Vol.2017-BIO-49 No.3
2017/3/23

コンテナ型仮想化による分散計算環境における タンパク質間相互作用予測システムの性能評価

青山 健人[1,2,a)]　山本 悠生[1,2]　大上 雅史[1,3]　秋山 泰[1,2,3]

概要：近年，軽量かつ性能に優れた仮想化技術としてコンテナ型仮想化が大規模な並列計算環境に導入されはじめている．様々なライブラリやソフトウェア群を併用する機会の多い生命情報解析分野において，ライブラリや実行ファイルを既存の環境から隔離し，即時実行可能な形式でソフトウェア共有を可能とするコンテナ型仮想化技術は，大きな影響を与えると予測される．本研究では，我々の研究室で開発したタンパク質間相互作用予測システム（MEGADOCK）について，クラウド環境上の仮想マシンにコンテナ型仮想化の実装の 1 つである Docker を用いた分散計算環境を構築し，並列計算性能を評価する．

キーワード：コンテナ型仮想化, クラウドコンピューティング, MPI, Docker, MEGADOCK, タンパク質間相互作用

Evaluation of Container Virtualized MEGADOCK System in Distributed Computing Environment

Kento Aoyama[1,2,a)]　Yuki Yamamoto[1,2]　Ohue Masahito[1,3]　Yutaka Akiyama[1,2,3]

Abstract: Container virtualization, is a lightweight and superior performance virtualization technology, has begun to be introduced into a large-scale parallel computing environment. In the bioinformatics field, where there are many opportunities to combine various libraries and binary files, container virtualization technology that isolates the existing software environment and enables rapidly software distribution in an immediate executable format is expected to have many effects. In this study, we have selected the Docker which is an implementation of container virtualization, constructed the distributed computing environment of a protein-protein interaction prediction system (MEGADOCK) in virtual machines on cloud computing environment, and have evaluated the results of parallel computing performance.

Keywords: Container Virtualization, Cloud Computing, MPI, Docker, MEGADOCK, Protein-Protein-Interaction

1. 導入

生命情報解析の分野では様々なソフトウェアが研究活動に利用されるが，依存するライブラリなどのソフトウェア環境の管理は研究上の課題である．近年では，ソフトウェア環境の複雑化に対する解決として，軽量かつ性能に優れた仮想化技術の 1 つであるコンテナ型仮想化技術 [1], [2] の導入が進んでいる．特にゲノム研究の分野で普及するパイプラインソフトウェアなどでは複数ソフトウェアを連結するため環境が複雑になりやすく，コンテナ型仮想化技術に

[1] 東京工業大学 情報理工学研究院 情報工学系
Department of Computer Science, School of Computing, Tokyo Institute of Technology
[2] 東京工業大学 博士課程教育リーディングプログラム情報生命博士教育院
Education Academy of Computational Life Sciences, Tokyo Institute of Technology
[3] 東京工業大学 科学技術創成研究院スマート創薬研究ユニット
Advanced Computational Drug Discovery Unit, Institute of Innovative Research, Tokyo Institute of Technology
[a)] aoyama@bi.c.titech.ac.jp

よる環境管理や分散処理の研究成果を含めた導入事例等が報告されている [3], [4].

コンテナ型仮想化では，依存するライブラリや実行バイナリを含めたソフトウェアの実行環境をコンテナとして隔離し，即座に実行可能な形式のソフトウェア配布を実現する [5], [6]．この特徴はソフトウェア環境の管理を容易にし，新規ソフトウェアの配布や導入を円滑に行うことを可能とする．また，コンテナ型仮想化は一般的な仮想マシンを実現するハイパーバイザ型仮想化よりも優れた性能を発揮すると言われており，適切な設定を施した場合には，物理マシン上の実行とほぼ同等の性能を発揮すると報告されている [7].

これまで，コンテナ型仮想化は迅速な環境構築とアプリケーションの抽象化が可能であることから，クラウド環境上の並列分散基盤における動的な負荷分散などの領域で発展してきた [8]．研究機関や大学内の計算機環境では仮想化による性能低下の懸念などの理由から導入は進んでいなかったが，並列計算環境上のベンチマーク結果や応用研究の事例報告による追い風があり，近年では大規模な並列計算機環境においてもコンテナ型仮想化技術が採用されはじめている．実例として，スーパーコンピュータの計算性能指標である Top500 上位の大規模並列計算機 Cori を有するバークレー研究所のエネルギー研究部門（NERSC）では，HPC 向けコンテナ型仮想化のオープンソースソフトウェア Shifter[9] を公開するほか，同研究所における様々な応用研究でのコンテナ型仮想化の利用報告がある [10]．国内においても，東京工業大学が 2017 年夏に稼働を予定する TSUBAME 3.0 ではコンテナ型仮想化技術が採用されると示唆されている．以上のことから，生命情報解析分野においてもコンテナ型仮想化への対応は急務である．

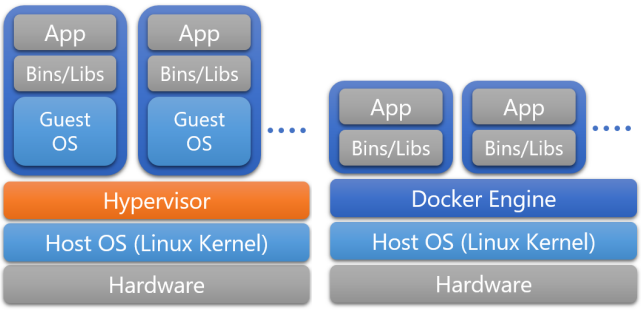
本報告では，Microsoft 社のクラウドコンピューティングサービス Microsoft Azure[11] 上に構築されたタンパク質間相互作用予測ソフトウェア（MEGADOCK-Azure[12]）システムに対して，コンテナ型仮想化の実装の 1 つである Docker[1] によるコンテナを用いた分散処理を導入し，システムの性能を評価する．

2. コンテナ型仮想化 (Docker) について

仮想化技術は大きく（1）ハードウェアレベルの仮想化（ハイパーバイザ型仮想化），および（2）OS レベルの仮想化（コンテナ型仮想化）の 2 つに分けることができる．

（1）ハイパーバイザ型仮想化

ハイパーバイザ型仮想化では，アプリケーションを管理する OS（スーパーバイザ）をさらに管理する上位のハイパーバイザによって仮想環境が提供される（**図 1**）．クラウド環境で一般的に普及する「仮想マシン」はハイパーバイザ型仮想化により実現されており，ハードウェア（またはホスト OS）上で稼働するハイパーバイザが別の OS（ゲスト OS）を管理することで，Windows，Linux OS など様々な OS をユーザは仮想マシン上で利用できる．



(a) ハイパーバイザ型仮想化　(b) コンテナ型仮想化（Docker）

図 1 仮想化技術の種類

Fig. 1 Types of Virtualization

ハイパーバイザ型仮想化には様々な実現方式が存在し，Linux Kernel の提供する仮想化機能による Kernel Virtual Machine (KVM)[13]，Microsoft Azure を支える Hyper-V[14]，Amazon Web Service を支える Citrix 社の XEN[15]，VMware[16] などが存在する．

（2）コンテナ型仮想化

コンテナ型仮想化では，OS 上で動作するプロセス（コンテナ）に対して仮想的に個別の名前空間を割り当てることで仮想環境が提供される（図 1）．コンテナ型仮想化は Linux カーネル機能の `namespace`[5] によって実現されている．`namespace` はグローバルな名前空間から個別のインスタンスを隔離して，内部で別のファイルシステム，プロセス，ユーザ，ネットワーク，ホスト名などを扱うことを可能とする．これにより，ユーザはホスト OS のファイルシステム等から隔離された環境でアプリケーション専用の実行環境を構築できる．コンテナ型仮想化では，各コンテナはホスト OS カーネル（スーパーバイザ）を共有するため，カーネル共有型の仮想化とも呼ばれる．

各コンテナのプロセスは隔離されて見えるが，それらのリソース管理はホスト OS の Linux の `cgroups` による制御下に置かれており，仮想化による性能への影響は小さい [7]．また，ハイパーバイザ型仮想化とは異なり仮想イメージに OS 機能のすべてを含む必要がないため，仮想イメージの容量を小さく抑えられる．Linux 上でコンテナを管理してコンテナ型仮想化を実現するツールとしては，Docker[1]，LXC[2] などがある．

2.1 Docker

Docker は Linux 上のコンテナの管理ツール，およびプラットフォームの総称である [1]．Go 言語で記述されており，GitHub 上でオープンソースソフトウェアとして活発に開発されている [17]．Docker で扱うコンテナ（以下，

Docker コンテナ）を管理する様々なツール・サービスが活発に開発されており，コンテナの実行・作成・管理などを担う Docker Engine，コンテナを共有するサービスである Docker Hub[6]（Docker Registry），複数コンテナ管理を担う Docker Compose[18]，クラスタ上のコンテナ管理を担う Docker Swarm[19] などが存在する．また，ハイパーバイザ型仮想化と併用して Windows，Mac OS 上で Docker を利用可能な Docker for Windows (Mac) も提供されている．

2.1.1 Docker イメージの共有

ユーザはコンテナに構築したソフトウェアの実行環境を Docker イメージとして出力して，別の環境上で同じように展開できる．また，DockerHub 等のレジストリサービスを利用して，作成したアプリケーションの実行環境を含むイメージを他人と共有したり，他人が作成したイメージをダウンロードできる．イメージにはアプリケーションに必要なライブラリやバイナリを格納できるため，取得したイメージを元にビルド等の必要なくアプリケーションを実行できる（図 **2**）．

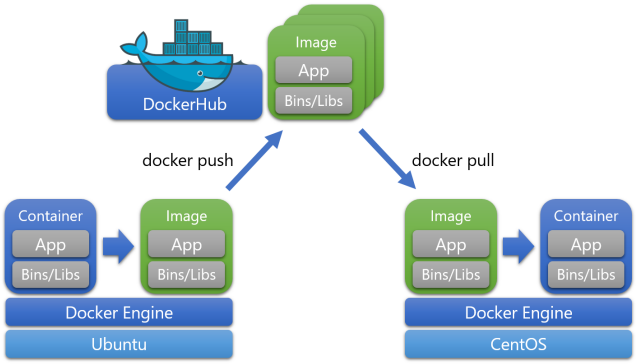


図 **2** DockerHub による Docker イメージの共有
Fig. 2 Sharing Docker Image via DockerHub

2.1.2 Docker コンテナのファイルシステム

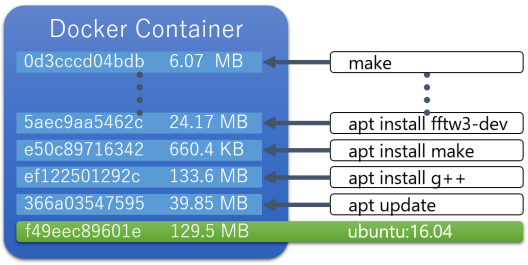


図 **3** Docker コンテナおよび Docker イメージのファイルシステム
Fig. 3 Filesystem of Docker Container and Docker Image

Docker では Docker イメージのファイルシステムに積層型の特徴を持つ AUFS（Another Union Filesystem)[20] を採用している（図 **3**）．AUFS はファイルの操作などで変化があると既存のファイルシステムとの差分を計算し，ハッシュ値を付けた新たな層を積み重ねていく．この特徴により，Docker では元のイメージからの操作履歴の閲覧，ロールバック，さらに同じ構造の層の再利用によるイメージ容量の削減を実現している．

なお，AUFS は大量の I/O には適さないため，データ処理の際には Docker の `volume` オプションを利用してコンテナ内にホスト OS のファイルパスをマウントして，データの入出力を行うことが多い．

3. MEGADOCK

MEGADOCK[21] は Ohue らにより開発された，大規模並列計算環境を想定したタンパク質ドッキングに基づくタンパク質間相互作用予測ソフトウェアである．MPI や OpenMP による並列化，GPU を利用した高速化に対応しており，東京工業大学のスーパーコンピュータ「TSUBAME 2.5」，理化学研究所のスーパーコンピュータ「京」における大規模並列計算の実績がある．近年では，より汎用な環境におけるソフトウェアの利用と普及を目的に，商用クラウド環境である Microsoft Azure 上の仮想マシンを利用した大規模並列化が可能な MEGADOCK-Azure[22]，Web アプリケーションとして利用可能な MEGADOCK-WEB[23] についても報告している．MEGADOCK-Azure を用いた Microsoft Azure[11] 上の仮想マシンによる並列処理基盤のシステムの構成を図 **4** に示す．

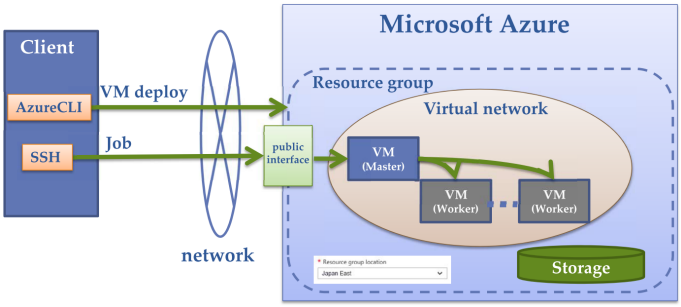


図 **4** MEGADOCK-Azure のシステム構成の概略図 [22]
Fig. 4 System Architecture of MEGADOCK-Azure [22]

一方で，ソフトウェアの環境構築の汎用性および性能の向上は依然として課題である．MEGADOCK-Azure による商用クラウド環境の利用は解決策の 1 つであるが，既存の計算機環境とソフトウェア環境を共用できない問題，ベンダーのロックイン問題，ハイパーバイザ型仮想化による性能低下などの懸念が残る．

本研究では，MEGADOCK に対してコンテナ型仮想化の実装の 1 つである Docker を利用して，これらの課題の解決を試みる．これには以下の利点がある．

- 他の商用クラウド環境や新規の計算機環境でも同一のコンテナイメージを利用して同一のソフトウェア環境を短時間で構築可能である
- 物理マシン上で実行した場合には，ハイパーバイザ型

仮想化に対する性能面の優位性がある

- 東京工業大学が予定する TSUBAME 3.0 でコンテナ型仮想化の導入が示唆されており，今後の大規模並列計算機上の実行性能のモデルケースとしての役割が期待できる

4. 評価実験

本報告では，以下の 2 つの実験を実施する．

(1) Microsoft Azure 上の MEGADOCK-Azure で構築した仮想マシン上で，MEGADOCK の MPI 利用が可能な Docker コンテナを実行して，仮想マシン数を増加させた際の並列実行性能を計測する

(2) ローカルな物理マシン上で，MEGADOCK の GPU 利用が可能な Docker コンテナを実行して，単一ノード上の実行性能を計測する

4.1 実験 1: クラウド環境上の並列実行性能

実験 1 では，MEGADOCK-Azure で構築した Microsoft Azure 上の仮想マシンを用いて，以下の 2 条件における並列実行性能を比較する．

(a) 仮想マシン間の MPI 通信を用いた MEGADOCK の並列実行性能

(b) 仮想マシン上で MPI 通信が可能な Docker コンテナを実行し，コンテナ間の MPI 通信を用いた MEGADOCK の並列実行性能

4.1.1 仮想インスタンス環境，ソフトウェア環境

Microsoft Azure の仮想マシンのインスタンスとして `Standard_D14_v2` を計測に利用した．仮想インスタンスの環境情報を**表 1** に，ソフトウェア環境を**表 2** に示す．

表 1 実験 1: `Standard_D14_v2` の仮想インスタンス仕様
Table 1 Experiment I : Specification of `Standard_D14_v2`

CPU	Intel Xeon E5-2673, 2.40 [GHz] × 16 [core]
Memory	112 [GB]
Local SSD	800 [GB]

表 2 実験 1: ソフトウェア環境
Table 2 Experiment I : Software Environment

	Virtual Machine	Docker
OS (image)	SUSE Linux Enterprise Server 12	ubuntu:14.04
Linux Kernel	3.12.43	3.12.43
GCC	4.8.3	4.8.4
FFTW	3.3.4	3.3.5
OpenMPI	1.10.2	1.6.5
Docker Engine	1.12.6	N/A

4.1.2 実験条件

実行時間の計測には，MPI プログラムの実行開始から終了までを `time` コマンドにより計測し，計測を 3 回繰り返した中から中央値を選択する．

計測時には，データの転送による影響を回避するため，各ノードにおける MEGADOCK の出力結果は各ノード内のファイルシステムに出力する．Docker コンテナ実行時も同様に，`Volume` オプションを指定して仮想マシンのファイルシステムをコンテナ内にマウントする．これは同時に AUFS によるファイル I/O の性能低下を回避する意図を含む．

仮想マシン上の Docker コンテナは Docker Swarm[19] により，コンテナ間ネットワーク機能を用いた仮想オーバーレイネットワークを構築しており，相互に MPI 通信が可能である（**図 5**）．仮想オーバーレイネットワークで相互接続された Docker コンテナ内で複数の MPI プロセスを起動させて，複数ノードによる並列処理を行う．

コンテナ型仮想化の実行性能を比較するため，1 つの仮想マシンの中では 1 つの Docker コンテナのみが実行されるようにコンテナ数をノード数と同数に調整する．また，Docker コンテナ 1 つに対して 4 つの MPI プロセス数が割り当たるよう合計の MPI プロセス数は決定される．OpenMP による並列化のスレッド数は `OMP_NUM_THREADS=4` で固定される．これらの MPI プロセス数と OpenMP のスレッド数の指定は，仮想マシン上の比較実験 (a) においてもノードあたり同様に設定される．

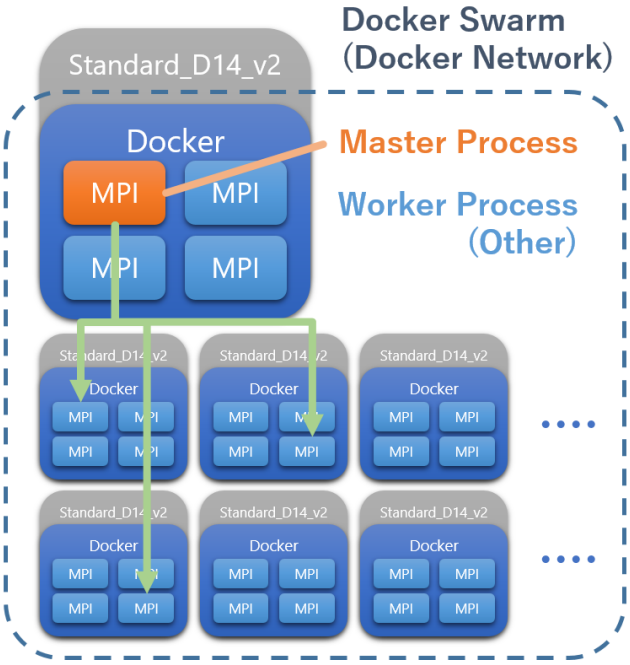


図 5 Docker Swarm によるコンテナネットワーク構成の概略図
Fig. 5 Overview of Container Network using Docker Swarm

4.1.3 データセット

ProteProtein-Protein Docking Benchmark version 1.0 [24] の複合体構造データセットを用いる．データセットには二量体の複合体構造が 59 個含まれており，二量体の各構成タンパク質がレセプターとリガンドとして区別されている．本研究では，レセプターとリガンドのすべての組み合わせとして $59 \times 59 = 3,481$ 通りのドッキング計算を行う．

4.1.4 実験結果

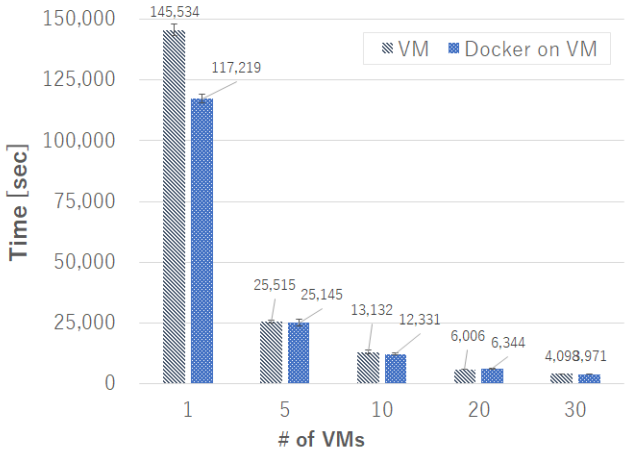


図 6　実験 1: 仮想マシン数と実行時間
Fig. 6　Experiment I : Execution Time

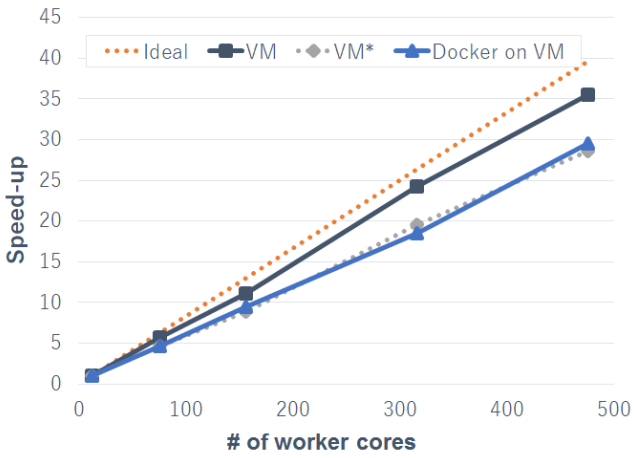


図 7　実験 1: ZDOCK ベンチマークの並列実行性能
Fig. 7　Experiment I : HPC Performance of ZDOCK Benchmark

データセットに対して仮想マシン数（コア数）を増加させたときの実行時間，および並列実行性能を図 **6**，図 **7** に示す．図 6 では，縦軸は各実行時間，横軸は仮想マシンの台数，エラーバーは測定値の標準偏差である．図 7 では，縦軸は仮想マシン 1 台のときの実行時間に対する高速化の割合，横軸はドッキング計算を実行するコア数である．青色の直線 (Ideal) はコア数に対して線形に高速化された場合の並列実行性能を示している．

なお，図 7 の VM* は，VM 1 台のときに Docker (VM) と同じ実行時間を仮定した場合の並列実行性能である．これは VM 1 台の実行時間が過去の計測時 [12] と比較して増大しており，高速化率の値が極端に向上していたために導入した．

図 7 より，コア数の増加にともない処理が高速化されており，VM 30 台（コア数 476）のときに VM では 35.5 倍，VM* では 28.6 倍，Docker (VM) では 29.5 倍の処理の高速化が達成されている．VM* と Docker (VM) の高速化率に大きな差はなく，計測箇所によっては VM* の実行性能を上回る箇所もあった．

Docker (VM) による実行性能は，ホスト OS によるプロセスの制御が通常の処理と極めて近いことから，VM* の実行性能とほぼ同等の性能を示した．MEGADOCK の処理は極めて計算律速の処理であり，ファイル I/O やネットワーク律速ではないため，コンテナ型仮想化による性能低下が小さかったと考えられる．今回のクラウド上の仮想マシン環境条件と試行回数では，VM* の実行性能と Docker (VM) の実行性能には大きな差は見られなかった．

4.2 実験 2: コンテナ型仮想化による性能低下の計測

実験 2 では，Docker によるコンテナ型仮想化のオーバーヘッドを計測するため，GPU を搭載したローカル環境の物理マシン上で，単一ノードにおける実行性能を計測する．

4.2.1 システム構成

実験 2 では，以下の 4 条件で MEGADOCK による計算の実行時間を比較する．イメージ図を図 **8** に示す．

(a) 物理マシン上で MEGADOCK（MPI 版）並列実行する．
(b) 物理マシン上の Docker コンテナ内で MEGADOCK（MPI 版）を実行する．
(c) 物理マシン上で MEGADOCK（GPU 版）を実行する．
(d) 物理マシン上で NVIDIA Docker を用いて，Docker コンテナ内で MEGADOCK（GPU 版）を実行する．

4.2.2 計算機環境，ソフトウェア環境

実験 2 で用いた物理マシンの環境情報を表 **3** に，ソフトウェア環境を表 **4** に示す．

表 3　実験 2: 物理マシンの計算機仕様
Table 3　Experiment II : Specification of Physical Machine

CPU	Intel Xeon E5-1630, 3.7 [GHz] × 8 [core]
Memory	32 [GB]
Local SSD	128 [GB]
GPU	NVIDIA Tesla K40

4.2.3 実験条件

Docker コンテナ内からの GPU デバイスの利用には，NVIDIA 社の NVIDIA Docker[25] を用いる．

表 4　実験 2: ソフトウェア環境
Table 4　Experiment II : Software Environment

	Physical Machine	Docker	NVIDIA Docker (GPU)
OS (image)	CentOS 7.2.1511	ubuntu:14.04	nvidia/cuda:8.0-devel
Linux Kernel	3.10.0	3.10.0	3.10.0
GCC	4.8.5	4.8.4	4.8.4
FFTW	3.3.5	3.3.5	3.3.5
OpenMPI	1.10.0	1.6.5	N/A
Docker Engine	1.12.3	N/A	N/A
NVCC	8.0.44	N/A	8.0.44
NVIDIA Docker	1.0.0 rc.3	N/A	N/A
NVIDIA Driver	367.48	N/A	367.48

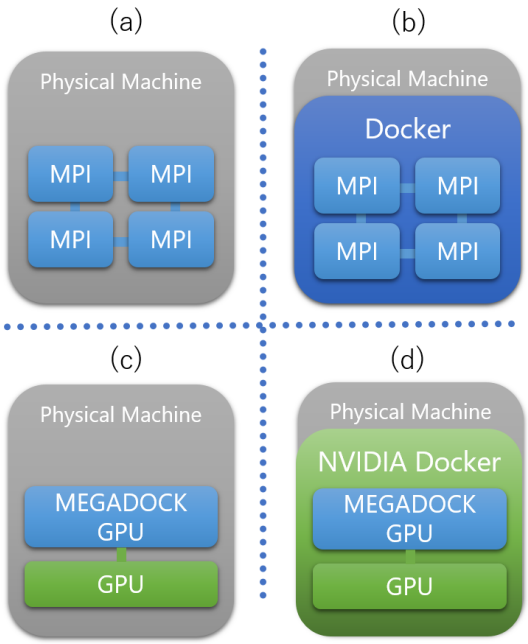


図 8　実験 2：概要
Fig. 8　Overview of Experiment II

実行時間の計測には，MPI プログラムの実行開始から終了までを `time` コマンドにより計測し，計測を 6 回繰り返した中から中央値を選択する．物理マシン，Docker コンテナともに GPU 版単体は MPI 版のタスク分散機能に未対応のものを利用したため，同等の処理を実行するスクリプトを作成して，すべての実行が終了するまでの時間を計測した．

MEGADOCK の出力結果は，同一ネットワーク上の NAS (Network Attached Storage) に出力する．Docker コンテナの実行時も `Volume` オプションを指定して，物理マシン実行時と同一のネットワーク上の NAS に出力する．

Docker コンテナを利用した計測のとき，物理マシン上では 1 つの Docker コンテナのみが同時に実行される．MPI による並列実行の場合は，Docker コンテナ 1 つに対して 4 つの MPI プロセス数が稼働する．OpenMP による並列化のスレッド数は `OMP_NUM_THREADS=4` で固定される．MPI プロセス数と OpenMP のスレッド数の指定は，Docker コンテナを利用しない比較実験 (a)(c) においても共通で設定される．

4.2.4　データセット

KEGG pathway[26] から取得した pdb データを無作為にレセプターとリガンドとして選択して，合計 100 ペアの組み合わせのドッキング計算を行う．

4.2.5　実験結果

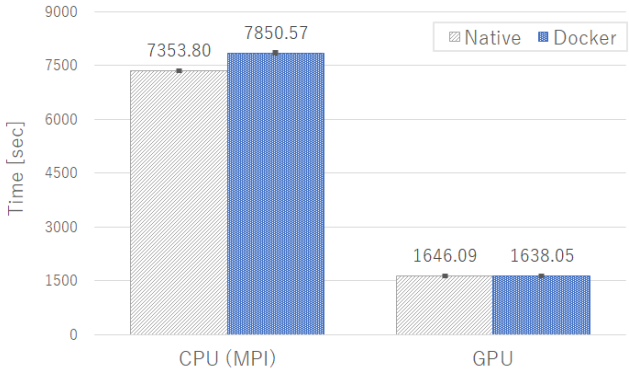


図 9　実験 2: 実行時間の比較
Fig. 9　Experiment II : Comparison of Execution Time

表 5　実験 2: 実行時間
Table 5　Experiment II : Execution Time

Test case	Time [sec]	Std.
(a) Native CPU (MPI)	7353.50	1.72
(b) Docker CPU (MPI)	7848.88	6.112
(c) Native GPU	1646.03	0.814
(d) Docker GPU	1638.16	0.817

図 9 および**表 5** に，データセットに対して物理マシン上で MEGADOCK の計算を実行したときのドッキング計算に所要した実行時間を示す．MPI 並列による MEGADOCK のドッキング計算では，Docker コンテナを利用した実行時間は物理マシン上の実行時間から約 6.32 %の増加がみられた．今回の実験は単一ノード上の計測であり，MPI 並列時

の実行時間には通信時間の影響は少なく，ファイル I/O に関しても Docker コンテナ実行時には `Volume` オプションを指定しており，影響は少ないと考えられる．一方で，GPU による MEGADOCK のドッキング計算では，Docker コンテナを利用した場合は物理マシン上の実行時間と比較して，ほぼ同等の実行時間を示した．GPU 版ではドッキング計算をほぼ GPU 上で処理するため，Docker コンテナ内からプログラムを実行していた場合でも，実行時間に与える影響が小さいためと考えられる．

5. 考察

実験 1 では VM 1 台のときの仮想マシン上の実行時間が極端に増大しており，極端に良好な並列実行性能を示していた．特に台数が少ない場合にはクラウド環境上の影響を受けやすいと考えられるため，回数を重ねることでより正確な時間を測定する必要がある．

今回の報告では，MEGADOCK の GPU+MPI 並列版の計測を行わなかった．これは本報告で Docker コンテナからの GPU 利用に使用した NVIDIA Docker が，Docker コンテナのクラスタリングに使用した Docker Swarm に未対応 [27] のためである．Microsoft Azure 上では 2016 年冬期から GPU 搭載の N シリーズの一般利用が解放されており，大上ら [22] が既に GPU インスタンスを利用した計測を報告している．Docker コンテナによる MEGADOCK GPU+MPI 並列版の対応は，今後の課題である．

なお，Microsoft Azure の HPC 向けの仮想インスタンス H シリーズ・A シリーズでは，MPI アプリケーション用に InfiniBand で相互接続されたクラスタ間での RDMA (Remote Direct Memory Access) をサポートしており，これらの HPC 向け仮想インスタンス上の MEGADOCK-Azure との性能比較も必要である．

今回は従来版の MEGADOCK との比較のため複数ノード上の実行には MPI 並列化を用いているが，MEGADOCK における MPI 通信は Master-Worker モデルによるタスクの分配のみであり，別のフレームワークによる代替が可能である．現状の実装では MPI プロセスの耐障害性への考慮が少ないため，今後はコンテナを実行単位として，動的な実行規模の拡大／縮小，障害時のコンテナの再起動や冗長性のある実行が可能なフレームワークに移行すべきである．Kubernetes[28] や Apache Mesosphere[29] 等のコンテナ間の協調を前提としたフレームワークを導入することは今後の課題である．

6. 結論

本報告では，商用クラウド環境である Microsoft Azure の仮想マシン上でコンテナ型仮想化の実装の 1 つである Docker を利用して，MEGADOCK によるタンパク質ドッキングの MPI 並列実行時の並列実行性能を評価した．仮想マシン上で直接 MEGADOCK を計算した場合と，仮想マシン上で MEGADOCK の Docker コンテナを利用して計算した場合の両方で並列コア数の増加にともなう高速化が達成されており，ほぼ同等の並列実行性能を発揮した．

また，GPU 搭載の物理マシン上でコンテナ型仮想化の実装の 1 つである Docker を利用したときの，単一ノードにおける MEGADOCK によるタンパク質ドッキングの計算時間を評価した．CPU コアのみで MPI 並列実行した場合には，物理マシン上の素の実行時間に対して Docker コンテナを利用した実行時間は約 6.32 %の増加が見られた．一方，GPU を利用した場合には，両者の実行時間に大きな差は見られなかった．

コンテナ型仮想化の利用によってアプリケーションの実行環境と既存の環境を隔離して扱うことが可能であり，研究成果ソフトウェアの配布や導入に効率的である．商用クラウド分野の基盤技術として主に普及してきたが，大規模並列計算機においてもコンテナ型仮想化の導入がはじまっている．コンテナ型仮想化によるソフトウェア実行環境の仮想化は，研究の生産性と再現性を向上させる上で大きな意味を持つと考えられる．

謝辞 本研究の一部は，文部科学省 博士課程教育リーディングプログラム 東京工業大学「情報生命博士教育院」，JSPS 科研費 基盤研究（A）(24240044)，JST CREST「EBD: 次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，および JST 研究成果展開事業世界に誇る地域発研究開発・実証拠点（リサーチコンプレックス）推進プログラム「世界に誇る社会システムと技術の革新で新産業を創る Wellbeing Research Campus “Tonomachi”」，Microsoft Business Investment Funding，リバネス研究費の支援を受けて行われた．

参考文献

[1] “Docker.” https://www.docker.com/.
[2] G. Stphane, “LXC - Linux containers.” https://linuxcontainers.org/.
[3] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame, “The impact of Docker containers on the performance of genomic pipelines,” *PeerJ*, vol. 3, p. e1273, 2015.
[4] A. Paolo, D. Tommaso, A. B. Ramirez, E. Palumbo, C. Notredame, and D. Gruber, “Benchmark Report : Univa Grid Engine , Nextflow , and Docker for running Genomic Analysis Workflows,”
[5] E. W. Biederman, “Multiple Instances of the Global Linux Namespaces,” *Proceedings of the 2006 Ottawa Linux Symposium*, vol. 1, pp. 101–112, 2006.
[6] “DockerHub.” https://hub.docker.com/.
[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” *Technology*, vol. 25482, pp. 171–172, 2015.
[8] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, “Dy-

namically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.
[9] W. Bhimji, S. Canon, D. Jacobsen, L. Gerhardt, M. Mustafa, and J. Porter, "Shifter : Containers for HPC," pp. 1–12, 2016.
[10] B. Debbie, "Using containers and supercomputers to solve the mysteries of the Universe," dockercon16, 2016.
[11] "Microsoft Azure." https://azure.microsoft.com/.
[12] M. Ohue, Y. Yamamoto, H. Sato, T. Matsushita, and Y. Akiyama, "MEGADOCK-Azure: High-performance protein-protein interaction prediction system on Microsoft Azure HPC," *Informatics in Biology, Medicine and Pharmacology 2016 (IIBMP2016)*, p. P52, 2016.
[13] A. Kivity, U. Lublin, A. Liguori, Y. Kamay, and D. Laor, "kvm: the Linux virtual machine monitor," *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.
[14] A. Velte and T. Velte, "Microsoft Virtualization with Hyper-V," 2010.
[15] "Xen Project." http://www.xen.org/, 2002.
[16] "Virtualization Overview." http://www.vmware.com/pdf/virtualization.pdf.
[17] "DockerEngine." https://github.com/docker/docker.
[18] "Docker Compose." https://docs.docker.com/compose/, 2016.
[19] "Swarm mode overview." https://docs.docker.com/engine/swarm/, 2016.
[20] "Advanced multi layered unification filesystem." http://aufs.sourceforge.net/, 2014.
[21] M. Ohue, T. Shimoda, S. Suzuki, Y. Matsuzaki, T. Ishida, and Y. Akiyama, "MEGADOCK 4.0: An ultra-high-performance protein-protein docking software for heterogeneous supercomputers," *Bioinformatics*, vol. 30, no. 22, pp. 3281–3283, 2014.
[22] 大上雅史, 山本悠生, and 秋山泰, "Microsoft Azure 上でのタンパク質間相互作用予測システムの並列計算と性能評価," **情報処理学会研究報告 バイオ情報学（*BIO*）**, *2017-BIO-49(4)*, pp. 1–3, 2017.
[23] 長澤一輝, 松崎由理, 大上雅史, and 秋山泰, "タンパク質間相互作用予測結果データベース及び表示系の構築," **情報処理学会研究報告 バイオ情報学（*BIO*）**, *2016-BIO-45(2)*, pp. 1–4, 2016.
[24] R. Chen, J. Mintseris, J. Janin, and Z. Weng, "A protein-protein docking benchmark," *Proteins: Structure, Function and Genetics*, vol. 52, no. 1, pp. 88–91, 2003.
[25] "NVIDIA - nvidia-docker." https://github.com/NVIDIA/nvidia-docker, 2016.
[26] E. V. Wasmuth and C. D. Lima, "KEGG: new perspectives on genomes, pathways, diseases and drugs," *Nucleic Acids Research*, vol. 45, no. November 2016, pp. 1–15, 2016.
[27] "Support for swarm mode in Docker 1.12 #14." https://github.com/NVIDIA/nvidia-docker/issues/141, 2016.
[28] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *acmqueue*, vol. 14, no. 1, p. 24, 2016.
[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pp. 295–308, 2011.