

複数種・不均一資源要求に対応した 実行時間推定非依存なクラスタスケジューラ

藪内 秀仁^{1,a)} 品川 高廣¹

概要: 近年のクラスタスケジューラでは、推定誤差によるスケジューリング性能の低下を避けるため、各タスクの実行時間推定に依存しない手法が提案されている。しかし、これらのスケジューラは各タスクが要求する計算資源が1種類かつ均等であると仮定しており、各タスクが複数種かつ不均一な計算資源を要求する現実の環境ではフラグメンテーションなどの問題が生じる。本研究では、複数種かつ不均一な計算資源要求量に対応した、各タスクの実行時間推定に依存しないクラスタスケジューラを提案する。提案手法では、クラスタスケジューリングを多次元ビンパッキング問題とみなし、計算資源の使用可能量と要求量の類似度に着目したノード選択や、プリエンプションするタスク数を最小化する組み合わせ選択などのヒューリスティクスを適用する。これにより、計算資源使用効率の向上とプリエンプション回数の削減を図り、各タスクのスケジューリング待機による遅延 (slowdown rate) を短縮する。シミュレーションによる評価の結果、1種類かつ均等な計算資源要求量を仮定したスケジューラ Kairos に比べ、提案手法はタスクの slowdown rate の 95 パーセンタイルを約 28.2%、99 パーセンタイルを約 30.7% 短縮した。

1. はじめに

ビッグデータ処理などのデータ集約型アプリケーションや科学計算アプリケーションは、強力な計算能力と大容量のストレージを必要とする。そのため、これらのアプリケーションの実行には、多数の計算ノードと大容量のストレージからなる計算機クラスタ（以下、クラスタ）が使われている。今日では多くの大学や研究所、企業がクラスタを所有し、複数のユーザが共有して使用している。

クラスタでは、各アプリケーションはジョブとして管理される。各ジョブは1つまたは複数のタスクからなり、各タスクの実行に必要な計算資源の種類および量（例：CPU 2 コアとメモリ 4GB）をユーザが事前に指定する。ジョブの実行はクラスタスケジューラが管理する。クラスタスケジューラは、未完了のジョブの集合について、実行するタスクを選択し、それが要求している計算資源が使用可能なノードを選択し、計算資源をタスクに割当て実行するという役割を担う。クラスタスケジューラの性能は主に、

- 多くのジョブをクラスタで実行するスループット
- 主に実行時間が短いジョブについて、ユーザが投入してから実行が完了するまでのレイテンシ
- ユーザ間・ジョブ間の公平性

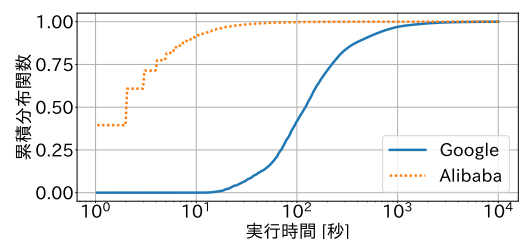


図 1: タスクの実行時間の累積分布関数。

によって決定される。

近年のクラスタにおけるワークロードの特徴として、まず、実行時間の不均一性が挙げられる [1,2]。図 1 は Google のクラスタのトレース [3] および Alibaba のクラスタのトレース (2018 年版) [4] について、タスクの実行時間の累積分布関数を示したものである。大部分のタスクは数秒から数百秒と短いですが、数時間以上かかる長いタスクも存在し、実行時間の分布は裾が長くなっている。変動係数は Google および Alibaba でそれぞれ 2.97 および 6.60 であり、非常に分散が大きい。多くの場合、短いタスクからなるジョブはレイテンシを短くする要求が厳しく、ユーザが投入してからクラスタスケジューラにより実行が開始されるまでの遅延に大きく影響を受ける。一方で、長いタスクをもつジョブはレイテンシを短くする要求が比較的緩い。

このような分散が大きく不均一な実行時間をもつワークロードを適切にスケジューリングするために、近年のクラ

¹ 東京大学 / The University of Tokyo

^{a)} yabuuchi@os.ecc.u-tokyo.ac.jp

表 1: タスクの計算資源要求量に関する統計.

	CPU 要求量 の変動係数	メモリ要求量 の変動係数	CPU とメモリ の相関係数
Google	0.81	0.91	0.33
Alibaba	0.38	0.34	0.28

スタスケジューラは各タスクの実行時間が事前に推定できることを前提とし、この情報に基づいてスケジューリングしている [5–11]. 例えば、実行時間が短いと推定されたタスクを優先して実行することで、このタスクをもつジョブが完了するまでの時間を短縮できる.

推定実行時間を用いるスケジューラの性能は、その推定精度に大きく影響を受ける. 前述の例では、実行時間が短いと推定し優先して実行したタスクが実際には長かった場合、このタスクが後続のタスクをブロックしスケジューリング待機時間を長くする Head-of-Line (HoL) ブロッキングが発生する. しかし、今日まで十分な精度で効率よく推定する方法は知られておらず、推定誤差により最適でないスケジューリング結果に陥っている [12].

推定誤差によるスケジューリング性能の低下を避けるために、各タスクの実行時間推定に依存しないクラスタスケジューラが提案されている [13,14]. これらは、スケジューリングする時点において各ジョブまたはタスクをそれまでに実行した時間を累積し、累積実行時間が短いものから優先することで、合計実行時間が短いジョブのレイテンシを短縮する. これらのスケジューラは各タスクが要求する計算資源が 1 種類かつ均等であるという仮定に基づいている.

しかし、実際のクラスタにおけるワークロードの 2 つ目の特徴として、各タスクが要求する計算資源は CPU やメモリを含む複数の種類からなり、かつ分散が大きく種類間の相関が小さい不均一な量であることが挙げられる [1,6]. 表 1 は、前述の Google および Alibaba のクラスタのトレースについて、タスクの計算資源要求量に関する統計を示したものである. Alibaba のクラスタでは各計算資源の要求量の変動係数は大きくないものの、Google では 0.81, 0.91 とある程度大きな分散が見られる. また、両方のクラスタについて CPU とメモリの要求量間の相関は小さい.

各タスクが要求する計算資源が複数種であること、また分散が大きく種類間の相関が小さいことを考慮せずスケジューリングすると、計算資源のフラグメンテーションや過剰割当てといった問題を生じさせる. 例えば、slot-based クラスタスケジューラ [15] は slot と呼ばれる固定量の CPU とメモリの組を単位として計算資源をタスクに割当ててため、内部フラグメンテーションが発生しやすく計算資源の利用効率が低下する. また、例えば CPU のみを考慮しメモリ要求量を見捨てるスケジューラは、メモリ集約型のタスクを多く同じノードで実行する可能性があり、メモリ競合を引き起こしタスクの実行速度を低下させる.

そこで本研究では、複数種かつ不均一な計算資源要求量に対応した、各タスクの実行時間推定に依存しないクラスタスケジューラを提案する. 提案するスケジューラでは、特に短いジョブについて投入から実行完了までのレイテンシを短縮し、また、多くのジョブを実行するスループットを向上させる. このスケジューラは、1 種類かつ均等な計算資源要求量を仮定しているスケジューラ Kairos [6] を拡張して、複数種かつ不均一な計算資源要求量を適切に扱えるようにしたものである.

提案手法では、クラスタスケジューリングを多次元ビンパッキング問題とみなしさまざまなヒューリスティクスを適用することで、各タスクのスケジューリング待機による遅延 (slowdown rate) を短縮する. まず、各タスクをノードに割当ての際、使用可能な計算資源量がタスクの計算資源要求量と類似度が高いノードを選択することで、計算資源の使用効率を向上させる. また、累積実行時間が長いタスクをプリエンプションし短いタスクを開始する際、全種類の計算資源を考慮し不要なプリエンプションを回避することで、全体のプリエンプション回数を削減する.

シミュレーションによる評価の結果、提案手法は Kairos に比べ、タスクの slowdown rate の 95 パーセンタイルを約 28.2%, 99 パーセンタイルを約 30.7% 短縮した.

以降、まず 2 章で関連するクラスタスケジューラの研究について述べる. 次に 3 章で提案手法の設計を説明し、4 章で既存研究との比較により評価をおこなう. 最後に 5 章で本論文の結論と今後の展望を述べる.

2. 関連研究

2.1 実行時間推定によるクラスタスケジューリング

近年の多くのクラスタスケジューラは、各タスクの実行時間が不均一な状況下で高スループット、低レイテンシ、および公平性を達成するために、実行時間の推定を利用しスケジューリングしている. EASY [5] は backfilling を導入することで、計算資源の使用効率を向上させ短いタスクのレイテンシを短縮した. Backfilling は FIFO スケジューリングを拡張した手法であり、キューの先頭にあるタスクが計算資源の不足により実行できない場合、後方にある短いタスクで要求する計算資源が使用可能なものを先に実行する. Tetris [6], Graphene [7], Yaq [8] は Shortest Remaining Time First に基づき、推定実行時間が短いタスクや未実行タスクの合計実行時間が短いジョブを優先することでレイテンシを短縮した. Hawk [9] および Eagle [10] はクラスタの一部を短いタスクのために予約しておき、短いタスクが投入され次第すぐに実行することでレイテンシを短縮した. Big-C [11] は短いタスクが要求する計算資源が不足しているとき、長いタスクから短いタスクへ計算資源を動的に再割当てし実行することで、短いタスクのレイテンシを短縮した.

各タスクの実行時間は、繰り返し実行されるジョブであればその履歴から推定できる [16]。また、事前のプロファイリング [17] やシミュレータでの実行 [18] により推定する手法も提案されている。しかし、十分な精度で効率よく推定することは困難であり、不正確な推定に依存したスケジューラは HoL ブロッキングなどの問題を生じさせ、最適なスケジューリングを得られない [12]。

2.2 実行時間推定の精度不足への対策

実行時間の推定誤差によるスケジューリング性能の低下を抑制するために、JamaisVu [19] は推定精度を向上させる手法を提案した。計算資源要求量やユーザ名などタスクの属性の集合と実行時間の履歴を紐付け、さらにどの属性の部分集合を用いると最も正確に推定できるか追跡しておくことで、新しく投入された各タスクの実行時間を精度よく推定する。また、3Sigma [12] は実行時間を点推定するのではなく確率分布として表現し、分布全体を考慮してスケジューリングすることで、推定の不確実性によるスケジューリング性能の低下を緩和した。

これらの方向性は一定の成功を収めているものの、推定誤差は避けられないものであり、推定に依存したスケジューラは依然として最適な性能を得られていない。

2.3 実行時間推定に非依存なクラスタスケジューラ

実行時間推定の精度不足に対する別のアプローチとして、実行時間推定に依存しないクラスタスケジューラが提案されている。LAS.MQ [13] および Kairos [14] はともに Least Attained Service (LAS) [20] スケジューリング戦略に基づいている。LAS はスケジューリングする時点においてジョブまたはタスクがそれまでに実行した時間を累積し、累積実行時間が短いものを優先する戦略である。LAS.MQ はジョブごとに累積実行時間を計測し、短いジョブに多くの計算資源を割当て多くのタスクを実行する。Kairos はタスクごとに累積実行時間を計測し、長いタスクをプリエンプションし短いタスクを実行することで、実行時間が短いタスクのレイテンシを短縮する。

これらのスケジューラは、各タスクの計算資源要求量が 1 種類かつ均一であることを前提としている。しかし、実際のクラスタでは要求計算資源は複数種であり、かつ分散が大きく種類間の相関が小さい不均一な量である [1, 6]。この特徴を考慮せずスケジューリングすると、計算資源のフラグメンテーションや過剰割当てを生じさせる。

2.4 複数種・不均一な計算資源要求量への対応

Dominant Resource Fairness (DRF) [21] は複数種かつ不均一な計算資源要求量に対応した、公平性を満たすスケジューリング手法である。各タスクについて支配的な種類の計算資源を基準に max-min fairness を適用することで、

公平性のために満たすべき性質をもつスケジューリングを達成する。Tetris [6] はヒューリスティックスの適用によりフラグメンテーションを抑えるようスケジューリングすることで、クラスタの計算資源の使用効率を向上させスループットを改善する。また、短いと推定されたタスクをもつジョブを優先することで完了までの時間を短縮する。

しかし、DRF はスループットやレイテンシといった公平性以外の性能指標には着目していない。また、Tetris は各タスクの実行時間推定に依存している。

3. 設計

本研究では、実行時間推定に依存しないクラスタスケジューラ Kairos [14] をもとに、複数種かつ不均一な計算資源要求量を適切に扱いスケジューリングできるよう拡張する。本章では、まずシステムの前提となる要件を述べ、スケジューリングアルゴリズムの設計方針を説明し、提案するスケジューラのアーキテクチャを説明した後、各部のアルゴリズムを詳細に述べる。

3.1 前提

本研究では、一定の処理をしたのち終了するジョブを対象とし、ウェブサーバなど常駐型サービスは考えない。提案手法では、タスクがプリエンプション（一時停止）および再開できることを前提とする。また、簡単のため各ジョブは 1 つのタスクからなると仮定する。

3.2 多次元ビンパッキング問題との相似

各タスクの計算資源要求量が複数種かつ不均一な量のと看、クラスタスケジューリングは多次元ビンパッキング問題とみなせる。すなわち、計算資源が d 種類あるとき、各ノードに対応する d 次元の容量をもつビンに、各タスクに対応する d 次元の大きさをもつ直方体を詰め、ビン数を最小にする、すなわち効率よく直方体を詰める問題である。ビンパッキング問題は一次元かつオフラインの場合でも NP 困難であり、多次元の場合は ($P = NP$ でない限り) 効率的な近似アルゴリズムも存在しないことが知られている [22]。さらに、クラスタスケジューリングではタスクが動的に投入されたり完了したりするオンライン問題となり、最適なスケジューリングを計算することは現実的には不可能である。そこで、本研究ではクラスタスケジューリングを多次元ビンパッキング問題とみなし、さまざまなヒューリスティックスを適用することで、計算資源の利用効率および各ジョブのレイテンシを改善する。

3.3 アーキテクチャ

Kairos と同様、提案するクラスタスケジューラは、

- システムに 1 つ存在する中央スケジューラ (3.4 節)
- 各ノードに存在するノードスケジューラ (3.5 節)

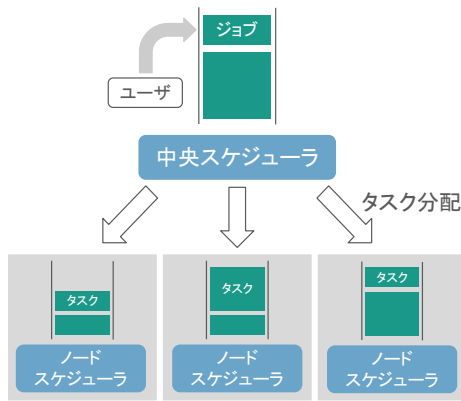


図 2: 提案するクラスタスケジューラのアーキテクチャ。

からなる2階層の分散型アーキテクチャをとる(図2)。投入されたジョブは、まず中央スケジューラがもつキューに保持される。中央スケジューラはジョブを投入された順にキューから取り出し、そのタスクを適切なノードスケジューラに分配する。その後、ノードスケジューラが各ノードで局所的にタスクをスケジューリングする。

プリエンプションされたタスクを他のノードにマイグレーションし再開することはコストが大きい[14]。したがって、分配されたタスクをノードごとにスケジューリングすることで、タスクの局所性を確保する。

3.4 中央スケジューラ

3.4.1 目標

提案手法は各タスクの推定実行時間に依存せず、またタスクのスケジューリングはノードスケジューラがおこなうため、中央スケジューラが各ジョブの完了までのレイテンシを短縮させることは困難である。そこで、中央スケジューラはスループットを向上させることを目標にタスクをノードスケジューラに分配する。レイテンシの短縮はノードスケジューラが担う(3.5節)。

また、分散型アーキテクチャを採ったため各ノードにおける負荷分散が重要になる。あるノードスケジューラにはノードの計算資源容量を超えるタスクが割当てられている一方、ほかのノードでは計算資源が余っているという状況では、計算資源が効率よく使用できていない。中央スケジューラはこのような状況を回避し、ノード間で割当てられた負荷のバランスをとることも目指す。

3.4.2 アルゴリズム

まず、中央スケジューラは計算資源の利用効率を向上させることでスループットを改善するよう、タスクをノードスケジューラに分配する。多次元ビンパッキング問題は現実的には解けないが、直方体を効率的にビンに詰めるヒューリスティックスが提案されており[23]、Tetris[6]によりクラスタスケジューラに適用された。この手法では、

各タスクについてその計算資源要求量と各ノードの使用可能な(他のタスクに割当てられていない)計算資源量との類似度を定義し、類似度が大きいノードを選択する。これにより、タスクの計算資源要求量がよく合致するノードを選択でき、フラグメンテーションを抑制し利用効率を向上させることができる。Tetrisは次に実行するタスクを決定するためにこのヒューリスティックスを用いたが、提案手法ではタスクを分配するノードを決定するために用いる。

具体的にはまず、各タスクについてノードとの類似度を「タスクの計算資源要求量」と「ノードで使用可能な計算資源量」の内積 $\sum_{r \in \mathcal{R}} \frac{D_r F_r}{C_r^2}$ と定める。 $\mathcal{R}$ は計算資源の種類の集合であり、CPUやメモリが含まれる。 D_r , F_r はそれぞれ計算資源 r についてのタスクの要求量およびノードの使用可能量である。種類間の単位の違いに影響されないようにするために、これらをノードの容量 C_r で割ることで正規化する。このとき、全ての計算資源 r について $D_r \leq F_r$ でなければ、容量を超えるためこのノードは除く。そして、最も類似度が大きいノードがあれば、タスクをそのノードに分配する。

分散型アーキテクチャのクラスタスケジューラでは、ノードでタスクが完了したときすぐに他のタスクを開始できるよう、ノードの容量を超えてタスクを分配する[24,25]。また、提案手法では新しく投入されたタスクをすぐ実行するために、容量を超えてタスクを分配されているノードにも新しいタスクを分配する(3.5節)。このとき、分配する合計タスク数の制御が必要である[8]。

そこで、全ノードがタスクを分配すると容量を超える場合、最も負荷の小さいノードに分配する。ノードの負荷は、分配され未完了の全タスクについて計算資源要求量の合計をとり、そのノルム $\sqrt{\sum_{r \in \mathcal{R}} \left(\frac{D_r}{C_r}\right)^2}$ と定める。このように、負荷の小さいノードに分配することでノード間の負荷分散を図る。ここで、負荷が一定値 L を超えているノードにはタスクを分配しない。全ノードの負荷が L を越えている場合、中央スケジューラはタスクの分配を停止し、少なくとも1つのノードの負荷が L を下回るまで待機する。

3.5 ノードスケジューラ

3.5.1 目標

1章で述べたとおり、近年のクラスタにおけるワークロードには実行時間の不均一性が見られる[1,2]。短いタスクが大部分を占める一方、長いタスクも存在する裾の長い分布であり、実行時間の分散は非常に大きくなっている。短いタスクからなるジョブは、ユーザが投入してから完了するまでのレイテンシを短くする要求が厳しい。そこで、ノードスケジューラは短いタスクを優先して実行することで、HoLブロッキングを回避しレイテンシを短縮することを目標にスケジューリングする。

3.5.2 LAS

各タスクの実行時間が不明な状況において短いタスクを優先するために、LAS スケジューリング戦略 [20] が有効である。LAS はスケジューリングする時点においてジョブまたはタスクがそれまでに実行した時間を累積し、累積実行時間が短いものを優先する。累積実行時間で真の実行時間を近似することで、短いタスクのレイテンシを短縮することを意図した戦略である。実行時間推定に依存しないクラスタスケジューラである LAS_MQ [13] や Kairos が LAS を用いている。LAS はタスクの実行時間が裾の長い分布になっているとき特に効果が高いため [26]、この特徴をもつ近年のクラスタに用いると有効であることが期待される。累積実行時間が短いタスクは合計実行時間も短い確率が高いため、これを優先することで高確率で短いタスクのレイテンシを短縮できる。

提案手法でも、ノードスケジューラが各ノードで LAS に従いタスクをスケジューリングする。各タスクの累積実行時間を追跡し、あるタスクが使用可能な計算資源の不足により実行できない場合、それより累積実行時間が長いタスクをプリエンブション（一時停止）し、短いタスクを代わりに実行する。提案手法のノードスケジューラでは、この過程でタスクの複数種かつ不均一な計算資源要求量を適切に扱い LAS を実現する。

3.5.3 アルゴリズム

まず、新しいタスクがノードスケジューラに分配されたとき、このタスクをすぐに実行する。タスクの実行時間が裾の長い分布のとき、新しいタスクは短い可能性が高いため、すぐに実行することでレイテンシを短縮できる。

ノードの使用可能な計算資源が十分にあればこのタスクを実行できるが、不足している場合、他のタスクをプリエンブションする。プリエンブションされるタスクは次のように決定する。まず、実行したいタスクを t とし、実行中のタスクを累積実行時間が長い順に並び替え $r_0, r_1, r_2, \dots$ とする。 $r_0, r_1, r_2, \dots$ から 1 つまたは複数プリエンブションし、 t を代わりに実行することとなる。基本的には累積実行時間が長いタスクをプリエンブションするが、計算資源要求量の不均一性のため、長いタスクから貪欲にプリエンブションすると不要なプリエンブションを発生させることがある。例えば、ノードの計算資源が全て割当て済みの状態で、 t が 〈CPU 2 コア、メモリ 4 GB〉 要求し、 r_0, r_1 にそれぞれ 〈1 コア、1 GB〉、〈2 コア、4 GB〉 割当てられていた場合、 r_0 をプリエンブションする必要はなく、 r_1 のみプリエンブションすればよい。そこで、次の順でプリエンブション候補を走査する。

$r_0,$
 $r_1, \quad r_0 + r_1,$
 $r_2, \quad r_0 + r_2, \quad r_1 + r_2, \quad r_0 + r_1 + r_2,$
 $\dots$

リスト 1: プリエンブション候補を生成するアルゴリズム。

```

1 running_tasks = [r0, r1, r2, ..]
2 candidates = []
3 while not running_tasks.is_empty():
4     r = running_tasks.pop()
5     candidates.push(r)
6
7     len = candidates.length()
8     for i from 0 to len - 1:
9         candidates.push(r + candidates[i])

```

$r_0 + r_1$ は r_0 と r_1 の両方をプリエンブションすることを意味する。この順でプリエンブション候補を走査し、 t が要求する計算資源が得られるのが見つかり次第、その候補をプリエンブションし t を実行する。前述の例では r_0 をプリエンブションしても t が要求する計算資源は得られないため、 r_0 はプリエンブションせず r_1 を次に調べる。このように不要なプリエンブションを回避することで、全体のプリエンブション回数を削減しレイテンシを短縮する。

このプリエンブション候補はリスト 1 のアルゴリズムで生成できる。実行中タスクが n 個あるとき、候補は最大で $2^n - 1$ 個生成される。 n が大きいと全候補の走査にかかる時間が長くなるため、一定値 N 個まで、すなわち r_0 から r_{N-1} までのタスクを調べることにする。

累積実行時間の長い N 個の実行中タスクを全て仮にプリエンブションしても、新しいタスクに要求する計算資源を割当てられない場合、他の新しいタスクのスケジューリングに移る。以上の処理を新しいタスク全てについて実行する。

プリエンブションし一時停止中のタスクは累積実行時間が長い順にソートしておき、この順にスケジューリングを試す。すなわち、ノードの使用可能な計算資源が十分にあれば先頭のタスクを実行し、不足している場合、実行中タスクのプリエンブションを試す。先頭の停止中タスクより累積実行時間が長い実行中タスクを長い順に最大 N 個考え、新しいタスクと同様の処理をおこなう。

このとき、実行中タスクが頻繁にプリエンブションされることを防ぐため、実行開始または再開から一定時間 W 経っていない実行中タスクはプリエンブション候補から除く。クラスタで実行されるワークロードに W より実行時間が短いタスクが多い場合、多くのタスクはプリエンブションされずに完了できるためレイテンシが短縮される。しかし、 W を大きく設定すると停止中タスクが実行を再開するまでの時間が長くなる。

4. 評価

4.1 比較対象とワークロード

提案するクラスタスケジューラの評価のため、Kairos [14] との比較実験をおこなった。ただし、Kairos は各タスクの計算資源要求量が CPU 1 コアのみと仮定しているため、複数種かつ不均一な計算資源要求量を持つタスクを入力できるよう単純な拡張をおこなった。具体的には、Kairos のノードスケジューラが停止中タスクを実行するために実行中タスクをプリエンブションする際、停止中タスクが要求する計算資源が得られるまで、実行中タスクを累積実行時間が長い順に貪欲にプリエンブションするよう変更を加えた。なお、Kairos の中央スケジューラのパラメータは、最良の性能を示すよう $Q = 32$ と設定した。

評価に用いたワークロードは、Google のクラスタのトレース [3] および Alibaba のクラスタのトレース (2018 年版) [4] の 2 つである。トレース全体に含まれるタスク数が多いため、Google のトレースは 256 個、Alibaba のトレースは 2,048 個おきにタスクを抽出し、それぞれ全体で 69,524, 648,052 個のタスクをもつワークロードを生成した。これらのトレースは各タスクの CPU およびメモリの要求量と、タスクの投入、開始、完了時刻のデータをもつ。開始から完了までの時間をタスクの実行時間とした。

4.2 実験環境

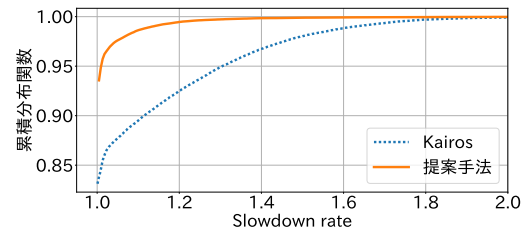
提案手法の評価のため、クラスタのシミュレータを作成した。シミュレータ内でスケジューリングにかかる時間、およびタスクのプリエンブションや再開にかかる時間はゼロと仮定した。

以下、特に断りのない限り、シミュレートしたクラスタは 256 ノードからなる。使用したトレースでは実行環境の計算資源容量の具体値は公表されていないため、Kairos でスケジューリングした際に各計算資源の使用効率が平均して 90% 程度になるよう設定した。入力するジョブは、計算資源が CPU とメモリの 2 種類あるため、クラスタの負荷が約 $\sqrt{2}$ に保たれる頻度で投入した。

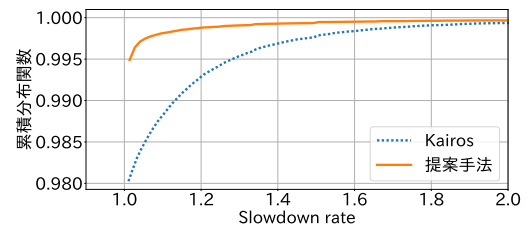
4.3 評価結果

本節の実験では、提案するスケジューラにおけるノードの負荷の最大値は $L = 1.5$ 、ノードスケジューラで走査する実行中タスクの最大数は $N = 4$ 、各タスクが停止中タスクにプリエンブションされるようになるまでの最小実行時間は、Google のトレースを用いるときは $W = 60$ s, Alibaba のトレースを用いるときは $W = 30$ s とした。

図 3 は各タスクのスケジューリング待機による遅延 (slowdown rate) の累積分布関数を示す。Slowdown rate は、各タスクについて $1 + \frac{\text{スケジューリング待機時間}}{\text{実行時間}}$ で定義され



(a) Google のトレースを用いた場合。



(b) Alibaba のトレースを用いた場合。

図 3: Slowdown rate の累積分布関数。

る指標である。実行時間が短いタスクを重視し投入から完了までのレイテンシを測る指標となっている。図 3 より、提案手法により slowdown rate の改善が見られ、多くのタスクについて完了までのレイテンシが短縮されたことがわかる。Google のトレースでは、提案手法は Kairos に比べ 95 パーセンタイルを約 28.2%, 99 パーセンタイルを約 30.7% 短縮した。また、Alibaba のトレースでは 95 パーセンタイルは Kairos においても 1.00 だったが、99 パーセンタイルを約 11.5% 短縮した。

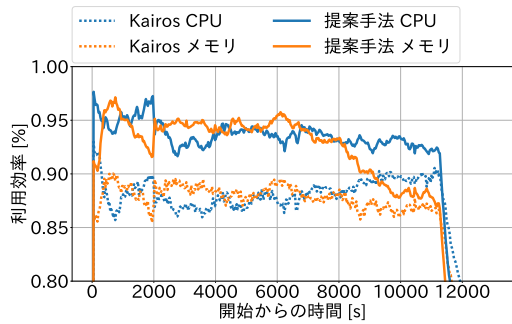
この改善は、提案手法が中央スケジューラで計算資源の利用効率を向上させ、またノードスケジューラでプリエンブション回数を削減したこと起因すると考えられる。図 4 はクラスタ全体における計算資源の利用効率を示す。提案手法では、CPU、メモリともに利用効率が Kairos に比べ向上していることがわかる。また、ワークロード全体のプリエンブション回数は、Kairos に比べ、Google のトレースでは 246,664 回から 24,025 回へ約 9.74% まで削減し、Alibaba のトレースでは 79,531 回から 6,318 回へ約 7.94% まで削減した。

4.4 パラメータ設定の感度分析

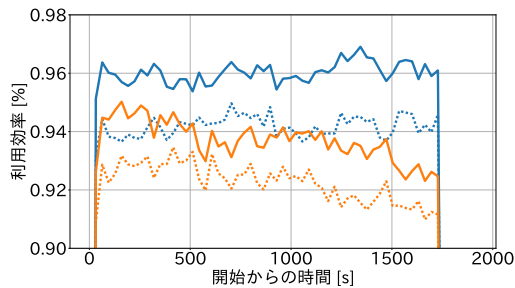
本節では、提案するスケジューラがもつパラメータ L , N , W の設定値によるスケジューリング性能の変化を評価する。また、ノード数による性能の変化も評価する。なお、各評価において固定したパラメータの値は 4.3 節で用いたものと同じである。

図 5 は、ノードの負荷の最大値 L を変えたときの slowdown rate の累積分布関数を示す。 L の値によるスケジューリング性能の変化はほぼ見られず、提案手法が L の設定に頑強であることがわかる。

図 6 は、ノードスケジューラでプリエンブション時に走



(a) Google のトレースを用いた場合.



(b) Alibaba のトレースを用いた場合.

図 4: クラスタ全体における計算資源の利用効率.

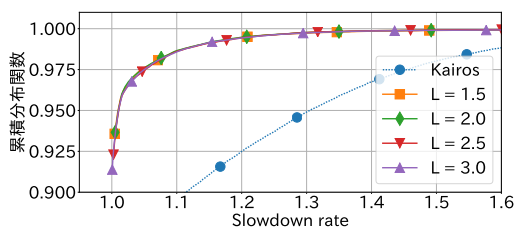


図 5: L を変えたときの slowdown rate の累積分布関数.

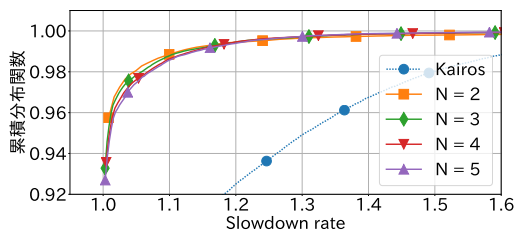


図 6: N を変えたときの slowdown rate の累積分布関数.

査する実行中タスクの最大数 N を変えたときの slowdown rate の累積分布関数を示す。 N の値によらず、提案手法は Kairos より小さい slowdown rate を示している。約 99 パーセント以上までは N が小さいほど slowdown rate も小さい。これは、 N が小さい場合はプリエンブションが起りにくいこと、少ない被プリエンブション回数で完了したタスクが多かったことによると考えられる。一方、約 99 パーセント以上では大きい N で slowdown rate が小さくなっている。これは、 N が大きい場合はプリエンブションが起りやすいため、多くのタスクが他のタスクをプリエンブションし実行されたことによると考えられる。

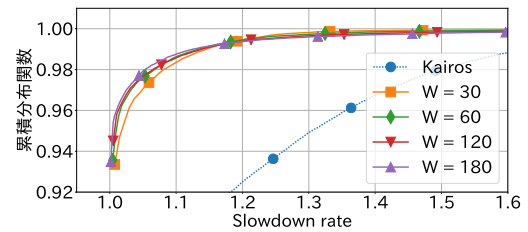


図 7: W を変えたときの slowdown rate の累積分布関数.

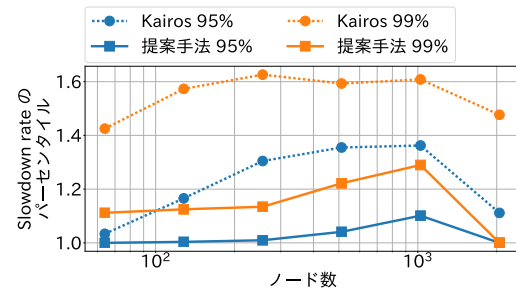


図 8: ノード数を変えたときの slowdown rate の累積分布関数.

図 7 は、各タスクがプリエンブションされるようになるまでの最小実行時間 W を変えたときの slowdown rate の累積分布関数を示す。 W の値によらず、提案手法は Kairos より小さい slowdown rate を示している。 W が大きい場合、実行中タスクがプリエンブションされにくいこと、約 99 パーセント以上において slowdown rate が小さくなっている。一方、 W が小さい場合、停止中タスクが他のタスクをプリエンブションしやすいこと、約 99 パーセント以上で slowdown rate が小さくなっている。

図 8 は、クラスタのノード数を変えたときの slowdown rate の累積分布関数を示す。実験の範囲では、ノード数によるスケジューリング性能の変化の傾向は見られなかったが、ノード数によらず、提案手法は Kairos より小さい slowdown rate を示している。

5. 結論

本研究では、複数種かつ不均一な計算資源要求量に対応した、実行時間推定に依存しないクラスタスケジューラを提案した。提案手法では、クラスタスケジューリングを多次元ビンパッキング問題とみなし、さまざまなヒューリスティックスを適用することで、各タスクの slowdown rate を短縮した。シミュレーションによる評価の結果、提案手法は既存研究に比べ、slowdown rate の 95 パーセントを約 28.2%、99 パーセントを約 30.7% 短縮した。

今後の課題として、まず、複数のタスクからなるジョブへの対応が挙げられる。また、ジョブ間の公平性を保証する機構の開発も必要だと考えている。さらに、提案手法を広く使われているスケジューラフレームワーク上に実装し、実環境での評価もおこなう予定である。

参考文献

- [1] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H. and Kozuch, M. A.: Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis, *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, ACM (2012).
- [2] Lu, C., Ye, K., Xu, G., Xu, C. Z. and Bai, T.: Imbalance in the Cloud: an Aalysis on Alibaba Cluster Trace, *Proceedings of 2017 IEEE International Conference on Big Data*, BigData 2017, IEEE, pp. 2884–2892 (2017).
- [3] Google: Borg cluster traces from Google, <https://github.com/google/cluster-data>.
- [4] Alibaba: cluster data collected from production clusters in Alibaba for cluster management research, <https://github.com/alibaba/clusterdata>.
- [5] Lifka, D. A.: The ANL/IBM SP scheduling system, *Job Scheduling Strategies for Parallel Processing*, pp. 295–303 (1995).
- [6] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S. and Akella, A.: Multi-Resource Packing for Cluster Schedulers, *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, ACM, pp. 455–466 (2014).
- [7] Grandl, R., Kandula, S., Rao, S., Akella, A. and Kulkarni Microso and, J.: Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters, *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, USENIX Association, pp. 81–97 (2016).
- [8] Rasley, J., Karanasos, K., Kandula, S., Fonseca, R., Vojnovic, M. and Rao, S.: Efficient Queue Management for Cluster Scheduling, *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, ACM (2016).
- [9] Delgado, P., Dinu, F., Kermarrec, A.-M. and Zwaenepoel, W.: Hawk: Hybrid Datacenter Scheduling, *Proceedings of 2015 USENIX Annual Technical Conference*, USENIX ATC '15, USENIX Association, pp. 499–510 (2015).
- [10] Delgado, P., Didona, D., Dinu, F. and Zwaenepoel, W.: Job-aware Scheduling in Eagle: Divide and Stick to Your Probes, *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, ACM, pp. 497–509 (2016).
- [11] Chen, W., Rao, J. and Zhou, X.: Preemptive , Low Latency Datacenter Scheduling via Lightweight Virtualization, *Proceedings of 2017 USENIX Annual Technical Conference*, USENIX ATC '17, USENIX Association, pp. 251–263 (2017).
- [12] Park, J. W., Tumanov, A., Jiang, A., Kozuch, M. A. and Ganger, G. R.: 3Sigma: Distribution-based cluster scheduling for runtime uncertainty, *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, ACM (2018).
- [13] Hu, Z., Li, B., Qin, Z. and Goh, R. S. M.: Job Scheduling without Prior Information in Big Data Processing Systems, *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*, ICDCS 2017, IEEE, pp. 572–582 (2017).
- [14] Delgado, P., Didona, D., Dinu, F. and Zwaenepoel, W.: Kairos: Preemptive Data Center Scheduling Without Runtime Estimates, *Proceedings of ACM Symposium on Cloud Computing 2018*, SoCC '18, ACM, pp. 135–148 (2018).
- [15] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B. and Baldeschwieler, E.: Apache Hadoop YARN: Yet Another Resource Negotiator, *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, ACM (2013).
- [16] Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I. and Zhou, J.: Re-optimizing Data-Parallel Computing, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, USENIX Association (2012).
- [17] Delimitrou, C. and Kozyrakis, C.: Quasar: Resource-efficient and QoS-aware Cluster Management, *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, ACM, pp. 127–144 (2014).
- [18] Ferguson, A. D., Bodik, P., Boutin, E. and Fonseca, R.: Jockey : Guaranteed Job Latency in Data Parallel Clusters, *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, ACM, pp. 99–112 (2012).
- [19] Tumanov, A., Jiang, A., Park, J. W., Kozuch, M. A. and Ganger, G. R.: JamaisVu: Robust Scheduling with Auto-Estimated Job Runtimes, Technical report (2016).
- [20] Nuyens, M. and Wierman, A.: The Foreground-Background queue: A survey, *Performance Evaluation*, Vol. 65, pp. 286–307 (2008).
- [21] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S. and Stoica, I.: Dominant Resource Fairness : Fair Allocation of Multiple Resource Types, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, USENIX Association (2011).
- [22] Woeginger, G. J.: There is no asymptotic PTAS for two-dimensional vector packing, *Information Processing Letters*, Vol. 64, No. 6, pp. 293–297 (1997).
- [23] Panigrahy, R., Talwar, K., Uyeda, L. and Wieder, U.: Heuristics for Vector Bin Packing, Technical report (2011).
- [24] Ousterhout, K., Wendell, P., Zaharia, M. and Stoica, I.: Sparrow: Distributed, Low Latency Scheduling, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, ACM, pp. 69–84 (2013).
- [25] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M. and Zhou, L.: Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing, *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, USENIX Association, pp. 285–300 (2014).
- [26] Rai, I. A., Urvoy-Keller, G. and Biersack, E. W.: Analysis of LAS Scheduling for Job Size Distributions with High Variance, *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, ACM, pp. 218–228 (2003).