

大学院博士前期課程修士学位論文

題 目

OS仮想化基盤を用いたSBCマルチディスプレイシステムの
フレーム処理並列化

指導教員

下條 真司 教授

報告者

高畠 勇我

2022年2月9日

大阪大学 大学院情報科学研究科

マルチメディア工学専攻

OS仮想化基盤を用いたSBCマルチディスプレイシステムのフレーム処理並列化

高畠 勇我

内容梗概

複数のディスプレイを1台の大型ディスプレイとして仮想化するマルチディスプレイシステム (MD) は、高解像な動画像を複数人で見る目的で利用される。先行研究では、MD構築コストの低減を目的としてシングルボードコンピュータ (SBC) を用いたMDが提案された。このMDはヘッドノードとディスプレイノードの2種類のノードから構成されており、SBCで実装したディスプレイノードの負荷を低減するためにヘッドノードで画像フレームの処理を行っている。しかし、ヘッドノードで行われている処理がボトルネックとなり、高フレームレートまたは高解像度な動画を再生すると動画表示に遅延が発生する。また、分割後のフレームごとに圧縮処理と送信処理を行うため、これらの処理を単一のプロセスで行う場合には処理の待ち時間が発生し、MDのパフォーマンスおよびスケーラビリティを低下させている。

本研究では、OS仮想化技術を利用し、ヘッドノードのフレーム圧縮部分を並列化し独立したプロセスとして実装することでフレームの処理時間を短縮し、MDのボトルネック解消を目指す。

提案手法におけるヘッドノードは、画像フレームの分割を行う分割コンテナ、画像フレームの圧縮と送信を行う圧縮コンテナ、ディスプレイノード群の同期制御を行う同期制御コンテナの3種類のコンテナで構成する。分割コンテナおよび同期制御コンテナはそれぞれヘッドノード内に単一のプロセスとして動作する。また、圧縮コンテナはMDを構成するディスプレイと同数が動作しており、対応するディスプレイノードと1対1で接続する。分割コンテナと圧縮コンテナの間で必要となる画像フレームの受け渡し処理には、高速なプロセス間データ通信が可能な共有メモリ (SYSTEM V IPC) を使用し、フレームデータ受け渡しにかかるオーバーヘッドを抑える。

評価では、提案手法によるヘッドノードでのフレーム処理時間の変化を確認するため、4K解像度の映像を用いてヘッドノードのフレーム処理に要する時間を計測、比較した。ディスプレイ4面構成および9面構成のMDを動作させることを想定し、ヘッドノード内で1つの分割コンテナと、ディスプレイ数と同数の圧縮コンテナを動作させた。評価結果より、4面構成時、9面構成時ともに提案手法ではフレーム処理に要する時間が短縮され、

分散並列化によりシステムのボトルネックが解消されたことが確認できた。また、ディスプレイ数を増加させても既存手法と比べて処理時間の増加は小さく、スケーラビリティも改善されていることが確認できた。

キーワード

マルチディスプレイ、小型低性能計算機、OS仮想化、コンテナ並列化、プロセス間通信

目 次

1	序論	1
2	小型低性能計算機を用いたマルチディスプレイシステムの構築	2
2.1	従来のマルチディスプレイ構築手法	2
2.2	先行研究で提案されたマルチディスプレイシステム	7
2.3	ヘッドノードの処理負荷軽減を行った関連研究	12
2.4	本研究での技術的課題	15
3	提案	16
3.1	OS仮想化基盤とコンテナ技術	16
3.2	MDのコンテナ化	17
3.3	コンテナを通したディスプレイの制御	18
3.4	ヘッドノードでのフレーム処理の改良	19
4	評価	27
4.1	実験環境	27
4.2	フレーム処理時間の評価	29
4.2.1	実験方法	29
4.2.2	実験結果	29
4.3	MDのコンテナ化の評価	31
4.3.1	実験方法	31
4.3.2	実験結果	31
4.3.3	結果に対する考察	31
5	関連研究	32
6	結論	33
	謝辞	34
	参考文献	35

1 序論

共同研究などでシミュレーション結果や動画像の共有などに利用 [1, 2]

2 小型低性能計算機を用いたマルチディスプレイシステムの構築

本章では、小型低性能計算機、特にSBCを利用した低成本なマルチディスプレイシステム構築手法の必要性と、その実現に向けて行われた先行研究について説明する。まず2.1節では、従来利用されてきたマルチディスプレイシステムの構築手法とそのコストについて述べる。続く2.2節では、先行研究でも使用されており、本研究でも取り扱うシングルボードコンピュータ(SBC)について説明し、2.3節では先行研究である、SBCを用いて構築されたマルチディスプレイシステムについて説明する。さらに2.4節で、先行研究で提案されたSBCを用いたマルチディスプレイシステムについての問題点と、その問題点を解決することを目的として行われた関連研究を紹介する。最後に2.5節では関連研究において将来的な課題とされていた部分の改善策と、それを実現するための技術的な課題点について述べる。

2.1 従来のマルチディスプレイ構築手法

これまでMDの構築には数多くの手法が提案されており、それらは大きく分けると2種類の手法に分類することができる[3, 4]。1つはマルチディスプレイを構築可能にする専用のハードウェアを用いて構築する手法であり、もう1つはマルチディスプレイを構築可能にする専用のミドルウェアを用いて構築する手法である。本節では、この2種類の構築手法を簡単に解説してそれぞれの構成例を示し、構築に要する金銭的コストについて説明する。

専用ハードウェアを用いたマルチディスプレイの構築

マルチディスプレイ構築用のハードウェアは接続した複数台のディスプレイに対して映像出力用の信号を送信する機能を持った機器であり、主にデジタルサイネージ[5]や会議用のディスプレイを構築する際に利用される。マルチディスプレイ構築用ハードウェアの多くは、HDMI(High-Definition Multimedia Interface), DisplayPort[6]等の規格に対応した映像出力ポートを複数搭載しており、各ポートに映像信号を分配することによって、接続したディスプレイを連動させて制御する。マルチディスプレイ構築用ハードウェアは、表示映像フレームの分割、接続ディスプレイ間の同期処理などをハードウェアレベルで行う。ハードウェアベースで処理を行うことにより高速なフレーム処理が可能となり、ディスプレイ上に高フレームレートで映像を出力することができる。一方で、マルチディスプレイ構築用ハードウェアは入力ポート、出力ポートそれぞれの設置数が固定であるため、入出力に使用できるポート数には上限が存在することになる。そのため、マルチディスプレイを構築するディスプレイ数にも上限が存在することになり、より大規模なディスプレイや高解像なディスプレイを構築する事が難しい。これがマルチディスプレイシステムのスケーラビリティという観点からすると欠点となっている。マルチディスプレイシステム

構築用ハードウェアの具体例としては、グラフィックボードやマトリックススイッチャが存在する。図 2.1 に、マトリックススイッチャを用いて構築したマルチディスプレイの概要図を示す。



図 2.1: 専用ハードウェアを用いたマルチディスプレイシステム

グラフィックボードは、PCなどのコンピュータにおいて、映像を信号として入出力するための機能を拡張ボードとして独立させたものである。また、一般的には内部に GPU (Graphics Processing Unit) と呼ばれるグラフィックスを描画する際の計算処理を行う半導体チップが組み込まれており、高速な画像処理が可能である。各グラフィックボードに搭載されているチップやメモリによって、描画可能な解像度や表現色数、2D/3D 描画性能などが異なる。マルチディスプレイを構築する目的でグラフィックボードを使用すると、ある程度の高解像な画像を描画可能であり、信号の入出力可能ポート数が多層備え付けられているものが必要となる。マルチディスプレイ対応型のグラフィックボードは、NVIDIA 社や AMD 社から発売されている。マルチディスプレイ対応型のグラフィックボードは、一般的に利用可能な映像出力ポート数が多く、表示解像度が優れているものほど価格が高くなる。

専用ミドルウェアを用いたマルチディスプレイシステム

マルチディスプレイ構築用ミドルウェアは、ディスプレイに接続した汎用 PC に画像フレームまたは描画命令を送信することによって動画や画像を表示する。マルチディスプレイ構築用ミドルウェアは主に科学的可視化の分野で広く利用されており、高性能なコンピュータで行われたシミュレーションの結果を表示する大規模可視化装置の構築などに使

用される。接続可能なディスプレイ数に上限は設けられておらず、多数の汎用PCを連携して動作させることでスケーラブルなマルチディスプレイの構築が可能である。しかし、マルチディスプレイ構築用ミドルウェアは大規模な科学的可視化の用途を想定して設計されたものが多く、ディスプレイと接続された汎用PC群に対して高負荷な描画処理を要求するものが多い。そのため、マルチディスプレイを構築するために使用される汎用PCには一定の処理性能を満たすものが必要となり、マルチディスプレイを構成するディスプレイ数が増加するごとに構築費用が高騰するという問題点がある。図2.2に、マルチディスプレイ構築用ミドルウェアを用いて汎用PC群を連携させることで構築したマルチディスプレイの構成例を示す。

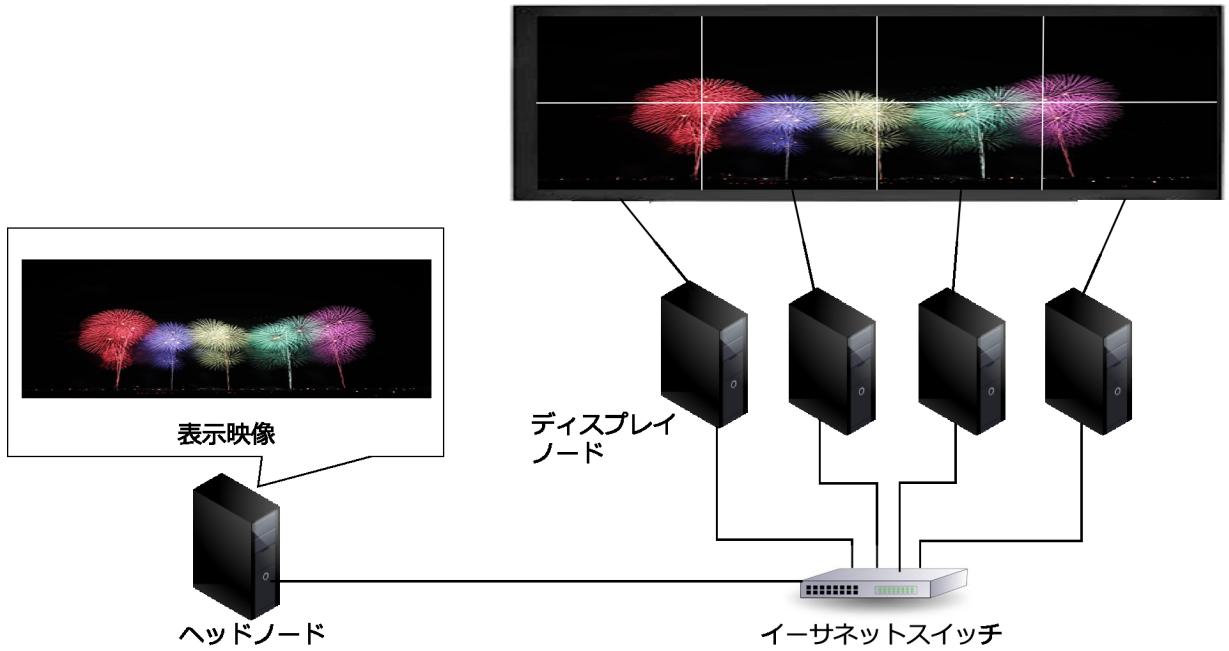


図2.2: マルチディスプレイ構築用ミドルウェア

それぞれのPCはマルチディスプレイ構築用ミドルウェアによってヘッドノードまたはディスプレイノードとして動作することが可能になる。ヘッドノードは各ディスプレイに画像を表示するための信号の送信やヘッドノード-ディスプレイノード間の同期処理などの役割を担う。ディスプレイノードはマルチディスプレイを構築するディスプレイに接続され画像の描画や表示に関する計算や処理を行う。ディスプレイノードとディスプレイは1対1で接続されるため、ディスプレイノードはマルチディスプレイを構築するディスプレイの数と同数用意する必要がある。

以下、専用ミドルウェアを用いたマルチディスプレイシステムの詳細な動作について説明する。各ミドルウェアによって行われる連携表示処理には、大きく分けて2通りの手法がある。1つはフレーム転送方式と呼ばれる手法である。フレーム転送方式はヘッドノードがディスプレイノードに対して表示映像のフレームを送信し、ヘッドノードとディスプレイノード間でデータをやり取りする方式である。

レイノードの同期処理によってフレームをディスプレイ上に連携表示させる方式である。フレーム転送方式を採用するマルチディスプレイの例として、SAGE [7] や SAGE2 [8] などがあげられる。

フレーム転送方式は、以下の 5 種類の動作から成り立つ。

- フレーム圧縮
- フレーム送信
- フレーム展開
- 同期制御
- フレーム表示

まずヘッドノード上で表示映像からフレームが切り出され、フレームの圧縮処理が行われる。このとき、フレームを各ディスプレイノードの表示領域に合わせて分割する手法と、フレーム全体をそのまま圧縮する手法の 2 通りが存在する。前者は SAGE [7]、後者は DisplayCluster [9] といったミドルウェアでそれぞれ採用されている。フレームの圧縮には DXT (DirectX Texture Compression) [10] や JPEG (Joint Photographic Experts Group) [11] などが利用される。

続いて、ヘッドノードで圧縮されたフレームを各ディスプレイノードへ送信する。ディスプレイノードは、ヘッドノードから圧縮フレームを受信したあと展開処理を行い、続いて同期処理を開始する。同期処理では、まずフレームの展開を終え表示待機状態になったディスプレイノードからヘッドノードへ表示準備完了通知が送信される。ヘッドノードは、接続された全てのディスプレイノードから表示準備完了通知を受け取った後、各ディスプレイノード宛に表示命令を送信する。表示命令を受け取ったディスプレイノードは、該当フレームをディスプレイ上に表示する。この一連の動作を繰り返し行うことで、ディスプレイ上に連携して動画を表示することが可能になる。

もう 1 つは、描画命令転送方式と呼ばれる方式である。この方式では、ヘッドノードはディスプレイノードに対して画像フレームの送信を行わず、アプリケーションの出力する描画命令を同期処理を行いつつ送信することで動画の連携表示を行う。この方式を採用しているミドルウェアとしては、DMX (Distributed Multihead X) や Chromium [12] などがあげられる。

描画命令転送方式は、主に以下の 4 種類の動作から成り立つ。

- 描画命令のキャプチャリング
- 描画命令転送

- 同期制御

- フレーム描画

まず、ヘッドノードはアプリケーションから出力された描画命令のキャプチャリングを行う。続いて、ヘッドノードがキャプチャリングした描画命令をディスプレイノードへ送信する。描画命令を受信したディスプレイノードは、命令を受け取ったことを通知するメッセージをヘッドノードへ送信する。ヘッドノードは全てのディスプレイノードから描画命令受信通知を受信し次第、フレーム描画命令を送信し、ディスプレイノードがこの描画命令を受信しディスプレイ上にフレームを描画する。この一連の動作を繰り返し行うことで、ディスプレイ上に連携して動画を表示する。

シングルボードコンピュータ (SBC)

シングルボードコンピュータ (SBC) は、1枚の基板上に必要最低限の部品を取り付けることで構築された小型計算機である。SBC は他の汎用 PC と比較しても非常に低価格で手に入れることができ、消費電力も少ないため、IoT (Internet of Things) 向けの機器として一般に利用される [13, 14]。また、軽量プログラミング言語が利用できることから、教育用やプログラミング学習用としても使用されている。SBC には Raspberry Pi シリーズを代表として、様々な種類のものが存在する。表 2.1 に主な SBC の機種とその仕様を示す。また、図 2.3 に主な SBC の画像を示す。

表 2.1: SBC 機種の仕様および価格

機種	Raspberry Pi 4 Model B [15]	Orange Pi Zero Plus2 [16]	Banana Pi BPI-M64 [17]
CPU	ARM Cortex-72 (1.5 GHz × 4)	ARM Cortex-A53 (1.8 GHz × 4)	ARM Cortex-A53 (1.2 GHz × 4)
GPU	Broadcom VideoCore IV (500 MHz)	Mali T720MP2 (650 MHz)	Mali-400 MP2 (500 MHz)
メモリ	4.0 GB	4.0 GB	8.0 GB
通信帯域	1 Gbps	100 Mbps	1 Gbps
映像出力端子	HDMI × 2	HDMI × 1	HDMI × 1
OS	Linux	Linux	Linux
価格	約 8,000 円	約 14,000 円	約 9,000 円

SBC が持つ CPU の特徴は、主に低周波数のコアがデュアルコア、クアッドコアなどの形式で複数搭載されているものが多いことである。低周波数コアの複数搭載は消費電力を



Raspberry Pi Model 4 B [15] Orange Pi Zero Plus2 [16] Banana Pi BPI-M64 [17]

図 2.3: 主な SBC の画像

抑えるため、また価格を低く抑えるためであり、こうした省電力や低価格という特徴を維持しつつCPU自体の処理能力を向上させる目的で、近年ではコア数の増加が進んでおり、マルチコア方式を採用した機種が多く販売されるようになってきている。搭載メモリや通信帯域に関しては、近年のSBCの性能向上により4.0 GB以上の比較的大きなメモリ容量を持つものも増えており、ギガビットイーサネットに対応したNIC (Network Interface Card) を有するものも多くの機種でHDMI端子による出力が可能であり、Full HD解像度での出力が標準的である。また、各機種のOSには軽量なLinuxベースのOSが広く採用されている。例えばRaspberry Piに対応したOSであるRaspbian [18]もLinuxをベースとするOSの一種である。

2.2 先行研究で提案されたマルチディスプレイシステム

2.1節で述べたように、ハードウェアを用いたマルチディスプレイの構築ではハードウェア自体の価格によりマルチディスプレイシステムの構築コストが増大する。また、ミドルウェアを用いてマルチディスプレイシステムを構築する場合でも、ディスプレイと1対1で接続して映像の表示を制御する役割を持ったディスプレイノードとして動作する汎用PCが複数台必要になるため同様に構築コストが増大する。このようなマルチディスプレイシステム構築におけるコスト面の問題に対して、マルチディスプレイシステムの構築費用を低減する目的でディスプレイノードとしてSBCを活用するミドルウェアの開発が行われた。

本節では、先行研究 [19] で提案された、SBCを用いて構築されたマルチディスプレイの特徴について述べる。まず、マルチディスプレイを構築する各ノードの動作について説明する。その後フレームの連携表示処理を可能にする仕組みについて概説する。

連携表示処理

先行研究で構築されたマルチディスプレイシステムは、1台のヘッドノードと、SBCを用いて実装された複数台のディスプレイノードで構成されている。

システムの概要を図 2.4 に示す.

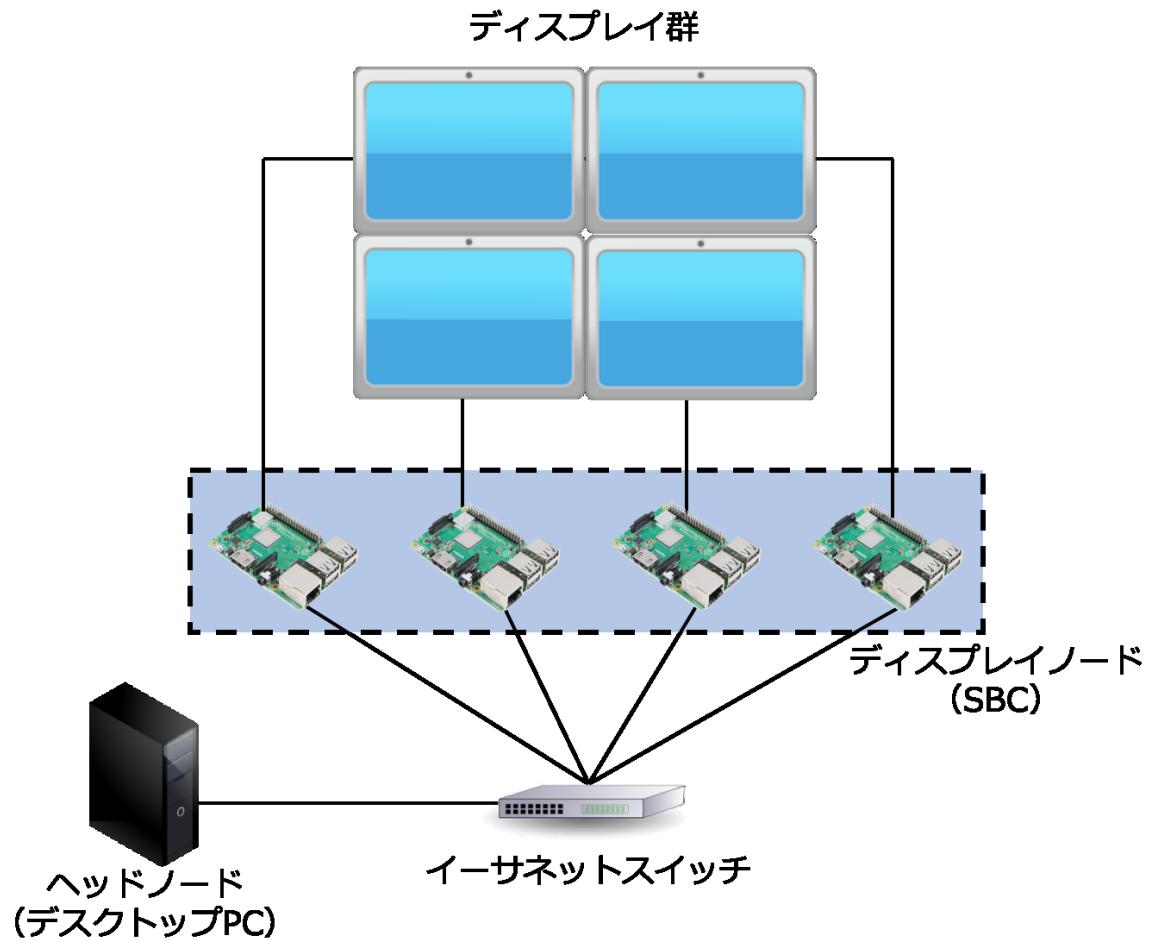


図 2.4: SBC を用いて構築したシステムの概要図

このマルチディスプレイシステムは、2.1 節で述べた 2 種類の方式のうち、フレーム転送方式を採用している。そのため、SBC をディスプレイノードとして用いることとなり、従来のフレーム転送方式をそのまま適用すると SBC 上で負荷の高い処理を行う必要がある。しかし、SBC は市販の汎用 PC などと比べると処理速度、描画性能などの面で劣っているため、可能な限りディスプレイノード上での処理負荷を小さく抑えなければならないという制約が存在する。そのため、SBC がディスプレイノードとして十分高速に動作するような設計を目的として、JPEG 圧縮や連携表示処理のパイプライン化などが実装されている。

以下では、先行研究において提案されたマルチディスプレイシステムの詳細な動作について述べる。ヘッドノードでは、動画ファイルからフレームを切り出した後、接続ディスプレイノード数に応じてフレームの分割を行う。分割したフレームはそれぞれ JPEG 方式で圧縮され、各ディスプレイノードへと送信される。ヘッドノードから送信された圧縮

フレームを受信したディスプレイノードでは、フレームを展開し、その展開が終了した後表示バッファに格納してヘッドノードに表示準備完了メッセージを送信する。ヘッドノードは全てのディスプレイノードからの表示準備完了メッセージを受け取り次第表示命令を送信し、命令を受け取ったディスプレイノードはディスプレイ上にフレームを表示する。

また、ノード上の各処理はスレッド処理によって行われており、ヘッドノード上では圧縮スレッド、送信スレッド、同期制御スレッドの3種類、ディスプレイノード上では受信スレッド、展開スレッド、表示制御スレッドの3種類のスレッドが動作している。

各スレッドの操作を図2.5に示す。

圧縮スレッドは、表示映像の切り出しと分割、圧縮までを行う。送信スレッドは、圧縮されたフレームを各ディスプレイノードに送信する。同期制御スレッドは、表示準備完了メッセージの受信と表示命令の送信を行う。受信スレッドは、ヘッドノードから圧縮フレームを受信する。展開スレッドは、圧縮フレームの展開を行う。表示スレッドは、ヘッドノードからの表示命令を受け取りフレームをディスプレイ上に表示する。

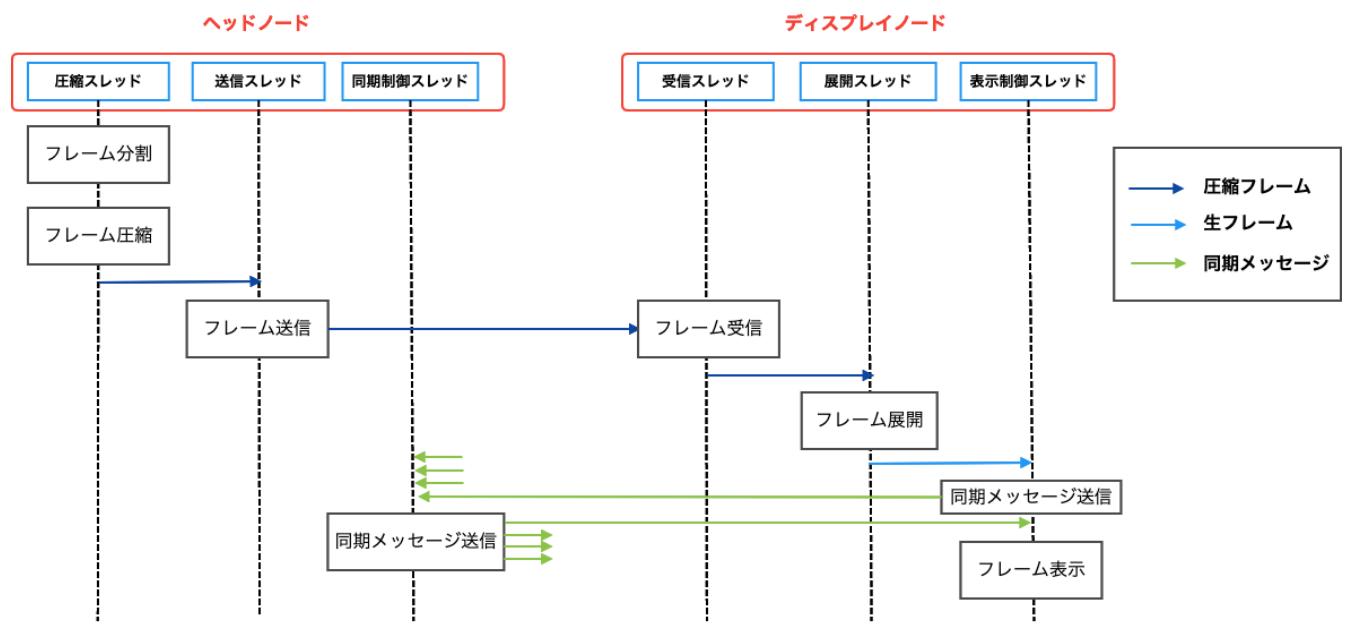


図 2.5: 各スレッドの動作

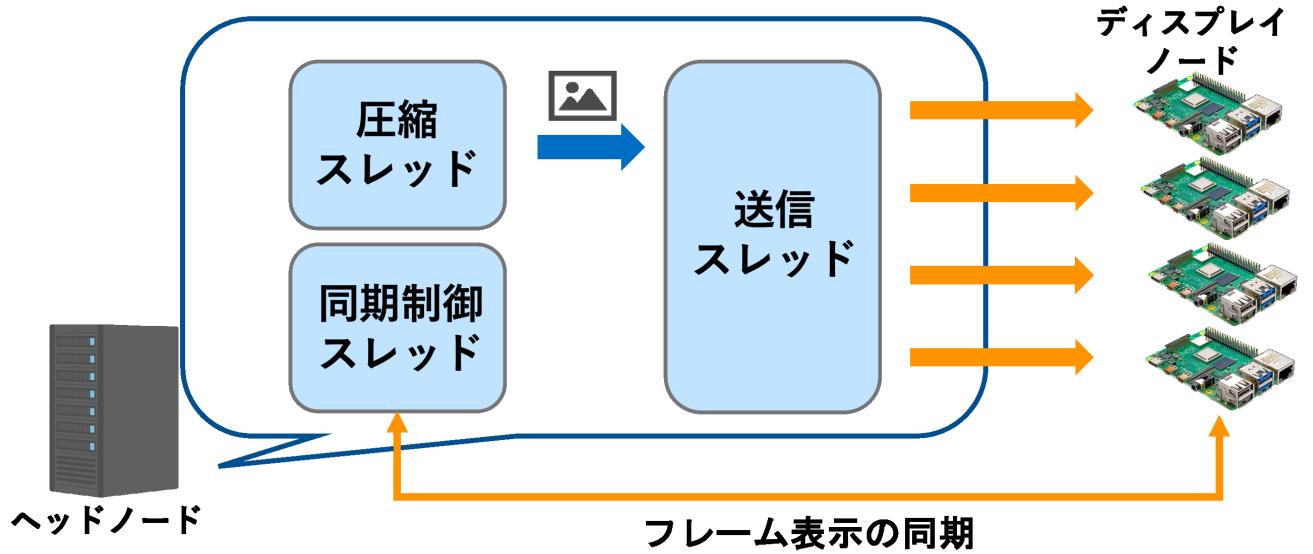


図 2.6: 各スレッドの動作

フィードバック処理

JPEG 圧縮処理では、まず入力画像のピクセルごとの画素値を RGB 形式から YCbCr 形式に変換する。YCbCr は色彩表現法の一種であり、色を人間の目が認識しやすい輝度成分と認識しづらい色差成分とに分けて表現する [20]。YCbCr 形式に変換された色成分はデータ量を削減するために間引かれることになるが、この時 YCbCr サンプル比という間引き方を決定するパラメータを変更することで、JPEG 圧縮に要する時間を変更することができる [21]。また、量子化を行う際の品質係数と呼ばれるパラメータを変更することによっても、JPEG 圧縮に要する時間を短縮することができる。先行研究のマルチディスプレイシステムでは、ディスプレイノードで動画再生時のフレームレートを計算し、目標フレームレートとの差に応じてこれらのパラメータを変更することで、フレームレートを目標値に近づけるフィードバック処理を実装している。

問題点と技術的課題

本節では、前節で述べた先行研究におけるマルチディスプレイシステムの問題点と、それを解決するための技術的課題について述べる。まず、先行研究のマルチディスプレイシステムにおける具体的な問題について説明する。その後、問題点解決に向けた手法の考察と利点、欠点について述べ、比較を行う。

先行研究で提案されたマルチディスプレイシステムの問題点

先行研究で構築されたマルチディスプレイには、高解像度な動画や高フレームレートな動画を表示しようとするとパフォーマンスが低下するという問題点がある。この問題の原因となっているのが、ヘッドノードでのフレーム処理である。前述したように、ヘッドノード内では動画フレームに対して分割、圧縮、送信の3つの処理が行われている。高解像度な動画では1フレームあたりのデータ量が大きくなるためフレームの圧縮に時間がかかり、パフォーマンスが低下する。また、高フレームレートな動画ではフレーム圧縮に要する時間が動画における1フレームの表示時間を上回り、動画本来のフレームレートを維持したまま動画を再生することが困難になる。

また、大規模なマルチディスプレイを構築した際のパフォーマンス低下も問題点として挙げられる。先行研究で構築されたMDの中では圧縮スレッド、送信スレッド、同期制御スレッドがそれぞれ1つのスレッドとして動作している。マルチディスプレイを構成するディスプレイの数が増加した場合にはそれに伴ってフレーム圧縮処理の回数も増加し、圧縮スレッドでのフレーム処理負荷も同様に増加することとなる。しかし、ヘッドノード内で動作する圧縮スレッドは1つであるため、処理負荷の増加に伴って処理に要する時間も増加することとなり、動画再生時のフレームレート低下につながる。実際、4面構成の場合では30fpsの動画をMD上に20~22fpsで再生することが可能だが、ディスプレイを9面構成にしたMDでは8~10fpsに低下することが確認できる(図2.6)。

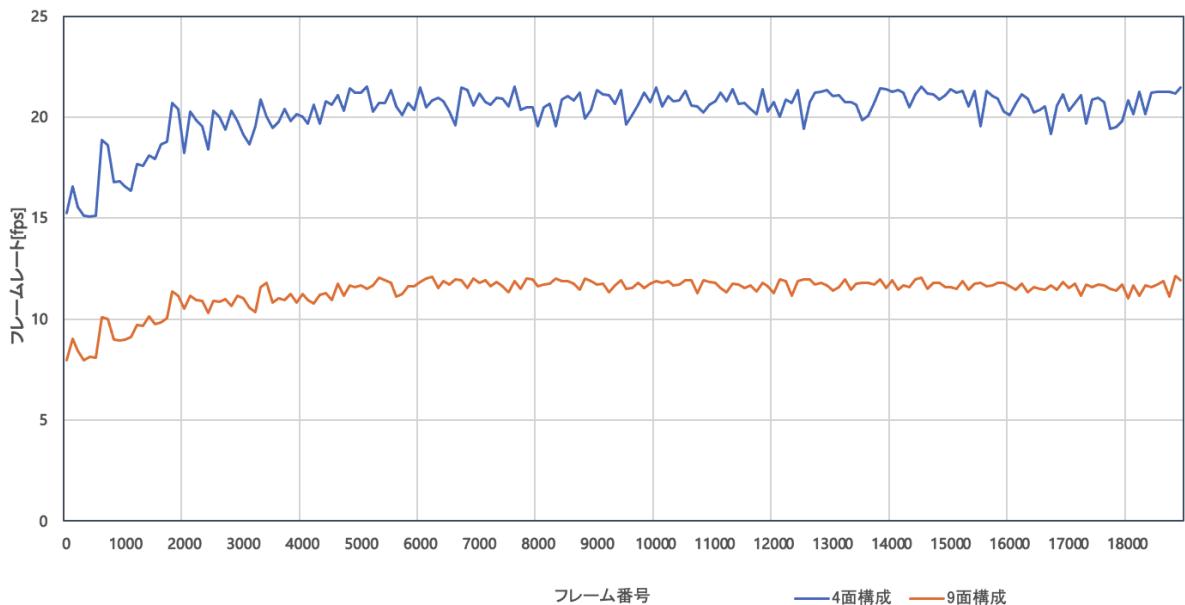


図 2.7: 画面構成変更時のフレームレート比較

2.3 ヘッドノードの処理負荷軽減を行った関連研究

ヘッドノードでのフレーム処理負荷が増加しフレームレートが低下するという問題に対して有効な対策として、ヘッドノードでのフレーム処理を並列化することにより処理負荷を軽減し、フレーム処理に要する時間の短縮を試みる手法が考えられる。この手法を用いた関連研究として、自身の卒業研究がある。

実装の概要

関連研究では、ヘッドノードでの処理を分散並列化する目的で、処理のノード分散を行っている。ノード分散では、新たにノードとしてCPUを追加実装することによって物理的にCPUを増加させ、高負荷なフレーム処理を複数のCPUで分担して実行する。また、ヘッドノードの性能に依存することが少なくなるため、ヘッドノードとしてCPUコア数が少ないような比較的低価格なPCを用いた場合にも大きく性能を落とさずに動作することが期待できる。

関連研究では、ヘッドノードの処理を複数のノードに分散するために、ヘッドノードとディスプレイノードとの間に新たにSBCを用いて中継ノードを実装した。また、連携表示処理の並列化を行うため、中継ノードでの処理を受信スレッド、画像処理スレッド、送信スレッドという3種類のスレッドで分担して行うようマルチスレッド化している。

各スレッドの動作を図2.7に示す。

中継ノードの動作

受信スレッドはヘッドノードから送信される一次分割済み非圧縮画像フレームを受信し、受信バッファへの格納を行う。分割・圧縮スレッドは受信バッファに格納された非圧縮フレームを取り出し、接続されているディスプレイノードの数に応じてフレームの二次分割処理を行う。その後、それぞれの分割済フレームに対してJPEG圧縮処理を行い、送信バッファへと格納する。送信スレッドは、送信バッファに格納されたJPEG圧縮済フレームを、各接続ディスプレイノードへと送信する処理を行う。以上の3種類のスレッドによって、中継ノードはヘッドノードとディスプレイノードとの間で連携表示処理を実現する。

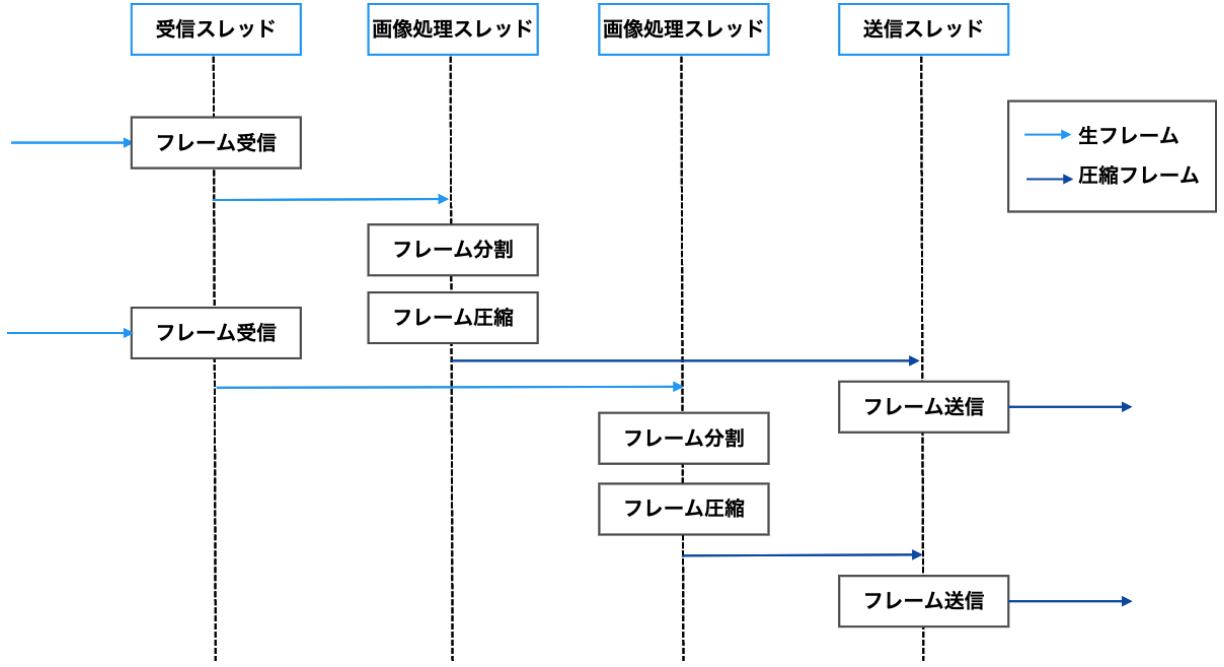


図 2.8: 各スレッドの動作

ヘッドノードの動作

中継ノードを実装したマルチディスプレイシステムでは、ヘッドノードは接続中継ノード数に応じて画像フレームの分割を行う。分割された画像フレームは、送信スレッドへ渡され、各フレームに対してフレーム番号が付与される。そして、フレーム番号の付与が終了したフレームは圧縮処理を行わずに中継ノードへと送信され、中継ノードは非圧縮フレームを受信して受信バッファに格納する。その後、画像処理スレッドが非圧縮フレームを受信バッファから取り出し、接続ディスプレイノード数に応じて分割し、パラメータに従い JPEG 形式に圧縮してディスプレイノードへ送信する。

ディスプレイノードの動作

ディスプレイノードは中継ノードから送信された圧縮フレームを受信し、受信バッファに格納する。その後、展開スレッドが圧縮フレームを展開する。展開処理が終了したフレームは表示バッファに格納され、表示制御スレッドがこれを認識してヘッドノードへ表示準備が完了した旨の同期メッセージを送信する。そして、ヘッドノードからの表示命令を受信し、表示制御スレッドが表示フレームの切り替えを行うことにより順次フレームがディスプレイ上に表示される。この一連の動作を繰り返すことにより、マルチディスプレイシステムとしての連携表示処理が可能となる。

図 3.2 に提案手法を用いて構成した 4 面構成のマルチディスプレイシステム全体の論理構成図と動作フロー図を示す。

表 2.2: 中継ノード実装によるフレームレートの比較

手法	平均	最大	最小
中継ノード実装システム	1.95	2.25	1.85
中継ノード非実装システム	9.44	10.42	7.35

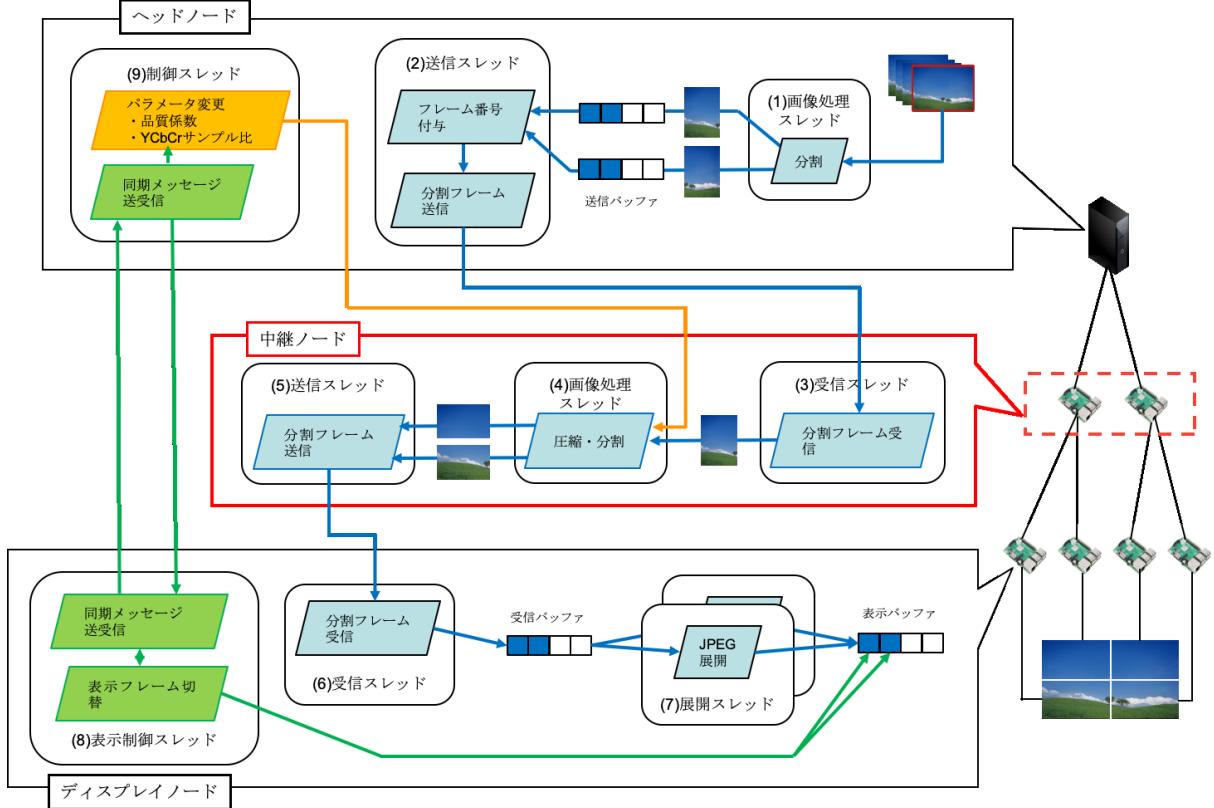


図 2.9: 関連研究で提案された手法を用いた MD の動作フロー

関連研究の問題点と将来課題

ヘッドノードとディスプレイノードの間に中継ノードを実装してフレーム処理の分散並列化を図った関連研究では、ヘッドノードの高負荷状態を解消する事ができた。しかし、フレームレートなどシステムのパフォーマンスの面では、元々のマルチディスプレイシステムと比較して性能が低下した（表 2.2）。

原因として考えられるのは、中継ノードを新たに実装したことで生じたフレームの送受信処理である。関連研究で提案されたマルチディスプレイシステムでは中継ノード内で画像フレームの圧縮処理を行うため、画像フレームはまずヘッドノードから中継ノードへと送信され、中継ノードで圧縮処理がなされた後ディスプレイノードへと送信される。この2度のフレーム送受信処理によって生じるオーバーヘッドが影響し、システムのパフォー

マンスが低下したと考えられる。

2.4 本研究での技術的課題

関連研究の結果から、ヘッドノードの処理を外部のノードへ分散させる手法は画像フレームデータの送受信のオーバーヘッドが生じるためパフォーマンスの向上への貢献度が低い事がわかった。フレームデータ送受信のオーバーヘッドを生じさせずにフレーム処理の並列化を行うためには、ヘッドノード内のCPUリソースを活用する事が必要である。

図2.9に、関連研究のヘッドノードにおけるCPU使用率の推移を示す。

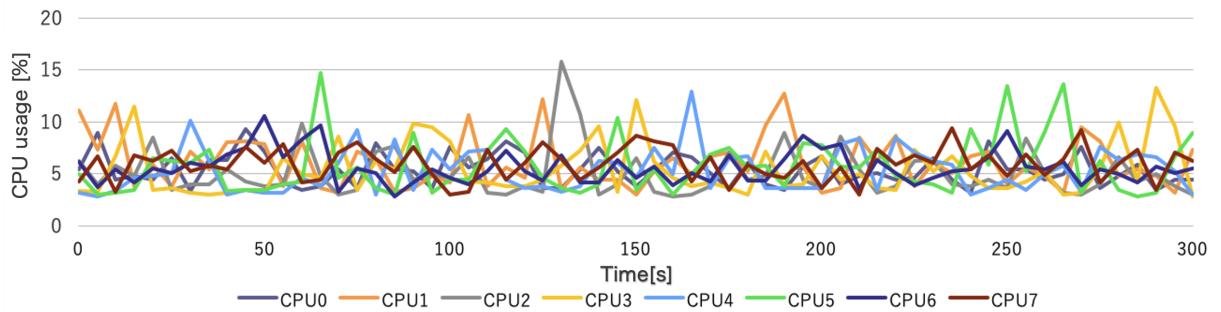


図 2.10: 関連研究のヘッドノードにおける CPU 使用率

これは中継ノードを実装して構築したマルチディスプレイ上に4K解像度の動画を表示し、映像の表示開始から300秒間のヘッドノードにおける各CPUコアの使用率の推移を計測したものである。最も使用率が高い瞬間においても使用率は15%程度にとどまっており、ヘッドノードの計算能力には十分な余裕が存在していることが確認できる。この計算能力の余裕を活用することで、ヘッドノード内で完結した処理が可能となり、画像フレームデータ送受信のオーバーヘッドを生じさせることのないフレーム処理が可能になると考えられる。

本研究では、SBCを用いて構築したマルチディスプレイに対して、ボトルネックとなっているフレーム処理部分の改善とMDのパフォーマンス向上を目的として設定し、ヘッドノード内のCPUリソースを活用したフレーム処理機構の実装について取り組む。

3 提案

本章では、SBC を用いたマルチディスプレイシステムにおける OS 仮想化技術を用いた仮想化と、コンテナ技術を用いた SBC マルチディスプレイシステムのフレーム処理並列化について設計・提案する。本章では、まず 3.1 節で OS 仮想化基盤とコンテナ技術について簡単に説明する。そして、続く 3.2 節で OS 仮想化基盤を利用した SBC マルチディスプレイシステムのコンテナ仮想化についての設計と実装について述べる。3.3 章では映像ベースのアプリケーションをコンテナ仮想化することによって生じる問題と、その解決法について述べる。3.4 章では、本研究の中心部分となるヘッドノード内でのフレーム処理の改良について、その設計指針を述べ、具体的な実装について記す。

3.1 OS 仮想化基盤とコンテナ技術

OS 仮想化基盤には、代表的なものとして Docker [22] が挙げられる。Docker は、Docker 社が開発しているプラットフォームであり、Docker を用いることで、マシン内にコンテナと呼ばれる仮想環境を作成、実行することができる。

Docker をはじめとした OS 仮想化と比較されるのが、ハイパーバイザ型仮想化である。ハイパーバイザ型仮想化は、ホストマシンとなる物理マシン上でハイパーバイザと呼ばれる仮想マシン (VM) を作成および実行するソフトウェアを動作させ、それを通じて仮想化環境を制御するものである。ハイパーバイザ型仮想化は、移植面で汎用性が高いことが長所である。一方で、ホスト OS を介して仮想環境の制御を行うため、制御のオーバーヘッドが大きく処理が低速になるという欠点を持つ。それに対して、コンテナを用いた OS 仮想化技術では仮想化環境 (コンテナ) はホストマシンの OS を使用する。そのため、コンテナは軽量に動作し仮想環境の起動や終了も小さいオーバーヘッドで行えるのが特徴である。

ハイパーバイザ型仮想化とコンテナ技術を用いた OS 化の概要を図 xxx に示す。



図 3.1: ハイパーバイザ型仮想化とコンテナを用いた OS 仮想化

さらに, Docker を利用することで様々な利点も生まれる.

まず, 仮想化環境をコード化して管理する事で, どのような環境の上にでも同一の環境を作成する事ができる点である. Docker では仮想化環境の構築情報をコード化されたファイルとして管理する. このファイルを保存し配布する事で, 特定の環境を再現し, 複数のマシン上で全く同一の環境を作り出すことが可能となる. この機能は, ソフトウェアの開発やテストなど, 異なるマシンで同一の環境を作成し, その上でアプリケーションを動作させる必要のある場合などによく利用されている.

また, 環境の作成や廃棄が簡単であることも利点の 1 つである. 複数のサーバを連携させて全体で 1 台のサーバであるかのように動作させるクラスタを構築する場合も、Docker イメージがあれば、それを元に複数の環境（コンテナ）を起動できる. この機能を活用することにより環境を一から作る作業もなくなり、クラスタ構成を構築するのも容易になる. さらに, Kubernetes [23] などのコンテナオーケストレーションシステムを用いてクラスタ環境を管理することも容易になる.

3.2 MD のコンテナ化

前節で紹介したような仮想化技術を用いることで, 異なるマシンでもアプリケーションを同一の環境で動作させる事ができる. この仮想化技術を用いて, 2 章で先行研究として紹介した SBC を用いたマルチディスプレイシステムを仮想化することを考える.

先行研究では, SBC マルチディスプレイシステムにおいてディスプレイノードに Raspberry Pi を使用している. しかし, 2022 年 2 月現在においては SBC の開発は盛んに行われており, 市場には数百種類もの SBC が存在している [24]. それらの SBC の間ではア-

キテクチャや OS, ディスプレイの描画方法, 対応言語などに違いが存在する. そのため, これらの SBC を用いてマルチディスを構築すると構築の手順や動作に違いが発生する事が考えられる. そのため, 異なる SBC を用いた際に発生するこれらの差異を吸収するため, 様々な SBC に対応することができるようなミドルウェアの開発が必要である. そこで, 前節で紹介したコンテナ仮想化技術である Docker の利用を検討する.

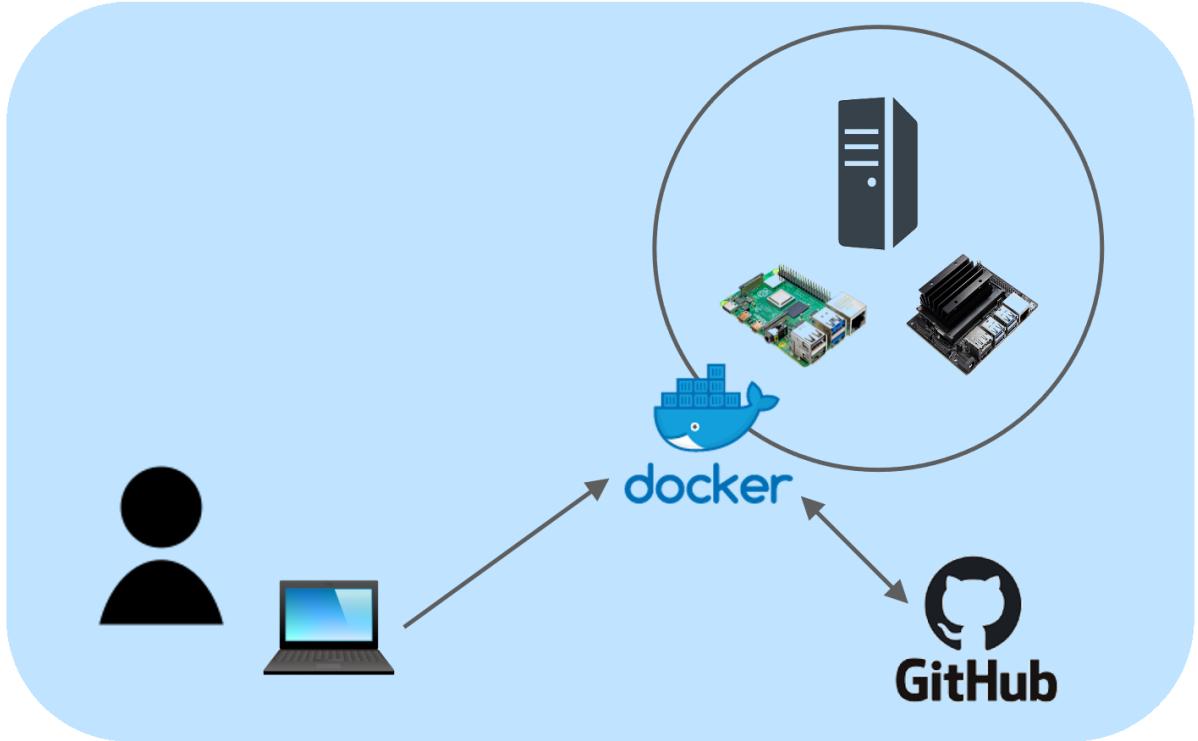


図 3.2: Docker を用いた環境構築

Docker を用いた環境構築の概要図を図 xxx に示す. ユーザは, まずヘッドノードもしくはディスプレイノードとして利用するマシンに対して Docker のインストール作業を行う. 次に, Docker のコマンドを用いて GitHub 上のリポジトリからソースコードを取得し, その中に含まれている Dockerfile を用いてホストマシン内にコンテナ環境を作成する. ここで Dockerfile とはコンテナ環境の構成情報 (OS, パッケージ, ネットワーク設定情報など) を保存したファイルであり, これを用いてコンテナ環境を構築することにより簡単にあらかじめ準備されていた環境を構築する事ができる. 最後に, ホストマシン上に構成したコンテナ環境内でプログラムのビルドを行う事で, そのマシンをヘッドノードもしくはディスプレイノードとして使用する事が可能になる.

3.3 コンテナを通したディスプレイの制御

本節では, コンテナ技術を用いて構築した MD における, フレームのディスプレイ時に生じる問題とその対処について説明する. マシンに接続されたディスプレイに画像を表示

する際には、一般にはフレームバッファと呼ばれる領域が使用される。フレームバッファは /dev ディレクトリに存在するデバイスファイルである。このファイルにディスプレイに表示するデータを格納することで、OS がディスプレイの画像を描画するという仕組みである。フレームバッファを用いたディスプレイへの画像表示の概要を図 xxx に示す。



図 3.3: ハイパーバイザ型仮想化とコンテナを用いた OS 仮想化

Docker を用いて構築したコンテナ環境の内部からは、ホストマシンのリソースへのアクセスが制限される場合がある。例えば、デフォルトの状態では Docker コンテナは Docker コンテナの内部で Docker daemon の起動を行うことができない。これは、デフォルトではコンテナからホストマシンのデバイスへのアクセスが許されていないためである。同様に、フレームバッファも Docker コンテナの内部から見ればホストマシンのデバイスファイルであるため、ファイルが操作できずにディスプレイへの画像表示が不可能になっている。この状態を、コンテナは unprivileged な状態であるという。

対して、privileged なコンテナはホストマシンの全てのリソースにアクセスする事が可能である。コンテナを privilege な状態で起動するのは、起動時のコマンドで --privileged コマンドを指定する必要がある。

3.4 ヘッドノードでのフレーム処理の改良

ここまでで、SBC マルチディスプレイシステムをコンテナ基盤上で動作させる事ができた。続いて本節以降では、本研究の提案の中心部分であるヘッドノードでのフレーム処理の改良について述べる。

2 章でも説明した通り、先行研究で提案されたマルチディスプレイには高解像度な動画や、高フレームレートな動画を MD 上に表示しようとするとヘッドノードでの処理がボ

トルネックとなり動画表示に遅延が発生するという課題がある。高解像度な動画はフレームの分割や圧縮にかかる時間が大きくなるため、ヘッドノード内の処理時間が増加し、フレームの表示処理への影響も大きくなる。また、高フレームレートな動画の場合には、フレーム処理時間の影響により元々のフレームレートを維持たまま動画を表示することが困難になる。さらに、大規模なMDを構築した際の性能低下も課題点としてあげられる。先行研究のMDではヘッドノード内でフレーム圧縮を行うスレッドが1つであるため、MDを構成するディスプレイ数が増加するのに伴ってスレッドでの処理負荷が増加する。その結果、MDの大規模化に伴ってフレーム処理にかかる時間も増加し、動画再生時のフレームレートが低下する。

この問題に対して提案では、ヘッドノードでのフレーム処理を並列化したプロセスとして実装し、ヘッドノード内の処理時間を短縮する。

フレーム処理のコンテナ並列化

提案手法の実装にはOS仮想化技術を用いたコンテナ技術を使用し、プロセスレベルでの並列化を図ります。また、処理の並列化に伴うプロセス間での画像フレームの受け渡しについては、高速なプロセス間通信を行える共有メモリを使用することでオーバーヘッドの低減を目指します。

先行研究では、ヘッドノード内の圧縮スレッド内でフレーム切り出し、フレーム分割、フレーム圧縮の3種類の処理が行われている。

フレーム切り出しとフレーム分割はディスプレイ数に関わらず1フレームに対して1回の処理のみが行われるが、フレームの圧縮はマルチディスプレイを構成するディスプレイ数に応じて処理回数が変化するため、この部分がシステムのボトルネックとなる。このボトルネックを解消するために、圧縮処理をヘッドノード内で並列化して行い、全体での処理時間を短縮する。

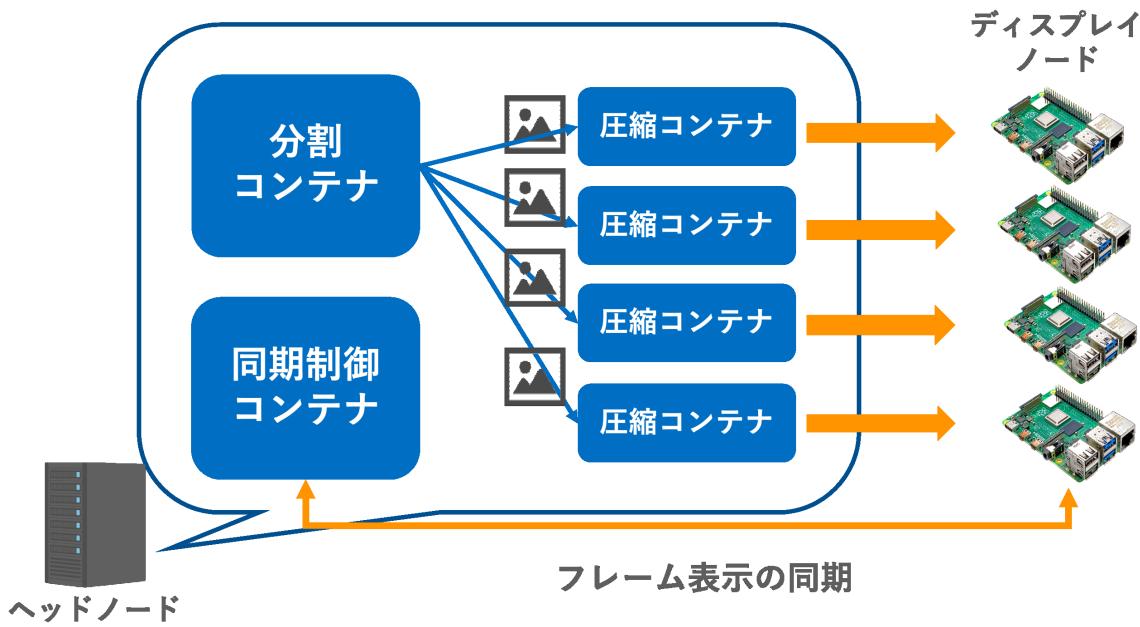


図 3.4: フレーム処理の並列化

提案手法では、ヘッドノード内で行われる処理それぞれを単一のコンテナに分割し、画像フレームの切り出し・分割を行う分割コンテナ、フレームの圧縮処理を行う圧縮コンテナ、そしてディスプレイノードとの同期用通信を行う同期制御コンテナの3種類のコンテナとして実装する。圧縮コンテナは構成ディスプレイと同じ数だけ用意し、ヘッドノード内で並列化してフレームの圧縮処理を行う。

ヘッドノードでのコンテナを用いたフレーム処理の並列化について書くメインコンテナ、圧縮コンテナの実装や動作などについて図やフローチャートを用いて解説する

コンテナ間でのフレーム受け渡し処理

処理の並列化を目的としてフレームの分割を行うコンテナと圧縮を行うコンテナを分けたことにより、ヘッドノード内のコンテナ間で画像フレームの受け渡しを行う必要が生じる。この処理によるオーバヘッドを抑えるために、高速なプロセス通信が可能な共有メモリ (System V IPC) [25, 26] を使用する。

以下、共有メモリ (System V IPC) について簡単に説明する。IPC とはプロセス間通信 (InterProcess Communication) の略であり、ユーザモードプロセスから

- セマフォを利用した他のプロセスとの同期
- 他のプロセスとの間でのメッセージ送受信
- 他のプロセスとのメモリ領域の共有

などの操作を行う事ができる。

System V IPC は、現在では Linux を含むほとんどの UNIX システムで使用できるようになっている。IPC のデータ構造は、プロセスが IPC 資源（セマフォ、メッセージキュー、共有メモリリージョン）を要求した際に動的に作成される。IPC 資源を要求したプロセスが獲得した資源を明示的に解放しない限り IPC 資源はメモリ上に残り続け、他のどのプロセスからでも使用できる状態になる。各資源は、ファイルシステムツリーにおけるファイルのパス名に相当する 32 ビットの IPC キーによって識別される。また、各 IPC 資源は 32 ビットの IPC 識別子を持つ。IPC 資源はカーネルによって決定されるが、IPC キーは自由に決める事が可能である。複数のプロセスが IPC 資源を利用する際には、IPC 識別子が利用される。本研究の実装では System V IPC を利用して共有メモリ領域を獲得し、異なるプロセス間でのデータ共有を高速に行うこととする。

続いて、IPC 資源の利用方法について説明する。System V IPC を用いて共有メモリ領域を獲得するのは、`shmget()` 関数を用いる。`shmget()` 関数は引数として渡された IPC に対応する IPC 識別子を取得し、その IPC 識別子を利用してプロセスが共有メモリ領域にアクセスすることができるようになる。別々のプロセスから同一の IPC 資源を共有するための方法は 2 通り存在する。1 つはプロセス間であらかじめ固定の IPC キーを決定しておく方法である。この方法は単純であるため、多くのプロセスが関連する複雑なアプリケーションで利用してもうまく動作する。しかし、全く関係ないプロセスが偶然同じ IPC キーを使用してしまう可能性がある。もう 1 つは、一方のプロセスで IPC キーに `IPC_PRIVATE` を指定する方法である。この方法では新規の IPC 資源が割り当てられ、その IPC 識別子によってアプリケーション内の他のプロセスとの通信を行うため、他のプロセスから誤って IPC 資源を利用されてしまうことを防げる。

以上の方を検討した結果、本研究では IPC 資源が複数のプロセスから参照されるため、前者の手法を採用して実装を行った。

IPC 共有メモリ

IPC 共有メモリでは、共有するデータ構造を IPC 共有メモリリージョンに配置することにより、複数のプロセスから共有データ構造にアクセスする事ができる。以下、IPC 共有メモリの使用方法について述べる。プロセス 1 とプロセス 2 の 2 つの異なるプロセスの間で共有メモリ領域を作成するメモリ領域を作成するとする。まずプロセス 1 はプロセス 2 との間で一意となる IPC キーを作成する。そして、このキーを用いて `shmget()` 関数により IPC 識別子を取得する。この時に、獲得する共有メモリ領域のサイズやパーミッションなどを決める事ができる。続いて、IPC 識別子を引数として `shmat()` 関数を実行することにより、共有メモリがプロセス 1 のメモリ領域にアタッチされ、プロセス 1 から共有メモリ領域へのアクセスが可能になる。

もう一方のプロセス 2 では、プロセス 1 が生成した IPC キーを取得する。続いてプロセス

ス 1 と同様に取得した IPC キーを引数として `shmget()` 関数を実行し、共有メモリの IPC 識別子を取得する。その後、IPC 識別子を引数として `shmat()` 関数を実行して共有メモリをプロセス 2 のアドレス空間にアタッチすることでプロセス 2 からもプロセス 1 と同じ共有メモリ領域を使用することができるようになる。以上に述べたような一連の手続きを行うことで、作成した共有メモリ領域を利用して異なるプロセスとの間でデータを共有することが可能になる（図 3.5）。

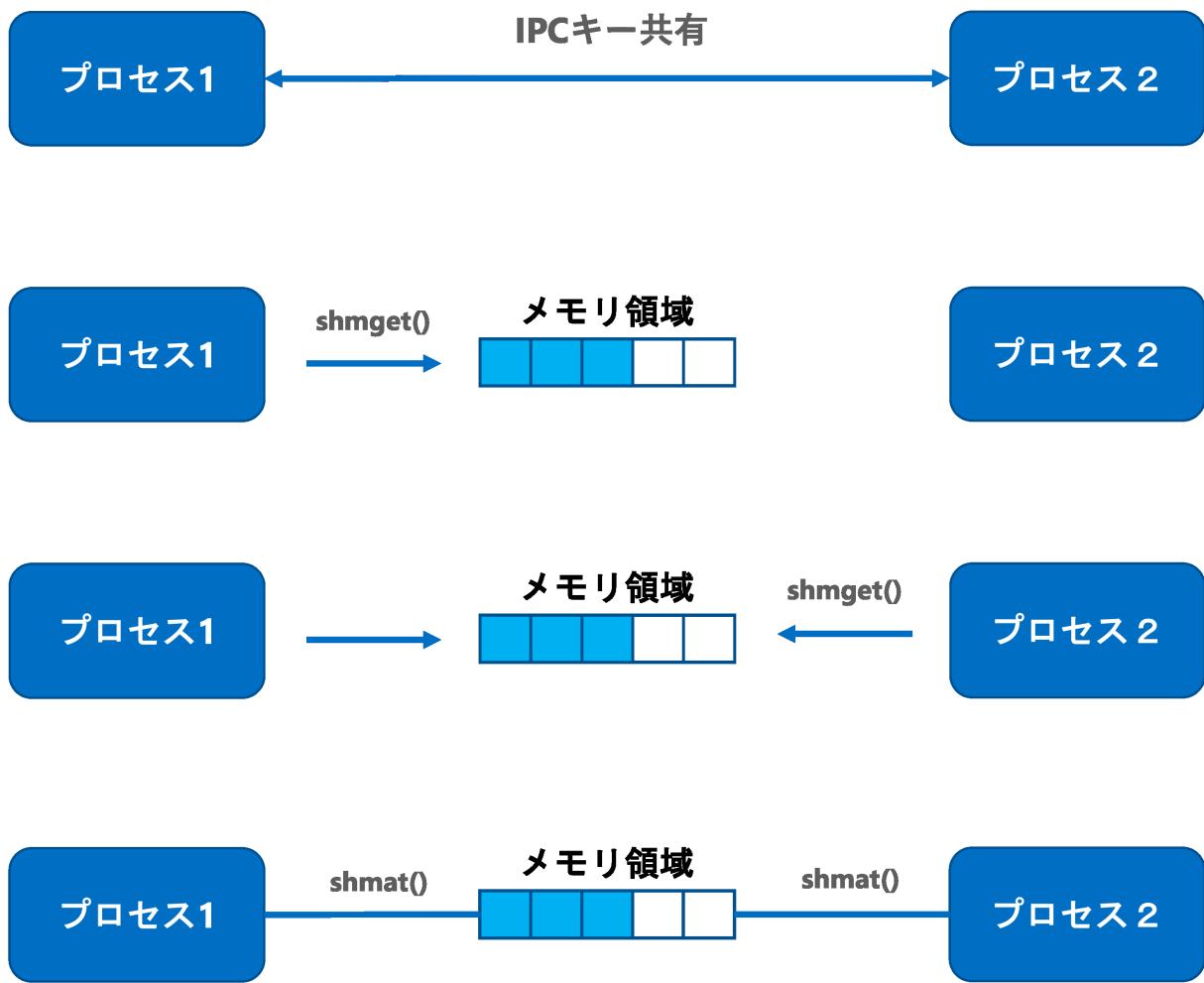


図 3.5: 共有メモリ使用の流れ

フレーム受け渡し機能の実装

続いて、フレーム受け渡し機能の具体的な実装について説明する。共有メモリは、ヘッドノード内で動作する分割コンテナと圧縮コンテナとの間で画像フレームデータを共有するために利用する（図 3.6）。分割コンテナは、動画ファイルから画像フレームの切り出し処理を行う。画像処理にはオープンソースのコンピュータビジョンライブラリである OpenCV を使用する。OpenCVにおいて画像フレームは `Mat` という型で管理される。`Mat`

型は画像フレームに関するパラメータを持った構造体であり、画像フレームの縦横ピクセル数、画像フレームの色深度、画像フレームのデータへのポインタなどをメンバを持つ。Mat 構造体の内容を図 3.7 に示す。



図 3.6: OpenCV における Mat 構造体

プロセス間でのフレームデータ共有を行うためには、この Mat 型の構造体を共有メモリに格納し、プロセス間で共有できるようにする事が必要である。しかし、Mat 型の構造体を単純な操作で共有メモリに格納するだけではフレームデータそのものの受け渡しは実現できない。これは Mat 構造体はフレームデータのポインタを持つのみであり、フレーム切り出しの際に画像フレームの画素データ本体が格納されるのはプロセス固有のメモリ空間であるためである。そこで、本実装では、分割コンテナと圧縮コンテナとの間に作成した共有メモリ領域を char 型の配列とし、Mat 構造体のもつフレームデータへのポインタの値を利用して memcpy() 関数によるコピーを行っている。memcpy() 関数は引数を 3 つ持ち、引数 1 を先頭としたメモリ領域に、引数 2 を先頭としたメモリから引数 3 byte 分のデータをコピーする。memcpy() 関数によるメモリのコピーは非常に高速に動作するため、共有メモリ領域への書き込み時間は小さくなる。

実際の処理では、まず分割コンテナがフレームの切り出しと分割を行った後、圧縮コンテナとの間に作成された共有メモリにフレームデータを格納する。圧縮コンテナは共有メモリに格納されたフレームを取得し、圧縮処理を行う。

共有メモリはヘッドノードの中に 1 つのみを作成し、その中でそれぞれの圧縮コンテナに対してメモリ領域を割り当てるという方式を取る。また、共有メモリにはフレーム切り出しの際に使われる OpenCV の Mat 形式からキャラクタ型の配列に変換してフレーム

データを格納する。

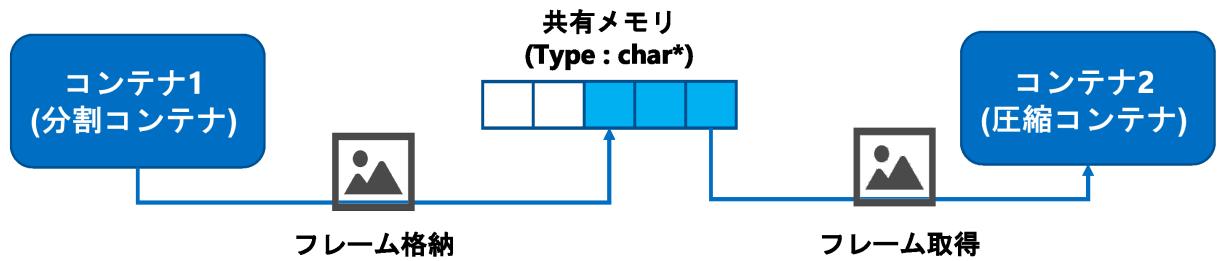


図 3.7: 共有メモリの使用例 2

図 3.7 に提案手法を用いて構築した MD の動作フローを示す。ヘッドノードでは分割コンテナ, 圧縮コンテナ, 同期制御コンテナの 3 種類のコンテナが動作している。まず画像フレームが分割コンテナによって分割され, バッファを通じて圧縮コンテナへ渡される。圧縮コンテナはこのフレームに対して JPEG 圧縮処理を行い, それぞれが接続しているディスプレイノードへと送信する。

ディスプレイノードではフレームを受信したのち展開処理を行い, フレームバッファに書き込むことでディスプレイへ画像を表示する。

ヘッドノードで操作している同期制御コンテナはディスプレイノードからの同期メッセージを受け取り, フレーム表示命令を送信するとともに, 圧縮コンテナ内で行われているフレーム圧縮のパラメータを制御することでフレームレートを一定に保つ役割を持つ。

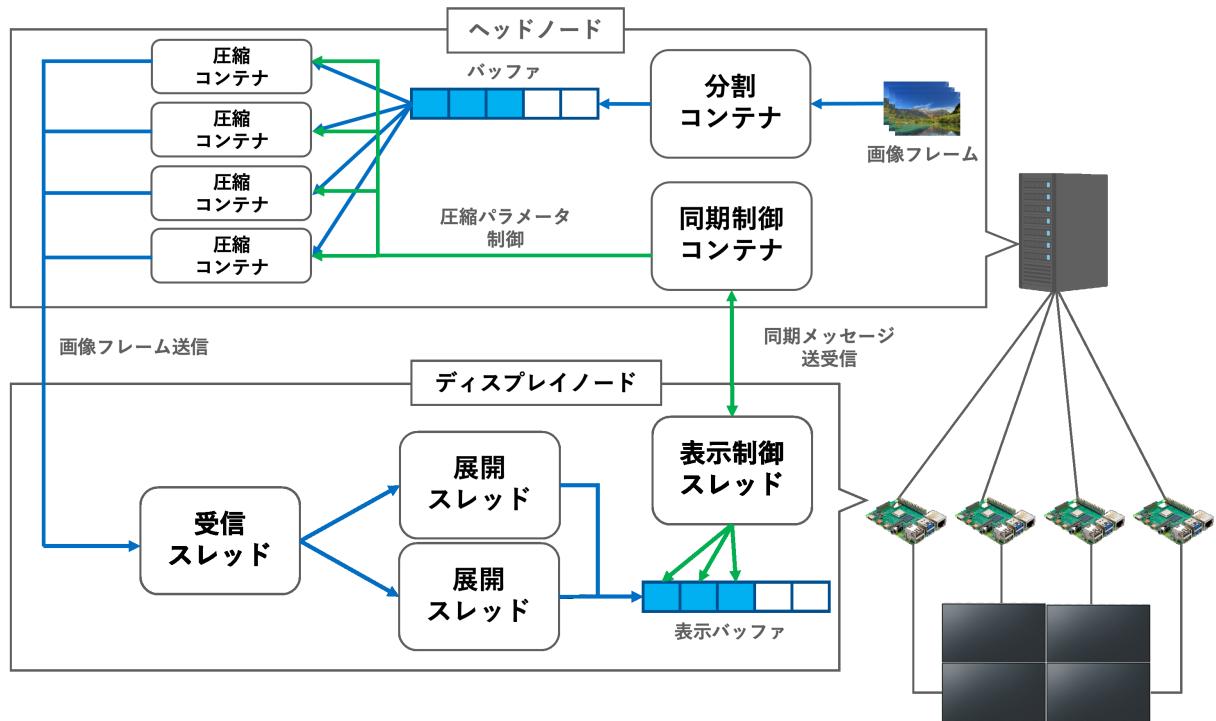


図 3.8: 提案手法を用いた MD の動作フロー

4 評価

本章では、全章で提案したヘッドノードにおけるフレーム処理並列化の有効性を確認するため、ヘッドノード内に構築したコンテナ環境において2種類の実験（実験1、実験2）を行う。実験1では、提案手法は既存手法と比較してヘッドノード内のフレーム処理時間が短縮されているかを確認する。実験2では、ヘッドノードの各CPUコアの使用率を計測し、提案手法を用いてもうちくしたマルチディスプレイシステムのスケーラビリティについての確認を行う。本章では、まず4.1節で、分評価で利用した実験環境について説明する。次に、4.2節で、実験1における実験方法と実験結果について述べる。さらに、4.3節では、実験2における実験方法と実験結果を述べる。

4.1 実験環境

本実験で用いたヘッドノード内のコンテナ構成を図4.2に示す。当実験環境は、ヘッドノードとして用いるデスクトップPC内のコンテナ仮想環境を用いて構築したものである。実験環境内では1つの分割コンテナと複数の圧縮コンテナを用意する。また、本実験では様々な画面構成のマルチディスプレイを構築することを想定し、圧縮コンテナの数は構成ディスプレイと同じ数になるように増減させる。本実験では1面、2面、4面、6面、9面構成のそれぞれのマルチディスプレイを構築することを想定し、評価を行った。

表4.1: ヘッドノード用デスクトップPCの仕様

要素	仕様
CPU	i7-5960X (3.0 GHz × 8)
メモリ	64.0 GB
通信帯域	1 Gbps
OS	CentOS 7.3

評価に用いる映像としては、クリエイティブ・コモンズ・ライセンスのもとで利用できる3DアニメーションであるBig Buck Bunny [27]を使用した。表4.4に、Big Buck Bunnyのプロパティを示す。さらに、図4.3にBig Buck Bunny再生時のスクリーンショットを示す。

表 4.2: Big Buck Bunny のプロパティ

項目	内容
動画形式	MP4
コーデック	H.264 [28]
長さ	10 分 34 秒
総フレーム数	19020 枚
フレームレート	30 fps
解像度	4K (3840 × 2160)



図 4.1: Big Buck Bunny のスクリーンショット

ヘッドノードで行われる処理の実装には, OpenCV [29], libjpeg-turbo [30], Boost.Asio [31] という 3 種類のライブラリを利用した. OpenCV は画像処理ライブラリ, libjpeg-turbo は JPEG の圧縮と展開を行うライブラリ, Boost.Asio はソケット通信を行うライブラリである. また, ディスプレイノードの処理の実装には libjpeg-turbo, Boost.Asio, fbdev を利用した. 実装には C++ 言語を使用し, GCC コンパイラ (GNU C Compiler) [32] でコンパイルを行った.

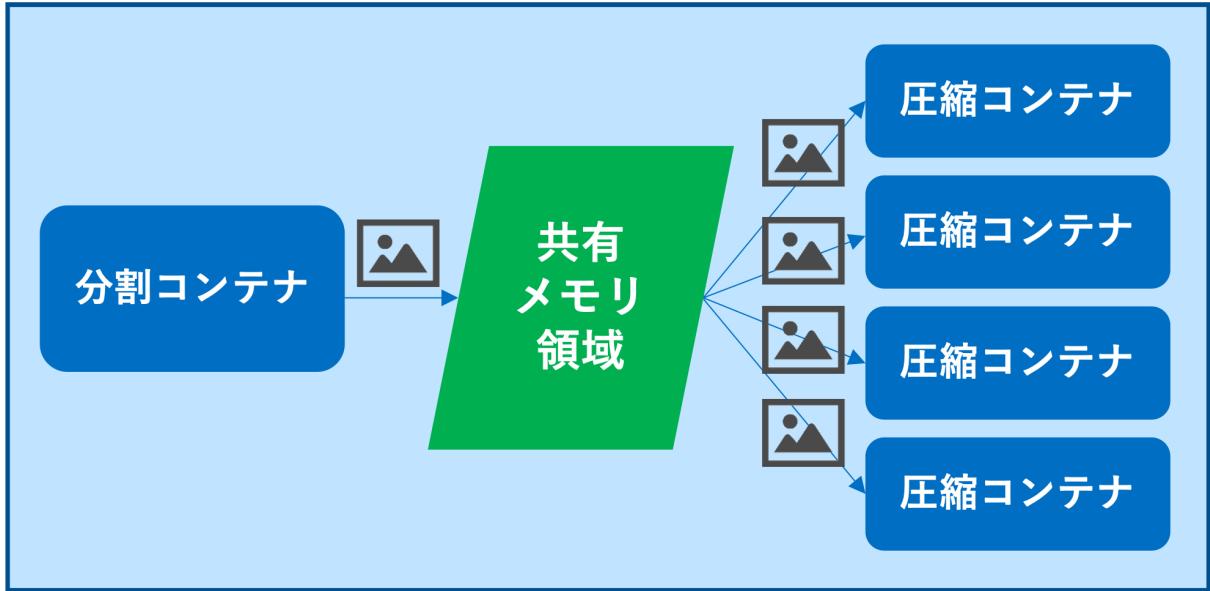


図 4.2: 実験環境のコンテナ構成

4.2 フレーム処理時間の評価

4.2.1 実験方法

先行研究で提案された手法と提案手法において、4面構成のMDを想定した環境で4K解像度の画像を表示し、動画のフレーム開始から1000フレーム目の処理が終了するまでの1フレームあたりに要するフレーム処理時間を計測した。また、フレーム処理時間の計測にはC++11の時間ライブラリである chrono を用いた。図4.6に先行研究で提案されたシステムと提案手法でのフレーム処理時間を示す。

4.2.2 実験結果

既存手法ではヘッドノードでのフレーム処理時間の平均が52.35msであったのに対し、提案手法では21.72msとなっており、フレーム処理時間が既存手法と比較して40%程度まで短縮された事が確認できた。

また、本提案で新たに生じた処理である共有メモリを用いたプロセス間での画像フレームデータ受け渡しに要する時間も平均で1ms未満となっており、オーバーヘッドは非常に小さくなっていることも確認できる。

以上の結果より、フレーム処理プロセスの並列化を行った提案手法によって、システムのボトルネックが解消された。

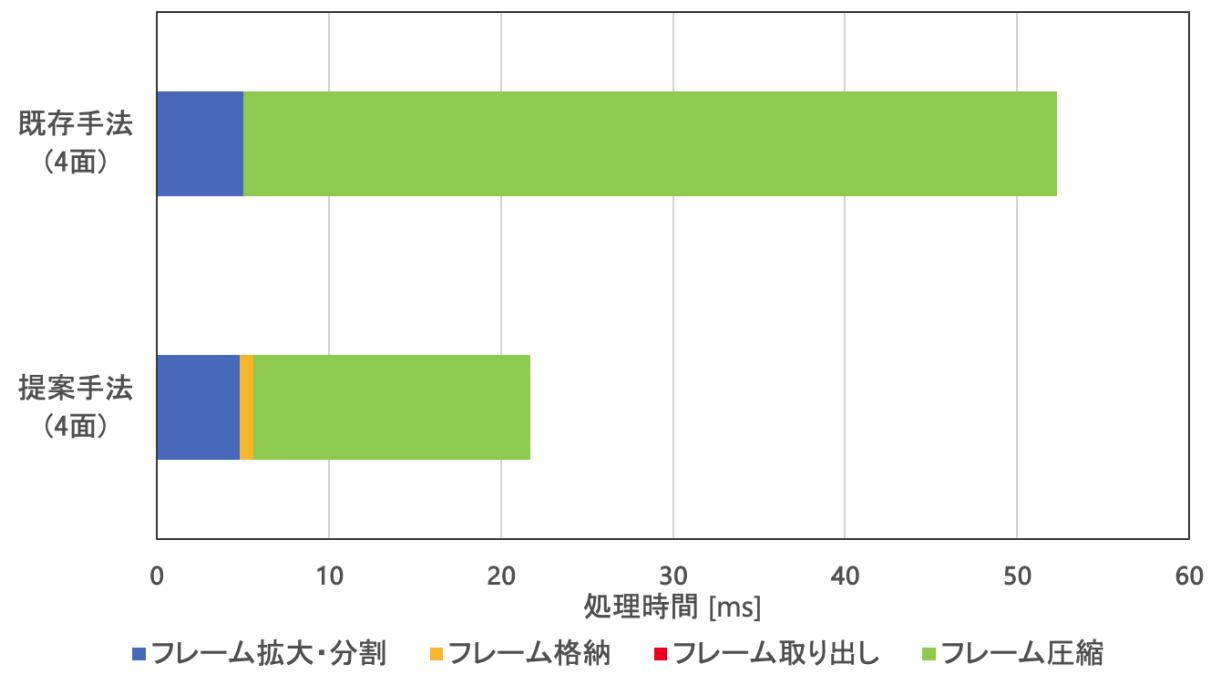


図 4.3: フレーム処理時間の比較 (4面構成時)

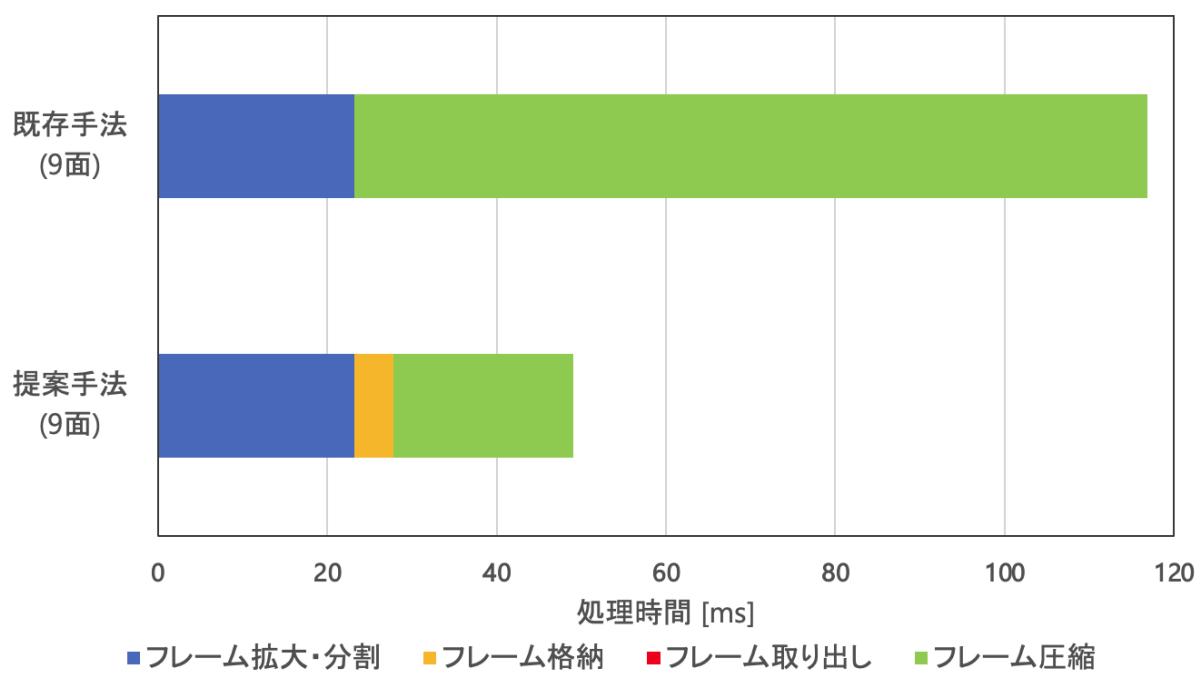


図 4.4: フレーム処理時間の比較 (9面構成時)

4.3 MD のコンテナ化の評価

4.3.1 実験方法

4.3.2 実験結果

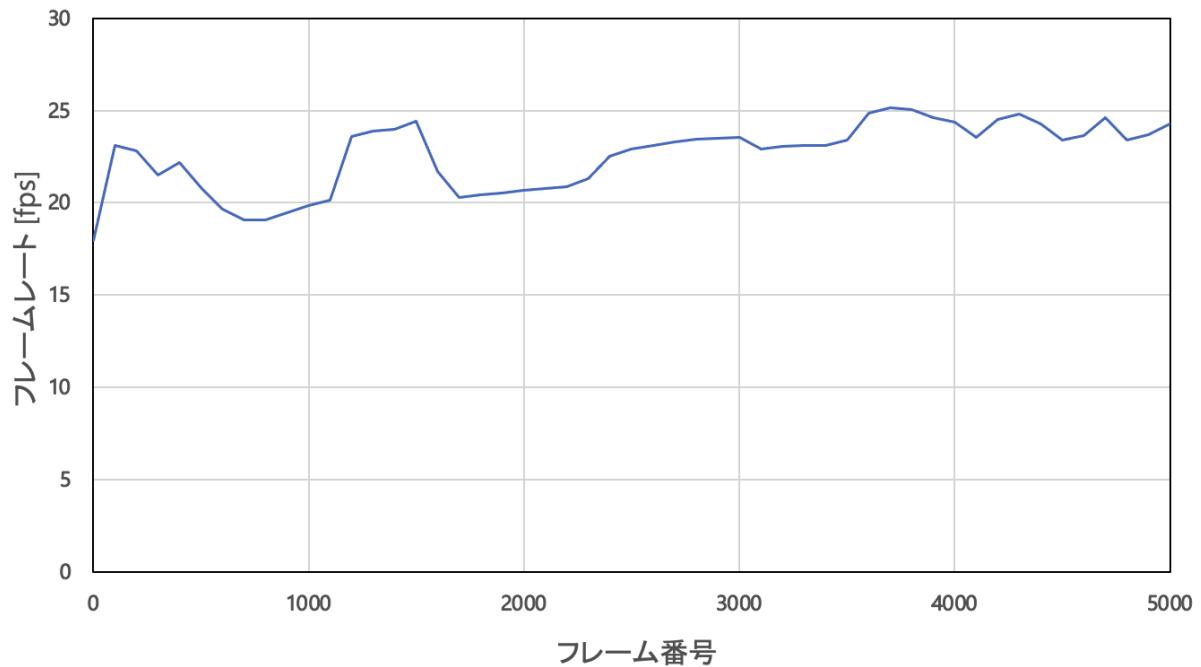


図 4.5: フレームレートの時間変化

4.3.3 結果に対する考察

5 関連研究

大規模ディスプレイシステムにおいて動画再生時のフレームレート向上を試みた例としては、Bundulis らの研究 [33] がある。Bundulis らは、以前に Infiniviz [34] とよばれる大規模ディスプレイシステムを提案している。Infiniviz は、仮想マシンベースの高解像度ディスプレイウォールシステムであり、既存の他の大規模ディスプレイシステムと比較して、ネットワーク帯域幅の消費量と計算性能を向上させようとするものである。また、シームレスな方法で可視化タスクにアプローチし、大規模な高解像度ディスプレイウォール上で、一般的なデスクトップオペレーティングシステムのソフトウェアを変更することなく実行できるようにすることを目的としている。Bundulis らの研究では、Quake 3 Arena を解像度 9600x5400, 24fps で動作させた際の Infiniviz の実際の性能を計測した。この研究では、仮想化を利用することで、ソフトウェアに依存しない仮想マシンベースの高解像度ディスプレイウォールシステムを構築している。

6 結論

謝辞

本研究を行うにあたり、懇切なる御指導、御鞭撻を賜りました大阪大学サイバーメディアセンター 下條真司教授に心より感謝の意を表します。

本研究の全過程を通じ、研究の方向性検討や論文の執筆等、熱心な御指導、御鞭撻を賜りました大阪大学サイバーメディアセンター応用情報システム研究部門 木戸善之講師に心より感謝申し上げます。

本研究の進行から論文執筆まで多岐にわたり、手厚い御指導、御鞭撻を賜りました大阪大学サイバーメディアセンター応用情報システム研究部門 伊達進准教授、小島一秀講師に心より感謝の意を表します。

本研究において、種々のご意見、ご指導を賜りました大阪大学情報推進本部 大平健司准教授に心より感謝いたします。

本研究において、厳しくも優しい御指導を賜りました大阪大学データビリティフロンティア研究機構 春本要教授に心より感謝の意を表します。

最後に、日々御支援と心からの激励をかけて頂いた下條研究室の皆様ならびに事務補佐員 片岡小百合氏に心より感謝いたします。

参考文献

- [1] Muhammad Saleem, Hugo E Valle, Stephen Brown, Veronica I Winters, and Akhtar Mahmood. The hiperwall tiled-display wall system for big-data research. *Journal of Big Data*, Vol. 5, No. 1, pp. 1–45, 2018.
- [2] Simon Su, Vincent Perry, Nicholas Cantner, Dylan Kobayashi, and Jason Leigh. High-resolution interactive and collaborative data visualization framework for large-scale data analysis. In *2016 International Conference on Collaboration Technologies and Systems (CTS)*, pp. 275–280. IEEE, 2016.
- [3] Luciano Soares, Bruno Raffin, and Joaquim Jorge. PC clusters for virtual reality. *IJVR*, Vol. 7, pp. 67–80, Jul. 2008.
- [4] H. Chung, C. Andrews, and C. North. A survey of software frameworks for cluster-based large high-resolution displays. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 20, No. 8, pp. 1158–1177, Aug. 2014.
- [5] 中村伊知哉. デジタルサイネージの動向. 情報管理, Vol. 55, No. 12, pp. 891–898, Mar. 2013.
- [6] 長野英生. Displayport. 映像情報メディア学会誌, Vol. 66, No. 11, pp. 942–945, May 2012.
- [7] L. Renambot, A. Rao, R. Singh, B. Jeong, N. Krishnaprasad, V. Vishwanath, V. Chandrasekhar, N. Schwarz, A. Spale, C. Zhang, G. Goldman, J. Leigh, and A. Johnson. SAGE : the scalable adaptive graphics environment. *Proceedings of the 4th Workshop on Advanced Collaborative Environments (WACE2004)*, Sep. 2004.
- [8] Luc Renambot, Thomas Marrinan, Jillian Aurisano, Arthur Nishimoto, Victor Mattevitsi, Krishna Bharadwaj, Lance Long, Andy Johnson, Maxine Brown, and Jason Leigh. SAGE2: A collaboration portal for scalable resolution displays. *Future Generation Computer Systems*, Vol. 54, pp. 296–305, Jan. 2016.
- [9] G. P. Johnson, G. D. Abram, B. Westing, P. Navr’til, and K. Gaither. Displaycluster: an interactive visualization environment for tiled displays. In *Proceedings of the 16th International Conference on Cluster Computing*, pp. 239–247, Sep. 2012.
- [10] Petr Holub, Martin Srom, Martin Pulec, Jiri Matela, and Martin Jirman. GPU-accelerated DXT and JPEG compression schemes for low-latency network transmis-

sions of HD, 2K, and 4K video. *Future Generation Computer Systems*, Vol. 29, pp. 1991–2006, Oct. 2013.

- [11] G. K. Wallace. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics*, Vol. 38, No. 1, pp. 18–34, Feb. 1992.
- [12] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH*, Vol. 21, pp. 693–702, Jul. 2002.
- [13] 小林敬. IoT を指向したシングルボードコンピュータの食品成分分析およびセンシングへの応用. 日本食品工学会誌, Vol. 20, No. 3, pp. 107–113, Oct. 2019.
- [14] M. S. D. Gupta, V. Patchava, and V. Menezes. Healthcare based on IoT using Raspberry Pi. In *Proceedings of the International Conference on Green Computing and Internet of Things (ICGCIoT2015)*, pp. 796–799, Oct. 2015.
- [15] Raspberry Pi 4 Model B. <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>, Feb. 2022.
- [16] Orange Pi Zero Plus2. <http://www.orangepi.org/OrangePiZeroPlus2/>, Feb. 2022.
- [17] Banana Pi オープンソースプロジェクト. <http://www.banana-pi.org/m64.html>, Feb. 2022.
- [18] Raspbian. <https://www.raspbian.org/>, Feb. 2022.
- [19] 石田和也. 小型低性能算機向けマルチディスプレイ構築用ミドルウェア, Feb. 2019.
- [20] Noor Ibraheem, Mokhtar Hasan, Rafiqul Zaman Khan, and Pramod Mishra. Understanding color models: A review. *ARPN Journal of Science and Technology*, Vol. 2, pp. 265–275, Jan. 2012.
- [21] P. T. Chiou, Y. Sun, and G. S. Young. A complexity analysis of the JPEG image compression algorithm. In *Proceedings of the 9th Computer Science and Electronic Engineering (CEEC2017)*, pp. 65–70, Sep. 2017.
- [22] Empowering app development for developers — docker, Feb. 2022.
- [23] Kubernetes. <https://kubernetes.io/>, Feb 2022.
- [24] The single board computer database. <https://hackerbords.com/>, Feb. 2022.

- [25] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.
- [26] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* ” O'Reilly Media, Inc.”, 2005.
- [27] Big Buck Bunny. <https://peach.blender.org/>, Feb. 2022.
- [28] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 13, No. 7, pp. 560–576, Jul. 2003.
- [29] OpenCV. <https://opencv.org/>, Feb. 2022.
- [30] libjpeg-turbo. <https://libjpeg-turbo.org/>, Feb. 2022.
- [31] boost C++ libraries. <https://www.boost.org/>, Feb. 2022.
- [32] GCC,GNU コンパイラコレクション. <https://gcc.gnu.org/>, Feb. 2022.
- [33] Rudolfs Bundulis and Guntis Arnicans. Infiniviz: Taking quake 3 arena on a large-scale display system to the next level. In *2018 23rd Conference of Open Innovations Association (FRUCT)*, pp. 91–98. IEEE, 2018.
- [34] Rudolfs Bundulis and Guntis Arnicans. Infiniviz-virtual machine based high-resolution display wall system. In *DB&IS (Selected Papers)*, pp. 225–238, 2016.