Vol.2017-OS-140 No.17
2017/5/17

可視化ミドルウェアのスイッチングモジュールの設計と実装

石田 和也[1,a)]　木戸 善之[2,b)]　伊達 進[2,c)]　下條 真司[2,d)]

概要：Tiled Display Wall (TDW) を制御するために用いられる可視化ミドルウェアは，ハードウェアアクセラレーション機能を用いることで高速化を図っているミドルウェアが多く，システムライブラリ等の特定のバージョンに依存関係を持っている場合がある．そのため，複数の可視化ミドルウェアを TDW に導入する場合，ライブラリのバージョン依存性から来る競合が発生し，導入および共存が困難になる．本研究では，コンテナ型仮想化によって可視化ミドルウェア環境を分離し，必要に応じて切り替える（スイッチングする）ことで，複数の可視化ミドルウェアの共存を可能にする手法を提案する．また評価としては，提案手法による可視化ミドルウェア環境のスイッチング時間と，コンテナ型仮想化によるオーバーヘッドを計測し，提案手法の実用性について検討する．

キーワード：Tiled Display Wall，可視化ミドルウェア，スイッチング，Docker，SAGE2，ParaView

1. 背景

仮想実験，いわゆるシミュレーションによって科学的知見を得るためには，結果を可視化することが重要である．「京」や Oakforest-PACS といった様々な超並列計算機やスーパーコンピュータが普及し，運用されるようになったことで，スーパーコンピュータを用いたシミュレーションが少人数や個人での研究においても利用できるようになりつつある．こうしたシミュレーションの結果は膨大な分量の数値データとして得られるため，高解像度の可視化装置を用いて可視化し，実験結果を目視で確認することが必要とされる．例えば，建築物や重要文化財の構造健全性評価はシミュレーションによって行われるが，シミュレーション結果を可視化することで地震発生時における構造物限界性能や破断後の構造物の変形を確認でき，構造物の補強すべき箇所を特定することが可能となる [1]．また，化合物とタンパク質との複合体の結合は，未知の化合物とタンパク質との分子動力学シミュレーションによって複合体の可能性を知ることができるが，シミュレーション結果を可視化することでタンパク質の変形や結合部位の詳細を把握でき，実際に薬物候補になりえるかを検討することが可能となる [2]．

4K や 8K の解像度を持つディスプレイが市販される昨今，大規模可視化装置もまた重要な意味を持っている．大規模可視化装置の 1 つである Tiled Display Wall (TDW) [3] は，モニタを複数連動させることによって，仮想的に 1 枚のディスプレイを構成する技術である．TDW は，複数のモニタをタイル状に並べることにより，市販のディスプレイよりも巨大なディスプレイを構築することができる．人の高さよりも大きく壁一面に配置された TDW は，高解像度な可視化コンテンツを人の視野を占有するほど大きく表示することができ，かつそれを複数人で視聴することを可能にする．すなわち TDW は，研究グループにおいて同じ可視化コンテンツを視聴しながら複数人で議論することを可能にする技術であると言える．

こうした大規模な TDW を個人もしくはグループ単位で購入・所有することは，設置場所やコスト面において簡単とは言い難い．そこで，大学や研究所などの公共機関が保有する大型計算機センターに大規模可視化装置を設置し，それを研究者間で共有利用することがある．大阪大学サイバーメディアセンターでは大規模可視化装置として TDW を設置し，可視化サービスとして運用している [4]．可視化サービスでは，TDW およびその附置システムにインストールされている可視化アプリケーションあるいはミドルウェアに対応した形式の可視化コンテンツをユーザが持参し，それを TDW に表示する．したがって，可視化サービスとして TDW を運用する場合，その TDW には多様な可

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[2] 大阪大学サイバーメディアセンター
Cybermedia Center, Osaka University
[a)] ishida.kazuya@ais.cmc.osaka-u.ac.jp
[b)] kido@ais.cmc.osaka-u.ac.jp
[c)] date@ais.cmc.osaka-u.ac.jp
[d)] shimojo@ais.cmc.osaka-u.ac.jp

視化アプリケーション及びミドルウェアをインストールすることが求められる．

TDW を制御するために用いられる可視化ミドルウェアには，Scalable Amplified Group Environment (SAGE2) [5], ParaView [6], COllaborative VIsualisation and Simulation Environment (COVISE) [7] など，多種多様なものが存在する．ユーザは，可視化の目的や扱うデータ形式に適した可視化ミドルウェアを選択して利用することを求めるが，異なる可視化ミドルウェア間では使用するライブラリの競合などが生じることがある．共有利用する TDW に複数の可視化ミドルウェアを導入し，運用する場合，こうした問題が運用を複雑にする．そこで本研究では，TDW を可視化サービスとして運用する場合を想定し，可視化ミドルウェア環境を仮想化技術によって分離する手法を提案する．

本論文の構成は以下の通りである．2 節で技術的背景と本研究の課題について概説し，3 節で提案手法の詳細とその実装について述べる．4 節でまとめと今後の課題について述べる．

2. 技術的背景と問題

この章では TDW のアーキテクチャを説明し，続けて可視化ミドルウェアの特徴について述べる．その後，TDW を共有利用するための運用における問題点を述べる．

2.1 TDW のアーキテクチャ

図 1 に，TDW のアーキテクチャを示す．TDW は，モニタ群と PC クラスタという 2 つの要素から構成される．モニタ群は，通常サイズのモニタをタイル状に複数配置することで構成される．PC クラスタは TDW のバックエンドシステムであり，モニタを 1 台以上接続したノード群から構成される．一般的に，TDW がモニタ群を連動させてディスプレイを構成する際には，PC クラスタ上で可視化ミドルウェアのプロセスをモニタ数と同じ数だけ連携駆動させる．これらのプロセスはいずれかのモニタと一対一対応し，それぞれが自身と対応するモニタが表示すべき領域に関するレンダリングと表示処理を行う．このようにして，可視化ミドルウェアは TDW を制御する役割を果たす．

2.2 本研究で利用する可視化ミドルウェア

2.2.1 SAGE2

Scalable Amplified Group Environment (SAGE2) [5] は，米国のイリノイ大学シカゴ校の Electronic Visualization Laboratory (EVL) が開発した，オープンソースの可視化ミドルウェアである．SAGE2 は Chromium ブラウザを用いたウェブベースでの動作が特徴であり，ネットワークを介してリモート可視化 (遠隔地のデータの可視化) を行うことができる．具体的には，遠隔地で動作する SAGE2 Server が，仮想デスクトップ上の SAGE2 アプリケーション (SAGE2 に対応した可視化アプリケーション) のウィンドウをストリーミング配信することで，リモート可視化を実現する．

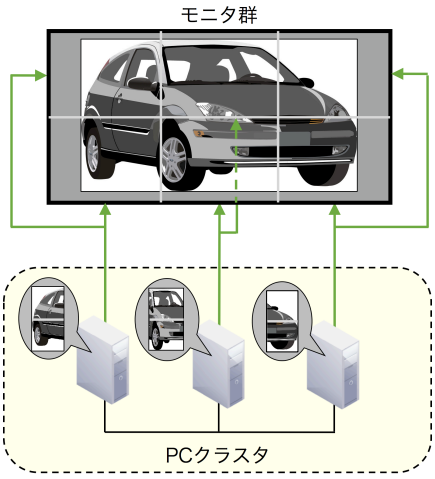

図 1 TDW のアーキテクチャ

SAGE2 のアーキテクチャを図 2 に示す．SAGE2 は SAGE2 Server，Display Client，Interaction Client の 3 つの要素から構成される．SAGE2 のアーキテクチャでは，TDW のモニタ群を構成する各モニタは，0 から始まる個別の番号 (Client ID) を持つ Display Client となる．SAGE2 Server は，表示するウィンドウの映像ストリームを WebSocket を用いて配信する．各 Display Client は，フルスクリーンモードの Chromium ブラウザで自身の Client ID に対応した URL を開くことにより，SAGE2 Server からの映像ストリームの受信と表示処理を行う．各ウィンドウの位置やサイズの変更は，Interaction Client 上で起動した SAGE UI を用いて行う．SAGE UI も SAGE2 Server が WebSocket を用いて配信しており，Chromium ブラウザで専用の URL を開くことで利用できる．

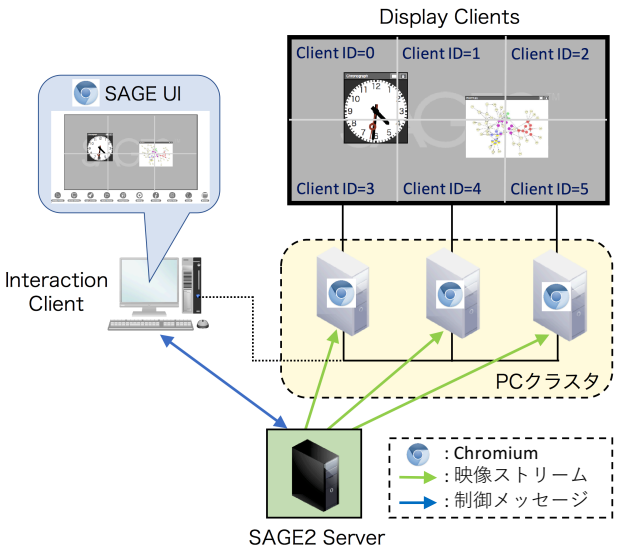

図 2 SAGE2 のアーキテクチャ

2.2.2 ParaView

ParaView [6] は，米国のサンディア国立研究所や Kitware 社等が開発した，オープンソースの可視化ミドルウェアである．ParaView は VTK や OpenFOAM，Plot3D といった多様な形式のデータを扱うことができる．また，可視化したデータを柔軟かつプログラマブルに加工するための Python スクリプティング機能を有している．

ParaView のアーキテクチャを図 3 に示す．ParaView はクライアント，サーバ，モニタ群の 3 つの要素から構成される．ParaView のアーキテクチャでは，モニタ群を構成する各モニタは，0 から始まる個別の番号 (ID) を持つ．また，PC クラスタはサーバとしての役割を果たす．サーバ上では，Message Passing Interface (MPI) を用いて pvserver (ParaView が提供する分散レンダリング用プログラム) をモニタ数と同じ数だけ連携駆動させる．各 pvserver は自身の持つランクと同じ値の ID を持つモニタと対応し，そのモニタが表示すべき領域に関するレンダリングと X Window System を用いた表示処理を行う．一方，クライアント上では，可視化されたデータを操作するための GUI が起動し，ランク 0 の pvserver との間に制御用コネクションを確立する．GUI 上でのユーザの操作内容は，制御用コネクションを介して送られた制御メッセージに基づき，各 pvserver が再度レンダリングと表示処理を行うことで反映される．

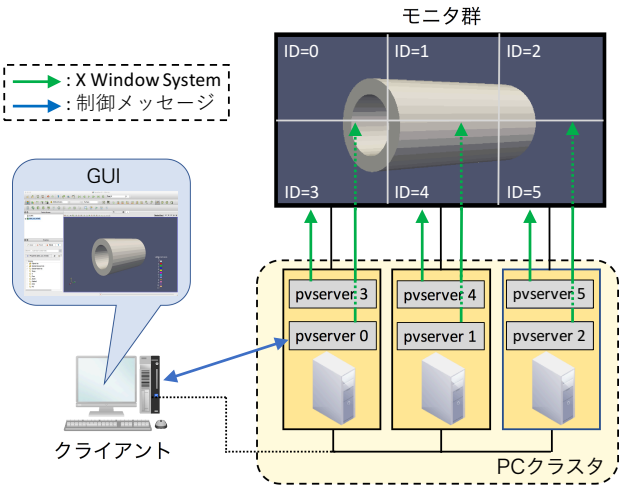

図 3 ParaView のアーキテクチャ

2.3 可視化ミドルウェア環境の分離の必要性

2.2 節で述べたように，可視化ミドルウェアは種類によってアーキテクチャが全く異なるが，使用するライブラリの種類については同様のものが多い．一方で，多くの可視化ミドルウェアはハードウェアアクセラレーション機能を用いることで高速化を図っているため，ライブラリの特定のバージョンに依存関係を持っている場合がある．例えば，表 1 に示した SAGE2・ParaView・COVISE が持つ依存関係のように，共通するライブラリを使用するが，そのバージョンが異なるような依存関係がある．表 1 はこれら 3 つの可視化ミドルウェアが持つ依存関係のごく一部であり，これらのミドルウェアを同一環境，同一計算機上に導入する場合は，ライブラリの競合毎に管理方法やインストールディレクトリを変更する必要がある．

可視化ミドルウェア	依存関係にあるライブラリ
SAGE2 (v2.0.0)	Node.js (v6.9 以上) FFmpeg (v3.0 以上) ImageMagick (v6.9 以上)
ParaView (v5.1.2)	VTK (v6.0 以上) Python (v2.7 以上) FFmpeg (v2.3 以上)
COVISE (v2016.12)	VTK (v6.0 以上) Python (v3.0 以上) OpenSceneGraph (v3.2 以上)

表 1 SAGE2・ParaView・COVISE と依存関係にあるライブラリ

可視化サービスのユーザは，利用する可視化ミドルウェアとして特定の種類もしくは特定のバージョンのミドルウェアを要求する場合がある．しかし，共有環境下で全種類・全バージョンの可視化ミドルウェアを導入・運用するのは困難である．また，可視化サービスのユーザは，これら特定の可視化ミドルウェアの他にユーザ独自の可視化ソフトウェアを作成している場合や，システムライブラリやグラフィックライブラリを独自カスタマイズしている場合などもあるため，可視化サービスのユーザの要求に対して，可視化環境は多様性を求められる．

3. 提案手法

本研究では，2.3 節で述べた可視化環境の構築を目指し，可視化ミドルウェア環境の仮想化技術による分離を提案する．仮想化技術には多様な方式があり，独立した仮想マシンにて自由に OS をインストールすることができるハイパーバイザ型，OS 部分をホストと共有するコンテナ型などがある．本研究では，実用可能性を探るための準備段階として，コンテナ型仮想化技術を用いた可視化ミドルウェア環境の分離，およびそのスイッチング機能の実装を試みる．

本章では，まず提案モジュールの構成技術であるコンテナ型仮想化技術 Docker について説明し，続けて提案モジュールのアーキテクチャを示す．その後，本研究で実装した提案モジュールについて概説する．

3.1 Docker

Docker [8] は，Docker 社によって開発されたコンテナ型仮想化技術である．Docker は，Docker イメージと呼ばれる仮想イメージを基にして，コンテナと呼ばれる仮想環境を構築することができる．コンテナは，ホスト OS 上に構成された論理的な区画であり，ホスト OS 上で動作する Docker Engine によって管理される．

Docker をはじめとするコンテナ型仮想化の特徴は，非常に軽量な仮想環境を構築できるという点である．ハイパーバイザ型仮想化で構築される仮想マシン (Virtual Machine: VM) とコンテナ型仮想化で構築されるコンテナのアーキテクチャの比較を，図 4 に示す．VM は，ホスト OS 上 (またはハードウェア上) で動作するハイパーバイザによって管理される．VM はホスト OS と完全に隔離された区画であるため，内部で独立にゲスト OS を起動し，その上にアプリケーションを起動する必要がある．一方，コンテナはカーネル部分をホスト OS と一部共有するため，内部でゲスト OS を起動することなくアプリケーションを起動できる．そのため，コンテナ内のアプリケーションは，VM に比べて極めて小さいオーバーヘッドで動作させることができる．また，コンテナは内部にゲスト OS を持たないため，起動や停止を VM よりも高速に行うことが可能である．

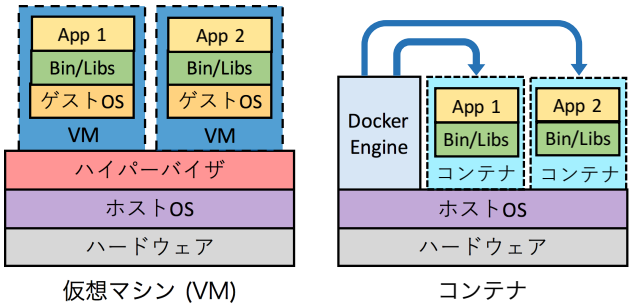

図 4 VM とコンテナの比較

また，Docker は Docker Swarm [9] と呼ばれる，複数のホスト上で起動するコンテナのオーケストレーション (一括操作) を行うための機能を有している．Docker Swarm のアーキテクチャを図 5 に示す．Docker Swarm を利用する際には，コンテナが起動する複数ホストをまとめて Docker Swarm クラスタを構成する．Docker Swarm クラスタに含まれる各ホストは，マネージャノードもしくはワーカノードとなる．マネージャノード上の Docker Engine は，コマンドラインで入力された Docker コマンドに応じて，コンテナ操作 (起動・停止・再起動など) の指示を Docker Swarm クラスタの全ノードに送る．ワーカノードと指示を送ったマネージャノード上の Docker Engine は，この指示に従って必要なコンテナ操作を行う．

3.2 提案モジュールのアーキテクチャ

提案モジュールのアーキテクチャを図 6 に示す．提案モジュールは Web UI，制御サーバ，Docker Swarm クラスタの 3 つの要素によって構成される．

Web UI はワンタッチ操作が可能な提案モジュールの UI であり，タブレット PC 上の Web ブラウザ上で利用される．Web UI 上には，可視化ミドルウェア環境をスイッチングする際に利用するボタンが用意されている．Web UI

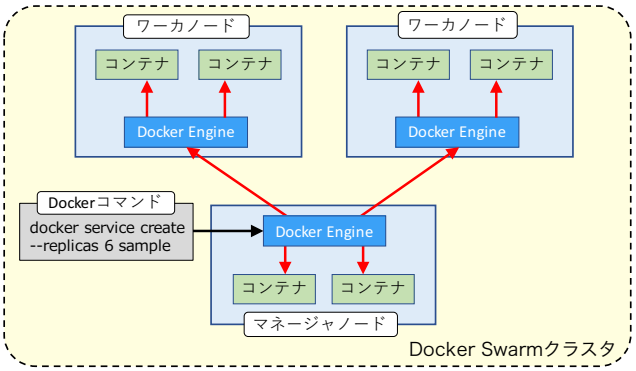

図 5 Docker Swarm のアーキテクチャ

上のいずれかのボタンをタッチすると，そのボタンに対応するパラメータ (ボタン名を表す文字列) を含む GET リクエストが制御サーバに送信される．

制御サーバは Web UI の配信とスイッチングの制御を行う Web サーバであり，Docker Swarm クラスタのマネージャノード上で起動する．制御サーバは Web UI からの GET リクエストを受信すると，このリクエストに含まれるパラメータの内容に対応するスイッチング用スクリプトを選択・実行する．各スイッチング用スクリプトは Docker Swarm クラスタ上のコンテナの停止・起動やコンテナ内での初期設定を行うための Docker コマンド群を含んでおり，実行することで可視化ミドルウェア環境のスイッチングに必要な一連の処理が行われる．

Docker Swarm クラスタは，可視化ミドルウェア環境のコンテナが起動するクラスタで，TDW の PC クラスタ全体から構成される．マネージャノード上の Docker Engine は，スイッチング用スクリプト内で実行された Docker コマンドに従って Docker Swarm クラスタ上のコンテナ全体のオーケストレーションを行う．一方，ワーカノード上の Docker Engine は，マネージャノードからの指示に従ってコンテナの停止と起動を行う．

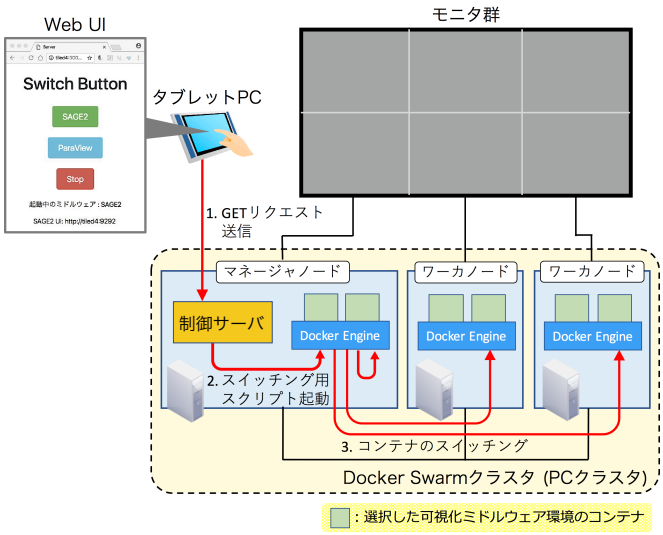

図 6 提案モジュールのアーキテクチャ

3.3 提案モジュールの実装

本研究では，TDW 上で SAGE2 の実行環境が起動している状態 (SAGE2 モード) ・ParaView の実行環境が起動している状態 (ParaView モード) ・両者ともに停止している状態 (Stop モード) という 3 状態間でのスイッチングが可能な提案モジュールを実装した．本節では，実装の概略を説明する．

Web UI 上には，SAGE2 ボタン・ParaView ボタン・Stop ボタンの 3 種類のボタンを実装した．これらのボタンは，それぞれ TDW の状態を SAGE2 モード・ParaView モード・Stop モードにスイッチングするためのものである．3.2 節で述べたように，Web UI 上のいずれかのボタンがタッチされると，そのボタンに対応するパラメータを含む GET リクエストが制御サーバに向けて送信される．制御サーバがこの GET リクエストを受信すると，このリクエストに含まれるパラメータに対応するスイッチング用スクリプトが実行される．このスイッチング用スクリプトとして，SAGE2 モード起動スクリプト・ParaView モード起動スクリプト・Stop モード起動スクリプトという 3 種類のシェルスクリプトを実装した．以下では，各スイッチング用スクリプトによって行われる処理の大まかな流れを説明する．なお，本実装では，SAGE2 モードで利用する Chromium ブラウザは，コンテナ内ではなくホスト OS 上で動作させるものとした．また，ParaView モードでは，各コンテナ内で起動する pvserver の数は 1 プロセスとした．

- SAGE2 モード起動スクリプト
  (1) TDW の現状態が ParaView モードである場合，Docker Swarm クラスタ上で起動中のすべての pvserver のコンテナ群を停止する．
  (2) マネージャノード上に，SAGE2 Server のコンテナを起動する．
  (3) (2) で起動したコンテナ内で，SAGE2 の初期設定と SAGE2 Server の起動を行う．
  (4) 各モニタ上にて，フルスクリーンモードの Chromium ブラウザを表示し，自身の持つ Client ID に対応する URL を開く．
- ParaView モード起動スクリプト
  (1) TDW の現状態が SAGE2 モードである場合，起動中のすべての Chromium ブラウザ及び SAGE2 Server のコンテナを停止する．
  (2) Docker Swarm クラスタ上で，pvserver のコンテナ群を，モニタ数と同じ数だけ起動する．
  (3) (2) で起動した全コンテナの IP アドレスを収集し，マネージャノード上の pvserver のコンテナ内に MPI 用のホストファイルを作成する．
  (4) (3) で作成したホストファイルを用いて，全コンテナ内の pvserver を MPI によって連携駆動させる．
- Stop モード起動スクリプト
  (1) TDW の現状態が SAGE2 モードであるなら，起動中のすべての Chromium ブラウザを停止する．
  (2) Docker Swarm クラスタ上のすべてのコンテナを停止する．

4. 評価

本研究では，提案手法の実用性について検討するため，2 つの実験 (実験 1・実験 2) を行った．実験 1 では，3.3 節で実装した提案モジュールを用いて，可視化ミドルウェア環境のスイッチング時間を計測した．実験 2 では，コンテナ内の可視化ミドルウェアがレンダリングを行う際に発生する，Docker によるオーバーヘッドを計測した．

4.1 評価環境

評価のため，3.3 節で実装したモジュールを，2 × 2 のモニタ群と 4 ノードの PC クラスタから成る評価用 TDW 上で利用した．図 7 に，評価用 TDW の構成を示す．利用したモニタの解像度は全て 1366 × 768 ピクセルである．TDW の PC クラスタを構成する 4 ノードの性能は，表 2 に示す通りである．また，本評価で使用した各種ソフトウェアのバージョンは，表 3 にまとめた．

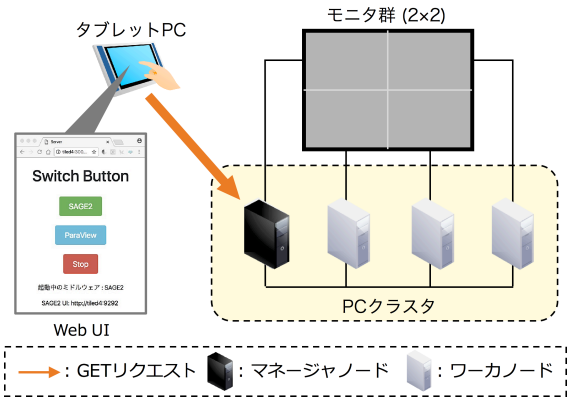

図 7 評価用 TDW

CPU	Intel Core i3-4150 (3.5GHz)
Memory	8.0GB
GPU	Intel HD Graphics 4400
OS	CentOS 7.3

表 2 PC クラスタのノードの仕様

ソフトウェア名	バージョン
Docker	1.12.6
SAGE2	2.0.0
ParaView	5.1.2
Chromium	55.0.2883.87

表 3 評価で使用したソフトウェアのバージョン

4.2 実験内容

4.2.1 実験 1

実験 1 では，実装したモジュールが実行できる以下の 6 通りのスイッチング (A) 〜 (F) について，それぞれの所要時間 (スイッチング時間) を計測した．この計測は，スイッチング時に停止・起動する必要のある可視化ミドルウェアのプロセス数を変えながら行った．可視化ミドルウェアのプロセスとは，SAGE2 モードで利用する Chromium ブラウザのプロセス及び ParaView モードで利用する pvserver のプロセスのことである．これらのプロセスは TDW のモニタ群を構成するモニタの台数と同じ数だけ必要になるため，プロセス数の変化範囲は 1〜4 とした．

- **(A)** SAGE2 モード → ParaView モード
- **(B)** SAGE2 モード → Stop モード
- **(C)** ParaView モード → SAGE2 モード
- **(D)** ParaView モード → Stop モード
- **(E)** Stop モード → SAGE2 モード
- **(F)** Stop モード → ParaView モード

4.2.2 実験 2

実験 2 では，コンテナ内の pvserver がレンダリングを行う際に生じる Docker によるオーバーヘッドを計測した．この計測では，2.2.2 節で述べた ParaView の Python スクリプティング機能を利用した．本計測で用いた Python スクリプト (test.py) のソースコードは，本論文の付録に示す．このスクリプトは，可視化された 3D データに対して「x 軸方向に 1 度回転させて再描画する」という処理を 360 回繰り返すことで 1 回転させ，その実行時間 (10 回の実行の平均) を計測するという内容である．また，計測の際に ParaView によって可視化するデータとして，図 8 に示す can.ex2 (ParaView に付属する 17MB のサンプル 3D データ) を用いた．

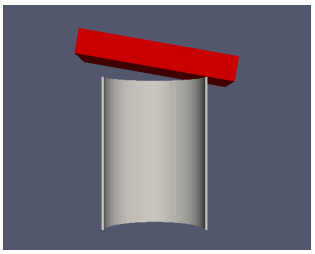

図 8 ParaView で可視化した can.ex2

実験 2 の具体的な手順を以下に示す．以下の手順を，pvserver のプロセス数を 1〜4 の範囲で変えながら行った．

(1) Docker を利用してコンテナ内で起動した pvserver によって can.ex2 を可視化し，test.py を適用して実行時間を計測する．
(2) Docker を利用せずにホスト OS 上で起動した pvserver によって can.ex2 を可視化し，test.py を適用して実行時間を計測する．
(3) (1) と (2) で得られた実行時間を次式に代入し，コンテナ内の pvserver がレンダリングを行う際に生じる，Docker によるオーバーヘッドを実行時間に対する比率として求める．

$$\text{オーバーヘッド } [\%] = \left(\frac{\text{(1) での実行時間 [s]}}{\text{(2) での実行時間 [s]}} - 1 \right) \times 100$$

4.3 評価結果

4.3.1 実験 1

図 9 に，実験 1 での計測結果を示す．図 9 の縦軸はスイッチング時間 [s] であり，横軸はスイッチング時に起動・停止する必要のある可視化ミドルウェアのプロセス数である．図 9 から，プロセス数が変化しても，(A) は約 13.5 秒，(B) は約 5.6 秒，(C) は約 17.5 秒，(D) は約 5.2 秒，(E) は約 12.6 秒，(F) は約 8.3 秒のスイッチング時間を維持することを確認した．この結果から，提案モジュールによるスイッチング (A) 〜 (F) は，可視化ミドルウェアのプロセス数が変化しても，実用上問題無い程度の時間で完了することを確認できた．

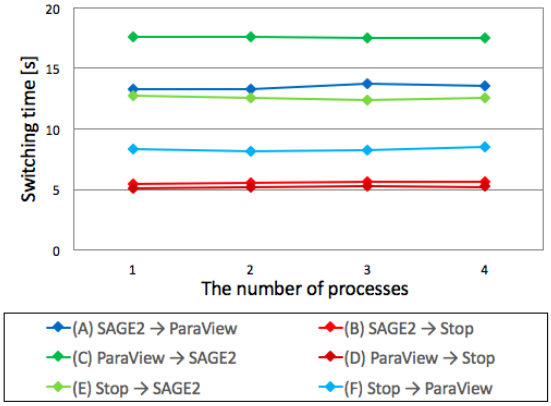

図 9 提案モジュールによるスイッチング時間

4.3.2 実験 2

図 10 に，実験 2 での計測結果を示す．図 10 の縦軸は Docker によるオーバーヘッドの比率 [%] であり，横軸は pvserver のプロセス数である．図 10 から，Docker によるオーバーヘッドの比率は，pvserver のプロセス数が変化しても 3%前後で維持されることを確認した．3.1 節で述べたように，プロセス実行時のコンテナ型仮想化のオーバーヘッドは極めて小さい．実験 2 で観測されたオーバーヘッドの主な原因は，可視化するデータを各コンテナ内の pvserver のバッファにロードするために，Docker のオーバーレイネットワーク機能 (複数ホストにまたがるコンテナ間通信を実現する機能) を利用する際に生じた遅延にあると考えられる．

ParaView は，OpenGL によるグラフィックボードのアクセラレーションを用いて 3D の描画を行う．このような OpenGL を用いた描画では，ダブルバッファリング等のバッファ多重化技術が用いられる．バッファを多重に持つことで，描画とデータロードを並行して行うことが可能となり，レンダリング速度の向上につながる．したがって，pvserver がオーバーレイネットワーク機能を利用する際に生じる遅延の影響は，pvserver のバッファを増やすことである程度軽減できると考えられる．

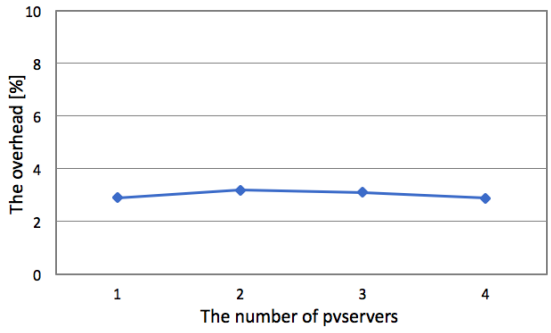

図 10 レンダリング時の Docker によるオーバーヘッド

5. まとめと今後の課題

本研究では，共有利用を想定した TDW での可視化ミドルウェア環境の分離とスイッチング機能の提案を行った．環境の分離にはコンテナ型仮想化技術 Docker を利用し，スイッチング機能としては SAGE2 モード・ParaView モード・Stop モードという 3 状態間でのスイッチングを可能にする提案モジュールを実装した．評価としては，提案手法の実用性について検討するために，2 つの実験 (実験 1・実験 2) を行った．実験 1 の結果，提案モジュールによるスイッチング時間は，可視化ミドルウェアのプロセス数が変化しても実用上問題無い程度で維持されることを確認した．また，実験 2 の結果，コンテナ内の pvserver によるレンダリングの際に生じる Docker のオーバーヘッドの比率は，pvserver 数が変化しても 3%前後で維持されることを確認した．これらの結果から，提案手法の実用性を確認することができた．

今後の課題としては，より大規模な TDW を用いた評価実験の実施や，ハイパーバイザ型仮想化を用いることによる提案モジュールの改良がある．

謝辞 本研究は JSPS 科研費 JP26540053 の助成を受けたものです．

参考文献

[1] 山田知典，野口紘一，淀薫，和田義孝，藤井秀樹，吉村忍：大規模耐震シミュレーション結果のサーバーサイドスクリーニング，日本計算工学会論文集，Vol. 2016, pp. 1–7 (2016).
[2] Kido, Y., Ichikawa, K., Date, S., Watashiba, Y., Abe, H., Yamanaka, H., Kawai, E., Takemura, H. and Shimojo, S.: SAGE-based Tiled Display Wall enhanced with dynamic routing functionality triggered by user interaction, *Future Generation Computer Systems*, Vol. 56, pp. 303–314 (2016).
[3] Humphreys, G., Buck, I., Eldridge, M. and Hanrahan, P.: Distributed rendering for scalable displays, *Proceedings of the 2000 ACM/IEEE conference on SuperComputing*, No. 30 (2000).
[4] 大阪大学サイバーメディアセンター大規模可視化システム，`http://vis.cmc.osaka-u.ac.jp/`.
[5] Marrinan, T., Aurisano, J., Nishimoto, A., Bharadwaj, K., Mateevitsi, V., Renambot, L., Long, L., Johnson, A. and Leigh, J.: SAGE2: A New Approach for Data Intensive Collaboration Using Scalable Resolution Shared Displays, *Proceedings of the 2014 IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pp. 177–186 (2014).
[6] Cedilnik, A., Geveci, B., Moreland, K., Ahrens, J. and Favre, J.: Remote large data visualization in the paraview framework, *Proceedings of the 2006 Eurographics Symposium on Parallel Graphics and Visualization*, pp. 163–170 (2006).
[7] Wierse, A.: Performance of the collaborative visualization environment (COVISE) visualization system under different conditions, *Proceedings of the 1995 International Society for Optics and Photonics's Symposium on Electronic Imaging: Science & Technology*, Vol. 2410, pp. 218–229 (1995).
[8] Kan, C.: DoCloud: An elastic cloud platform for Web applications based on Docker, *Proceedings of the 2016 International Conference on Advanced Communication Technology*, pp. 478–483 (2016).
[9] Naik, N.: Building a virtual system of systems using docker swarm in multiple clouds, *Proceedings of the 2016 IEEE International Symposium on Systems Engineering*, pp. 1–3 (2016).

付　　録

付録として, 4.2.2節で述べたPythonスクリプト (test.py) のソースコードを以下に示す.

```
1  # test.py
2
3  import paraview
4  import time
5
6  # 変数の初期化
7  average = 0
8
9  # 計測を10回実行
10 for i in range(1, 11):
11     # 開始時刻の取得
12     start_time = time.time()
13
14     # 360回再描画を実行
15     camera = GetActiveCamera()
16     for angle in range(1, 361):
17         # x軸方向に1度回転
18         camera.Roll(1)
19         # 再描画
20         Render()
21
22     # 終了時刻の取得
23     end_time = time.time()
24
25     # 実行時間の計測
26     interval = end_time - start_time
27     average += interval
28
29 # 実行時間の平均を計算
30 average /= 10
31 print average
```