Vol.2021-OS-151 No.3
2021/3/1

不均質マルチクラスタのためのスマートスケジューラの設計と実装

齋藤 峻[1]　廣津 登志夫[2]

概要：大規模な HPC クラスタでは，同一構成のノードを複数台用意し、ジョブを実行することが多い．しかし，近年では使用するソフトウェアやハードウェアなど，ジョブに応じて要求するハードウェアやソフトウェアの構成が違い，多種多様なジョブが増加している．そのため，今後の HPC クラスタでは構成に差異があるノードが混在しているマルチクラスタ環境が増えると考えられる．しかし従来のジョブスケジューラではクラスタ内部のソフトウェアやハードウェアの差異を管理していないため，マルチクラスタ環境において適切なスケジューリングができない．そこで本研究ではマルチクラスタ環境における不均質性を考慮した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実際のマルチクラスタ環境で実装し，既存のジョブスケジューリングアルゴリズムとの比較をする．

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．これまではこのような HPC クラスタを構成するノードにおいて，特定の仕様のハードウェア上で，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た．しかし昨今，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，幅広い用途で使われている．そのため構成が同一のノードで構成されたクラスタでは対応できる計算に限界があり，近年ではその限界に対してどのように対応するかが課題となっている．これに対する一つの解決策は，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，プロセッサの単体の性能向上が難しくなっていくことを考えると，この機能特化したコンピュータが複合し，様々なジョブに対応するような複合型クラスタ環境が一般的になる．

多様なジョブに対応するために，従来の計算処理を行う基本計算ノードの一部に例として 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するといった構成法が考えられる．このような環境をここではマルチクラスタと呼ぶ．この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が，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タ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差が生じるものが考えられる．さらに複数ノードで並列実行する処理を行うジョブもあり，並列ジョブにおけるノードの選択やジョブの配置するタイミングなどにおいてもジョブの実行性能差が発生すると考えられる．

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質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．

[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

本研究ではこれを実現するアルゴリズム提案し，実環境に実装することにより，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考慮して制御することで，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し，複数ノードで処理を行うジョブに関してもこれらの条件を考慮してスケジューリングすることができるようになる．

2. 関連研究

ジョブマネージャーの Slurm(Simple Linux Utility for Resource Management)[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質性に対するスケジューリングをすることができない．

Kubernetes はコンテナ型仮想化基盤であり，コンテナオーケストレーションツールの一つである．Kubernetes がコンテナを配下のノードにスケジューリングする際，配置するコンテナが必要とするリソース量などを確認した後，コンテナをどのノードに配置するかは既存のスケジューリング手法と同様に各ノードの残リソースを鑑みて行っている．コンテナ型仮想化基盤における特性上，ユーザーがジョブにおいてソフトウェアやライブラリを自在に使用することができる．しかし，ジョブの中にはソフトウェアを実行したりコンパイルするのに CPU の拡張命令セットが必要である場合や GPU を使用する際に Capalibity の高い GPU が必要になる場合がある。例えば TensorFlow がそのソフトウェアに該当する．TensorFlow は 1.5 以降のバージョンに関しては GPU の Capability が 3.5 以降でないと動作しない．また TensorFlow を動作させるには CPU の拡張命令セットである AVX を使用するため，AVX 拡張命令セットを持たない CPU は実行することができない．そのための解決手法として Kubernetes ではクラスタ内のノードに対してクラスタ管理者がタグを付与することが挙げられる．このタグ付けは管理者の自由に付与することが可能なため，タグの名前を自由に決定し，そのノードにあったタグを付けることができる．例えば HDD や SSD などのタグを使用することで，ディスクの書き込みが多く発生するコンテナは SSD のノードに配置したり，Capability のタグを作成することで，特定の Capability 以上の GPU を搭載しているノード上にコンテナを配置することができる．スケジューリングの際にタグを使用する場合は，コンテナ情報を記述した YAML ファイルにタグ名をのせることで，スケジューリングされるときにそのタグ情報も考慮してノードを選択することができる．しかしながらこのタグ付けはクラスタの管理者が行う必要があり，タグがない場合は使用できない．また，このタグは手動で付与・削除を行うため，ノードに新しいハードウェアを換装した場合や，ノードを増設した場合は再度タグをノードに対して付与しなければならない．このようなタグの管理の手間はクラスタの管理者にとって煩雑である．

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質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．

本研究では既存のスケジューリングアルゴリズムではアルゴリズムの対象とされていないマルチクラスタ環境におけるハードウェアやソフトウェア・ライブラリなどの差異を考慮したスケジューリングアルゴリズムを提案する．これによって既存のアルゴリズムではマルチキューの設定をせずに適切なジョブ配置を可能にし，キューごとに発生する混雑具合の差が発生しないスケジューリングを実現する．

3. マルチクラスタ環境における不均質性

ここ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質性について述べる．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質性が何であるか，また不均質性における問題点を明確にする．

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．

従来の HPC クラスタでは，同一構成のコンピュータで大規模なクラスタリングを組むことが多い．このようなクラスタをシングルクラスタと呼ぶ．シングルクラスタでは構成する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・ライブラリのバージョンも統一される．そのため，どのコンピュータにおいても計算できる処理において差がない．計算の実行を割り当てられたコンピュータが実行できない場合は，他の全コンピュータにおいてその計算は実行不可能である可能性が高い．

この問題を解決するために，多種類のコンピュータを用意しインストールされているソフトウェア・ライブラリの種類を増やすことがある．これはハードウェアが異なるコンピュータを同一クラスタ内に搭載することで，単一種類のコンピュータでクラスタリングした時より，多くの種類の計算処理を実行することができる．またハードウェア以外にも，ソフトウェアやライブラリも同様に，異なる種類・バージョンのものをインストールすることで，多種類の計算を実行できる．その他にも既存のクラスタ環境において GPU などのハードウェア的機能の追加や特定のコンピュータにソフトウェアをインストールするなど，既存の構成を保ちつつ，内部のリソースを変更することで，構成の異なるコンピュータが生じる．このようにクラスタ内部において，構成の異なるコンピュータでクラスタリングを形成しているシステムをマルチクラスタと定義する．

マルチクラスタはクラスタに構成の異なるコンピュータが混在している環境のことを指し，ヘテロクラスタのような元々異なる計算処理に応じて計算するコンピュータを変更するクラスタも包含している．またマルチクラスタは 2 つ以上のクラスタが混在していることで発生する環境であり，最低でも 2 種類以上の構成が混在している環境である．マルチクラスタにおける構成の差異とは 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やネットワークなどがある．

図 1 はマルチクラスタの概念を表したもので，発生する要因と流れを示している．マルチクラスタの概念は前段で述べたように，構成の異なるコンピュータ群が複数合わさることで形成される．複数のクラスタが混合する環境になる要因として，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がある．リプレースによるマルチクラスタ発生の要因は既存のクラスタにおいて経年劣化における交換や正常に動作しないコンピュータを交換するような場合が考えられる．経年劣化ではクラスタ内部のコンピュータを全て入れ替えることが多いが，既存のクラスタリングされたコンピュータを維持しつつ，新しいコンピュータを導入しクラスタを拡大しつつ，既存のコンピュータと併用して使用することもある．このような場合，元々構成されていたコンピュータ群に対し，新規でコンピュータを追加したことで，ハードウェア的構成の違うコンピュータが混合するマルチクラスタ環境が形成される．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．クラスタの投入では既存のクラスタ環境に対し，新しいクラスタを増設することで発生する．これは利用しているユーザーの要望や新規のソフトウェアを実行可能にするために，すでにクラスタリングされているコンピュータではなく，新規のコンピュータを導入することでマルチクラスタの環境にすることがある．これによって既存の環境と新規の環境で複数の環境が混在するマルチクラスタになる．

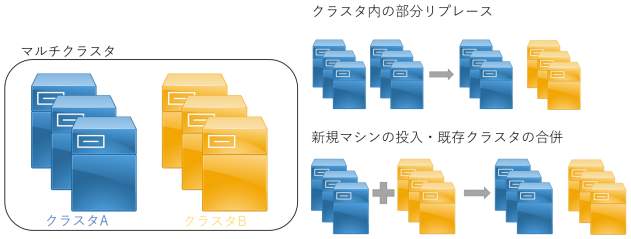

図 1: マルチクラスタの概念と発生要因

3.2 マルチクラスタ環境における不均質性

第 3.1 節で定義したマルチクラスタ環境における差異を不均質性と呼ぶ．不均質性は様々な要素があり，これには CPU・GPU などのハードウェアやライブラリ・ソフトウェア，ネットワークといったものが考えられる．

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．

ライブラリ・ソフトウェアにおける不均質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特定のソフトウェアを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生じる．

このようなマルチクラスタ環境において不均質性がもた

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質性を考慮し，計算処理を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. スケジューリングアルゴリズム

この節では第 3 節で記述したマルチクラスタ環境における不均質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．不均質性の要素を定義し，既存のアルゴリズムと提案する不均質性を考慮したアルゴリズムを比較する．

4.1 不均質性をもたらす要因の表現

第 3.2 節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質性が発生する．この不均質性をアルゴリズムにするため，各不均質性の定義をする．ハードウェアの不均質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 Capability 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．

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

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

IsVersionIsSSE，*IsCapability* 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質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

本研究では Slurm のスケジューリングアルゴリズムを元に、不均一性への対処の機能を導入することで改良をおこなす。Slurm のアルゴリズムをアルゴリズム 1 に示す．このアルゴリズム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質性を考慮してスケジューリングする方法である．その不均質性については第 3.2 節で述べ，アルゴリズムの側面から見た不均質性の要素定義については第 4.1 節にて述べた．ここでは不均質性を考慮したアルゴリズムについて述べる．

本研究で提案するアルゴリズムをアルゴリズム 2 に示す．これは Slurm のアルゴリズムと同様にハードウェアリソースの確認を行った上で，*CPUGPULibrariesCheck(J)* でハードウェア・ソフトウェアの環境依存性の判定を行っている．この *CPUGPULibrariesCheck(J)* はその内部で第 4.1 節で述べた式 1，式 2，式 3 を呼ぶことでマルチクラスタの不均質性を考えたスケジューリングを行うことができる．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を満たしているか判断する．

5. 実装

この節は第 4 節で述べたマルチクラスタ環境における提案するスケジューリングアルゴリズムの既存のジョブスケジューラへの実装について述べる．実装元のジョブマ

アルゴリズム 2 マルチクラスタの不均質性に対応した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
```

ネージャとして Slurm を使用する．Slurm のジョブスケジューラに対して提案するアルゴリズムを実装する．実装元とするアルゴリズムは cons_tres と呼ばれる Slurm の Consumable Resource Allocation Plugin を使用し，マルチクラウド環境における不均質性を考慮するスケジューリング要素を追加実装する．

5.1 Slurm のジョブスケジューラの仕組み

Slurm のジョブスケジューラはジョブが投入された後，そのジョブが実行可能か判断しジョブをスケジューリングする．ジョブが投入されるとマスターノード内でジョブオブジェクトが作成され，内部でジョブを発行する．発行されたジョブはスケジューラによってどのノードに配置されるか計算される．実行できるノードの確認においてジョブが実行できるかどうかの確認をする．その後ノードの選択において，上記の確認を通過したノード群の中からジョブが実際に実行されるノードが選択され，ジョブが配置・実行される．

Slurm にはジョブスケジューリングのアルゴリズムが複数用意されている．アルゴリズムはプラグインとして扱われており，Slurm の設定ファイルである slurm.conf の SelectType パラメータにアルゴリズム名を記述することで実行時に読み込み，設定することができる．アルゴリズムは HPC 環境では Consumable Resource Allocation Plugin が使用される．このアルゴリズムはジョブごとに割り振られたリソースを各ノードの残リソース量と照らし合わせ，最適なノードにジョブを配置するアルゴリズムである．アルゴリズムについては linear，cons_res，cons_tres などがあり，linear を基準として cons_res や cons_tres がある．cons_res と cons_tres の違いは GPU の管理にあり，ジョブスケジューリングに対して GPU 数といったパラメータを使用することができる．

ジョブスケジューリングに関しては slurmctld が管理している．slurmctld ではジョブが投入された後，ジョブが内部で発行され，スケジューリングが開始する．スケジューリングでは choose_nodes 関数で受け取った情報からノードを選択する．choose_nodes 関数内ではクラスタ内のノードから残リソース量を見てスケジューリングする．_can_job_run_on_node 関数はジョブとノードの組み合わせが入力として与えられる．ノード情報として，キューイングされたキューの対象となっているノードのリストがあり，リソース情報がまとめられている．このノードのリストとジョブを照らし合わせ，ジョブが実行できるかどうかをリソースによって判断している．

本研究ではこの_can_job_run_on_node 関数においてマルチクラスタ環境における不均質性を考慮する仕組みを実装する．その仕組みにおいてジョブを実行できないノードだと判断した場合はジョブを実行する対象ノードの一覧から排除され，ジョブが実行できるノードのみ一覧に残る．

5.2 マルチクラスタ環境の不均質性を考慮するスケジューリング機能の実装

提案するアルゴリズムの実装として cons_mytres としてアルゴリズムの実装を行った．ここでは cons_tres を拡張した cons_mytres を新たに用意した．そして新しく不均質性を考慮する実装のコードとして mytres_multi_cluster.h と mytres_multi_cluster.c を実装した．

前節で記述した_can_job_run_on_node 関数においてマルチクラウド環境の不均質性を考慮する条件を付ける関数を実装する．_can_job_run_on_node 関数にて，マルチクラウド環境の不均質性から見て，そのジョブが対象ノードで実行することができるかどうかを判断できるようにする．

まず不均質性を判断するために，各ノードの不均質性を保持する構造体を作成した．構造体には CPU・GPU・ライブラリの情報が格納されており，各ノードの CPU におけるバージョンや拡張命令セットの有無，GPU の Capabilirty，インストールされているライブラリ情報やそのバージョンがある．リスト 1 ではその構造体を広げたものを表現している．実際には SSE やライブラリの情報はポインタで格納されており，ポインタ先には各要素の配列が存在する．配列には搭載されている拡張命令セットやインストールされているライブラリの情報が要素として格納されている．またジョブに関して追加の情報を保持するために構造体を用意した．この構造体はノードの不均質性を保持している構造体と同様に，ジョブの実行における情報を格納した．これはジョブを実行するのに必要な CPU のバージョンや GPU の Capability，ライブラリの情報など，ノードで定義した構造体と同じような情報を有している．これらの構造体から各ジョブがどのような要素を持っているか確認し，そのノードの要素と比較することでジョブが実行可能か判断する．

_can_job_run_on_node 関数における評価はリソース情報によるが，その評価の最後に不均質性の評価をする関数

choose_nodes_multi_cluster 関数を実行することで，マルチクラスタ環境の不均質性を考慮した評価をするように実装した．これにより_can_job_run_on_node 関数内でそのジョブがそのノードで実行できるかを判断することができる．

実装した choose_nodes_multi_cluster 関数の定義はリスト 2 の通りである．構成はジョブ情報の構造体である job_record_t のポインタとノード情報の構造体である node_record_t のポインタを引数に取り，それらの情報からリスト 1 の構造体情報と比較することで 0 か 1 を戻り値とする関数である．引数にとった構造体に不均質性を考慮するのに必要なジョブの情報とノードの情報があり，それに基づいて不均質性を考慮する構造体 node_additional_information_t を使用し比較する．比較方法としては各ジョブの不均質性として与えられた情報と引数に取ったノードにおける不均質性の構造体の情報を 1 項目ずつ確認し，全て該当する場合はそのノードで実行可能であると判断し，1 要素でも欠けている場合は実行不可能であると判断する．

リスト 1: 不均質性を保持する構造体

```
typedef struct
  node_additional_information{
    // basic info
    char *node_name;
    // CPU info
    char *cpu_name;
    int cpu_version;
    sse *cpu_sse {
        {
          char *sse_name;
          int can_run_sse;
        }, ...
    };
    // GPU info
    char *gpu_name;
    float capability;
    uint32_t mem;
    // Library info
    library_t *libraries {
      {
        char *library_name;
        int version;
      }, ...
    }
}node_additional_information_t;
```

リスト 2: 不均質性を比較する関数

```
extern int choose_nodes_multi_cluster(
        node_record_t *node_ptr,
        job_record_t *job_ptr
        );
```

6. 評価

本研究で提案するマルチクラスタ環境のためのスケジューリングアルゴリズムを，実機環境で既存のアルゴリズムと比較する．1 つ目の評価として，このアルゴリズムの処理が全体のジョブスケジューリング機構においてオーバーヘッドになっていないかジョブの実行時間を測定する．2 つ目として，アルゴリズムにおけるジョブの実行可能率を測定する．最後に 3 つ目として，マルチキューを用いた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スケジューリングアルゴリズムを用いたジョブの実行時間を比較評価する．

6.1 ジョブの実行時間の評価

ジョブの実行時間の評価では実装したスケジューリングアルゴリズムの処理において，ジョブスケジューリング全体の処理のオーバーヘッドとなっていないか確認する．確認する方法として，既存のアルゴリズムを使用した場合と提案するアルゴリズムを使用した場合の 2 パターンで，それぞれジョブの実行時間の計測を行う．ここで用いるジョブは sleep を 1 秒するジョブとなっており，実行には 1 秒程度かかるジョブとなっている．また計測はジョブを 1 つずつキューに入れ，ジョブがキューイングされてからジョブが完了するまでの時間を計測する．ジョブを sbatch コマンドを用いてキューイングした時刻を開始時刻とし，squeue コマンドを用いて取得できるジョブの状態が complete になった時刻を終了時刻とする．そしてこの開始時刻と終了時刻の差をジョブスケジューリングとジョブ実行の実行時間とし，各 100 回実行した平均を評価した．

表 1 はジョブの平均実行時間を示している．既存のアルゴリズムでは 2.95 秒であり，実装した提案するアルゴリズムでは 2.94 秒である．それぞれのアルゴリズムにおいて大きな差はなく，不均質性を考慮する処理におけるオーバーヘッドは小さい．

6.2 ジョブの実行可能率の評価

ジョブの実行可能率の評価では既存のアルゴリズムと提案するアルゴリズムにおいて，ジョブを実行することのできる割合を測定する．ジョブを複数用意し，複数ノードがクラスタリングされている環境でジョブを実行し，各ジョブが適切なノードに配置され実行できるかを評価する．まずノードとジョブはそれぞれ 3 種類用意した．ノードは 1 台ずつ存在し，3 種類のジョブは動作できる環境に差がある．表 2 は使用した 3 種類のジョブを示している．それぞ

表 1: ジョブの平均実行時間

アルゴリズム	平均実行時間 (秒)
既存	2.95
本研究	2.94

表 2: 実行可能評価で使用したジョブの定義

ノード	ジョブ 1	ジョブ 2	ジョブ 3
ノード 1	○		
ノード 2	○	○	
ノード 3	○	○	○

表 3: アルゴリズムによる実行可能率

アルゴリズム	実行可能率 (%)
既存	60.4
本研究	100

表 4: 比較実験における環境のパターン

	ノード 1(台)	ノード 2(台)	ノード 3(台)
パターン 1	2	1	1
パターン 2	2	2	1
パターン 3	3	2	1

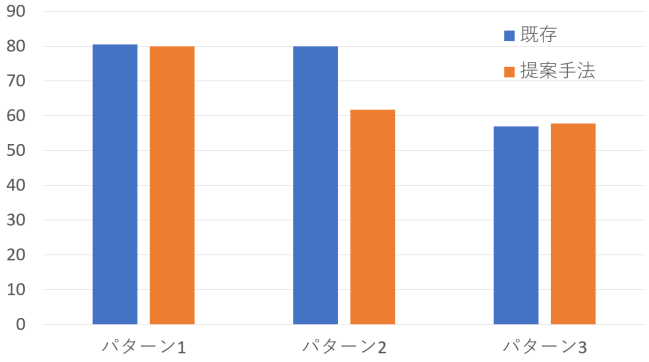

図 2: マルチキューを使用した既存アルゴリズムと提案するアルゴリズムのジョブ実行時間の比較

れのジョブが実行できないノードがある．ジョブ 2 はノード 2・3 で実行可能であり，ジョブ 3 はノード 3 のみで実行可能である．ジョブをそれぞれ 100 個ずつキューイングさせ，ジョブが実行できるか確認する．

表 3 はアルゴリズムによる実行可能率を示している．既存のアルゴリズムでは 60.4%であるのに対し，提案するアルゴリズムでは 100%であった．

6.3 マルチキューによる制御との比較

第 6.2 節の実行可能率の評価は比較対象としてシングルキューの既存アルゴリズムを用いていた．そこで，対象をマルチキューを用いた従来手法に変えて比較を行う．実際クラスタ環境において既存の手法が使用される場合，ノードの違いによってキューを分別するマルチキュー方式を採用する．マルチキューにすることでジョブを実行可能なノードにスケジューリングしている．そこでこの評価ではマルチキューを用いた既存のアルゴリズム環境と提案する手法で複数のジョブを実行したときの実行時間を比較し評価する．

評価に使用したジョブは表 3 の 3 種類である．それぞれのジョブにおいて sleep の時間を変更することで，各ジョブの実行時間に差異を表現した．ジョブ 1 は 3 秒，ジョブ 2 は 5 秒，ジョブ 3 は 10 秒とし，ジョブの実行可能ノードは表にある通りである．この評価ではジョブ 1 を 400 回，ジョブ 2 を 100 回，ジョブ 3 を 20 回とし，評価の 3 パターンに応じてノードの数を変更した．ノードの数は表 4 に示した通りである．ノードによる違いは実行可能なジョブに違いがあり，搭載している GPU やインストールされているソフトウェアが異なることを想定している．そのため，ジョブに応じて実行できるノードに差が生じる．それぞれのジョブを規定回数キューイングし，最初のジョブをキューイングした時刻を開始時刻とし，全てのジョブを実行し終えた時刻を終了時刻とした．既存手法においてはそのジョブ専用のキューを用意し，各ジョブはそのキューにのみキューイングする．提案手法はキューを単一にしすべてのジョブを 1 つのキューに投入する形でそれぞれジョブを実行する．

図 2 では各パターンのジョブ実行時間の比較となっている．それぞれパターン 1 では既存手法が 80.4 秒，提案手法が 80.0 秒となり，パターン 2 ではそれぞれ 79.9 秒，61.8 秒，パターン 3 では 57.0 秒，57.8 秒となった．

7. 考察

本研究ではマルチクラスタ環境における不均質性を考慮したアルゴリズムが有効であることを示すために 3 つの評価を行った．まず 1 つ目の評価としてジョブの実行時間の評価である．この評価において既存のジョブスケジューリングと実装したスケジューリングにおいてどちらも実行時間が 3 秒程度であった．これは実装した処理において重い処理をしておらず，ジョブとノードの対応を新たに実装した構造体を元に判断しているからである．今回の実装ではノードの情報やジョブの情報を取得しているという状況を念頭にしているため，今後ジョブがどのような状態を持っているのか，ノードのインストールされたソフトウェアはどのようなものがあるのかという情報をリアルタイムで取得した場合，処理に時間がかかる可能性がある．これはノードの総数に応じて変わり，大規模なクラスタではノード情報の更新が長くなる．これに対してはインターバルを設定し，決められた時間ごとに取得する等の処理をすることで対処できる．

2 つ目の評価としてジョブの実行可能率の評価を行った．この評価ではジョブの不均質性を考慮したジョブスケ

ジューリングができるかを確認している．既存の手法ではリソースなどの情報を元にスケジューリングしている．しかし既存手法はジョブがどのようなハードウェアを必要としているのかやどのバージョンのソフトウェアを必要としているのかという情報を抜きにスケジューリングするため，ジョブをソフトウェア的な懸念で実行できないノードに配置される可能性があった．評価結果によると，6 割のジョブが実行可能であることが判明した．今回評価に用いたワークロードでは，不均一性を考慮しないと確率的には 3 割程度のジョブが実行できないモノであったため，今回の評価における 3 割程度のジョブが実行できないという結果は合致している．しかし本来既存手法ではマルチキューを使用することで，ジョブを実行可能なノードがあるキューに投入するため，既存手法においてもマルチキューを使用すれば実行可能率は 100%になる．

3 つ目の評価として既存手法と提案手法の実行時間の比較を行った．環境としてパターンを用意し，それぞれのパターンにおける実行時間の差を評価した．パターン 1 はどのキューにおいても待ちが発生する場合，パターン 2 はノードを増設したことで待ちが少し解消された場合，パターン 3 はそれぞれのジョブにおける実行時間に差がない場合を想定した．パターン 1 のようにどのキューにおいてもリソースが足りていない場合は，既存手法と提案手法に限らずどちらも実行時間がかかってしまい差はあまり無かった．しかしパターン 2 のようにノードを追加するとリソースに余剰が見られ，空いているノードにてジョブを実行できるようになるため，提案手法では実行時間が短くなる．またパターン 3 のようにどちらもリソースが一定以上ある状態になると，どちらの手法においても実行時間に差がない状態になる．今回実施した評価では上記で述べたようなパターンであったが，提案手法の方が実行時間が遅くなるパターンもある．それは重いジョブが大量に投入されたときである．このときはジョブの実行時間が重いジョブに引きずられ，他のジョブの配置が重いジョブの実行完了を待つ必要がある．これに対しては軽量なジョブを優先する制御を入れることで十分に対処できると考えられる．

8. 結論

本研究ではマルチクラスタ環境における不均質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．提案した手法ではハードウェアやソフトウェアなどノード間の差を観測し，それらの不均質性を考慮した上でスケジューリングを行う．提案アルゴリズムと既存アルゴリズムを比較したところ，ジョブの実行時間や実行可能性と言った面で性能の向上が見られた．

参考文献

[1] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[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.