Vol.2020-OS-150 No.9
2020/7/30

マルチクラスタ環境における不均一性を考慮したジョブスケジューリングアルゴリズムの設計

齋藤 峻[1]　廣津 登志夫[2]

概要：大規模 HPC クラスタ環境では，一多数の同一構成のノードにより複数のジョブを処理することが一般的である．しかし，昨今の高性能計算に要求される多様性を考えると，タスクに応じて適切なノード構成が異なってくることが考えられる．このため，今後の計算クラスタでは一部に異なる構成のノードが混在するマルチクラスタ環境が増えてくることが想定される．しかし，従来ジョブスケジューラではクラスタ内部の状態を把握できないため，このマルチクラスタ環境におけるハードウェアやインストールされているソフトウェアの違いといった不均一性を考慮することができない．そこで本研究ではマルチクラスタ環境におけるジョブスケジューリングの制約に対応するマルチクラスタスケジューリングアルゴリズムを提案する．提案したアルゴリズムをシミュレータ上で実装し，既存のスケジューリングアルゴリズムと比較することで有効性を示す．

1. 序論

HPC(High Performance Computing) クラスタでは大規模な計算リソースを有したノードを複数台接続し，多数のジョブを並列に実行している．このクラスタを構成するノードは，一般的に同一のハードウェアリソースを有し，OS やソフトウェアなど内部構造が同一になるように構築されている．昨今，HPC クラスタの利用目的の多様化が進んでおり，従来から広く使われていた科学技術計算から深層学習の処理まで，幅広いジョブに対応する必要が生じている．これに対する一つのアプローチは，多様な処理に対応するプロセッサを開発することであるが，そのために必要となるコストが大きくなる可能性が高い．

従来型の計算処理のを行う基本計算ノードの一部に例えば GPU アクセラレータのような計算処理機能を付加したり，クラスタの一部に異なる計算に向いた拡張計算ノード群を導入したりといった構成法が考えられる．このようなマルチクラスタ環境は，それぞれの基本ノードや拡張ノードのそれぞれのアーキテクチャ向けのソフトウェア資産を活用することができることが利点であるが，その一方で，ジョブの実行要求に応じて適切に計算資源を割り当てるジョブスケジューラの構成が重要になる．

従来のスケジューラはハードウェアリソースの空きをみてスケジューリングを行っているが，マルチクラスタ環境ではハードウェアのバージョンなどを考慮してスケジューリングする必要がある．そのため従来のスケジューラではマルチクラス環境において十分なスケジューリングを行うことができない．またクラスタによってインストールされているライブラリやソフトウェアの種類やバージョンが異なる場合，その点を考慮してスケジューリングすることも必要となる．ジョブの種類によっては，一方のノード群の上でのみ実行可能なものや，両方のノード群で実行可能なもの，実行するノード群によって著しく性能がでるものが考えられる．

ジョブをどのクラスタに設置するか，またどのノードに配置するかということに対する一つのアプローチは，ノード群毎にジョブキューを用意することであるが，どちらでも動くジョブの存在を考えると，ノード群の稼働状況や要求されているジョブの状況に応じた柔軟な処理が実現されることが望ましい．そのためマルチクラスタ環境で発生するハードウェアやソフトウェアの違いといった特有の不均一性を考慮したジョブスケジューリングを行う必要がある．

本研究では既存のジョブスケジューリングアルゴリズムに加え，マルチクラスタ環境における不均一性を考慮したマルチクラスタジョブスケジューリングアルゴリズムを提案する．ハードウェア面ではリソース情報と CPU の拡張命令セットや GPU の Capability，またソフトウェア面ではバージョンやハードウェアにおける実行制約条件，インストールされているライブラリやソフトウェア間の依存を

[1] 法政大学大学院 情報科学研究科
Hosei University Graduate School
[2] 法政大学 情報科学部
Hosei University

ⓒ 2020 Information Processing Society of Japan

考慮したアルゴリズムにすることで，従来のハードウェアリソースのみのスケジューリングよりも幅広い条件でスケジューリングすることが可能になる．その結果従来のスケジューリング方法では実行が不安定なジョブや実行速度が低下してしまうジョブに対しても，適切なハードウェア・ソフトウェア上で実行することで安定性が増す．

本研究ではまず提案したマルチクラスタアルゴリズムを用いたジョブスケジューリングシミュレータを実装する．様々なジョブやクラスタ環境でシミュレーションを行うことで不均一性を考慮したアルゴリズムの優位性を示す．また全体的なジョブの実行時間を計測し，本研究のアルゴリズムが既存のハードウェアリソースベースのスケジューリングアルゴリズムよりもマルチクラスタ環境において効果的であることを示す．

2. 関連研究

ジョブマネージャーの TORQUE(Tera-scale Open-source Resource and QUEue manager)[1] や OpenStack[2]，Kubernetes[3] といった近年大規模に開発されているシステムではハードウェアリソースを考慮したスケジューリングを行っている．この方法はクラスタ内のノードにおけるハードウェアリソースの空きを監視し，全体のリソース使用率が均等になるようにスケジューリングされる．しかしながらこの方法では実際にジョブが実行できるのか，またジョブの実行時間においてノード間で差が生じるかなどといった，マルチクラスタの不均一性に対するスケジューリングをすることができない．

関連研究として Pigeon[4] では階層型のジョブスケジューラを実装することでジョブの実行時間や実行順序の優先度ごとにスケジューリングすることを可能にしている．ここでは，ジョブの実行時間やジョブの計算量といったジョブの性質について，それぞれのジョブキューを用意することで，個別のポリシでスケジュールすることができる．これによりマルチクラスタのような環境においてもジョブの実行時間を鑑みてスケジューリングすることができる．しかし，マルチクラスタにおける，ノードの種別やソフトウェアの制約などの不均一性を全て個別のキューで実現することは困難であり，さらにジョブの性質が増えるとその組み合わせでキューを作る必要が生じる．そのため本研究では新たな不均一性を定義しスケジューリングアルゴリズムを提案する．

3. マルチクラスタ環境における不均一性

この章ではマルチクラスタの定義とそのマルチクラスタ環境における問題点であるノードの不均一性について述べる．シングルクラスタとマルチクラスタの違いに言及し，そのマルチクラスタ環境におけるノードの差異である不均一性が何であるか，何故問題点となるか明確にする．

3.1 マルチクラスタ

コンピュータにおけるクラスタとは複数のコンピュータをネットワークで接続し一つの大きな群を形成しているシステムのことである．クラスタは単一のコンピュータでは得られない多くのハードウェアリソースを持つことが可能である．そのリソースを使用することで大規模な計算を実行することができる．クラスタを用いて行われる計算には数値計算から機械学習や自然言語処理など幅広い分野の計算処理があり，クラスタが利用される場面は増加している．

マルチクラスタは差異のあるノードがクラスタを形成しているシステムと定義する．そもそもクラスタ内のコンピュータは同一のハードウェアリソースを使用し，内部の OS やインストールされたソフトウェアやライブラリのバージョンも統一される．そのためクラスタ内は同一のコンピュータしか存在しない．しかしマルチクラスタではクラスタ内のコンピュータに差異がある．ここにおける差異とは CPU や GPU などのハードウェアリソースからソフトウェアの種類やバージョンの違いのことを指す．

マルチクラスタは部分リプレースや新規コンピュータのクラスタ投入，既存クラスタ間の合併などで発生する．クラスタ内部のコンピュータがなんらかの原因で正常に動作しない場合に該当コンピュータのみ交換すると部分リプレースが起こる．また既存クラスタ間の合併は複数のクラスタをまとめることで発生する．このように様々な要因からマルチクラスタが発生する可能性があると考える．

3.2 マルチクラスタ環境における不均一性

第 3.1 章で定義したマルチクラスタ環境における差異を不均一性と呼ぶ．不均一性は様々な要素があり，それは CPU・GPU などのハードウェアやライブラリ・ソフトウェア，ネットワークが挙げられる．

CPU・GPU などでは CPU の種類や世代によって搭載されている拡張命令セットや GPU の Capability が異なることがある．これによりライブラリ・ソフトウェアに対して依存性が発生し，拡張命令セットや Capability によって実行できないことや処理が遅くなるといった状況が生じることになる．

ライブラリ・ソフトウェアにおける不均一性はバージョンの違いによる差やインストールされているライブラリ・ソフトウェア間の依存，ライセンスによる制約などがある．バージョンによってプログラムの動作が異なることがある．また依存関係によりインストールするライブラリに違いが生じる．ライセンスによりクラスタ内部の全台にインストールできずにインストールされているコンピュータとされていないコンピュータが発生することもある．

マルチクラスタでは同一データセンター内の同一ネットワーク内にコンピュータが存在せず，地理的にも離れた場所にコンピュータが存在することもある．データのやりと

りや同期などをネットワークを介して行うと同一ネットワーク内で通信するよりも地理的な差による遅延が発生してしまう．

このようなマルチクラスタ環境において不均一性がもたらす制約を考えることでさらに最適なマルチクラスタ運用ができる．マルチクラスタ環境における不均一性を考慮し計算処理の実行するコンピュータを選択することで既存のハードウェアリソースで選択しているアルゴリズムよりも全体のジョブ実行時間を減少させることができる．

4. マルチクラスタ環境における不均一性を考慮したスケジューリングアルゴリズム

この章では第 3 章で記述したマルチクラスタ環境における不均一性を考慮したスケジューリングアルゴリズムを提案する．不均一性の要素を定義し，既存のスケジューリングアルゴリズムと本研究で提案する不均一性を考慮したアルゴリズムを比較する．

4.1 不均一性をもたらす要因

第 3.2 章で述べたようにマルチクラスタ環境ではシングルクラスタ環境と比較し様々な不均一性が発生する．この不均一性をアルゴリズムにするため，各不均一性の定義をする．本研究で定義する不均一性は 3 つであり，以下の通りである．

- CPU や GPU などハードウェアの不均一性
- ライブラリ・ソフトウェアの不均一性
- 実行環境によるジョブ動作の不均一性

ハードウェアの不均一性では既存のスケジューリングアルゴリズムで評価しているハードウェアリソースを用いるほか，CPU の世代やクロック数，拡張命令セットなどを評価する．同様に GPU ではコア数や Capabirity を評価し，CPU や GPU のジョブが実行できるか確認する．ジョブを $J = J_1, J_2, ...J_n$ とする．このときジョブ J_i がその CPU と GPU で動作することができるかどうかを $CPUCheck$ と $GPUCheck$ で表す．

$$CPUCheck(J_i) = IsVersion(J_i) \times IsSSE(J_i) \quad (1)$$

$$GPUCheck(J_i) = IsCapabirity(J_i) \quad (2)$$

$IsVersion$ と $IsSSE$，$IsCapabirity$ はその CPU や GPU で動作するか真偽値 (0 1) で返す．ジョブ J には必要条件が記載されているため，その該当箇所と比較することで動作するか判断する．

ライブラリ・ソフトウェアの不均一性ではジョブを実行するために必要なソフトウェアがインストールされているか判断する．またそのソフトウェアが依存しているライブラリ・ソフトウェアを確認する．この際，ライブラリ・ソフトウェア名のみではなく，バージョンまで確認する．これによりバージョンの違いによるジョブの動作への影響がなくなる．ライブラリ・ソフトウェアを L とし，このときの式を $LibraryCheck$ で示す．

$$LibraryCheck(L_i) = InstalledLibrary(L_i) \times InstalledDependencyLibraries(L_i) \quad (3)$$

この $LibraryCheck$ をジョブの動作に必要なソフトウェア数実行する．これによりそのノード上でジョブの必要ソフトウェアを確認することができる．

これらの式 1，式 2，式 3 を使用し，ノードの選択を行う．また従来のアルゴリズム同様ハードウェアリソースの確認も行う．

アルゴリズム 1 従来のスケジューリングアルゴリズム

```
Input: クラスタのノード情報 N_x，実行するジョブ J
Output: ジョブを実行するノード TN
  TN ← nil
  for i = 0, ··· , N.length − 1 do
    if N_i.available > J.usage then
      if N_i.available > TN.available then
        TN ← N_i
      end if
    end if
  end for
  return TN
```

4.2 ベースラインアルゴリズム

本研究で提案するスケジュールリングアルゴリズムに対するベースラインとなる比較対象として，従来のハードウェアリソースのみを考慮したスケジューリングを用いる．多くのシステムが初期のスケジューリングをこの方式で実行しているが，今回はジョブマネージャーの TORQUE を参考にした．ベースラインのアルゴリズムとしてアルゴリズム 1 に示す．このアルゴリズム 1 ではノードの空きリソースを取得し，実行するジョブの必要リソース量が確保できるか確認する．確保できるノードの中で，最適なノードを選択する．この選択の方法は複数あり，大きく分けるとクラスタ全体でリソース使用率が均等になるようにジョブを実行する方法と実行可能なノードを発見次第そのノードに配置することを決定しジョブを実行する方法である．今回は前者のクラスタ全体でリソース使用率が均等になるようにジョブを配置し実行する方法をとる．

4.3 マルチクラスタスケジューリングアルゴリズム

本研究で提案するマルチクラスタ環境におけるスケジューリングアルゴリズムは，マルチクラスタ環境で発生する不均一性を考慮してスケジューリングする方法である．その不均一性については第 3.2 章で述べ，アルゴリズムの側面から見た不均一性の要素定義については第 4.1 章にて述べ

た．ここではこれらの章で述べた不均一性を考慮したアルゴリズムについて述べる．

アルゴリズム 2 は本研究で提案するアルゴリズムである．これはベースアルゴリズムと同様にハードウェアリソースを確認するほか，第 4.1 章で述べた式 1，式 2，式 3 を使用することでマルチクラスタの不均一性を考えたスケジューリングを行うことができる．$CPUGPULibrariesCheck(J)$ が上記の式をまとめて判断している．ハードウェアの使用可能なリソースを確認したのち，CPU や GPU，インストールされているソフトウェアがジョブの実行条件にを満たしているか判断する．

5. シミュレータ

本研究ではマルチクラスタ環境における不均一性を考慮したスケジューリングの優位性を示すためにシミュレータを実装した．実装したシミュレータはジョブスケジューリングに発生するイベントでシミュレーションを行うイベント駆動型シミュレータである．イベント駆動型シミュレータは状態の変化が伴う事象をイベントとして定義し，そのイベントによって次の状態に遷移するというシミュレーション方法で，ノードやジョブに関するイベントを定義することでジョブスケジューリングのシミュレーションを行うことができる．実装は Go 1.14.2 を用いて行った．

5.1 ジョブスケジューリングのイベント定義

本研究のイベント駆動型シミュレータではジョブのスケジューリングをシミュレーションするため，ノードの状態が変化するであろう事象をイベントとして定義する．ノードの状態とは空いているハードウェアリソースの変化のことを指し，CPU 使用率やメモリ使用率，GPU の占有率の変化，ジョブの実行状態などを言う．このように状態変化が伴う事象をイベントとして定義し，そのイベントごとに応じた時間を観測することでシミュレーションした際の時間を計測することができる．本研究のジョブスケジューリングで発生するイベントとして今回のシミュレータでは 3 つ定義した．そのイベントは以下の 3 つであり，各イベントについて詳細を説明する．

- マルチクラスタ環境にジョブを投入
- ジョブのスケジューリング
- ジョブの終了

1 つ目のイベントはジョブのキューイングイベントである．これはジョブがクラスタのジョブキューにジョブが投入される．ジョブがキューに投入されるとジョブのスケジューリングイベントが追加される．追加されるとジョブのキューイング時間 Q_t かかる．2 つ目のイベントはジョブのスケジューリングイベントである．これはジョブの投入イベントから発火する．このイベントでは与えられたジョブをアルゴリズムに基づいて実行ノードを指定する．実行ノードが選択された場合，ジョブは正常に実行されたと見なしジョブの終了イベントを作成する．このイベントも同様にスケジューリング時間 S_t かかる．最後のイベントはジョブの終了イベントである．このイベントはジョブの実行が終了したことを示すイベントであり，実行ノードのリソースを解放する．またジョブの実行にかかる時間 E_t を全体の時間に追加する．以上のイベントを繰り返し実行することでジョブのスケジューリングのシミュレーションを実行できる．

アルゴリズム 2 マルチクラスタスケジューリングアルゴリズム

Input: クラスタのノード情報 N_x，実行するジョブ J
Output: ジョブを実行するノード TN

```
TN ← nil
for i = 0, ··· , N.length − 1 do
    if N_i.available > J.usage then
        if CPUGPULibrariesCheck(J) then
            if N_i.available > T.available then
                TN ← N_i
            end if
        end if
    end if
end for
return TN
```

5.2 シミュレータの設計

本研究で実装したシミュレータにはジョブスケジューリングに必要な要素をモデル化し，実装を行った．表 1, 表 2 ではそれぞれノードとジョブのモデルを表している．ノードではハードウェアリソースとソフトウェアをもつ．これにより CPU やメモリ，GPU の使用率やどのノードに何がインストールされているかの情報を有することができる．Cpu や Gpus にはノード上に搭載している CPU や GPU

表 1: Node のモデル

モデル	要素
Node	Id
	Name
	Cpu
	Gpus
	MaxCpu
	MaxMem
	MaxGpu
	MinCpu
	MinMem
	MinGpu
	AvailableCpu
	AvailableMem
	AvailableGpu
	Libraries

表 2: Job のモデル

モデル	要素
Job	Id
	Name
	RequiredCpu
	RequiredCpuSSE
	RequiredGpu
	RequiredGpuCapability
	RequiredMem
	RequiredSoftware
	Runtime

情報を所持しており，CPU の種類や拡張命令セット，GPU の Capability などが判断できるようになっている．また各 CPU のクロック数や GPU のコア数などもノードの情報として格納しているため，搭載されている CPU や GPU によって速度の差を表現できるようにしている．さらに Libraries ではインストールされているライブラリ・ソフトウェアとともに，それらのライブラリの依存先ライブラリを情報として格納している．ジョブは実行時のノードに対する必要条件をもつ．CPU のコア数や拡張命令セット，GPU の数や Capability，実行時に必要なソフトウェアである．この RequiredSoftware では Node 同様に必須ソフトウェアの依存関係を格納しているため，Node の Libraries と比較することでそのジョブを該当ノードで実行できるか判断することができる．

第 5.1 章で定義したイベントを使用したシミュレーションの流れをフローチャートで示したものが図 1 である．まずシミュレーションを実行する上で必要な要素の初期化を行う．その後最初のジョブが投入されシミュレーションが始まる．投入された時点の時刻を 0 とし，その後のイベントで発生する時刻を計測する．ジョブが投入されると次のジョブの投入が予定され，投入されたジョブのスケジューリングが開始する．このスケジューリングでもし実行可能なノードが存在しない場合，ジョブの待ちキューに入り，他のジョブの終了を待つ．ジョブの終了時はジョブが使用していたノードのリソースを解放する．これにより待ちキューに入っていたジョブが実行可能になる場合はそのジョブを配置し，ジョブの終了イベントを予定する．これらの一連の流れにより繰り返しジョブが実行または待ちの状態をシミュレーションすることができる．(図 1)

6. 評価

本研究で提案するスケジューリングアルゴリズムの優位性を示すため，シミュレータ上でジョブの実行時間を計測し評価を行った．シミュレータは Go 言語により実装し，実行時間の計測のために，ベースラインアルゴリズムと提案するマルチクラスタスケジューリングアルゴズムの二つのスケジューリングポリシを用意した．またジョブやノードの種類を様々なパターンにわけ評価を行い，クラスタ環境やジョブの種類といった要素による実行時間に変化が生じるか評価した．

6.1 ジョブの実行時間の評価

本研究で提案したジョブスケジューリングアルゴリズムのジョブの実行時間を評価する．ジョブの実行時間はあらかじめ用意したジョブを投入し，すべてのジョブが実行され終了するまでの時間を計測する．この評価で使用するジョブは 2 種類あり，一つはごく軽量のジョブで，もう一方はそのジョブに比べ実行に時間のかかるジョブである．

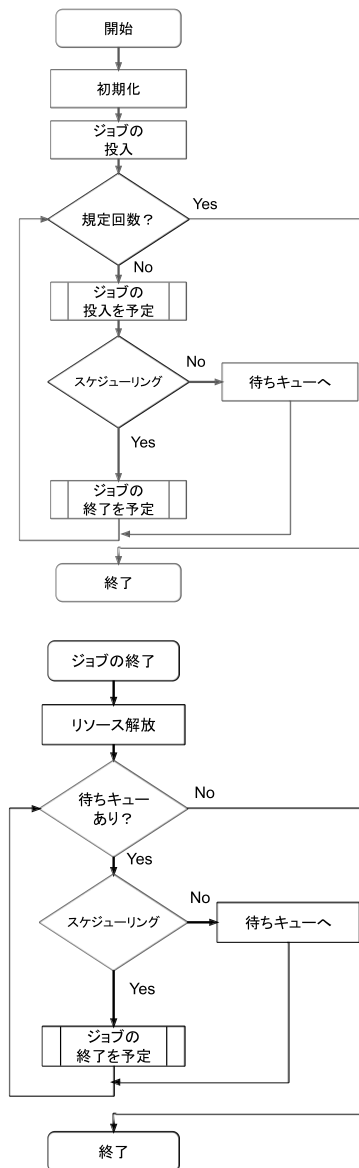

図 1: シミュレーションのフローチャート

それを示したのが表 3 である．これは表 2 での値をしている．軽量のジョブは実行にシミュレータ内の時間で 10 かかりもう一方のジョブは 100 かかる．この時全体のジョブ数を 100 から 10000 に変化させ，シミュレーションを行う．また投入するジョブは全体数の半数とし，ジョブを 100 個投入する場合，それぞれのジョブは 50 個ずつ投入される．さらにこの 100 個のジョブはランダムに投入され，シミュレーションごとに異なる．

図 2 は本研究で提案するアルゴリズムを使用した際に得た平均実行時間で，ジョブ数 100 個のときを 1 としたときの比において対数をとったものを表している．横軸はジョブ数の対数であり，縦軸はジョブの平均実行時間においてジョブが 100 個のときを 1 とした比の対数をとったもので

表 3: 実行時間評価で使用したジョブの定義

要素	ジョブ 1	ジョブ 2
RequiredCpu	100	1000(1core)
RequiredCpuSSE	Non	Non
RequiredGpu	1	1
RequiredGpuCapability	2.5	2.5
RequiredMem	0.5	2.0
RequiredSoftware	Non	Non
Runtime	10	100

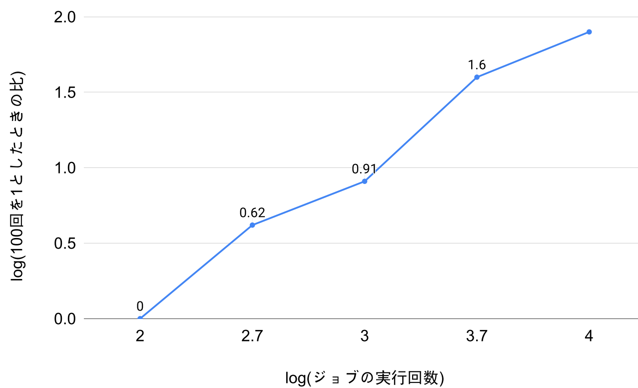

図 2: ジョブ数による実行時間の変化の比

ある．ジョブの数が 100 個のときを 1 とすると，500 個で 4.15，1000 個で 8.11，5000 個で 39.76，10000 個で 79.28 となり，それぞれ対数をとると図 2 になる．

6.2 ジョブの実行可能の評価

本研究で提案したアルゴリズムにおいて不均一性が考慮されることで実行できるジョブが増加することを評価する．ジョブの実行可能とはノードに投入されたジョブがそのノードで実行できるかを示したものである．この評価では 3 つのジョブを使用する．表 4 は使用したジョブの 3 種類を示している．必要なリソース量やジョブ実行に必要な条件がジョブ 5 の方が多い．実行時間もそれぞれ 10，100，300 とした．また実行するジョブ数は 3000 と，それぞれ 1000 個ずつのジョブをスケジューリングする．シミュレーションする環境として 8 ノード上で実行し．それぞれ CPU は 8 コアでメモリは 8GB，GPU は 2 枚搭載されている．この 8 ノードのうち 4 ノードは古い CPU・GPU を使用しており，ジョブ 4・5 を実行するのに必要なソフトウェア A・B がインストールされていない．残りの 4 ノードはソフトウェアに違いがあり，そのうちの 2 ノードはソフトウェア A のみがインストールされ，あとの 2 ノードはソフトウェア A・B の両方がインストールされている．シミュレーションはアルゴリズムごとに 10 回行った．表 5 はシミュレーション結果である．既存アルゴリズムは実行可能率が 58.3%となった．

表 4: 実行可能評価で使用したジョブの定義

要素	ジョブ 3	ジョブ 4	ジョブ 5
RequiredCpu	500	500	1000(1core)
RequiredCpuSSE	Non	Non	Non
RequiredGpu	1	1	1
RequiredGpuCapability	2.5	2.5	5.0
RequiredMem	0.5	1.0	2.0
RequiredSoftware	Non	A	A,B
Runtime	50	100	300

表 5: ジョブの実行可能率

アルゴリズム	平均実行ジョブ数 (個)	実行可能率 (%)
既存	1750	58.3
本研究	3000	100

6.3 マルチキューを使用した既存アルゴリズムと提案するアルゴリズムのジョブ実行時間

第 6.2 章で行ったジョブの実行可能率の評価では，この評価上の設定で行うシミュレーションでは既存アルゴリズムが 58.3%のジョブしか実行できないことが判明した．しかし既存のシステムではジョブの実行可能率を上げるため，マルチキューを採用している．そこでマルチキューを使用した既存アルゴリズムと提案するアルゴリズムでジョブの実行時間を評価した．

使用したジョブは表 4 に記載されているものを使用した．この評価ではジョブ 3 を 2500 個，ジョブ 4 を 300 個，ジョブ 5 を 200 個と合計 3000 個のジョブを実行するシミュレーションを行った．また実行されるノードは合計 6 台用意し，ジョブ 3 は全台で実行可能，ジョブ 4 は 4 台で実行可能，ジョブ 5 は 2 台で実行可能とした．このとき既存アルゴリズムでの評価はマルチキューを使用する．マルチキューを使用することでジョブを実行するノードを選択することができ，ジョブの実行可能率は提案するアルゴリズムと同様に 100%とすることができる．そのためジョブ 3，ジョブ 4，ジョブ 5 はそれぞれ各 2 台のノード上で実行される．提案するアルゴリズムではマルチキューを使用せず，ここまでの評価と同様の手法を用いた．これらのシミュレーションをそれぞれ 10 回実行し，ジョブの平均実行時間を評価する．

図 3 ではマルチキューを使用した既存アルゴリズムを基準として，それぞれのスケジューリングアルゴリズムにおけるジョブ実行時間の比を表している．ジョブの実行可能率はどちらも 100%となったが，平均実行時間はマルチキューを使用した既存アルゴリズムの平均実行時間を 1.00 とすると，提案するアルゴリズムでは 0.73 となった．

7. 考察

本研究ではジョブの実行時間と実行可能率を評価した．

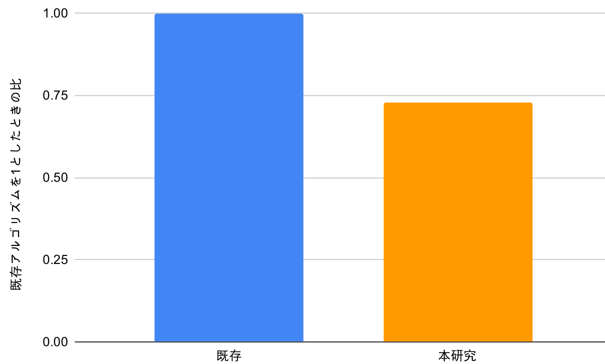

図 3: マルチキューを使用した既存アルゴリズムと提案するアルゴリズムのジョブ実行時間の比

ジョブの実行時間に関しては既存のアルゴリズムと提案したアルゴリズムにおいて差はほぼなかった．また図 2 からは，実行時間がジョブ数に比例して大きくなることがわかる．今回のシミュレーションでは，各ジョブの発行するイベント系列を固定しており，どの実験においてもイベントの差はないことから，スケジューリングによって大きなオーバヘッドは生じていないことを意味している．実行可能率に関しては表 5 から既存のアルゴリズムを使用すると 58%のジョブのみ実行されることがわかる．これは 42%のジョブは実行できないノードにジョブが投入され，実行できずに終了していることを示している．実行できるジョブが実行できないノードに投入されてしまう点に関しては提案したアルゴリズムがマルチクラスタにおいて優位性があることがわかる．またマルチキューを使用することでジョブの実行可能率を 100%にした既存アルゴリズムと提案するアルゴリズムの実行時間を比較した．図 3 からその実行時間に約 27%の差があった．これはマルチキューにおいて，片方のキューにジョブが集中すると一方のキューが空いているにも関わらず，ノードのリソースが空いていてもジョブが実行できないからである．しかし提案したアルゴリズムにおいてはすべてのノードからマルチクラスタの不均一性を考慮しスケジューリングするため，このような偏りがなく，既存アルゴリズムの実行時間との間に大きな差が出る．

TORQUE では第 4.2 章で挙げたベースラインのようなアルゴリズムが使用されるが，Kubernetes など近年利用需要が高まっているシステムで使用される場合は，その環境に合わせて設定されることが多い．Kubernetes ではノードごとにラベルとして属性を持たせることができ，特定のラベルが付与されたノード上で実行する・特定のラベルが付与されていないノードで実行するなど多様なスケジューリングが可能になっている．またラベルを付与する以外にも，カスタムコントローラと呼ばれるユーザーが記述し，動作を定義することができる仕組みが存在する．このカスタムコントローラはジョブスケジューリングにも用いることができ，どのジョブをどのノードで実行するか詳細に記述することができる．しかしこれらのラベル設定やカスタムコントローラの設定は管理者が行う必要があり，学習難易度も高い．さらにラベルの設定に関しては，ノードごとに行う必要があり，大規模なクラスタでは設定し管理することが難しい．そこで本研究で提案したアルゴリズムを実装する際は管理コストを減少させることが課題となる．

実装したシミュレータはジョブを繰り返し実行するのみであるため，異なる種類のノードを追加するイベントなどマルチクラスタ特有のイベントを定義し，それらを加味した評価を行う必要がある．これはマルチクラスタ環境でのイベントであり，既存のシングルクラスタ環境であれば同等のノードが追加されたと見なせば良いが，マルチクラスタ環境においてはそのノードがどのようなハードウェア・ソフトウェア環境があるか確認する必要がある．これはノードの追加や新しいソフトウェアがインストールされた場合などのイベントを実装し，シミュレーションする．また実装したシミュレーションにはマルチキューの概念がない．TORQUE などのジョブマネージャーではマルチキューを設定することができ，キューによって配置するノードを変えることができる．このマルチキューをシミュレーション上で実装することでさらに詳細な評価をすることができる．

実行環境によるジョブ動作の不均一性はジョブを実行することが可能であったとしても，動作するノードのパフォーマンスにより速度に違いがでる．これはハードウェアの差がある場合においてどちらのノードでも実行可能なとき，当然高速な CPU や GPU を搭載しているノードの方がジョブの実行が高速になり，実行時間が短縮されるということである．これはジョブのスケジューリング時に不均一性を考慮したアルゴリズムでジョブを配置可能と判断したノードにおいて CPU や GPU のスペックを考慮し判定し，シミュレーションに反映させる必要がある．

8. 結論

従来のクラスタ環境では同一のコンピュータを大規模に配置することが多いが，幅広い分野の計算を実行する必要があり，そのために拡張計算ノードを導入するなどが考えられる．そこで本研究ではそのような複数種類のコンピュータが混合したマルチクラスタ環境における不均一性を考慮したスケジューリングアルゴリズムを提案した．ハードウェアやソフトウェアの違いからジョブの実行動作に関して影響を与える要素を不均一性と呼び，アルゴリズムはこれらを考慮することで従来のハードウェアリソースのみを考慮したスケジューリングより，ジョブの実行時間に影響を生じるようなオーバヘッドなしに，ジョブの実行可能率を上げることができる．

ⓒ 2020 Information Processing Society of Japan

提案したアルゴリズムを用いたジョブスケジューリングのシミュレータを実装し，従来のスケジューリングアルゴリズムと比較し，ジョブの実行時間の評価を行った．実行時間の評価では既存アルゴリズムと速度の差がないことが示され，実行可能率の評価では従来のアルゴリズムよりもジョブの実行可能率が高いことが示された．これらの評価よりマルチクラスタ環境において本研究で提案したアルゴリズムが優位であると示された．

今後，今回提案したアルゴリズムを Kubernetesn のようなクラスタ管理基盤上のジョブスケジューリングアルゴリズムとして実現し，実践的な環境において評価を示していく予定である．

参考文献

[1] Adaptive Computing. Torque resource manager, 2020. https://www.adaptivecomputing.com/products/torque/.
[2] OpenStack Foundation. Openstack ussuri release delivers automation for intelligent open infrastructure, 2020. https://www.openstack.org/.
[3] CLOUD NATIVE COMPUTING FOUNDATION. Production-grade container orchestration, 2020. https://kubernetes.io/.
[4] Zhijun Wang, Huiyang Li, Zhongwei Li, Xiaocui Sun, Jia Rao, Hao Che, and Hong Jiang. Pigeon: An effective distributed, hierarchical datacenter job scheduler. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2019. Association for Computing Machinery.

ⓒ 2020 Information Processing Society of Japan