Native* ジョブ）の実行時間を 3 つのホスト上で計測した．*Guest* ジョブとしては，SORT プログラムを実行している．コンパイルプログラムは逐次に実行される．表 3 に結果を示す．ここで，ORG はオリジナルの実行時間を示し，AS はアクティブスケジューラを用いた場合の実行時間である．表からアクティブスケジューラによる実行時間が若干増加してことがわかるが，最大でもオリジナルの実行時間の 15%である．アクティブスケジューラが，*Native* ジョブの実行時間を妨げず，*Guest* ジョブを実行できることが確認できる．

最後の実験として，典型的な 2 つのアプリケーションプログラムに対するアクティブスケジューラの影響を調査する．利用したアプリケーションは，MP3 エンコーディングと FreeBSD のカーネルコンパイルである．MP3 エンコーディングは，データの読み込み，計算，データの書き込みのフレーズを逐次で実行する．FreeBSD のカーネルのコンパイルは，データの読み込みと書き込みの I/O 操作がランダムで頻繁に発生する．*Guest* ジョブとして，SORT プログラムを実行した．Host 2 を用いてアクティブスケジューラを用いた場合 (+AS) と，並列に実行するがアクティブスケジューラを用いない場合 (AS) で，アプリケーションの実行時間を計測した．結果を表 4 に示す．表からアクティブスケジューラの利用が，*Native* ジョブを妨げないという目的を達成している．アクティブスケジューラを利用せずに SORT を実行すると，400 秒以上もアプリケーションの実行時間が増えている．つまり，*Native* ジョブの実行を妨げている．アクティブスケジューラを利用すると，アプリケーション実行時間の増加を抑制できている．

ここで，注目する点は各アプリケーションで実行時間の増加する割合が異なっている．すなわち，MP3 のエンコーティングでは 10 秒程度の増加であるが，カーネルコンパイルでは，アクティブスケジューラを用いても 100 秒も増加している．



(a) カーネルコンパイル



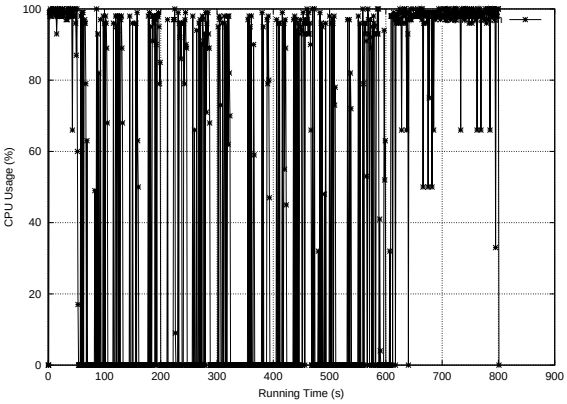
(b)MP3 エンコーディング

図 2: 閾値の遷移

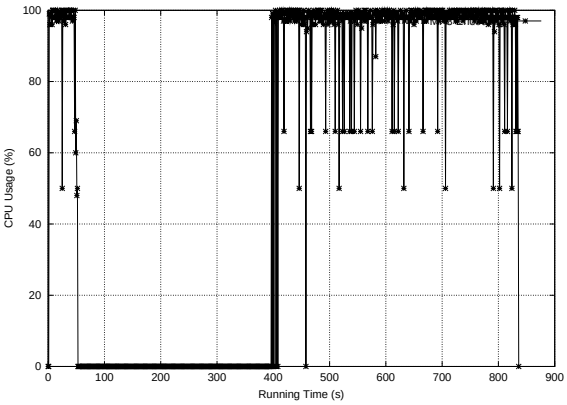
この現象は，次のように説明できる．I/O 操作間は，CPU アイドル時間が増加するために，閾値 T_L の値が連続的に増える．すなわち *Guest* ジョブの並列度が増加する．このため，I/O 操作からコンパイルに戻ったとき，コンパイルのための CPU 資源が残されていない．これからすべての I/O 操作毎に，コンパイルに割り当てられる CPU 時間が短くなる．

この現象を調べるために，各アプリケーションにおける閾値と CPU 利用率の遷移について調べた．図 2 は，各アプリケーションの閾値の遷移を示している．図 3 は，*Guest* ジョブの CPU 利用率の遷移を示している．これらの図から，上記現象が実際に発生していることがわかる．閾値の連続的な増加がカーネルコンパイルを実行を妨げている．

この問題は，3.3 節で述べた繰り返し回数 m を適切に調整することで解決できる．図 4 は，アプリケーションプログラムの実行時間に対する m の影響を示す．図から我々は，m を増やすことで，カーネルコンパイルの実行時間を短縮できることがわかる．これは，適切に m を増やすことで，I/O 操作で発生する問題を解決できることを意味している．



(a) カーネルコンパイル



(b)MP3 エンコーディング

図 3: CPU 利用率の遷移

5 おわりに

本稿では，並列度を制御することで並列プログラムの優先度を制御するアクティブスケジューラの概要について述べ，その実装と評価について述べた．提案システムのプロトタイプを FreeBSD3.4 上で実装し，幾つかの実験は，実際の環境で提案システムの有効性を示した．実験の結果から，アクティブスケジューラを用いることでオリジナルの実行時間の 15%のオーバヘッドがあることを示した．そして，多くのユーザがジョブを同時に実行する現実の分散環境で並列プログラムの実行を調整する有効性を示した．また，I/O 操作によるアクティブスケジューラの問題とパラメータとの関係について述べた．

今後の課題としては，提案システムで利用する閾値決定パラメータの決定方法がある．例えば，第 5 章の最後実験で，閾値の更新のタイミングに関するパラメータ m などである．また，これらのパラメータを自律的に調整する方式が必要である．さらに，本システムを利用するために必要なユーザインタフェースの開発を検討する必要がある．

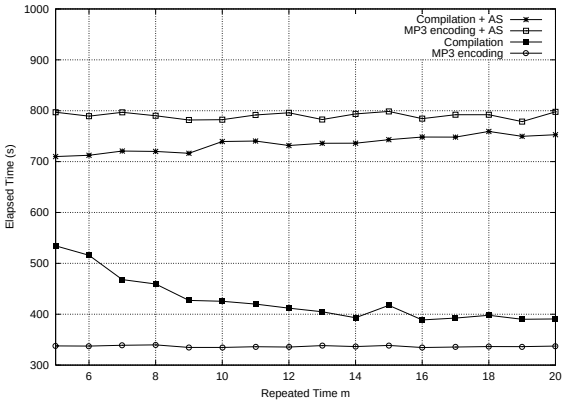


図 4: アプリケーションの実行時間に対する m の影響

参考文献

[1] R. D. Blumofe and C. E. Leiserson: Scheduling multithread computations by work stealing. J. ACM, 46(5):720–748, 1999.

[2] M. G. Norman and P. Thanisch: Models of machines and computation for mapping in multicomputers. ACM Computing Surveys, 25(3):263–302, 1993.

[3] A. Acharya, G. Edjlali, and J. H. Saltz: The utility of exploiting idle workstations for parallel computation. In ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 225–236, 1997.

[4] J. Ju, G. Xu, and J. Tao: Parallel computing using idle workstations. Operating Systems Review, 27(3):87–96, 1993.

[5] M. J. Litzkow, M. Livny, and M. W. Mutka: Condor a hunter of idle workstations. In Proc. ICDCS'88, pp. 104–111, 1988.

[6] D. A. Nichols: Using idle workstations in a shared computing environment. In Proc. 11th ACM Symp. on Operating System Principles, pp. 5–12, 1987.

[7] F. Tandiary, S. C. Kothari, A. Dixit, and E. W. Anderson. Batrun: Utilizing idle workstations for large-scale computing. IEEE Para. Distr. Techn., Summer:41–48, 1996.

[8] Deng Lei, 田頭茂明, 伊東靖英, 藤田聡: An Active Scheduler: Autonomous Concurrency Control of Parallel Programs in a Distributed Environment, 電子情報通信学会ソフトウェアサイエンス研究会, SS2000-31〜41, pp.73-80, 2001.