負荷テストによるKubernetes Pod構成の CPUとメモリ値の自動設定

伊藤 佳城[1] 串田 高幸[1]

概要：Kubernetes (K8s) は, Pod の CPU とメモリの使用量を設定することで，Pod の実行に必要な CPU とメモリを確保することが可能である．しかし，CPU とメモリの使用量の値を決めるためには実行する Pod 内のアプリケーションに対するテストシナリオを作成し，測定を行い値を決める．しかし，測定した値の決定はユーザの経験則に依存する．本研究の提案は，K8s 上にデプロイした Pod へ負荷を与え，CPU とメモリの使用量を取得し，これらの値をもとに limits の設定値を自動的に算出する事である．実験は，WordPress の Pod に対して HTTP リクエストを送信し，その際の CPU とメモリの使用量の最大値を limits に設定し，そこから CPU とメモリの limits を 1%ずつ減少させ最小値を決定する．本提案によって設定なしの場合と比較し，結果として HTTP リクエスト数が 40(req/s) の時，最大で 144 (ms) 削減でき，設定を行うことによる応答時間の短縮させることが可能となった.

1. はじめに

K8s は，Google で開発されたオープンソースのコンテナオーケストレーションシステムである [1]. K8s では，1 つ以上のコンテナのグループを Pod という単位で管理しており，Pod 内のコンテナに対する CPU とメモリに関する設定が存在する [2][3]．Pod 内のコンテナの設定として CPU とメモリ使用量の requests(要求量) と limits(制限量) を設定可能である．CPU は，1 (vcpu)=1000 (millicores) として定義されており 100 (millicores) と設定すると 1 (core) の 10%が使用される．requests は，コンテナをデプロイする際に必要とする CPU とメモリ使用量の必要な最小値を設定可能である．requests が設定されている場合，requests の要求量がノードに空いているかでデプロイが行われる．limits はコンテナの CPU，メモリ使用量の制限値である．これらの設定が無い場合，コンテナは実行中のノードで空き容量の CPU とメモリを無制限に使用可能となる．limits を超える量をコンテナが使用しようとした場合，CPU 使用量が制限されているため処理速度が遅くなる．

メモリの場合 Linux の OOM (out of memory) Killer プロセスが起動中の Pod を停止させてメモリ使用量を確保する [4]．その際に停止する Pod は limits の設定がない Pod が優先的に削除される [5].

K8s には，Horizontal Pod Autoscaler (HPA) と Vertical Pod Autoscaler (VPA) のオートスケーラの CPU とメモリ使用量の管理におけるオートスケーリングの機能が備わっている [6]．VPA では Pod の CPU とメモリ使用量の変更中に起動しているアプリケーションとサービスの継続性が損なわれる [7].

[1] 東京工科大学大学院 バイオ・情報メディア研究科 コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

1.1 課題

新しい Pod をデプロイすると同一ノード上で実行中の Pod の影響を受け, 新しく作成する Pod に必要な CPU とメモリ量が実行中の Pod に CPU とメモリを使用され，占有される場合がある．K8s のノードは 1 つの VM だとすると，ノードの CPU とメモリ, ストレージは VM の設定値があらかじめ決まっているため上限がある．設定値の中でノードの CPU とメモリが不足し, Pod の動作が停止することがある [8]．その際のノードと Pod の関係を図 1 に示す．CPU とメモリの使用量は，Pod 内のアプリケーションによって変化する．ノード全体の CPU が 1000 (millicores) の場合, Pod1 の必要 CPU が 300 (millicores) に対して設定 CPU が 700 (millicores) に値を決定した場合, Pod2 の必要 CPU の 600 (millicores) を満たすことができない．Pod2 では必要 CPU が 600 (millicores) であるのに対し，設定 CPU が 400 (millicores) となる．同一ノード上の Pod1 が Pod2 の CPU 使用量に影響を与える．Pod1 の設定値が必要量より多く，使用できる CPU 使用量に限りができる．Pod2 は必要な分の CPU 使用量が確保できない．メモリ使用量に関しても同様である.

©2021 Information Processing Society of Japan

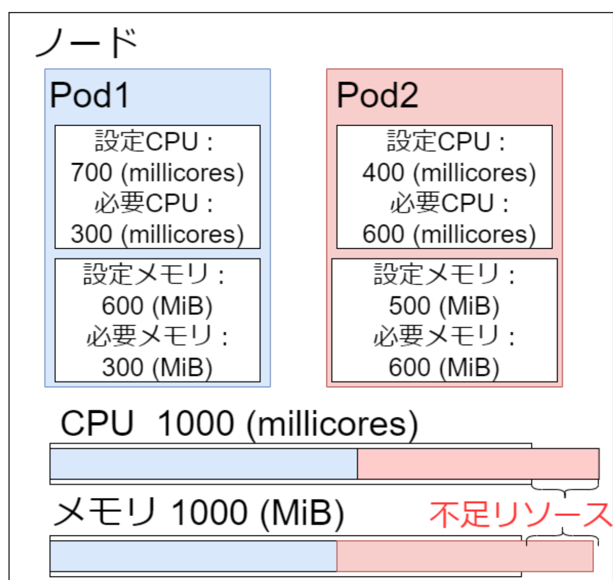

図 1 Pod の CPU とメモリ制限設定が無い場合に発生する課題

ノードの CPU とメモリが不足せず安定した動作をするために CPU とメモリの limits を設定する．しかし, limits の値を決めるためには実行する Pod 内のアプリケーションに対するテストシナリオを作成し，実際に CPU とメモリの使用量を計測および測定をする必要がある．また，CPU とメモリの limits の設定値を決定する場合，ユーザ自身の経験則に依存する場合や Pod 内のアプリケーションの内部構造を十分に理解する必要がある．これには膨大な時間と学習コストが発生するため，ユーザ全員が一意に limits の値を決定することは困難である．

各章の概要

この論文は, 次のように構成される．1 章では, 本研究の背景・課題について述べる．2 章は, 関連研究の説明を行う．3 章では, 本研究の提案について述べる．4 章では，実装とその実験環境について述べる．5 章では，評価と分析を行う．6 章では今後の課題や議論・考察を行う．最後に 7 章では, 最終的なまとめとする．

2. 関連研究

コンテナのシステムリソースに関する論文では, 挙動を監視するためコンテナの一連の評価を行っており．手動にデプロイされたクラスタとの比較をしている．比較対象を手動と自動化に絞ることで評価を比較している [9]．この研究では，K8s のクラスタを各種ベンダーが提供するクラウド環境でコンテナを実行した際のパフォーマンスを図っているがローカルで構築された環境下におけるパフォーマンスの評価は行っていない．

Docker コンテナの垂直弾性 (powering vertical elasticity) を自律的に強化する最初のシステムである ELASTIC-DOCKER を提案している．この研究では，アプリケーションのワークロードに応じて, 各コンテナに割り当てられた CPU とメモリの両方をスケールアップおよびスケールダウンを行い，ホストにリソースがない場合コンテナを別のホストに移す動作を取っている [10]．この研究では，スケールアウトでコンテナの弾性力の増加を図っているがコンテナ単体の設定を行っていない．

コンテナ内にデプロイされたアプリケーションは，同じマシン上の共有リソースが分離できておらず，パフォーマンスの問題を引き起こす原因となっている．より適切に特徴付けるために，K8s 内のリソース管理のリファレンスネットベースのモデルを開発している [11]．この研究では，コンテナのライフサイクルにおける時点で各コンテナに確率分布を付与することでオーバーヘッドを特定しているがリソースの分離レベルでの解決をするため，コンテナにおけるリソースの設定には, 関係していない．

K8s のポッド間でリソース制限を公平に割り当てることは困難な問題に焦点を当て Pod 間でリソース制限を公平に割り当てる新しい方法を実践している [12]．この研究では，公平にリソースを割り当てるが管理者が全体的な環境の空き状況を把握していることが前提となり実用性は低いとされる．

K8s のシステムパフォーマンスを分析してリソース管理のリファレンスネットベースのモデルを開発したこの研究では，Pod とコンテナからアプリケーションをどのように構築するかの提案をする [13]．しかし, この研究では, 実際の負荷情報における検証がなされておらずその部分が課題として挙げられている．

K8s に基づく動的なリソースプロビジョニングを容易にする汎用プラットフォームの開発を目的としたこの研究では，既存のリソースプロビジョニングに加え動的な管理と監視対象をリソース使用率と QoS メトリックとしている [14]．しかし，この研究では Pod に対する負荷があった場合のリソース量の判別は，デプロイ時の情報を元に行われるため，想定して設定を構築するわけではない．

これらの関連研究においては実際の環境下での検証がなかったり，個々のコンテナに対して設定を行なっていなかったりしている．この場合．コンテナごとに設定を変更できないため，コンテナ内部のアプリケーションに合わせたリソース管理が実現できない．

3. 提案

本研究の提案は，K8s 上にデプロイした Pod へ負荷を与え，CPU とメモリの使用量を取得し，これらの値を元に limits の設定値を自動的に算出する事である．本提案では Web アプリケーションサーバのコンテナを想定し，ユーザからのリクエストに応答するワークロードを対象とする．また，前提として Pod の起動した際の CPU とメモリの使用量の最大値を requests として設定し，固定する．

©2021 Information Processing Society of Japan

図2におけるアーキテクチャの説明の流れを下記に示す.

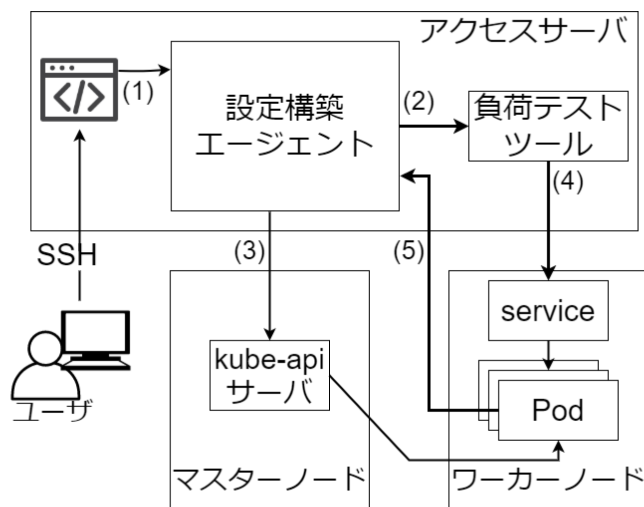

図 2 アーキテクチャ図

(1) ユーザがアクセスサーバに SSH アクセスし，設定構築エージェントを起動

(2) 設定構築エージェントが負荷テストツールを構築・起動

(3) 設定構築エージェントが kube-api サーバを介して Pod と service をデプロイ

(4) 負荷テストツールを使用して service に対して HTTP リクエストを送信

(5) HTTP リクエストを送信した Pod の CPU とメモリの使用量を設定構築エージェントで取得

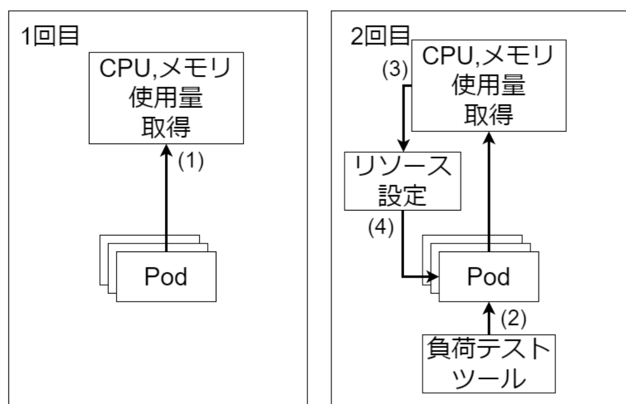

図 3 CPU とメモリ設定方法 1

細かい CPU とメモリ設定方法の流れを図 3 と図 4 に示す. 図 3 では, 基準となる CPU とメモリの limits の設定手順の流れを説明する. 前提として HTTP リクエストの受付時間は，HTTP リクエスト実行期間と同じとする.

(1) Pod に対し, 起動時にかかる CPU とメモリの使用量を取得

(2) 指定した HTTP リクエスト (req/s) を送信

(3) (2) で行った負荷テストの CPU とメモリ量を取得

(4) (3) で取得した最大値を limits に設定

図 4 では, 基準となる CPU とメモリの limits から減少させる手順の流れを説明する. 本研究では，ユーザが利用するアプリケーションによってワークロードが異なるとする．ワークロードが異なることで CPU とメモリの相互依存の有無はアプリケーションによって定まる．そのため今回は前提として独立していることとする.

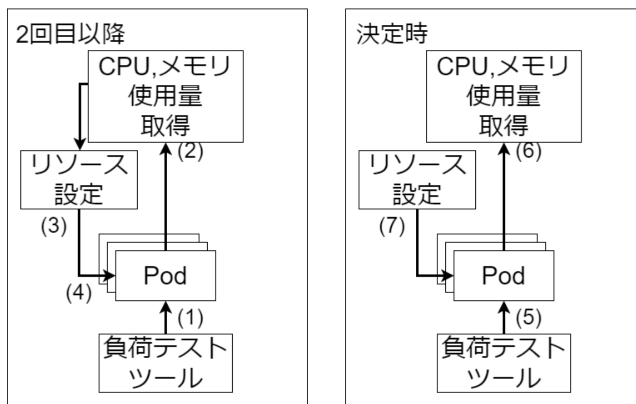

図 4 CPU とメモリ設定方法 2

1 (millicores) ずつ減少させる場合は，上限数を n とすると，計算量は O(n) となる．しかし，1%ずつ減少させることで計算量は O(100) となるため，設定値算出までの時間が短縮可能である．また，システムの環境によって CPU やメモリ量は異なるが，割合を用いることで環境依存しない減少値として扱うことができ，計算量も固定されるため 1%を用いる.

(1) 値を設定した Pod に対して指定した HTTP リクエスト (req/s) を送信

(2) (1) で行った負荷テストの CPU とメモリ量と処理された 1 秒あたりの HTTP リクエスト数を取得

(3) 指定した HTTP リクエスト数 (req/s) を満たせていたら Pod の設定値から 1%減らした値を設定

(4) Pod を設定した値に更新してデプロイ

(5) (1) と同じく HTTP リクエストを送信

(6) 負荷テストの CPU とメモリ量と処理された 1 秒あたりの HTTP リクエスト数を取得

(7) 指定した HTTP リクエスト数 (req/s) を満たせていなかったら 1 つ前の設定に戻して Pod をデプロイ

最終的にデプロイされる値は，ノードの空き容量を確保できる最大の値にする．この提案によって本研究で取り上げた課題である CPU とメモリの使用量の limits, requests の設定値がユーザの経験則に依存する点を解消し，limits, requests を決定する.

3.1 ユースケース

ユースケースとして K8s クラスタを利用する WordPress と MySQL を利用する Web サービスを開発および実装す

る例を取り上げる [15]．ユースケースのターゲットとしてWeb サービスを構築する開発者とする．そのユースケースを図 5 に示す．

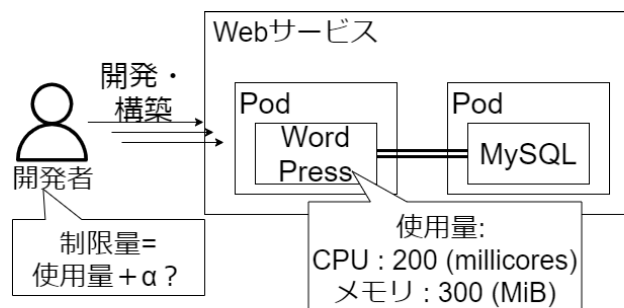

図 5 ユースケース図

環境構築するにあたり，WordPress と MySQL 2 つの Pod が使用される．この際に想定される問題は構築における Pod の CPU とメモリの値である制限量を手動で決めている点である．ここでの制限量は CPU とメモリの使用量の limits のことを指す．Web サービスを構築する際にどの程度のパフォーマンスや処理リクエスト数を出すべきかは Web サイト構築の性能用件で定義される．その要件を満たす制限量の決定はテストを行いそのときの CPU とメモリの使用量から開発者が決定する．図 5 のようにWordPress の CPU 使用量が 200 (millicores), メモリ使用量が 300 (MiB) の場合，この値から制限量を決定するには使用量にいくつ+α するのかを開発者が手動で決めなくてはならない．自動化で CPU とメモリの使用量を算出することで，実際の構築および運用での CPU とメモリの使用量の値を開発者が手動で決定する手順を省くことが可能となる．

4. 実装と実験環境

4.1 実装

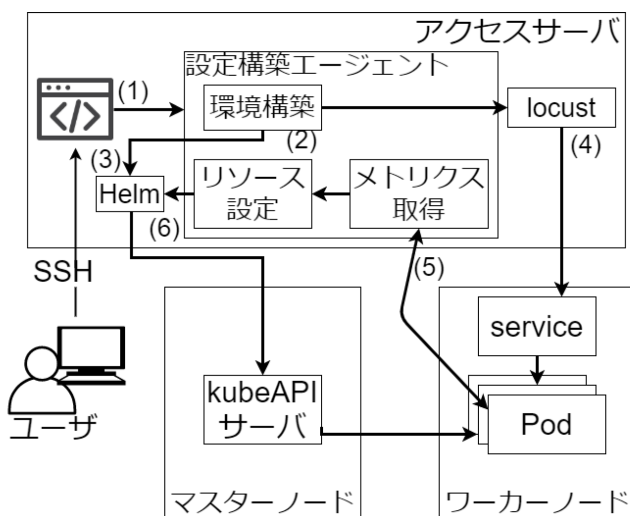

図 6 実装図

本研究の提案における実装のプログラムコンポーネントは下記の 3 つからなる．

- 環境設定
- メトリクス取得
- CPU とメモリ設定

個別のプログラムを一連の流れとして動作させるのが自動化プログラムである設定構築エージェントである． Python3 の subprocess を用いて基本的にコマンドで呼び出している．シェルスクリプトで書かれた環境構築を起動して実装に必要な負荷テストツールの Locust, K8s のパッケージマネージャである Helm をインストールする．Helm を使用し K8s 上に WordPress のデプロイを行った後，負荷テストツールである Locust を使用して Python3 の subprocess を用いて CLI から負荷テストを行っている．行った Pod のメトリクスを python3 による Kubernetes client api を使用し, Pod の CPU とメモリの使用量をメトリクス取得プログラムで取得する．そして取得したデータを元に CPU とメモ設定で CPU とメモリ制限の値を Helm の value.yaml に記述する．

さらに流れを下記に示す．

(1) ユーザがアクセスサーバに SSH アクセスし，設定構築エージェントを起動
(2) 環境構築プログラムが実行され環境構築に必要なソフトウェアをインストール
  - 環境設定に必要なソフトウェア (Helm, Locust) のインストールおよび構築
(3) Helm を使用し，コンテナ化されたアプリケーションの構築
(4) 負荷テストツールである Locust を構築，Pod に負荷テストとして HTTP リクエストを送信
  - 負荷テストに必要な設定ファイルを読み込み，Locust を実行
  - 実行される HTTP リクエスト (req/s) は 0 から 50 まで 10 刻みの 5 パターンで送信
(5) メトリクス取得を使用して負荷テスト中の Pod の CPU とメモリの使用量を収集
  - 負荷を与えられている状態の CPU とメモリの使用量を収集する．この際収集したデータは CSV ファイルで保存
(6) 収集した CPU とメモリの使用量を基に CPU とメモリ設定で Pod の CPU とメモリの limits を 1%ずつ減少させて設定
  - 設定を YAML ファイルのマニフェストに記入し，設定を反映した Pod をデプロイ

4.2 実験環境

実装環境は Ubuntu18.04 の VM3 台で構築した K8s クラスタ (マスターノード 1 台, ワーカーノード 2 台) を使用す

©2021 Information Processing Society of Japan

る．Ubuntu18.04 の VM 上から K8s クラスタのマスターノードにアクセスするため，kubectl をインストールした K8s クラスタへのアクセスサーバ 1 台を構築した．K8s クラスタとアクセスサーバ計 4 台の VM で実験を行う．下記に VM の基本構成を示す．

- vCPU x 2
- RAM 4GB
- HDD 50GB

本研究の実験方法では，負荷の対象とする Pod はユースケースと同じく WordPress を対象とする．パッケージマネージャである Helm を使用し，bitnami/WordPress の Chart を利用しデプロイを行う．どちらも共通の条件として，負荷テストツールを使用し，Pod の CPU とメモリの使用量を Kubernetes client api を使用して取得する．

5. 評価と分析

5.1 評価方法

評価方法は，図 7 の環境を用いて応答時間，処理リクエスト数，CPU とメモリ使用量を設定ありと設定なしで比較し，増加量および減少量から本提案による効果を定量的に判断する．

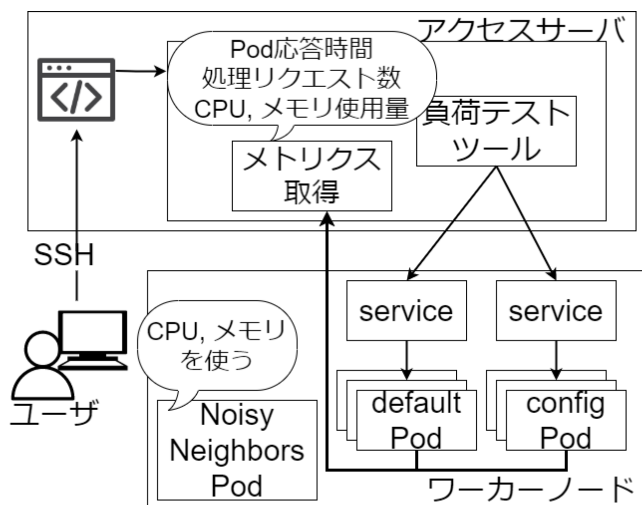

図 7 実験環境の概要図

負荷テストを CPU・メモリ変更以外は同じ条件で行うことを前提とし, Pod の limits を指定した後の Pod を config Pod, 変更を加えていない Pod (デフォルト値) を default Pod として設定を行う．コンテナ内で Stress-ng コマンドを実行し，CPU を 100%, メモリを 1500 (MiB) 使用する Noisy Neighbors Pod を作成する．Noisy Neighbors Pod を使用することで同一ノードにデプロイされた Pod が影響を受けるかを実験する．Noisy Neighbors Pod を 1 つワーカーノードにデプロイし，同一ノードに実験対象とする config Pod もしくは default Pod を 1 つデプロイする．その際に HTTP リクエストを処理した場合の Pod からの応答時間，処理リクエスト数、CPU とメモリの使用量から CPU とメモリ制限の有無でどのような影響が出るか評価を行う．

表 1 CPU とメモリの requests

requests CPU (millicores)	5
requests メモリ (MiB)	196

表 2 CPU とメモリの limits 初期値

HTTP リクエスト (req/s)	10	20	30	40	50
初期 limits CPU (mllicores)	318	643	922	1193	1484
初期 limits メモリ (MiB)	234	274	325	372	431

表 3 CPU とメモリの limits 決定値

HTTP リクエスト (req/s)	10	20	30	40	50
決定 limits CPU (mllicores)	267	553	747	954	1336
決定 limits メモリ (MiB)	197	236	263	298	388

5.2 実験結果

Pod における CPU とメモリの requests の値を表 1 に示すように固定値とし，CPU とメモリの limits の初期値を表 2 に示す．また，小数点以下の数値の設定はできないため小数点以下は四捨五入をして整数の値を入力する．

requests の CPU とメモリの値に関しては，CPU が 5 (millicores), メモリが 196 (MiB) で固定とする．

limits の初期値は 10-50 (req/s) ごとの CPU とメモリの値を算出し設定した．この値をもとに減少させ最終的な決定値を算出する．決定した CPU とメモリ使用量の limits の決定値を表 3 に示す．

本提案では，limits の減少幅を 1 (millicores) ずつではなく 1%ずつに決定した．10 (req/s) の CPU 使用量は初期 limits の設定値より 51 (millicores) 減少させた値を設定できた．この値を求めるために必要な処理数は，O(14) となる．減少幅を 1 (millicores) とした場合，必要な処理数は O(51) となる．20-50 (req/s) も同様に本提案の減少幅で計算量の削減を行うことができた．

Locust を用いて HTTP リクエストを 1 秒に 1 回送信し，送信した際のリクエスト応答時間と処理リクエスト数と Pod の CPU とメモリの使用量を 60 秒間取得する．ここでは limits の値を決定している Pod を設定あり, 値を決めていないものを設定なしとして評価する．Pod の応答時間の結果を図 8 に示す．設定ありの方が設定なしよりすべての HTTP リクエスト数において応答時間が早くなる結果となった．30 (req/s) 時，設定なしが 206 (ms), 設定ありが

©2021 Information Processing Society of Japan

164 (ms) となり，設定あり方がが 42 (ms) 早い結果となった．40 (req/s) 時，設定なしが 321 (ms)，設定ありが 177 (ms) となり，設定ありの方がが 144 (ms) 早い結果となった．これらの結果，設定ありの方が設定なしより 42-144 (ms) 間で応答時間が減少した．

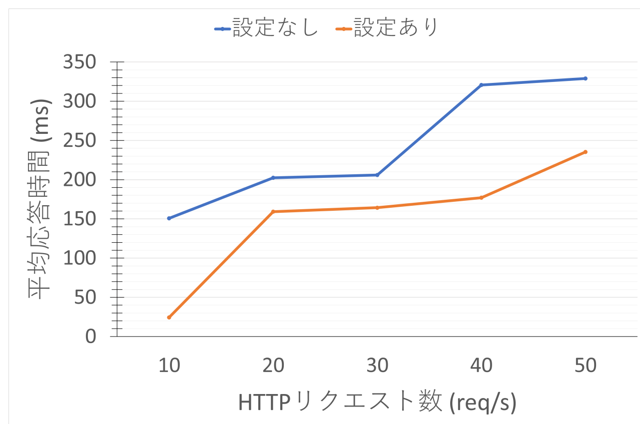

図 8 Pod 応答時間

Pod の処理リクエスト数の結果を図 9 に示す．10-30 (req/s) までは設定の有無ではほとんど変化が無い結果となった．40,50 (req/s) の時，設定ありは処理リクエスト数が 0.5 (req/s) 程度増加する結果となり，設定ありの処理リクエスト数が高い結果となった．

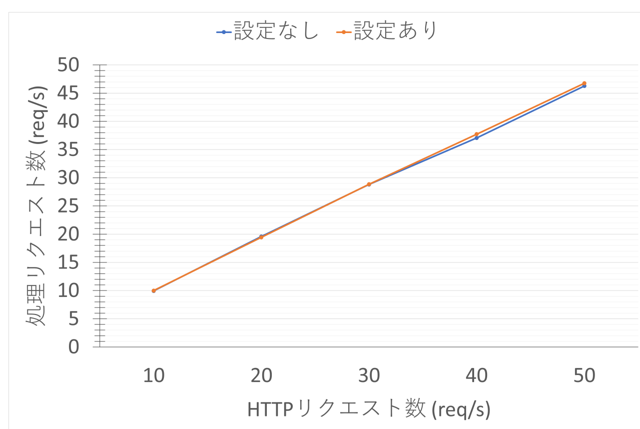

図 9 処理リクエスト数

Pod の CPU 使用量の結果を図 10 に示す．CPU 使用量は，設定なしと設定ありを比較すると，10 (req/s) の時，設定ありが 244 (mllicores)，設定なしが 173 (millcores) となり，設定ありが 70 (millicores) 程度増加した．20-50 (req/s) では設定ありが 1-27 (millicores) 増加した．

Pod のメモリの使用量の結果を図 11 に示す．メモリ使用量は，設定なしと設定ありを比較すると，20 (req/s) の時，設定ありが 199 (MiB)，設定なしが 146 (MiB) となり設定ありが 53 (MiB) 程増加した．

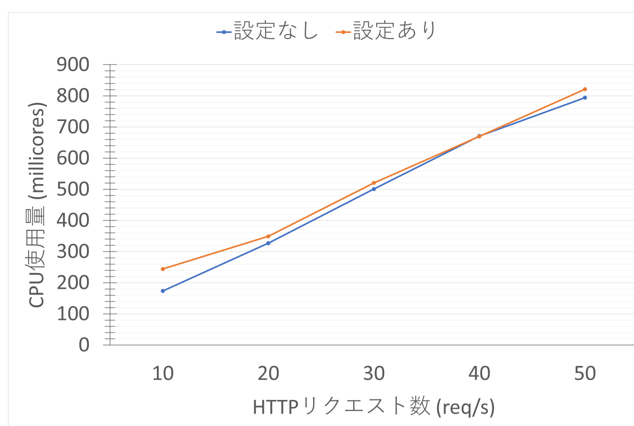

図 10 CPU 使用量

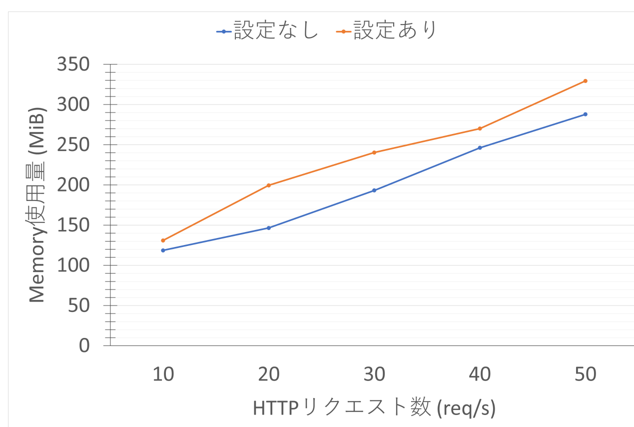

図 11 メモリ使用量

5.3 評価・分析

図 8 の実験結果から，Pod の応答時間は設定なしに比べ，設定ありの応答時間が早い結果となった．これは，図 10, 図 11 からわかるように，設定ありの方が CPU とメモリの使用量がともに多くなたことが要因である．同じ量の HTTP リクエストを処理した場合，設定ありではより高いパフォーマンスを引き出すことができたため，図 8 のように応答時間が設定なしよりも早くなった．HTTP リクエスト数が 40(req/s) の時，144 (ms) 削減でき，設定を行うことによる効果が大きいことが示された．

また，CPU 使用量およびメモリ使用量が増加することによって処理リクエストが増加することが実験結果で示された．

6. 議論

本研究の議論点として Pod の CPU とメモリの設定方法，設定の有無による CPU とメモリの使用量の違いについて，CPU とメモリのパラメータが相互依存している場合の 3 点を取り上げる．本研究では，CPU とメモリ使用量のから設定値を決定し，その値を基準値として 1%づつ減らしていく手法を取った．1%は基準値となる値に依存し，その値が大きければ大きいほど 1%における値は減少値として大

©2021 Information Processing Society of Japan

きくなる．これによって同じ 1%でも一度に変更する値が異なり最小値を求める最効率な手順とはならない．効率的な設定方法としてアルゴリズムの適応が必要となる．例として二分探索の適応方法がある．負荷テストにおけるリクエストの処理応答時間に対しての必要な CPU とメモリの最小値を求める．負荷テストを一定時間行った際の CPU とメモリ使用量を取得し，取得した値の中央値をとりその値を設定する．その設定値で処理応答時間が満たせていたらさらに中央値を設定し，探索を行っていく．1 つずつ足してく方法だと計算量は O(N) になるが二分探索を使うと計算量は O(log2N) になり計算量の削減ができる．

また，設定ありの Pod よりも設定なしの Pod のほうが CPU とメモリの使用量が低い結果になったことについて取り上げる．設定ありの Pod では，Pod 作成時に Noisy Neighbors Pod と同様に新しい Pod 内のアプリケーションを安定して実行するのに必要な CPU とメモリ量を設定値として要求している．しかし，設定なしの Pod では，Noisy Neighbors Pod が CPU とメモリ量を際限なく占有するため，設定値が設定ありよりも小さくなった．これは，Noisy Neighbors Pod の必要とする CPU とメモリ量が多くなるほど，新しく作成する Pod の使用可能な CPU とメモリ量も少なくなり，最終的に新しく Pod を作成することができなくなる．Noisy Neighbors Pod のように際限なく CPU とメモリ量を使用する Pod が存在した場合，CPU とメモリの残量によって，設定値がないと Pod が作成できなくなると考える．

本研究では，CPU とメモリの limits はそれぞれのパラメータの影響が独立であるとした．しかし，アプリケーションによってパラメータが相互依存している場合がある．その場合，減少幅を 1%ではなく CPU の 1 (millicores) に対するメモリの値を算出する．例えば 10req/s から 20req/s の CPU とメモリの上昇値から増加量を計算し，その上昇量から減少させる値を算出する．そうすることにより実際の負荷による測定ではなく算出された関係性から値を決定する．

7. おわりに

本研究では, K8s の CPU とメモリの limits の設定値はユーザの経験則に依存するため, limits を決定する事が困難である課題を解決するために K8s 上でシステムを運用する際に処理するリクエスト数から Pod に必要な CPU とメモリの使用量を決定するまでを自動化する提案行った．コンテナ内で Stress-ng コマンドを実行し，CPU を 100%, メモリを 1500 (MiB) 使用する Noisy Neighbors Pod を作成しその際に HTTP リクエストを処理した場合の Pod からの応答時間，処理リクエスト数、CPU とメモリ使用量から CPU とメモリ制限の有無でどのような影響が出るかを評価を行った．結果として limits を設定した Pod は, HTTP リクエスト数が 40(req/s) の時，最大で 144 (ms) 削減でき，設定を行うことによる効果が大きいことがわかった．本提案によって，ユーザの経験則に依存せず limits の正しい設定値を自動的に算出可能である．これによって，K8s のシステム運用や自動的な管理に貢献できる．

謝辞

本研究は，JSPS 科研費 JP20K11776 の助成を受けたものである．

参考文献

[1] Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J.: Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade, Vol. 14, No. 1, p. 70–93 (online), DOI: 10.1145/2898442.2898444 (2016).

[2] Medel, V., Tolón, C., Arronategui, U., Tolosana-Calasanz, R., Bañares, J. Á. and Rana, O. F.: Client-Side Scheduling Based on Application Characterization on Kubernetes, *Economics of Grids, Clouds, Systems, and Services* (Pham, C., Altmann, J. and Bañares, J. Á., eds.), Cham, Springer International Publishing, pp. 162–176 (2017).

[3] Yue, J., Wu, X. and Xue, Y.: Microservice Aging and Rejuvenation, *2020 World Conference on Computing and Communication Technologies (WCCCT)*, pp. 1–5 (online), DOI: 10.1109/WCCCT49810.2020.9170005 (2020).

[4] Larsson, L., Tärneberg, W., Klein, C., Elmroth, E. and Kihl, M.: Impact of etcd deployment on Kubernetes, Istio, and application performance, *Software: Practice and Experience*, Vol. 50, No. 10, pp. 1986–2007 (online), DOI: https://doi.org/10.1002/spe.2885 (2020).

[5] Wu, Q., Yu, J., Lu, L., Qian, S. and Xue, G.: Dynamically Adjusting Scale of a Kubernetes Cluster under QoS Guarantee, *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 193–200 (online), DOI: 10.1109/ICPADS47876.2019.00037 (2019).

[6] Balla, D., Simon, C. and Maliosz, M.: Adaptive scaling of Kubernetes pods, *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5 (2020).

[7] Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H. and Kim, S.: Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration, *Sensors*, Vol. 20, No. 16 (online), DOI: 10.3390/s20164621 (2020).

[8] Chiba, T., Nakazawa, R., Horii, H., Suneja, S. and Seelam, S.: ConfAdvisor: A Performance-centric Configuration Tuning Framework for Containers on Kubernetes, *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 168–178 (online), DOI: 10.1109/IC2E.2019.00031 (2019).

[9] Pereira Ferreira, A. and Sinnott, R.: A Performance Evaluation of Containers Running on Managed Kubernetes Services, *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 199–208 (2019).

[10] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. and Merle, P.: Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER, *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pp. 472–479 (online), DOI: 10.1109/CLOUD.2017.67 (2017).

[11] Medel, V., Rana, O., Á. Bañares, J. and Arronategui, U.:

©2021 Information Processing Society of Japan

Adaptive Application Scheduling under Interference in Kubernetes, *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pp. 426–427 (2016).
[12] Hamzeh, H., Meacham, S. and Khan, K.: A New Approach to Calculate Resource Limits with Fairness in Kubernetes, *2019 First International Conference on Digital Data Processing (DDP)*, pp. 51–58 (online), DOI: 10.1109/DDP.2019.00020 (2019).
[13] Medel, V., Rana, O., Á. Bañares, J. and Arronategui, U.: Modelling Performance Resource Management in Kubernetes, *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, pp. 257–262 (2016).
[14] Chang, C., Yang, S., Yeh, E., Lin, P. and Jeng, J.: A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning, *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6 (2017).
[15] Lee, S., Son, S., Han, J. and Kim, J.: Refining Micro Services Placement over Multiple Kubernetes-orchestrated Clusters employing Resource Monitoring, *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1328–1332 (online), DOI: 10.1109/ICDCS47774.2020.00173 (2020).

©2021 Information Processing Society of Japan