情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

メニーコアプロセッサにおける競合とスケーラビリティを考慮したスレッドスケジューリング

谷本輝夫[†1]　佐々木広[†2]
三輪忍[†1]　中村宏[†1]

本研究はメニーコアシステム上で並列アプリケーションを複数同時に実行した際のシステム全体の性能向上を目指す。メニーコアシステムではキャッシュやメモリなどの資源を複数のコアで共有しているため，アプリケーション間の競合を防ぐことが重要である．また，アプリケーション毎にスケーラビリティが異なるため、複数のアプリケーションを同時に実行する際にはスケーラビリティに応じて適切に資源を分配することが望ましい。本研究では同時に実行されるアプリケーションに割り当てるコア数を動的に制御することで効率的に実行する。本稿ではアプリケーションの振る舞いが一定であるという仮定のもとで実際にスケジューラを実装し，実行時情報からスケーラビリティを検出し，適切な割り当てコア数を決定できることを示す．

1. はじめに

プロセッサの周波数が向上しなくなったため近年では複数のプロセッサコアをひとつのチップに集積し，スレッドレベルの並列度を抽出することで性能向上を図るようになった．それに伴い，プロセッサに搭載されるコア数は増加傾向にある．サーバアプリケーションや数値科学計算などのアプリケーションは従来から並列化されてきたが，近年ではクライアント側であるデスクトップ用途のアプリケーションもメニーコアの利点を活かすためにシングルスレッドからマルチスレッドへと移っており[1)]，今後は複数の並列アプリケーションが同時実行される機会が増えると考えられる．

メニーコアプロセッサの特徴の一つは，複数のコアで最下位キャッシュや主記憶など下位のメモリ階層を共有していることである．コアごとに使用率の異なるリソースを共有することで効率良く活用できる反面，共有リソースにおける競合が起きると深刻な性能低下を引き起こす．同一のチップに搭載されるコア数が増え続ければ共有リソースにおける競合が起きる頻度も増加し，その影響もより大きくなるため，スレッド間の公平性やシステムスループット向上のために共有リソースの競合を解決する様々な手法が提案されてきた．OS のスケジューラでどのプロセスを同時に，どのコアに割り当てるか決定することを提案したものもある[2)] が，このような競合を考慮したスケジューリングの研究の多くは対象がシングルスレッドアプリケーションに限定されている．

スケーラビリティの高いプログラムに対する OS のスケジューラの役割はアプリケーション同士の干渉を防ぎ、それらに割り当てたハードウェアを独占させることである。しかし、メニーコアプロセッサにおける並列アプリケーションの性能向上はプログラム全体に占める逐次処理部分の割合，タスクやデータ分割の均一さ，共有データへの通信とデータの局所性，クリティカルセクションの量などのアプリケーションの性質に大きく依存しているため，デスクトップアプリケーションなどが並列化された場合，それらのスケーラビリティはそれほど高くなく，アプリケーションによって様々である[1)]．これらのことから，メニーコア向けの一般用途の OS は，様々なスケーラビリティを持つ複数の並列アプリケーションを効率よくスケジューリングする必要があると考えられる．

しかしながら，現在の Linux の標準スケジューラにはこのような環境で次のような問題がある．

- スケジューラは CPU 時間を均等に分配することでアプリケーション間のフェアネスを保とうとするが，アプリケーションのスケーラビリティが等しいとは限らない．そのような場合，よりスケーラビリティの高いアプリケーションにリソースを割り当てたほうが効率が良い．
- 複数のアプリケーションのスレッドが最下位キャッシュやメモリコントローラ，プリフェッチャやメモリバスを含むメモリ階層のリソースを共有している場合に競合を引き起こす．

本研究ではこれらの Linux スケジューラの問題を解決する複数並列アプリケーション環境向けのスケジューラを開発した．ここまでに述べたことから，複数の並列アプリケーションを実行する際にはアプリケーション間の競合とスケーラビリティの 2 つが重要な要素で

†1 東京大学
†2 九州大学

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

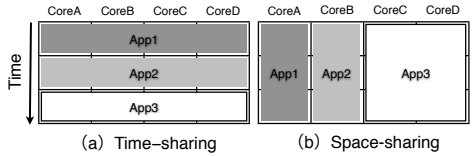

図 1 2 つのスケジューリング方法

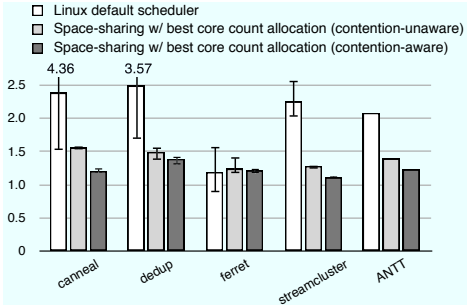

図 2 Normalized turnaround time (NTT)

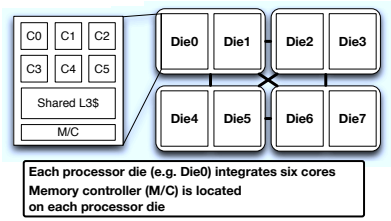

図 3 評価環境のアーキテクチャ概念図

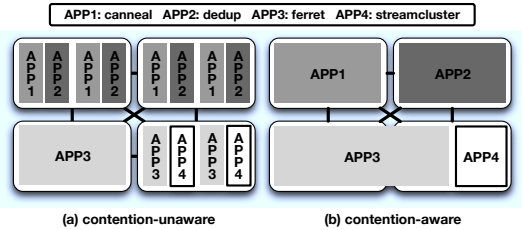

図 4 2 通りのスペースシェアリングによるプロセス―コア割り当て

あると考えられる．本研究で提案するスケジューラは共有リソースの分割と，適切な割り当てを行うことによって，これらの 2 つの要素を考慮したスケジューリングを行う．

本稿の構成は以下のとおりである．2 章で競合とスケーラビリティを考慮したスケジューリングについて述べ，3 章でスケジューラの実装について述べる．4 章で評価と考察を行い，5 章で関連研究について述べ，6 章でまとめる．

2. メニーコアプロセッサにおける複数並列アプリケーションの実行

並列アプリケーションのスケジューリング方法にはタイムシェアリングとスペースシェアリングの 2 つの方法がある．図 1 にこれらのスケジューリングの例を示す．(a) タイムシェアリングとは一般のマルチタスク OS で採用されている方法で，複数のアプリケーションにプロセッサコアを短時間ずつ時分割に利用させる．各アプリケーションに CPU 時間が均等になるように分配することでアプリケーション間のフェアネスを保証することができる一方，スケーラビリティの低いアプリケーションをこの方法で実行すると多数のコアを有効に活用することができないため，システムスループット向上の余地がある．(b) スペースシェアリングとは，アプリケーションにコアを分配し，占有させるという手法である．スケーラビリティの高いアプリケーションにより多くのコアを分配すると，CPU 時間は均等にならないがシステムスループットを向上させることができる．図 2 は複数並列アプリケーションを実行した際に Linux の標準スケジューラ（タイムシェアリング）が性能低下を起こしていることを示したものである．これは PARSEC Benchmark Suite[3] から選択した 4 つの並列アプリケーション（canneal, dedup, ferret, streamcluster）を 48 コアのシステム[⋆1]で実行した際の結果である．縦軸は normalized turnaround time（NTT）であり，エラーバーは評価全体における最高，最悪の場合を示している．NTT とはあるアプリケーションがシステムを占有した場合に対する複数のアプリケーションを同時実行した場合の実行時間の比であり、性能低下率を示す 1 以上の値が小さいほうが望ましい指標である．右端のバーは NTT の平均（ANTT）を示したもので，本研究で用いる性能指標である．各アプリケーションについて，Linux の標準スケジューラと，最適な割り当てコア数でのスペースシェアリングと呼ばれる 2 つの静的なプロセス―コア割り当て方法（競合を考慮しないものと考慮するもの． 実際の割り当てを図 4 に示す.）での結果を示した．このときの割り当てコア数は canneal と dedup に 12 コアずつ，ferret に 18 コア，streamcluster に 6 コアである．スペースシェアリングの際にはそれぞれのコアは単一のアプリケーションしか実行しないものとし，複数のアプリケーションがコアを共有することはないものとした．どちらのスペースシェアリングにおいても各アプリケーションに割り当てるコア数は同じだが，競合を考慮する場合としない場合で性能が異なることがわかる．

Linux スケジューラの ANTT は 2.07 であったが，各アプリケーションへの最適な数のコア割り当てを行った 2 つのスペースシェアリングでは，Linux よりも性能向上し，競合を考慮しない場合（図 4 の（a））では ANTT が 1.39，競合を考慮する場合（b）では 1.23 であった．この結果からアプリケーションのスケーラビリティを考慮して割り当てるコアの数を適切に制御することが重要だといえる．

⋆1 図 3 に本研究で用いた評価環境のアーキテクチャ概念図を示す．このシステムは 12 のプロセッサコアを持つ CMP を 4 つ搭載したもので，それぞれの CMP は 2 つの 6 コアからなるダイを持つ．評価環境の詳細は第 4 章で述べる．

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

それぞれのダイはメモリコントローラを持っており，2 つのスペースシェアリングを比較すると，競合を考慮しないばあいには canneal と dedup，そして ferret と streamcluster が最下位キャッシュとメモリーコントローラ，プリフェッチャとメモリバスを共有している事がわかる．共有資源を切り分けることは競合を回避するための根本的な手段であることから，本研究ではこのようにして競合を考慮した割り当てを行うこととする．

これまでに競合とスケーラビリティを考慮したスペースシェアリングをすることによって性能向上が期待できることを示したが，これを一般用途の OS に適応するのは簡単ではない．その主要な理由は，(1) アプリケーションのスケーラビリティは同時実行される他のアプリケーションの影響をうけるため．(2) アプリケーションはスケーラビリティや共有資源の利用度の異なる幾つかのフェーズから構成されている可能性があり，実行時間中に最適なコア割り当てが変わりうるため．(3) 一般に OS 上ではアプリケーションは様々なタイミングで実行され，どのアプリケーションがコスケジュールされるかはわからないため，である．本研究ではシステムソフトウェアがハードウェアと協調して動的にアプリケーションのスケーラビリティを検出することによってアプリケーションへの最適な割り当てコア数を決定し，競合を考慮した割り当てを行うことでこれらの問題を解決する．

3. CaSAS: Contention and Scalability-Aware Scheduler

マルチコアプロセッサ上でフェアネスを保ちながらシステムの高い性能を実現する Contention and Scalability-Aware Scheduler（CaSAS）を設計・実装した．CaSAS は各アプリケーションに割り当てるプロセッサコア数を動的に変えることでアプリケーションのスケーラビリティを検出し，最適な割り当てを決定する．さらに，CaSAS は割り当て資源量を決定するだけではなく，計算ノードのアーキテクチャを考慮し，アプリケーション間の競合が最小化する配置を行う．この章ではスケジューリングポリシー，アルゴリズムについて述べる．

3.1 スケジューリングポリシー

これまでに複数の並列アプリケーションをスペースシェアリングする際に競合とスケーラビリティを考慮することが重要であることを述べた．ここでは CaSAS のそれらに対する方針を述べる．

3.1.1 アプリケーション間競合の回避

第 2 章において，スペースシェアリングの際に，計算ノードのアーキテクチャ，とくにメモリ階層を考慮した割り当ての重要性を示した．競合回避の基本はキャッシュパーティショニング[4] のようにアプリケーション同士の干渉や競合を起こさないよう，共有資源を分割し独占させることである．本研究では最下位キャッシュとメモリコントローラにおける競合を避けるため，プロセッサダイ（図 3）をアプリケーションへの割り当ての最小単位とした．すなわち，今回用いた評価環境では，スケジューリングの際に各アプリケーションには 6 の倍数の数のコアを割り当てる．

このような割り当て方は今回の評価環境に特化した手法のようにみえるが，ページカラーリング[5] などを行うことでハードウェアの変更なしにシステムソフトウェアによってキャッシュパーティショニングする手法が提案されており，最下位キャッシュがすべてのコアから対称でかつ共有されている場合においても，アプリケーションごとに共有資源を切り分ける事ができる．また，今回評価に用いた環境のようなノード内で複数の CPU パッケージをインターコネクトで接続することによってノード内の並列度を大きくする傾向は今後も続くと考えられるため，その意味でもこの手法は有効であると言える．

3.1.2 アプリケーションのスケーラビリティと資源割り当て

もう一つの CaSAS の重要な点は各アプリケーションのスケーラビリティを考慮したスケジューリングを行うことである．そのためにスケジューラがすべきことは（1）アプリケーションのスケーラビリティの検出，(2) アプリケーションへの適切な量の計算資源の割り当て，(3) 可能であれば割り当てられたハードウェア資源の下でアプリケーションの性能が最大化するようアプリケーションを最適化することである．(1) については，ハードウェアと協調し，実行時情報を取得することによって検出する．詳細については 3.2 で述べる．

(2) については第 2 章でも述べたとおり，average normalized turnaround time（ANTT）を最小化するような割り当てを行う．ANTT は以下の式により求められる．

$$ANTT = \frac{1}{n}\sum_{i=1}^{n} NTT_i = \frac{1}{n}\sum_{i=1}^{n} \frac{C_i^{MP}}{C_i^{SP}} \quad (1)$$

C_i^{SP} と C_i^{MP} はそれぞれプログラム i を単独，複数のアプリケーションと実行した時のクロックサイクル数（実行時間に対応）である．ANTT が最小であることは，複数のアプリケーションがシステムを共有した際にすべてのアプリケーションが公平に単独実行時に近い実行であることを意味する．これまでの Linux のスケジューラは“ フェアネス ”を各スレッドに同じ量の CPU 時間を与えることによって保証している．これだと大量のスレッドを生成するプロセスに多くの CPU 時間を割り当ててしまうため，スレッドごとではなくプロセスごとに管理する手法もある[6] が，いずれにせよスケールしないアプリケーションに多くの CPU 時間を割り当てることは複数並列アプリケーション環境において公平とは言えない．

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

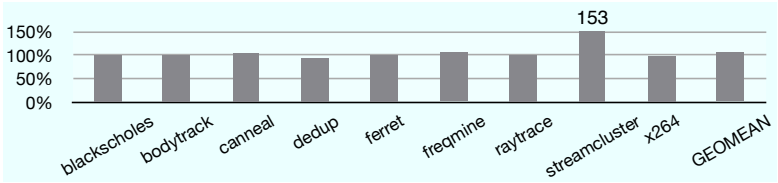

図 5 6 コア 6 スレッドに対する 6 コア 48 スレッドの相対的な実行時間

システムの性能を考える際に OS のスケジューラが目指すべきはすべてのアプリケーションが同様に僅かに性能低下する状況であると考え，本研究は ANTT の最小化を目的とする．

（3）については，大規模な数値科学計算にスペースシェアリングを適用した先行研究において，利用できる計算資源にあわせてプログラミング言語や OS のレイヤでスレッド数を制御する手法[7] が提案されている．数値科学計算では，データ並列度が高いため，処理を利用できるプロセッサ数に均等に分割してコンテキストスイッチの発生回数を減らし，スレッド間のワークロードインバランスを避けるのが最も効率が良い．しかしながら，CMP の普及に伴い様々なアプリケーションが並列化されるようになったため，数値科学計算のように処理を均等に分配することができずスレッドごとの処理量が異なる場合が増えている．OS のスケジューラは生成されるスレッド数は制御できないが，均等に並列化できていないアプリケーションではハードウェアのコア数以上のスレッド数を生成することで動的負荷分散できる利点がある．図 5 は PARSEC アプリケーションを 6 コアで実行する際に，並列度を 6 に設定した時に対する，並列度を 48 とした時の相対的な実行時間を示したものである．スレッドをハードウェアのコア数よりも多く作ることによる性能低下は streamcluster を除くと 10%以内に収まっており，6 コア以外でも同様の結果となった（グラフは紙面の都合上割愛する）．このことから，本研究ではアプリケーションはシステムのコア数以上のスレッドを生成するものとし，アプリケーションのスレッド数自体を制御するかわりに，各アプリケーションが使用するコアの数を変えることで動的な制御を行うことにする．

3.2 スケーラビリティの検出

アプリケーションのスケーラビリティは共有資源における競合などのハードウェアの要因と，逐次処理部分の割合やクリティカルセクションの量などのソフトウェア自体の性質の両方によって制限される．アムダールの法則は並列アプリケーションのスケーラビリティのモデルとして現実のアプリケーションにもよく当てはまるできることが知られているが，このモデルはアプリケーションを逐次処理部分と並列処理部分に完全に分けられるとしている．

表 1 コア数による実行命令数の増加量

アプリケーション	1 コアに対する実行命令数増加量 [%]（コア数）
blackshcoles	0.4（48）
bodytrack	2.2（48）
canneal	8.1（36）
dedup	8.7（48）
ferret	4.9（48）
fluidanimate	0
freqmine	0.9（42）
raytrace	0.8（48）
streamcluster	3.0（48）
x264	0.1（30）

逐次処理部分では，コアが複数ある利点はないため，スケジューラは 1 コアが確実に割り当てられるようにすればよいが，並列処理部分については，アムダールの法則の仮定が成り立つとすれば，コア数がいくつであれ完全に並列処理できるため，利用できるすべてのコアを割り当てるのが良いことになる．しかしながら，クリティカルセクションにおける競合を考慮した並列アプリケーションの性能モデル[8] によれば一般に並列処理部分であっても完全な並列化はできず，アプリケーションごとに異なるスケーラビリティを持つ．CaSAS はこのような，並列処理部分において異なるスケーラビリティを持つアプリケーションを主な対象とする．

スケジューラは複数のアプリケーションが同時実行された際にスケーラビリティを検出する必要があるため，近年ではほとんどのプロセッサで利用可能なパフォーマンスモニタリングユニット（PMU）を利用することにした．

本研究では，アプリケーションを単独実行した場合，つまりアプリケーションがシステムを独占した場合に対する同時実行時の相対性能を検出する必要がある．ただし，単独実行とは，システム上で実行する際に最も性能が高い場合のこととし，すべてのコアを使う場合とは限らない．そのための本研究の提案はアプリケーションの単位時間あたりの全てのスレッドの実行命令数（IPS:instructions per second）を利用することである．短時間ずつ割り当てコア数を変えて実行し，IPS を取得することで実行時に単独実行に対する相対性能を得る． しかし，同期待ちのスピンロックなど，プログラムが進まなくても命令数は増加するため，IPS からはうまくスケーラビリティを推定できない可能性があるため，実行コア数を変えた時のプログラム全体での実行命令数を調べた．表 hoge に生成スレッド数を 48 に固定し 6,12,18,24,30,36,42 コアで実行した場合の 1 コア時に対する命令数の増加量の最大値（括弧内は最大となるコア数）を示した．すべてのアプリケーションで 8%以内に収まって

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

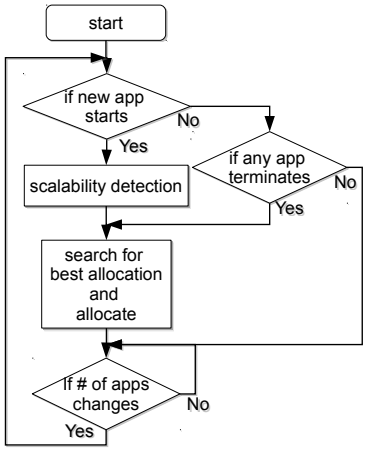

図 6 スケジューリングのフローチャート

Processor:	4 × AMD Opteron 6172
Sockets:	4
Cores per processor:	12
Num. of dies per socket	2
Num. of cores per die:	6
Total num. of cores:	48
L3 cache size:	12 MB per socket
Main memory:	96 GB DDR3 PC3-10600

表 2 評価環境の主なパラメータ

おり，この程度であればスケーラビリティ予測に利用できると考えられる．

3.3 スケジューリングアルゴリズム

本稿では，実行時情報から適切なコア割り当てが可能であることを確かめるために，アプリケーションの振る舞いは一定であるという仮定をし，シンプルなスケジューリングアルゴリズムの実装を行った．振る舞い一定の仮定から，アプリケーション実行中の割り当て変更は行わず，新たなアプリケーションが実行される時，実行中のアプリケーションのいずれかが実行終了する時にスケジューリングを行う．スケジューラは最適なコア割り当てを探索した後，各アプリケーションへのコア割り当てを行い，次のスケジューリングが行われるまで待つ．最適な割り当てを探索するアルゴリズムを図 6 に示した．スケジューラ関数は実行するアプリケーションの数が変わるたびに呼ばれ，新しいアプリケーションが実行を開始する場合は，割り当てるコア数を変えながら単独実行し，各割り当てでの IPS を調べてそのアプリケーションのスケーラビリティを検出する．スケーラビリティの検出が終わると，実行中のすべてのアプリケーションのスケーラビリティの情報から，ANTT を最小化する割り当てを求め，実際に割り当てを行う．一方，実行中のアプリケーションが実行を終了した場合には，残りのアプリケーションのスケーラビリティはすでに検出済みであるため，ANTT が最小となる割り当てを計算して割り当てを行う．

4. 評　　価

4.1 評価環境

今回評価には 4 ソケットの IBM System x3755 M3 サーバを用いた．これは 2.1GHz で動作する 12 コアの AMD Opteron 6172 プロセッサを 4 つ搭載し，合計で 48 のプロセッサコアを持つ．それぞれのソケットは 6 つのプロセッサコアを持つプロセッサダイを 2 つ集積したもので，それぞれのコアはプライベートな L1 と L2 キャッシュを持っており，ダイごとに 12MB の共有 L3 キャッシュを持つ．今回は CaSAS の試作版をユーザ空間のプログラムとして実装した．Linux kernel 2.6.37.6 を用い，PMU の取得には変更を加えた perf-tools を用いた．アプリケーションへの明示的なコア割り当てには Linux API である sched_setaffinity(2) を用いた．

ワークロードとしては，PARSEC Benchmark Suite 2.1 の中から 10 のアプリケーションを用いた．GCC-4.1 を用い，”-O3 -funroll-loops -fprefetch-loop-arrays” オプションを指定してコンパイルした．評価には入力セットとして native を用いた．

アプリケーションの並列部分におけるスケジューラの評価を行うために，アプリケーションのチェックポイント／リスタートを可能にする BLCR[9] を用いてベンチマークのソースコード中に示されている region of interest(ROI) だけを実行するようにした．

各アプリケーションの実行時間が異なるため，複数のアプリケーションの実行方法には SMT job scheduling[10] などで提案されているものに習い，すべてのアプリケーションが最低 3 回実行するまで各アプリケーションを繰り返し実行するものとした．アプリケーションの実行時間は time コマンドを用いて計測した．

4.2 評価結果

4.2.1 ベンチマークの分類

アプリケーションのスケーラビリティは CaSAS の性能に大きく影響するため，まず，評価環境におけるアプリケーションのスケーラビリティを調べた．図 7 は PARSEC ベンチマークのスケーラビリティを示したもので，横軸に割り当てコア数，縦軸にシステムを占有した場合（最大性能）に対する相対性能をプロットしたものである．各グラフの上部の数字は 1 コア時に対する最大性能の相対値である．各アプリケーションのスケーラビリティからアプリケーションを 3 種類に分類した．よくスケールする Green ，中程度のスケーラビリティを持つ Yellow，スケーラビリティが低い Red である．

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

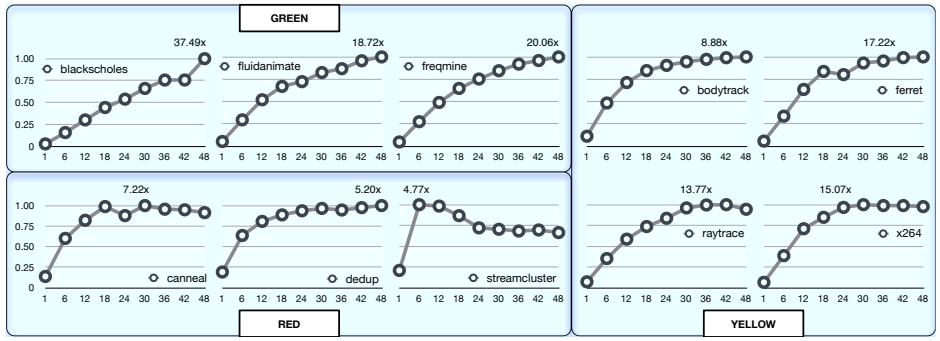

図 7 PARSEC ベンチマークのスケーラビリティ

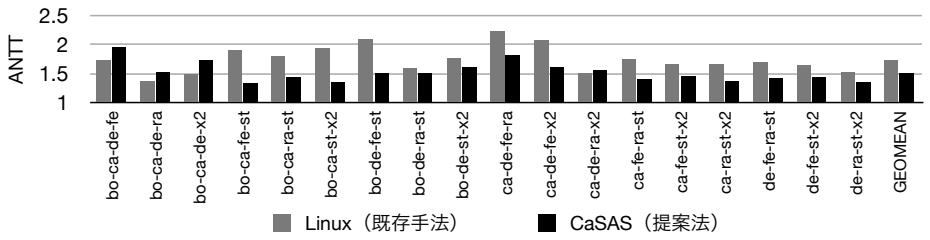

図 8 YYRR

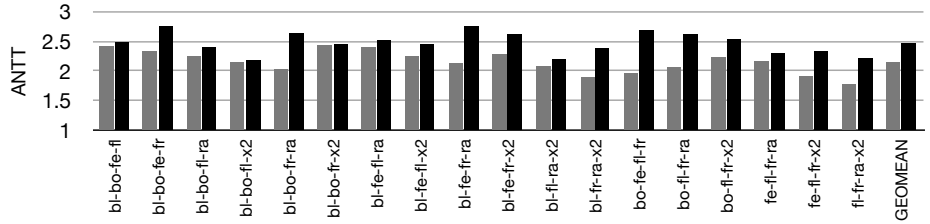

図 9 GGYY

4.2.2 4 ベンチマークによる結果

本稿では，4 つのアプリケーションを同時に実行したときの ANTT を Linux の標準スケジューラと比較した．すべてのアプリケーションの組（210 組）の ANTT の幾何平均は，Linux で 1.94，CaSAS で 1.95 と同程度となった．同時実行する組に含まれるアプリケーションのスケーラビリティが評価結果に大きく影響するため，アプリケーションのグループの組み合わせごとに結果を示す．紙面の都合上，Y グループから 2 つ，R グループから 2 つ選んだ組（YYRR）の結果を図 8 に，G グループから 2 つ，Y グループから 2 つ選んだ組の結果を図 9 に示す．YYRR（図 8）では，各組の幾何平均が Linux で 1.52，CaSAS で 1.34 となり，CaSAS による性能向上が見られた．一方，GGYY（図 9）では，各組の ANTT の幾何平均が Linux で 1.78，CaSAS で 2.21 となり，Linux のほうが性能が良い結果になった．この理由として，G 群のアプリケーションはスペースシェアリングの効果が小さいことがあげられる．G 群のアプリケーションを含まない組の ANTT の幾何平均は Linux で 1.74，CaSAS で 1.54 であった．このことから，本研究が主対象とする比較的スケーラビリティの低いアプリケーション群について，スペースシェアリングが有効であることと，CaSAS が実行時情報から適切な割り当てコア数を選択できていることがわかる．また，今回はアプリケーションのフェーズが一定であるという仮定をおいたが，bodytrack（Y 群）や freqmine（G 群）といったアプリケーションは実行中にフェーズ変化があることがわかっており，CaSAS で CPU がアイドル状態になる期間が生じていると考えられる．

5. 関 連 研 究

5.1 共有資源における競合の軽減

最下位キャッシュにおける競合は深刻な性能低下をもたらすため，キャッシュリプレースメントポリシーを見直すことで最下位キャッシュにおける競合を回避するキャッシュパーティショニング[4] が提案されている．また，最下位キャッシュによる競合に対処するために OS によるページカラーリング[5] といったシステムソフトウェアによる手法も可能である．

5.2 複数並列アプリケーションのスケジューリング

PDPA[11] はシステムスループット最大化を目的とし，割り当てコア数を変更した際の性能の変化を取得し，割り当てコア数のインクリメント／デクリメントを行う．スケーラビリティを考慮したコア割り当てを行なっているものの，スケーラビリティを検出するためにアプリケーションに手を加える必要があり，アプリケーション間の競合が考慮されておらず，フェアネスについても議論されていない．HOLYSIN Scheduler[12] はエネルギー遅延積を

Vol.2011-ARC-197 No.31
Vol.2011-HPC-132 No.31
2011/11/29

小さくすることを目的として動的に割り当てを変更するスペースシェアリングを行なっているが，競合によるスケーラビリティの阻害を主対象としており，また，フェアネスは CPU 時間を均等にすることによって保証している．

6. ま と め

メニーコア時代には様々なスケーラビリティを持つ並列アプリケーションが同時に実行されると考えられ，従来の OS のスケジューラがフェアネスをアプリケーションに割り当てる CPU 時間によって定義しているのに対し，本研究ではアプリケーション間の競合とスケーラビリティを考慮したスケジューリングが重要であるということを述べた．競合の解決については既によく研究されており，提案手法である CaSAS でもシステムのハードウェア構造に着目した共有資源の切り分けを行った．また，システムとして高い性能を実現するためにアプリケーションのスケーラビリティを考慮し，アプリケーションに適切な計算資源を割り当てることの重要性について述べた．

アプリケーションの振る舞いが一定であるという仮定の下，スケジューラを実装し，Linux の標準スケジューラと比較した．主対象とする比較的スケーラビリティの低いアプリケーションの組について Linux スケジューラの ANTT の幾何平均が 1.74 に対し，CaSAS が 1.54 と提案法の有効性を示した．

謝辞 本研究の一部は，科学技術振興機構 (JST) 戦略的創造研究推進事業 (CREST)，および科学研究費補助金 (基盤 (B) No.22300015) によるものである．

参 考 文 献

1) Blake, G., Dreslinski, R.G., Mudge, T. and Flautner, K.: Evolution of thread-level parallelism in desktop applications, *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, New York, NY, USA, ACM, pp. 302–313 (online), DOI:http://doi.acm.org/10.1145/1815961.1816000 (2010).
2) Zhuravlev, S. and Fedorova, A.: Addressing Shared Resource Contention in Multicore Processors via Scheduling, *ASPLOS '10 Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, No.1, pp.129–141 (online), DOI:http://dx.doi.org/10.1145/1736020.1736036 (2010).
3) Bienia, C., Kumar, S., Singh, J. P. and Li, K.: The PARSEC benchmark suite: characterization and architectural implications, *PACT*, pp. 72–81 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1454128⟩ (2008).
4) Chandra, D.: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture, *11th International Symposium on High-Performance Computer Architecture*, IEEE, pp.340–351 (online), DOI:10.1109/HPCA.2005.27 (2005).
5) Cho, S. and Jin, L.: Managing Distributed, Shared L2 Caches through OS-Level Page Allocation, *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, IEEE, pp. 455–468 (online), DOI:10.1109/MICRO.2006.31 (2006).
6) Wong, C.S., Tan, I., Kumari, R.D. and Wey, F.: Towards achieving fairness in the Linux scheduler, *ACM SIGOPS Operating Systems Review*, Vol.42, No.5, pp.34–43 (online), available from ⟨http://portal.acm.org/citation.cfm?id=1400097.1400102⟩ (2008).
7) Corbalan, J., Martorell, X. and Labarta, J.: Improving Gang Scheduling through job performance analysis and malleability, *Proceedings of the 15th international conference on Supercomputing - ICS '01*, pp. 303–311 (online), DOI:10.1145/377792.377852 (2001).
8) Eyerman, S. and Eeckhout, L.: Modeling critical sections in Amdahl's law and its implications for multicore design, *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, New York, NY, USA, ACM, pp. 362–370 (online), DOI:http://doi.acm.org/10.1145/1815961.1816011 (2010).
9) Hargrove, P.H. and Duell, J.C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters, *Journal of Physics: Conference Series*, Vol.46, No.1, pp.494–499 (online), DOI:10.1088/1742-6596/46/1/067 (2006).
10) Snavely, A. and Tullsen, D.M.: Symbiotic jobscheduling for a simultaneous multithreaded processor, *ACM SIGARCH Computer Architecture News*, Vol.28, No.5, pp.234–244 (online), DOI:10.1145/378995.379244 (2000).
11) Corbalan, J., Martorell, X. and Labarta, J.: Performance-driven processor allocation, *IEEE Transactions on Parallel and Distributed Systems*, Vol.16, No.7, pp. 599–611 (online), DOI:10.1109/TPDS.2005.85 (2005).
12) Bhadauria, M. and McKee, S.a.: An approach to resource-aware co-scheduling for CMPs, *Proceedings of the 24th ACM International Conference on Supercomputing - ICS '10*, p.189 (online), DOI:10.1145/1810085.1810113 (2010).