Vol.2019-OS-146 No.7
2019/5/31

バッジジョブのためのコンテナ環境と連携するジョブスケジューラの設計と実装

齋藤 峻[1]　廣津 登志夫[2]

概要：近年 HPC クラスタ環境におけるコンピューティングタスクが自然言語処理や人工知能など多様になってきたことから，それらを実行するために必要となるライブラリが多様になっている．この多様な実行環境を競合なく提供するためには，ジョブタスクをコンテナ化環境で実行することが考えられる．この場合，クラスタ上でのジョブ管理の観点では，従来のバッチジョブの実行管理を行うジョブマネージャとコンテナを管理するコンテナオーケストレータとの連携が重要である．本研究では, 従来のバッチジョブ型のタスクとコンテナ化されたタスクの双方を一元的に管理するジョブスケジューリングの仕組みを提供する.

1. 序論

HPC(High Performance Computing) クラスタでは共有リソースを元にしたクラスタにジョブと呼ばれるスクリプトを投入し，クラスタ上でそのジョブを実行する．以前は，数値計算など多量の計算を要する比較的限定的な問題が対象となってきたが，近年そのジョブは多様化の一途を辿っており，数値計算に留まらず機械学習や自然言語処理などの分野でも使用され，その利用需要は更に増している．

従来の数値計算などの用途においては，それを目的としたアプリケーションやライブラリを用いて計算が行われることが多かった．そのため，クラスタシステムとして単一のアプリケーション実行環境を用意して，利用者はその環境の範囲内で計算を処理するプログラムを作成・実行していた．このような環境でそれらのアプリケーションやライブラリを複数バージョン共存させる場合には，インストールパスをバージョン毎に分離することで競合が起きない環境が提供されていた．一方で，昨今，クラスタ計算の主要なアプリケーションとなりつつある人工知能の学習などにおいては，次々と新しいライブラリやアプリケーションが開発され，その更新頻度も高い．また，言語処理系，AI など目的に応じたライブラリ，それらのライブラリが使う汎用や並列化などのシステムライブラリと，相互のバージョンの組み合わせが多数あり，従来のような競合を解消させつつ共存させる環境を提供することが難しくなっている．

[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

そこでコンテナ仮想化の技術を用いると，特定のバージョン毎の独立した環境を提供し，バージョン依存性を気にせず自由に計算処理を実行することができる．その結果，ライブラリやソフトウェアがクラスタ環境に強く依存しているという課題を解決することができる．そこで従来の数値計算処理と昨今の機械学習や自然言語処理を共にコンテナ化することで，計算環境を併存させつつ，クラスタ全体で安全かつ安定的に実行できるコンテナ化ジョブとして一元管理する必要がある．

本研究では既存のジョブスケジューラ環境下で稼働していたネイティブなタスクとコンテナ化されたタスクの両方を一元的に扱う手法を提案する．具体的にはジョブマネージャーによりジョブを各クラスタノードに割り振り，その際にインストールされていないライブラリやソフトウェアを使用しているジョブの場合はクラスタ上に専用コンテナイメージを作成し，そのコンテナ環境でジョブを実行する．ジョブマネージャーとして TORQUE(Tera-scale Open-source Resource and QUEue manager)[1] を使用する．TORQUE のジョブはファイル内に必要な情報をメタ文字`#PBS` を使用し記述する．そのメタ文字に続く引数を読み取ることでリソースや実行ノード数などの設定を行う．これらの情報を有しているジョブスクリプトからコンテナ環境を作成する．またこの手法を実装する上でコンテナ環境を管理するプラットフォームが必要である．そこで Kubernetes[2] を使用し，コンテナの制御を行う．

本研究で提供するジョブスケジューラは，ジョブが投入された時点でそのジョブを解析し，必要なライブラリやソフトウェアがインストールされたコンテナ環境を作成，その環境でジョブを実行する．具体的には実行するバッチ

ジョブバイナリの共有ライブラリ情報から，そのコンテナ環境を作成し，その上でバイナリを実行する．これにより，ネイティブ環境にインストールされていないライブラリやソフトウェアでもジョブを実行することが可能となる．TORQUE と Kubernetes を連携させることにより，キューイングされたジョブに対応したコンテナを配置しジョブを実行することが可能になる．

2. 関連研究

HPC クラスタでのコンテナ基盤として Singurarity[3] がある．Singularity はコンテナ仮想化技術のひとつで主に HPC 環境で用いられる．Singularity の利点として，デフォルトで GPU を使用できること，コンテナの実行に root ユーザーである必要がないこと，MPI へ対応してることが挙げられる．これにより，Docker と比較し，可動性のある HPC 用コンテナ技術として提案されている．Singurarity では，TORQUE などのジョブスケジューラにおいて，タスク実行の際に singurarity_exec 等のコマンドを用いて実行することで，コンテナ化されたタスクを TORQUE の管理下で実行することが可能である．

Orchestrating Docker Containers in the HPC Environment[4] では HPC 環境において Docker コンテナを使用した際のパフォーマンス評価を行っている．その評価においてコンテナ環境において，ネイティブ環境と同等の演算処理を実行できることを示している．

本研究では，コンテナと TORQUE の一元的管理の下でクラスタ上のタスクを実行制御することを考えて，従来のネイティブ環境で稼働するジョブやクラスタ環境で用意しているコンテナではライブラリ等の実行資源が不足するようなジョブについても，コンテナ管理の下で安全に実行し TORQUE 等のジョブスケジューラで実行管理できるようにすることを目指す．

3. Kubernetes

コンテナのオーケストレーションなどの機能を提供しているプラットフォームが Kubernetes である．Kubernetes はコンテナの自動配置，スケジューリングなど管理に必要な機能を提供している．これらの機能を使用し，コンテナ単体では制御できないことを実現している．

Kubernetes では，コンテナは Pod という単位で制御される．この Pod に対する設定をマニフェストファイルに Yaml 形式で記述することによってコンテナ環境を作成し，その上でジョブを実行することができる．マニフェストファイルにはコンテナ名，コンテナイメージ，CPU やメモリといったコンテナに使用するリソース量，マウントするディレクトリ，開放するポート，生成時実行されるコマンドなどが情報として記述されている．1 つのマニフェストファイルによって同一のコンテナが作成される，そのため，一度作成するだけで同じ環境を何度も作成することができる．このマニフェストファイルをそのジョブごとに作成することでそのジョブ専用のコンテナ仮想化環境を構築することが可能である．

4. コンテナ化クラスタ

この章ではジョブマネージャーが導入された HPC 環境における問題点を挙げる．HPC 環境では 1 つの計算や学習などの処理をジョブという単位で扱う．このジョブを共有 HPC 環境クラスタ上で実行することで大規模な処理を可能にしている．このような HPC 環境において，ジョブを複数実行した際にリソース不足になる問題, クラスタのノードを管理する問題などがある．ジョブを実行する際, ジョブの上限などの制限がない場合, ジョブがクラスタのリソースの上限まで使い, クラスタのマシンが停止することやジョブの一部が動作不良に陥る．そのため, 適切にジョブをノードに分配する必要があり, 使用可能な資源を一元管理し, どのノードでどの程度のリソースが使用されているか監視し, 一定以上のリソースを使用しないようにする必要がある．

これらの問題を解決するべく，HPC 環境ではジョブを管理するジョブマネージャーというプラットフォームを使用する．ジョブマネージャーを導入することによって，大量のジョブの実行要望が存在しても, すべて同時にクラスタ上で実行されることがなく, リソースに応じてジョブが実行される．ジョブスケジューラは処理されているタスクやこれから処理するタスクを統括し, 各クラスタノードのリソース使用量を取得することで，次に割り当てるジョブを実行するノードを決定する．これらジョブマネージャーの機能を用いて，クラスタ環境全体のリソースやタスク処理を最適化する．

このようなジョブマネージャーが導入された HPC 環境における問題は, ジョブを実行するためのライブラリやソフトウェアがクラスタ依存であるということである．HPC 環境ではクラスタ内にインストールされているライブラリやソフトウェアがクラスタ管理者によって管理されている．そのため, ジョブを実行する場合, 実行環境のクラスタにインストールされているライブラリやソフトウェアを使用しなければならなず，バージョンも固定化されてしまう．そのため機械学習や自然言語処理などで使用するライブラリや GPU などのハードウェアライブラリ，並列化などのシステムライブラリといった数多くのライブラリでの相互のバージョン依存が激しく，それらを自由に利用者が使用することができない．その結果，利用者はクラスタが提供するライブラリの組み合わせで開発を行う必要があるため，利用者にとって不自由に感じる部分が多い．また既存開発環境のバージョンでジョブを実行することができず，再現環境でジョブを実行することができない．このようにライ

ブラリやソフトウェア間のバージョン依存性は様々な問題を生じる．またこれらのライブラリは更新頻度も高く，新たなライブラリが次々に誕生し，利用されている．その膨大なライブラリをすべてクラスタに導入し，バージョンによる依存関係を解決したクラスタを提供し続けることは困難である．

これらの問題を解決するべく，HPC クラスタ環境ではコンテナ仮想化技術が用いられている．これにより，クラスタ内で特定のバージョンを用いた独立の環境を作成することで，その環境でジョブの実行を可能にしている．しかし既存のジョブマネージャーにおいてコンテナ化ジョブの支援は乏しい．そのため，従来の数値計算処理と昨今の機械学習や自然言語処理を共にコンテナ化し，計算環境を併存させつつ，クラスタ全体でジョブとして一元管理し，コンテナ化ジョブに対応する必要がある．そこで本研究では既存のジョブスケジューラ環境下で稼働していたネイティブなタスクとコンテナ化されたタスクの両方を一元的に扱うことでクラスタ上でコンテナ化ジョブを実行する仕組みを設計する．

5. 設計

本研究では既存のジョブスケジューラ環境下で稼働していたネイティブなタスクとコンテナ化されたタスクの両方を一元的に扱う手法を提案する．本研究は，ジョブマネージャーとコンテナオーケストレータの連携を行うことでコンテナ仮想化環境でジョブを実行できるようにする．この機能拡張により，ネイティブ環境で動作しないジョブをコンテナ仮想化環境で動作させることが可能となる．またクラスタを使用する利用さ h 側からは既存のジョブスケジューラにジョブを挿入すると同様の手順でジョブをコンテナ環境で実行することができる．

5.1 ジョブマネージャーとコンテナオーケストレータの連携

ジョブマネージャはジョブが投入された際，ジョブスクリプトを解析し，そのジョブにおけるリソース使用量やクラスタノードの割り当てなどを行う．ジョブから必要なライブラリやソフトウェアの要件を解析し，その環境をコンテナイメージとして作成することで，実行するための環境をコンテナとして用意することが可能になる．それにより，ネイティブ環境では動作しないようなジョブが投入されたとしても，そのジョブをコンテナ環境で実行することが可能である．

ジョブキューにジョブが投入された際，そのジョブをどのノードで実行するかどうかを判断し，そのノード上にコンテナ環境を作成，ジョブを実行する必要がある．そのため，ジョブスクリプトで指定されたジョブへの設定やクラスタのリソース情報を保持しているジョブマネージャーがどのノードで実行するかの判断を行う．その後，指定された条件でコンテナオーケストレータはジョブ専用コンテナを作成する．そのコンテナ上でジョブを実行する．ジョブの実行が終了した際，ジョブマネージャーに終了の通知を行い，コンテナオーケストレータが専用コンテナを削除し，リソースの解放を行う．

5.2 コンテナイメージによる HPC クラスタ管理

実行するジョブスクリプトからコンテナイメージを作成することで，そのバッジジョブを実行可能なコンテナ環境を作成することができる．そのコンテナイメージを共有することによって，全 HPC クラスタノードでそのバッジジョブを実行することが可能になる．これにより，膨大な HPC クラスタノードにライブラリやソフトウェアのインストールやアップデートをする必要がなく，そのコンテナイメージを動作させることでそのライブラリやソフトウェアを使用することが可能である．またコンテナ環境により，単一ライブラリにおいて複数バージョンを提供することが可能になる．これにより利用者側が固定のバージョンを使用している場合など細かい要件に対応することができる．HPC クラスタにコンテナ環境を導入することにより，管理者側の負担が減るだけではなく，利用さ h 側の利点も増え，バッジジョブの幅が広がる．

またコンテナイメージを使用することで利用者が日常的に使用しているコンテナ環境をそのまま再現することができる．そのため，ジョブ実行時にコンテナイメージを指定することで，そのイメージ上で実行することが可能となる．これによって，利用者が意図した環境で動作させることができるため，ジョブの安定化を図ることが可能である．

6. 実装

本論文ではジョブマネージャーに投入されるジョブスクリプトから専用コンテナイメージを作成し，そのイメージを使用し Kubernetes で Pod 上でジョブを実行し，結果を出力する機構を実装した．これらの機構は Go 言語によってコマンドとして実装した．Kubernetes が Go 言語によって実装されているため，本論文のコマンドを Kunerentes に組み込むことが可能である．

6.1 ジョブスクリプトからマニフェストファイルへの変換

Torque に投入されるジョブファイルの内容はシェルスクリプトである．Torque 用のオプションについてはファイル内の上部にメタ文字を先頭にして記述される．Torque のメタ文字は`#PBS` である．このメタ文字に続けて，オプションの引数を記述する．オプションはジョブをジョブキューに投入する際のコマンドである `qsub`[5] のオプションと同等である．これらの情報から Kubernetes で Pod を構築するために必要なマニフェストファイルを作成する．

実行したいバイナリはジョブファイルの中に記述されている．そこで本論文では実行するバイナリを取得するために新しく-B というオプションを作り，バイナリを指定するようにした．また既存のオプションから専用コンテナ環境を作成するために必要な情報を取得する．使用する CPU やメモリ，ノード数などのリソース量やジョブを一意にするためにジョブ名である．それぞれ-l,-N というオプションにより取得する．これらをマニフェストファイルの `resources` や `metadata,name` とすることで設定を Pod に反映する．また専用コンテナが Pod として構築された後，ジョブを実行するためにマニフェストファイルの `command` に実行するジョブを記述する．さらにこのコマンドに対し，結果をファイル出力させるようにリダイレクトされ，結果はジョブ実行後に閲覧できる．

Listing 1: 自動作成されたマニフェストファイル

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: torque-examplejob
5  spec:
6    containers:
7    - name: torque
8      image: centos/torque-examplejob
9      imagePullPolicy: IfNotPresent
10     securityContext:
11       privileged: true
12     volumeMounts:
13     - name: tasktorque-examplejob
14       mountPath: /mnt/task/torque-
           examplejob
15     resources:
16       requests:
17         memory: "512"
18         cpu: "1"
19     command: [/bin/sh]
20     args: [-c, /mnt/task/torque-
         examplejob/torque_task_example.sh
         >& /mnt/task/torque-examplejob/
         result.log]
21   restartPolicy: OnFailure
22   volumes:
23   - name: tasktorque-examplejob
24     hostPath:
25       path: /mnt/task/torque-examplejob
```

Listing1 は `torque_task_example.sh` というジョブスクリプトを読み込み自動生成したマニフェストファイルである．/mnt/task/torque-examplejob は Pod 内とホスト OS でボリュームマウントしており，ファイルの共有が可能である．また，ホスト OS とジョブを投入するノード間で NFS ファイル共有を行っているため，ジョブ投入時に対象ディレクトリにジョブファイルなどのコピーを行うことで，ジョブ投入ノード-コンテナ実行ノード-Pod(コンテナ) 間でのファイル共有を可能にしている．`args` ではジョブの実行とその結果の出力をリダイレクトで行っている．結果出力ファイルはファイル共有されているため，ジョブ投入ノード上で閲覧可能である．

6.2 共有ライブラリ情報とインストールするパッケージの取得

ジョブファイルよりメタ文字を使用して取得したバイナリ情報から共有ライブラリ情報を取得する．これにより，そのバイナリが実行するための共有ライブラリを取得できるため，そのライブラリをインストールした環境を用意することでその環境で実行することが可能となる．

共有ライブラリの取得には `ldd` コマンドを使用した．`ldd` コマンドはバイナリが使用してる共有ライブラリの情報を取得するコマンドである．このコマンドにジョブファイルから取得したバイナリを引数にとり実行した結果を保持する．それにより，どの共有ライブラリを使用しているかが判明する．

`ldd` コマンドを実行した結果から得た共有ライブラリ情報を使用してインストールするパッケージ情報を取得する．そのパッケージ情報の取得にはパッケージ管理コマンドである `yum` を使用した．`yum` は Red Hat 系の Linux ディストリビューションで使用されるパッケージ管理コマンドであるが，このコマンドの機能である `whatprovides` サブコマンドが存在する．これを使用することで共有ライブラリをインストールするために必要なパッケージを取得することができる．これを各共有ライブラリで実行することでその共有ライブラリをインストールするために必要なパッケージの一覧を取得することができる．これにより，ジョブを実行するための環境をコンテナ環境で構築できる．

6.3 ジョブ専用コンテナイメージの共有

HPC クラスタにおいてコンテナイメージを共有するためにコピーを行う．コンテナイメージは Docker コマンドにおける save 機能を使用することにより，`.tar` ファイルに圧縮することができる．このファイルを他のクラスタノードにて Docker コマンドの load 機能を使用し，ローカルのコンテナイメージとして認識することができる．これらのコンテナイメージ共有処理は全クラスタノードにおいて ssh を用いて行う．これにより，Kubernetes 上で load したイメージを元に Pod を生成し，ジョブを実行することができる．本来 Kubernetes のイメージはイメージリポジトリ上からプルしてくる必要があるため，そのままイメージを指定するとエラーが発生する．そのため `imagePullPolicy`

をマニフェストファイルに記述することでローカルのイメージを使用するようにする．

6.4 マニフェストファイルを使用したジョブ実行

ジョブスクリプトからマニフェストファイルに変換したのち，そのマニフェストファイルを Kubernetes で実行し，Pod としてジョブを実行する．この Pod でジョブを実行する際に必要なジョブのバイナリは Volume マウントされたディレクトリ以下にあり，そこで実行される．またその Volume マウントしたディレクトリにジョブの結果を出力する．Volume マウントはホスト OS 上のディレクトリをその Pod のディレクトリにマウントできる機能である．この機能を使用しジョブを実行する．またこのディレクトリは Volume マウントのタイプを HostDir にすることで，Pod を削除した後もそのディレクトリは残る．さらにここでマウントしているディレクトリはホスト OS 上でも NFS を使用してマウントしており，実際はジョブを投入するノードに存在するディレクトリである．このディレクトリはジョブによって異なるため，競合もない．また，このディレクトリ以下にジョブスクリプトとバイナリがコピーされる．

Pod としてジョブを実行する際には Kubernetes のコマンドである，`kubectl` コマンドを使用した．このコマンドは Kubernetes クラスタに対して提供されている API を使用し，Kubernetes 上で様々な機能を使用するために用いるコマンドである．これを使用することで Kubernetes 上で Pod を構築することができる．またマニフェストファイルでジョブスクリプトにおける条件が変換されているため，構築後その条件を有した状態でジョブを実行することが可能である．

7. 評価

本論文で実装した専用コンテナ環境におけるジョブ実行時間の評価を行った．また実装した機構が実行できるジョブの種類を評価し，妥当性を示す．これによりバッジジョブにおいて，専用コンテナ環境においてジョブを実行し，結果を出力できることを示す．

今回評価に使用した環境はジョブを投入するためのマスターノード 1 台とジョブをコンテナ上で実行するための計算ノード 4 台の計 5 台構成で評価を行った．それぞれのノードには Kubernetes version 1.13.1 をインストールし，ジョブスクリプトからコンテナ環境でジョブを実行できる．

7.1 ジョブの妥当性

ジョブが専用コンテナ上で実行に成功するか失敗するかは共有ライブラリがインストールできるかできないかに基づく．そのため，yum コマンドでデフォルトのリポジトリからインストールできない共有ライブラリをバイナリで使用している場合，実行できない．DockerHub にアップロードされているコンテナイメージにおいてデフォルトで指定されているリポジトリは `base/7/x86_64,extras/7/x86_64, updates/7/x86_64` の 3 つからなる 12272 個である．これらのリポジトリから提供されている共有ライブラリは本論文の実装であるとインストールできない．そのため，本論文の評価で用いるジョブはこれらのリポジトリから取得できる共有ライブラリのみを使用するジョブを実行している．

7.2 スクリプトジョブの実行時間

Listing2 は今回使用したフィボナッチ数列を計算するジョブである．このジョブは 6 番目のフィボナッチ数までを表示する．ファイル内にはメタ文字を使用し，ジョブのリソースを指定している．実行ノードは 1 台，使用する CPU が 1 コアである．このジョブではバイナリが指定されていないため，共有ライブラリの取得を行わない．そのためコンテナ環境が作成されてからジョブが実行されるまでの時間となる．このジョブスクリプトを実行し，コンテナ環境で動作するジョブ実行時間を計測し，ネイティブ環境での実行時間と比較する．

Listing 2: フィボナッチ数列を計算するジョブスクリプト

```
1  #!/bin/bash
2  #PBS -l nodes=1:ppn=1
3  #PBS -N exampleJob
4  N=6
5  a=0
6  b=1
7  for (( i=0; i<6; i++ ))
8  do
9      echo -n "$a "
10     fn=$((a + b))
11     a=$b
12     b=$fn
13 done
```

この評価で実行したジョブは Listing2 のジョブスクリプトファイルを使用する．TORQUE ネイティブ環境はジョブを実行する際に qsub コマンドを使用し実行する．コンテナ環境では本論文で実装したコマンドを用いてジョブを実行する．これにより TORQUE からジョブを実行した際とコマンドからコンテナ環境でジョブを実行した際の実行時間の比較を行う．表 1 はスクリプトジョブの実行時間評価結果である．これにより，ネイティブ環境で 0.02 秒程度で動作するジョブスクリプトにおいて，コンテナ環境では約 3 秒かかる．時間はかかるものの，コンテナ環境においてスクリプトジョブが動作することがわかる．

環境	平均時間 (秒)	最大時間 (秒)	最小時間 (秒)
ネイティブ環境	0.024	0.027	0.023
コンテナ環境	2.90	3.08	2.80

表 1: スクリプトジョブの実行時間

環境	平均時間 (秒)	最大時間 (秒)	最小時間 (秒)
ネイティブ環境	0.24	0.25	0.15
コンテナ環境	8.64	12.03	6.10

表 2: バッジジョブの実行時間

7.3 バッジジョブの実行時間

バッジジョブの実行時間を計測し，ジョブが実行可能な時間で実行できることを示す．今回の計測にはジョブを単純なものにするため，簡単な計算を行うジョブを選択した．

Listing 3: フィボナッチ数列を計算するバッジジョブ

```
1 #!/bin/bash
2 #PBS -l nodes=1:ppn=1
3 #PBS -N exampleJob
4 #PBS -B a.out
5
6 ./a.out
```

Listing3 は Listing2 を C++で実装し，コンパイルしたものである．このジョブは同様に 6 番目のフィボナッチ数までを表示する．ファイル内にはメタ文字を使用し，ジョブのリソースを指定している．実行ノードは 1 台，使用する CPU が 1 コアである．開始時間はジョブを実行を送信した時間であり，終了時間はジョブの結果がファイル出力された時間である．この時間はイメージを作成する時間は含まれていない．

対象のジョブを 100 回実行し，その実行時間を評価した．表 2 はバッジジョブの実行時間評価の結果である．評価結果よりネイティブ環境は平均時間 0.24 秒，コンテナ環境は 8.64 秒であることが判明した．ネイティブ環境環境はすべて 1 ホストで実行されるため結果ファイル出力がコンテナ環境に比べ高速である．コンテナ環境では約 8 秒程度かかってしまう．

また専用のコンテナイメージを作成する場合，約 44 秒程度の時間がかかる．この時間は必要な共有ライブラリをインストールしたコンテナイメージを作成し，各ノードに配布している時間が大部分を占めている．この時間は共有ライブラリの種類や共有する HPC クラスタノード数に大きく影響する．クラスタノードに共有する時間は 1 ノードあたり約 4 秒程度である．しかしながら，一つのコンテナ環境を作成し，ジョブを実行するまでにかかる時間としては妥当な時間であり，実行不可能な時間ではない．

8. 考察

第 7 章で行った評価結果において，ジョブ実行時にコンテナを作成する際にかかる時間により，ネイティブ環境と比較し，大幅に時間がかかっている．その時間差はコンテナ環境生成にかかる時間が大部分を占めており，ジョブの実行時間においては大差ない．また専用コンテナを作成する手順である，共有ライブラリの検索やコンテナイメージの作成，共有を含む場合，約 44 秒かかる．コンテナイメージを作成しているためにこのような大幅な時間がかかる．表 2 よりバッジジョブの実行時間平均は 8.64 秒であるが，差分の 35 秒程度だとわかり，ジョブ専用コンテナイメージ作成に使用されている時間がその時間であるとわかる．実際数十秒から数分で終了するならば，実際の HPC クラスタ上で実行するジョブの計算時間と比べ，利用者から見て気になる時間ではない．この時間によって HPC クラスタで実行できなかったジョブが実行できるようになる利点の方が大きいと考える．

コンテナ環境作成時間の高速化を図るためには，コンテナイメージ作成時間を減少させる必要がある．本論文ではジョブ投入ノードでコンテナイメージを作成し，そのイメージを圧縮したファイルを他ノードにコピーし使用している．コンテナイメージを作成する際，元となるコンテナイメージを DockerHub からプルしている．予めこのイメージをプルしておくことで時間を減少させることができる．利用者がコンテナイメージをすでに作成している場合，そのイメージをクラスタにアップロードすることでそのイメージを使用することができ，新規にイメージを作成する時間もなくなる．

コンテナイメージの共有は圧縮したものをコピーせず，NFS 等のファイル共有を使用する手法がある．この手法では Docker のイメージ構造である OverlayFS での Low レイヤーのみを共有し，追加されるレイヤーは該当ノードで保持することで，初期イメージがネットワークから取得できる．イメージをメモリで保持することで，高速化を図ることができる．

コンテナイメージ自体の容量を軽くすることで高速化できる．これは最低限のコンテナイメージを作成することである．本論文の実装では CentOS をベースのコンテナイメージとして用いたが，このイメージにはジョブ実行に必要ないデータも含まれている．それらを取り除き，ジョブを実行するための最低限の環境を作成することにより，イメージ作成や転送にかかる時間が減少する．

本論文で実装した環境では CentOS の RedHat 系ディストリビューションを使用したため，パッケージ管理に yum を使用した．そのため yum で取得できる，ローカルの登録されたリポジトリを確認するため，特殊な共有ライブラリ

を使用している場合には対応できない可能性がある．それに対応するために，既存のコンテナ上で開発環境を利用者に作成させ，そのコンテナから共有ライブラリをコピーする手法がある．その場合，コンテナ環境を作成する知識が必要になり，既存のジョブマネージャーを使用してる利用者のラーニングコストが高い．さらにそのコンテナ環境を用意するリソースが別で必要になり，ジョブ実行能力が低下してしまう．

今後の課題として，バッジジョブ以外のジョブに対応することである．具体的には機械学習や数値計算である．機械学習のジョブから自動でコンテナ環境を構築することで，クラスタ管理者がライブラリのバージョン更新をすることなく，利用者が自分の使用したいライブラリのバージョンを用いてジョブを実行することができる．

しかし，利用者から受け入れたジョブを実行するだけではセキュリティの問題がある．Docker 内で不正なライブラリやソフトウェアが実行されることで，クラスタの環境を破壊したり，他人のジョブ出力結果を不正に取得したりなどが考えられる．これらに対応するため，ネイティブ環境で実行できない不正ジョブをコンテナ環境で実行しないよう，ジョブスケジューラで判断する必要がある．

また課題として，ジョブマネージャーとコンテナオーケストレータの連携がある．連携に関してはジョブマネージャーがどのノードでジョブを実行するか判断し，そのノードにコンテナオーケストレータがコンテナ環境を作成する．これにより，ジョブマネージャーが実行するノードの選択権があるため，コンテナイメージを全ノードに配布する必要がなくなる．

9. 結論

近年の HPC 環境ではコンテナを使用することで様々な課題を解決している．本論文では TORQUE を使用したクラスタ環境においてその環境の課題を説明し，TORQUE と Kubernetes を連動させる設計について述べた．TORQUE 用のジョブスクリプトからそのジョブ専用の Kubernetes 上のコンテナ上でジョブが実行され，結果がファイルとして出力されるコマンドを実現した．実際に TORQUE と Kubernetes が動作しているクラスタにて動作検証をしたところ，ジョブとして投入したスクリプトファイルがコンテナ内で実行され，出力がファイルとして保存されていることを確認した．現状として，TORQUE のキューからジョブを取得し，コンテナを作成から一連の流れを実行する機能は実現できていないため，今後 TORQUE のスケジューラやサーバーに機能を実装し，今回作成したコマンドと連携させる必要がある．

参考文献

[1] Adaptive Computing.: TORQUE Resource Manager, 入手先 ⟨https://www.adaptivecomputing.com/products/torque/⟩ (accessed 2019-05-02).
[2] CLOUD NATIVE COMPUTING FOUNDATION.: Production-Grade Container Orchestration, 入手先 ⟨https://kubernetes.io/⟩ (accessed 2019-05-02).
[3] Kurtzer, Gregory M. AND Sochat, Vanessa AND Bauer, Michael W.: Singularity: Scientific containers for mobility of compute, 入手先 ⟨https://doi.org/10.1371/journal.pone.0177459⟩ (accessed 2019-05-09).
[4] Higgins, Joshua and Holmes, Violeta and Venters, Colin.: Orchestrating Docker Containers in the HPC Environment, 入手先 ⟨https://www.researchgate.net/publication/300779185_Orchestrating (accessed 2019-05-09).
[5] Adaptive Computing.: A.21 qsub, 入手先 ⟨http://docs.adaptivecomputing.com/torque/6-1-2/adminGuide/torque.htm#topics/torque/commands/qsub.htm⟩ (accessed 2019-05-03).