

# 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独立したプロセスとして複数フレームを同時処理できるように並列化した実装とすることで、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	3
2.1	従来のマルチディスプレイ構築手法 . . . . .	3
2.2	シングルボードコンピュータ (SBC) . . . . .	7
2.3	先行研究で提案されたマルチディスプレイシステム . . . . .	8
2.4	先行研究の問題点と技術的課題 . . . . .	11
2.5	関連研究での技術的課題 . . . . .	16
3	提案	18
3.1	OS 仮想化基盤とコンテナ技術 . . . . .	18
3.2	マルチディスプレイシステムのコンテナ化 . . . . .	19
3.3	コンテナ環境におけるディスプレイの制御 . . . . .	21
3.4	ヘッドノードでのフレーム処理の改良 . . . . .	22
4	評価	33
4.1	実験環境 . . . . .	33
4.2	実験 1: フレーム処理時間に関する評価 . . . . .	36
4.3	実験 2: スケーラビリティに関する評価 . . . . .	37
4.4	実験 3: コンテナの起動時間に関する評価 . . . . .	39
4.5	実験 4: マルチディスプレイシステムのコンテナ化に関する評価 . . . . .	39
5	関連研究	43
6	結論	45
6.1	本研究の総括 . . . . .	45
6.2	将来課題 . . . . .	45
	謝辞	47
	参考文献	48

## 1 序論

近年では、デジタル化の進展により大量のデータを取得できるようになり、それらのデータを活用して研究機関が行う学術研究が盛んになっている [1]。学術研究においては、個々の研究機関の枠組みを越えて研究設備や研究データを利活用することでより大きな成果が期待できる。このような背景から、大学や研究機関の研究者たちによる共同研究のプロジェクト数が増加傾向にある [2, 3]。

また、最近では新たなスーパーコンピュータの開発による計算能力の向上を背景に、シミュレーションベースで行われる研究・プロジェクトの数が増加している。KBDD (K supercomputer-Based Drug Discovery project by Biogrid pharma consortium) プロジェクト [4, 5] は、新薬の開発を目指して立ち上げられた共同研究プロジェクトであり、タンパク質と化合物の結合可能性の予測に高性能計算機による大規模シミュレーションの結果が活用されている。国立研究開発法人海洋研究開発機構 (JAMSTEC) は、東京大学地震研究所と協力して「富岳」 [6] を用いた大規模数値シミュレーションによる地震・津波による複合災害の統合的予測システムを開発している [7, 8]。また、同様に「富岳」を用いて行われた COVID-19 の飛沫・エアロゾル拡散モデルシミュレーションに関する共同研究プロジェクト [9, 10] では、ゴーダンベル賞を受賞するなど目覚ましい成果を挙げている。

以上に挙げたような大規模シミュレーションを用いた共同研究では、行われたシミュレーションの結果を可視化し、複数人の研究者たちで議論を交わすことが必要となる。そのため、複数人で同時に閲覧できるような大型の高解像ディスプレイが不可欠である。高解像のディスプレイを構築する方法として、格子状に配置した複数のディスプレイを1台の仮想ディスプレイとして構成するマルチディスプレイシステム (MD) がある。MD は 4K などの高解像な動画や画像を大画面で表示することが可能であるため、多数の研究者が議論する場などで利用される。MD の導入事例を挙げると、大阪大学サイバーメディアセンターでは 24 面ディスプレイで構成される MD を吹田本館に、15 面の MD をうめきた VisLab に設置し運用しており、大規模シミュレーションやデータ同化シミュレーションの解析結果を可視化する高解像度ディスプレイとして活用している [11]。また名古屋大学情報基盤センターでは、16 面 MD を用いた可視化装置と連動する大規模情報システムを運用している [12]。

しかし、MD の構築および運用コストは高くなる傾向にある。MD の構築方法には、主に 2 通りの手法が存在する。まずマトリックススイッチャを用いた MD 構築用ハードウェアを用いる方法であるが、可視化用途の専用機器は一般に高価であるため、MD 構築にかかる費用が高くなる。もう一方は SAGE2 [13] などに代表される MD 構築用ミドルウェアを用いる方法である。この手法では、ディスプレイに映像を出力するための計算機を複数台用意し、それらとは別の制御用の計算機を用意する構成となる。ディスプレイを接続し

た計算機では、3Dレンダリングや描画を行うため、一般的に高価なグラフィックボードを用いる。また、制御用計算機では画像もしくは可視化データの送受信が頻繁に発生するため、高性能なネットワーク機器が用いられる。そのため、MDの構築および運用コストは高くなる。

こうしたMDの構築コストに関する問題に対して、先行研究でシングルボードコンピュータ(SBC)を用いて構築したMDが提案された[14]。MDの構築にSBCを用いることにより、ディスプレイノードの構築単価は抑えることができ、全体として構築コストを下げることができる。先行研究で提案されたMDは、画像フレームの分割・圧縮・送信、通信の制御を行うヘッドノードと、分割された画像フレームを描画するディスプレイノードとで構成されている。まず、ディスプレイノードにSBCを用いるため、ディスプレイノードに高負荷な処理が要求されないようフレーム転送方式による連携表示処理を行っている。ヘッドノードはフレームの分割および圧縮を行ってからフレームを送信し、ディスプレイノードが圧縮済フレームを受信し展開することで動画をディスプレイ上に表示する。

しかし、このMDではヘッドノード内のフレーム処理を单一のプロセスで行っているため、高解像度な動画や高フレームレートな動画を表示する場合には動作に遅延が生じる。また、MDの構成ディスプレイ数を増加させた場合にも動作遅延が生じるため、スケーラビリティにも問題を抱えている。そこで本研究では画像フレームの圧縮を行うプロセスを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圧縮処理を行う手法を提案する。フレーム処理の並列化はOS仮想化基盤を利用し、プロセスをコンテナとして動作させることで実現する。フレーム処理の並列化を行うことで、ヘッドノード内のフレーム処理時間を短縮とMDのスケーラビリティ向上を目指す。

以下、本論文の構成について述べる。2章では、主なマルチディスプレイシステムの構築方法について説明し、低価格なマルチディスプレイ構築方法の重要性について述べる。また、ディスプレイノードにSBCを用いて構築された先行研究を紹介し、その関連研究として自身の卒業研究にも触れる。3章では、本研究の目的を実現するための提案手法である、OS仮想化技術を用いたヘッドノードのフレーム処理並列化について説明し、その設計と実装、関連技術について詳細に説明する。4章では、提案手法の効果を確認するための評価実験について述べる。5章では、仮想化技術を用いてマルチディスプレイシステムを構築した関連研究について述べる。最後に6章で本研究を総括し、今後の課題点について述べる。

## 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 [15, 16]。1 つは専用のハードウェアを用いてマルチディスプレイを構築する手法であり、もう 1 つは専用のミドルウェアを用いてマルチディスプレイを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 [17] 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 [18] 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。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し、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ことが難しい。この点が、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、グラフィックボーディングカード (GPU) を用いた構築法がある。GPU は複数のディスプレイを同時に駆動する機能を持った専用チップであり、複数のディスプレイを同時に駆動するための機能を内蔵している。また、GPU は映像信号の生成や処理を行う機能を持った専用チップであり、映像信号の生成や処理を行う機能を内蔵している。そのため、GPU を用いた構築法は、映像信号の生成や処理を行う機能を内蔵している。そのため、GPU を用いた構築法は、映像信号の生成や処理を行う機能を内蔵している。

ドやマトリックススイッチャがある。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: ハードウェアを用いて構築したマルチディスプレイ

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場合には、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数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、主に 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ディスプレイ上に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。この構築手法では接続可能なディスプレイの数に上限はない。

く、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を構築する事ができる。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るため、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する汎用 PC には一定の処理性能基準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にしたがって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 群を連携することで構築したマルチディスプレイの構成例を示す。

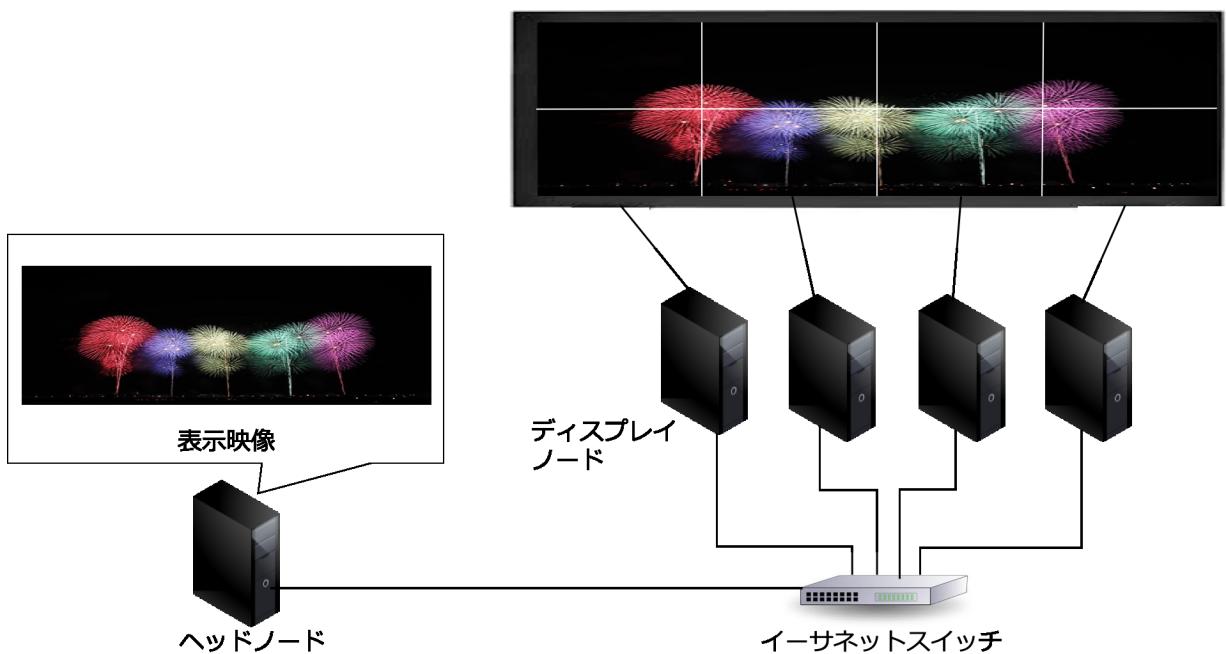


図 2.2: ミドルウェアを用いて構築したマルチディスプレイ

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を実行することで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。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およびヘッドノードと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群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必要とな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て構築し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。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行われる同期処理によって画像フレームをディスプレイ上に連携

表示する方式である。フレーム転送方式を採用したミドルウェアの例として、SAGE [19] や SAGE2 [13] 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 [19]、後者は DisplayCluster [20] 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) [21] や JPEG (Joint Photographic Experts Group) [22] 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後に展開処理を行い、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ると、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 [23] などがあげられる。

描画命令転送方式は、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、ヘッドノードが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。

## 2.2 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 [24, 25]。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	Orange Pi	Banana Pi
	Model B [26]	Zero Plus2 [27]	BPI-M64 [28]
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機種が多いことである。低周波数コアを複数搭載している目的は消費電力を抑えることと価格を低く抑えることであり、こうした省電力や低価格とい



Raspberry Pi Model 4 B [26]      Orange Pi Zero Plus2 [27]      Banana Pi BPI-M64 [28]

図 2.3: 主な SBC の画像

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発売されるようになってきている。映像出力の機能としてはほとんどの機種で 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 [29] も Linux をベースとする OS の一種である。

### 2.3 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担う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が開発された。

本節では、先行研究 [30] 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。

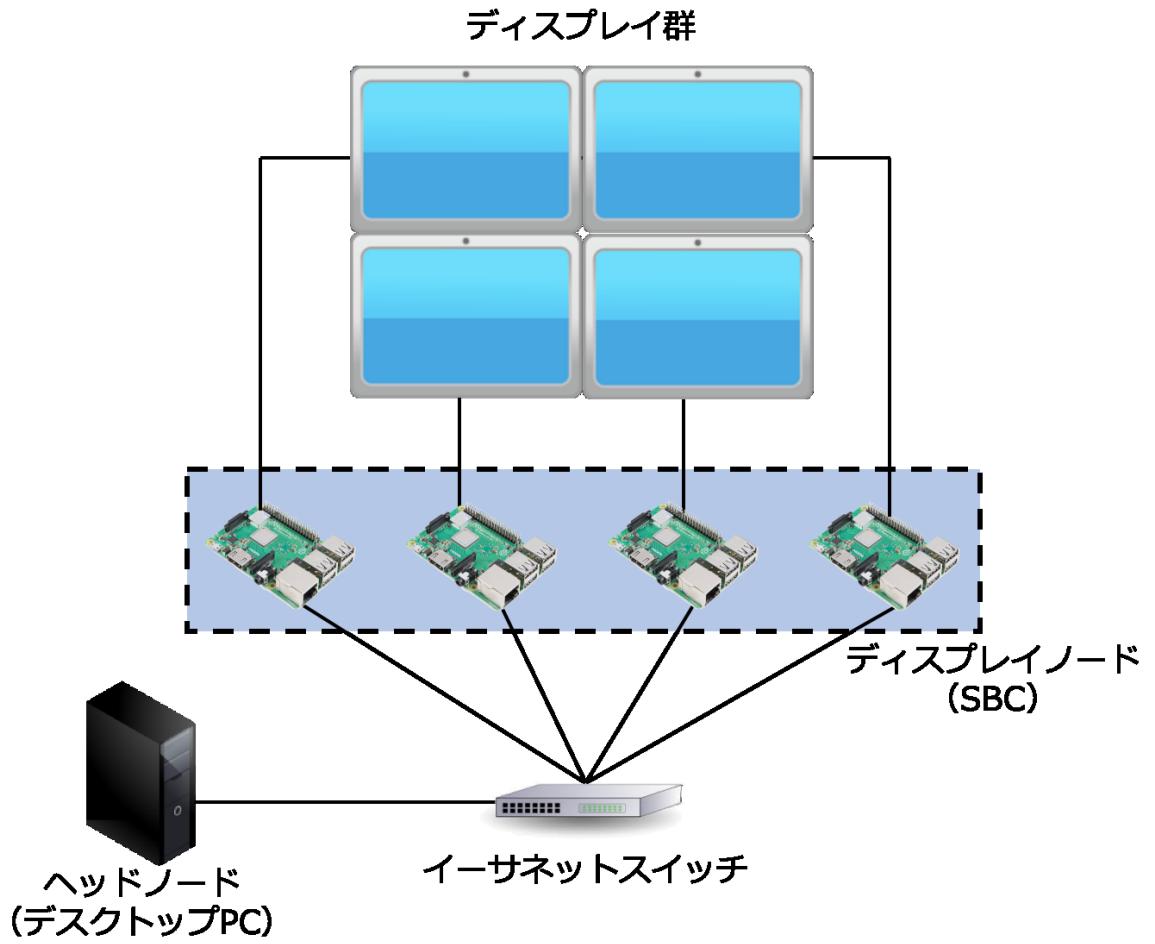


図 2.4: SBC を用いて構築したシステムの概要図

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と比較すると処理速度、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、図2.6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。

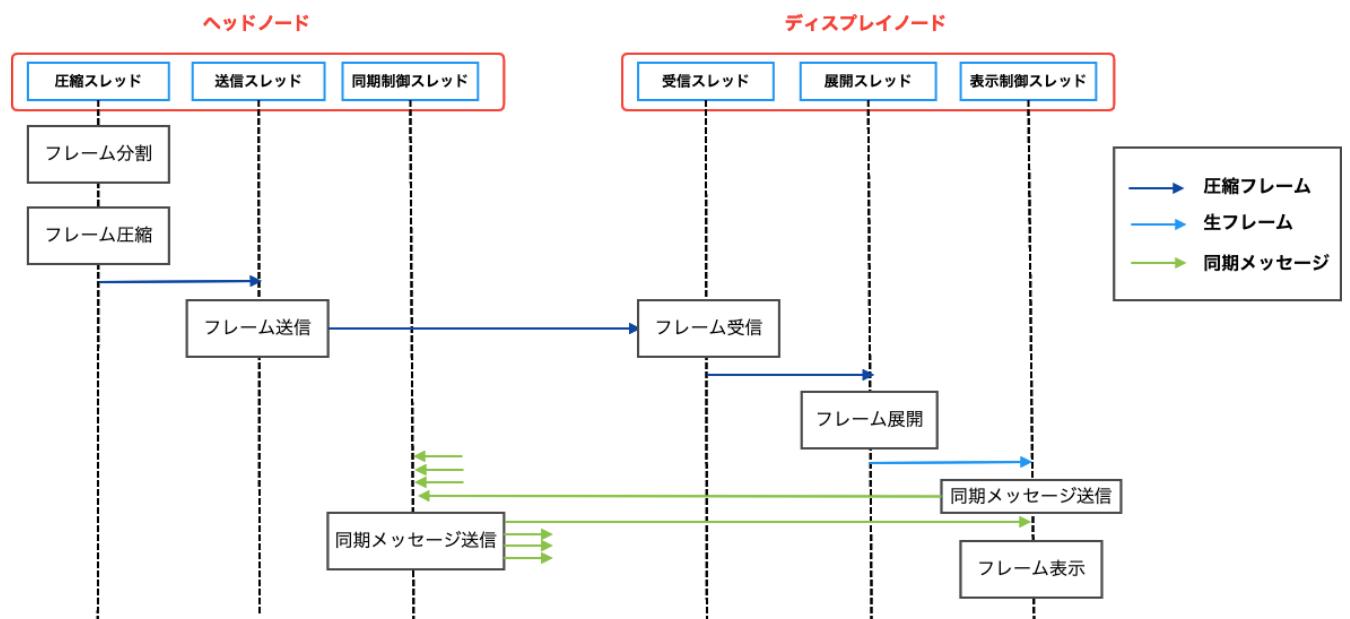


図 2.5: 各スレッドの動作

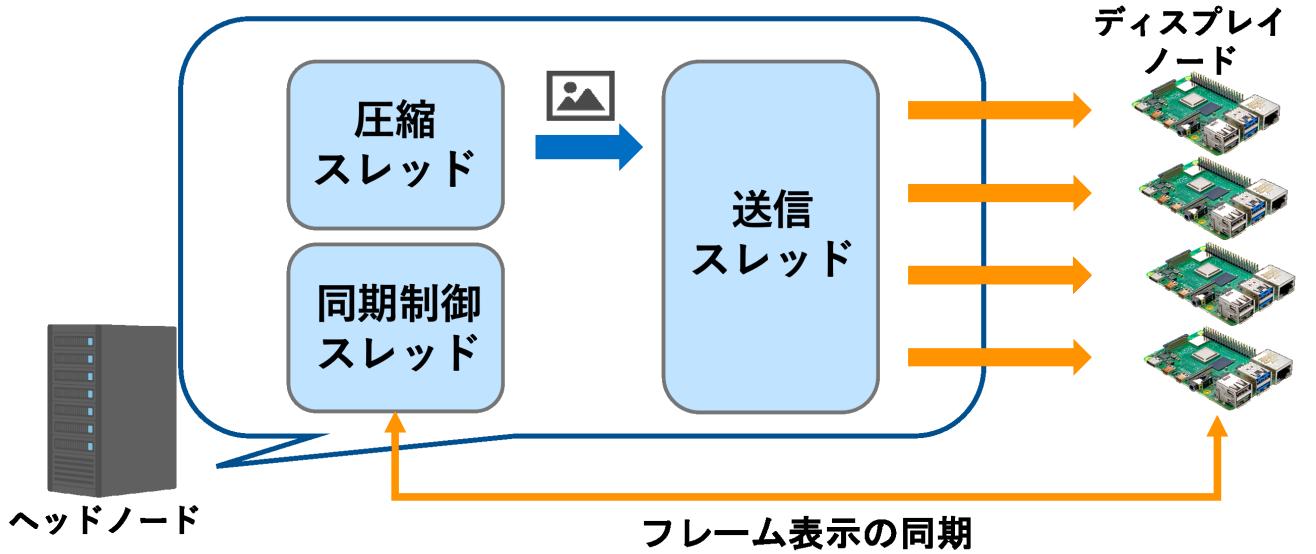


図 2.6: 各スレッドの動作

#### 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人間の目では認識しづらい色差成分とに分けて表現する [31]。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があるが、このとき 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 [32]。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。

#### 2.4 先行研究の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、マルチディスプレイの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処理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るため、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場合におけるマルチディスプレイ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単一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を引き起こす。実際、4面構成の場合では30fpsの動画をマルチディスプレイ上に20~22fps程度で再生することが可能だが、ディスプレイを9面構成にしたマルチディスプレイでは8~10fps程度に低下することが確認できる(図2.7)。

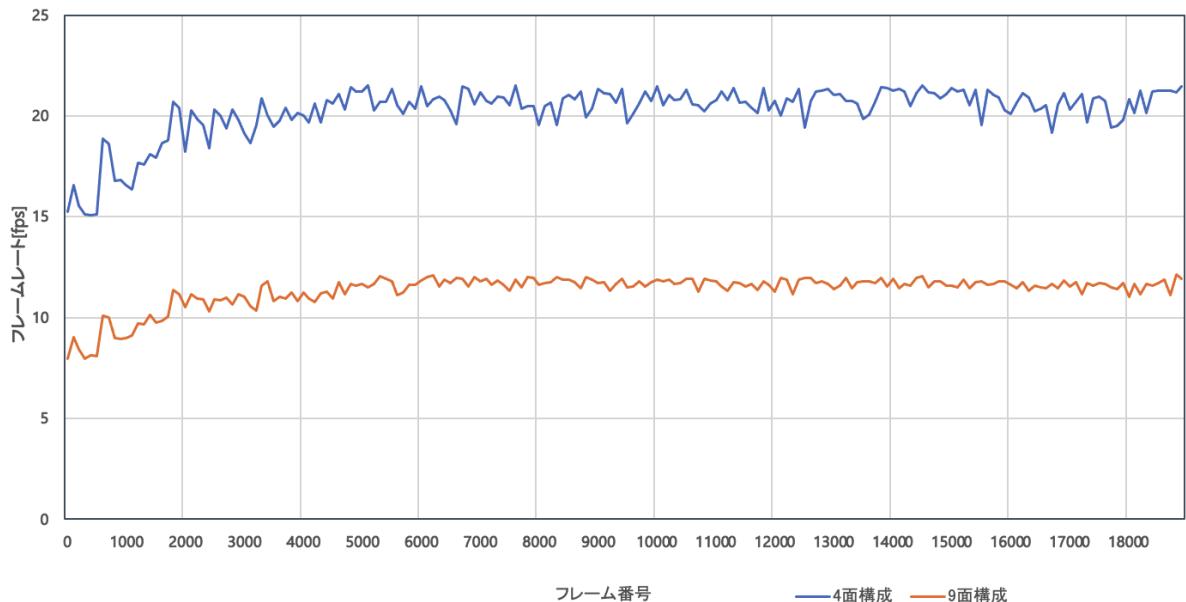


図2.7: 画面構成変更時のフレームレート比較

## 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手法として、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ことが考えられる。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2.8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。

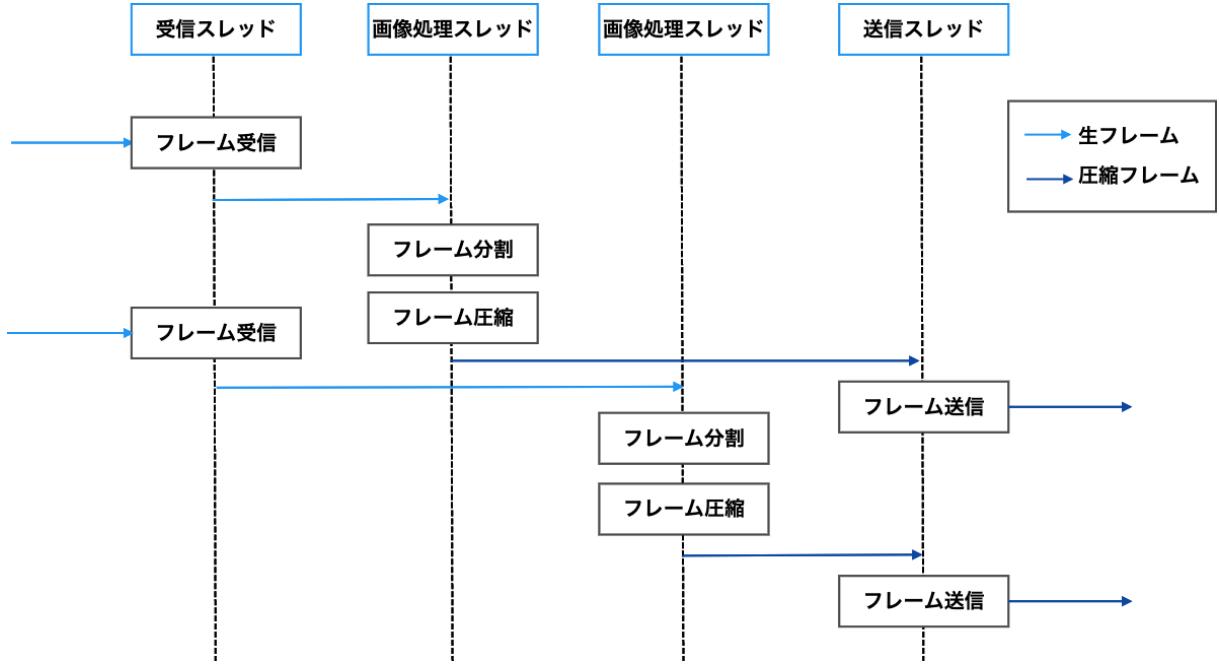


図 2.8: 各スレッドの動作

### 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、ヘッドノードはまず動画ファイルからのフレーム切り出しを行い、接続された中継ノード数に応じた分割処理を行う。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 2.9 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。

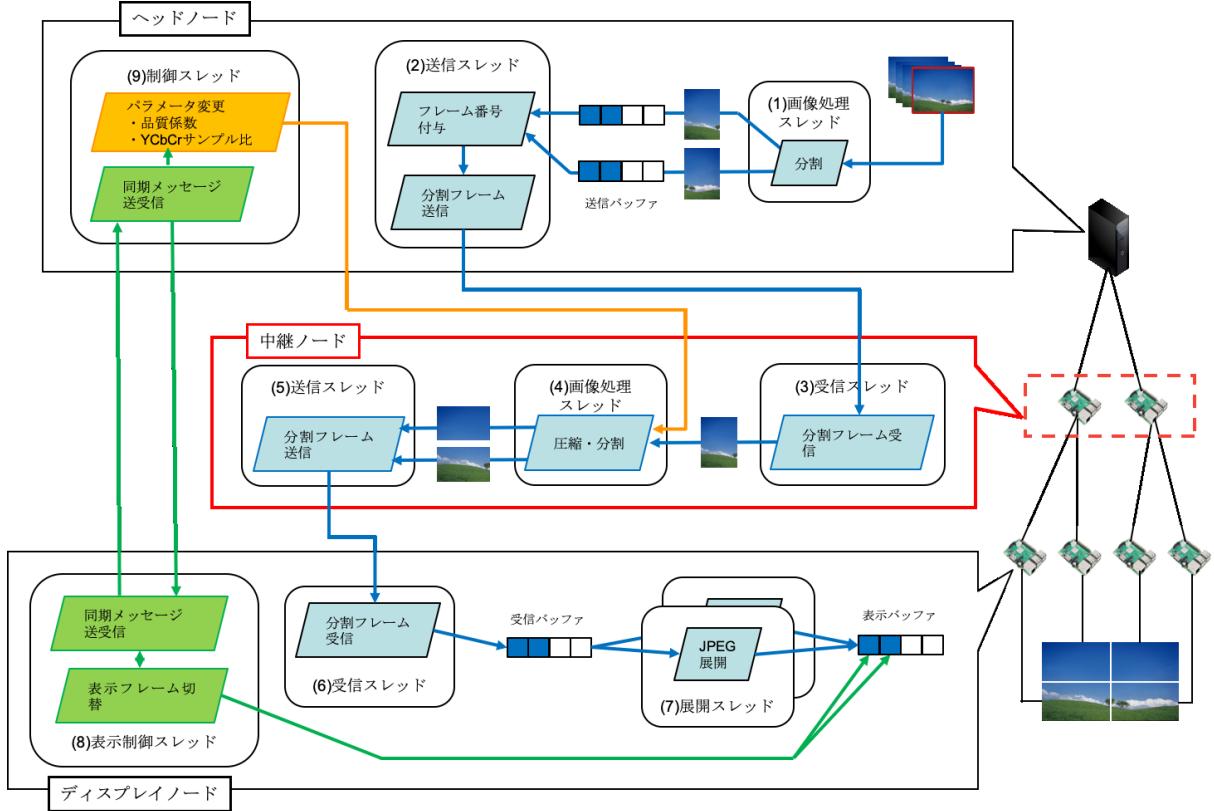


図 2.9: 関連研究で提案された手法を用いた MD の動作フロー

### 関連研究の問題点と将来課題

関連研究の評価では、中継ノードを実装した効果を検証するため、先行研究と提案手法において 9 面構成の MD を構築して 4K 解像度の動画再生時における CPU コア使用率を計測した。先行研究では HN の CPU コア使用率が 100% 近くに達していたのに対して、関連研究での提案手法では 20% 以下に抑えられていた。また、中継ノードの CPU コア使用率もも 70% 程度となり、分散並列処理による CPU コア時間のボトルネックは解消された。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)。

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

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によって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。

## 2.5 関連研究での技術的課題

関連研究では、ヘッドノードでのフレーム圧縮処理を分散並列化する目的で複数の中継ノードをヘッドノードとディスプレイノードの間に設置した。しかし、前節で述べたようにヘッドノードのCPU使用率は低下したものとの、マルチディスプレイシステムのパフォーマンス指標として最も重要なフレームレートは大幅に低下していた。この結果から、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CPUの負荷分散に対しては効果的であるが、マルチディスプレイシステム自体のパフォーマンス向上には貢献できていないと考えられる。すなわち、関連研究での技術的な課題は、ヘッドノードの処理の分散を外部のノードを用いて行うのではなく、ヘッドノード内のリソースを効率的に活用してフレーム処理の分散化を行うことである。ヘッドノード内のCPUリソースを活用することで、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事ができ、フレームレートの向上にも寄与できる可能性が高い。

図2.10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。

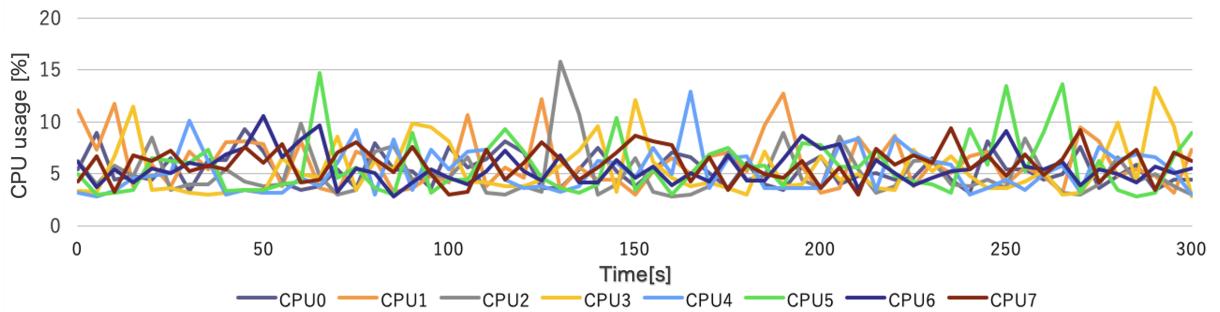


図2.10: 関連研究のヘッドノードにおけるCPU使用率

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を設計・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におけるフレーム処理の改良についての設計指針を述べ、具体的な実装について述べる。

#### 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 [33, 34] がある。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し、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させることができ、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ことが特徴である。

コンテナ技術を用いた OS 仮想化とハイパーバイザ型仮想化の概要を図 3.1 に示す。

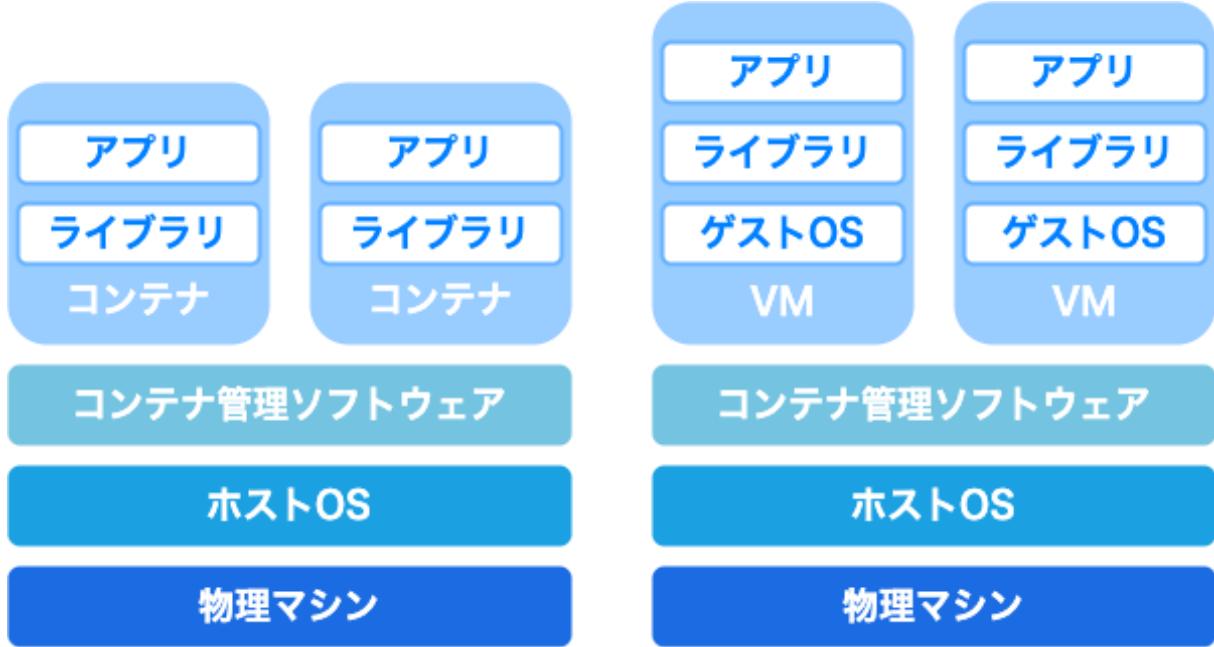


図 3.1: コンテナ技術を用いた OS 仮想化とハイパーバイザ型仮想化

さらに、Docker を利用することでさまざまな利点も生まれる。

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という点である。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で異なるマシンに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システムを構築する場合も、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開発環境、動作環境をはじめから作る必要がなくなり、クラスタシステムを構築することも容易になる。さらに、Kubernetes [35] などが有する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できる。

### 3.2 マルチディスプレイシステムのコンテナ化

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ことができる。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の仮想化を検討する。

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として Raspberry Pi を使用している。2022 年 2 月現在においては SBC の開発が盛んに行われて

おり、市場には数百種類もの SBC が存在している [36]。それらの SBC の間ではアーキテクチャ、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プレイを構築する場合には構築の手順や動作に異なる点が生じることが予想される。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には、さまざまな SBC に対応することができるマルチディスプレイ構築用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。

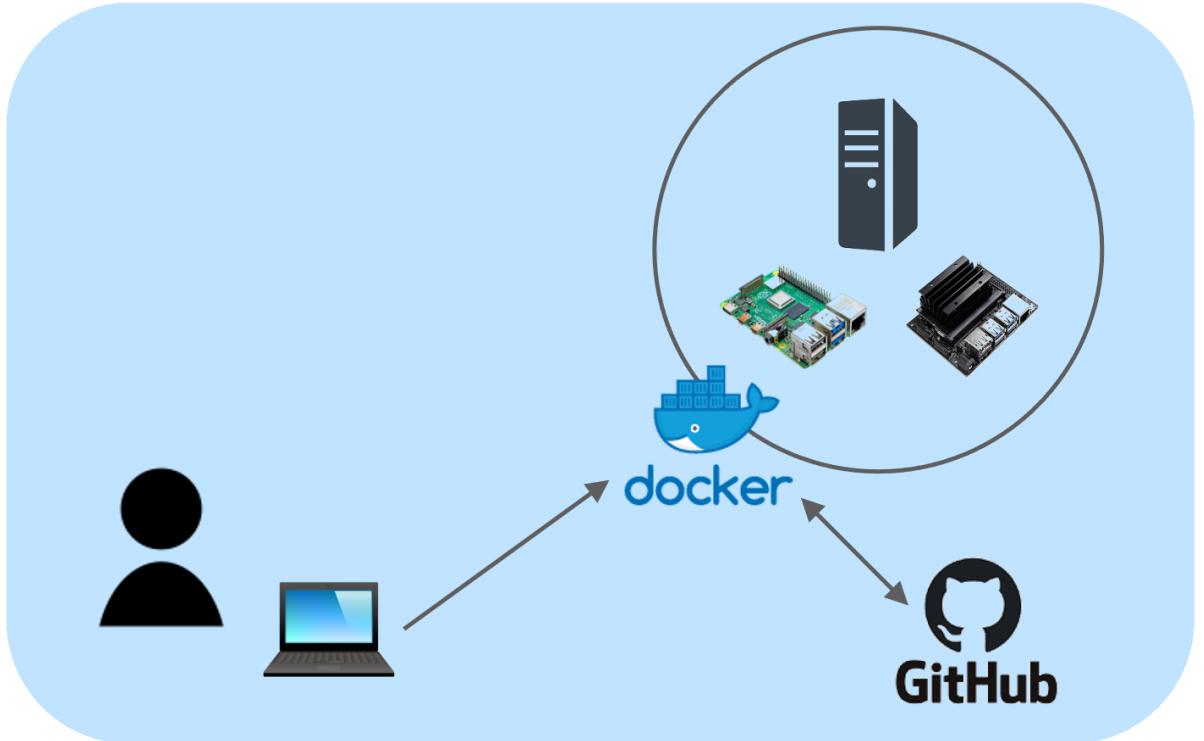


図 3.2: Docker を用いた環境構築

Docker を用いた環境構築の概要図を図 3.2 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 のインストールを行う。次に、Docker に用意されているコマンドを用いて GitHub [37] 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ファイルによってあらかじめ設定されていた環境を簡単に構築することができる。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ことが可能になる。

### 3.3 コンテナ環境における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におけるフレーム表示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フレームを表示する際には、一般にフレームバッファとよばれる領域が使用される。フレームバッファはマシンの/devディレクトリに存在するデバイスファイルである。このファイルの中にディスプレイに表示するデータを格納することで、OSがディスプレイに画像を描画するという仕組みになっている。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3.3に示す。

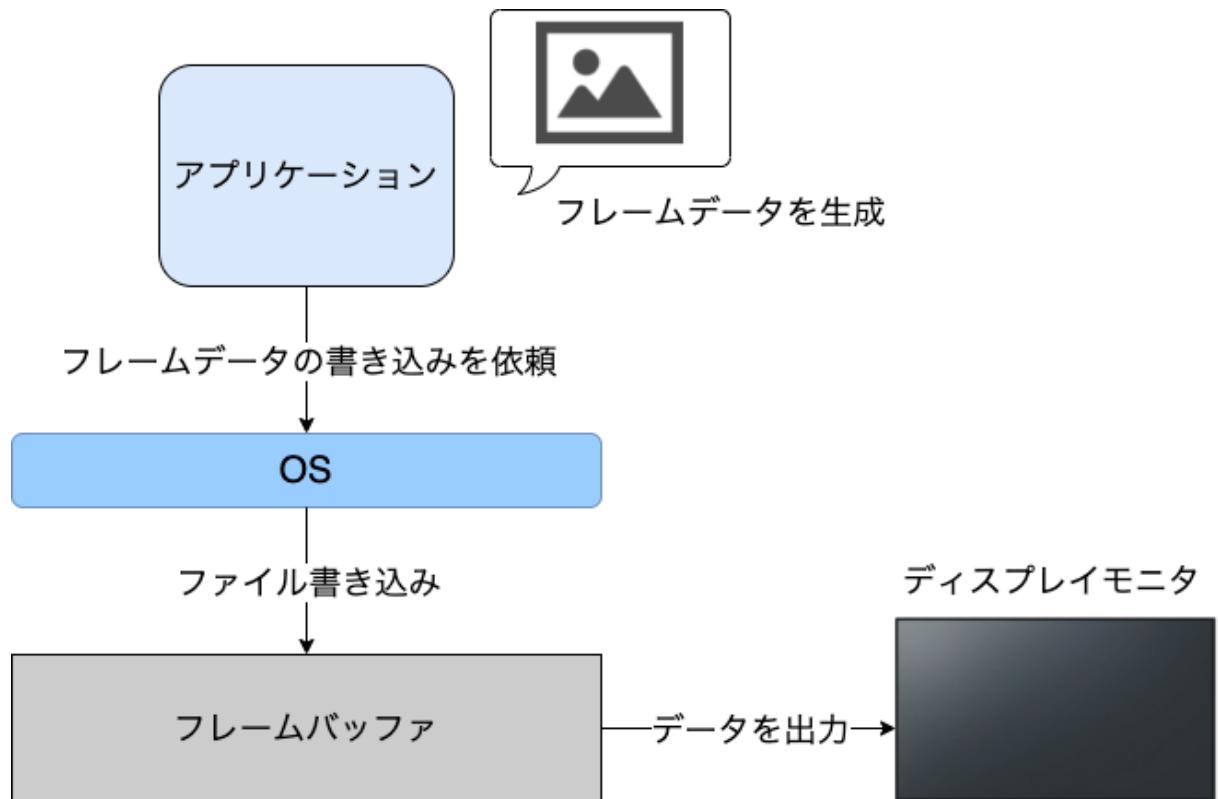


図 3.3: フレームバッファを用いたディスプレイ表示

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デーモン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ことが原因である。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内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画像フレームデータをディスプレイへの表示することができなくなっている。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できる。コンテナを privilege な状態で起動するためには、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

ここまで紹介した技術を用いることで、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ことができるようになる。続いて本節以降では、本研究の提案の中心部分とな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問題点がある。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で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し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 を構築した際のパフォーマンス低下も問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、提案手法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、ヘッドノード内でのフレーム処理時間の短縮を目指す。

#### 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処理並列化を図る。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が可能な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。しかし一方で、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。そのため、構成ディスプレイを増加させた場合には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を行い、全体でのフレーム処理時間の短縮を試みる。

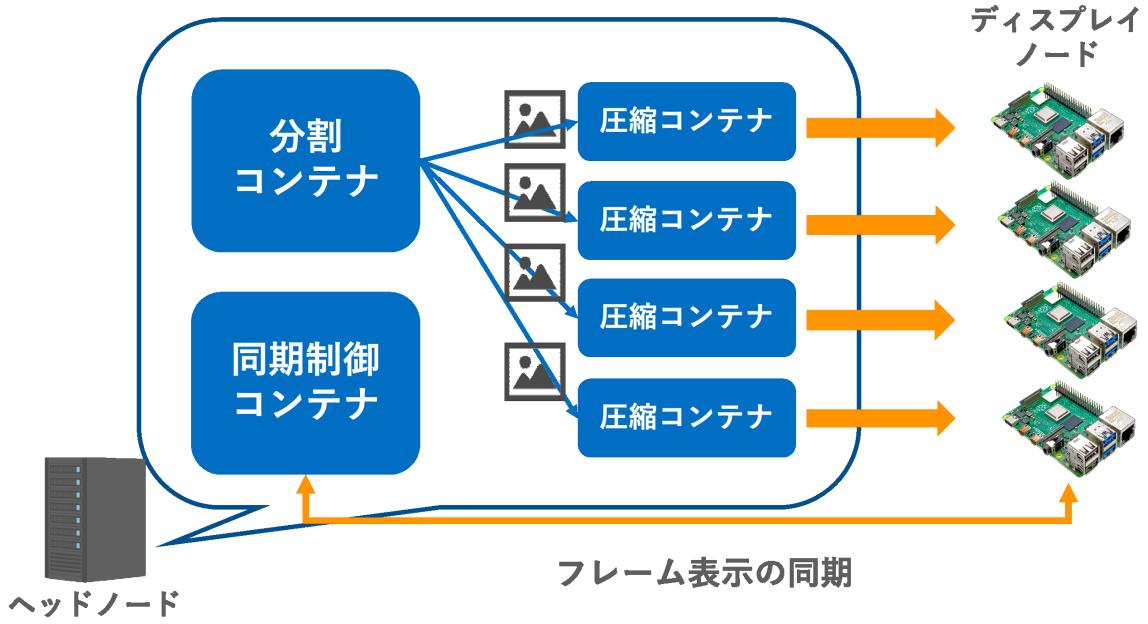


図 3.4: フレーム処理の並列化

図 3.4 に、提案手法におけるヘッドノード内のコンテナ構成を示す。提案手法ではヘッドノード内で行われる処理そ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。

#### 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) [38, 39] 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

などの操作を行うことができる。

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を目的とする。

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ためには、`shmget()` 関数を使用する。`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 識別子を利用してすることで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。

以上の方針を検討した結果、本研究では複数のコンテナ間でのデータの受け渡し処理に使用することを想定しており、IPC 資源が複数のプロセスから参照されることになるため、実装が容易になる前者の手法を採用した。

## 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）。

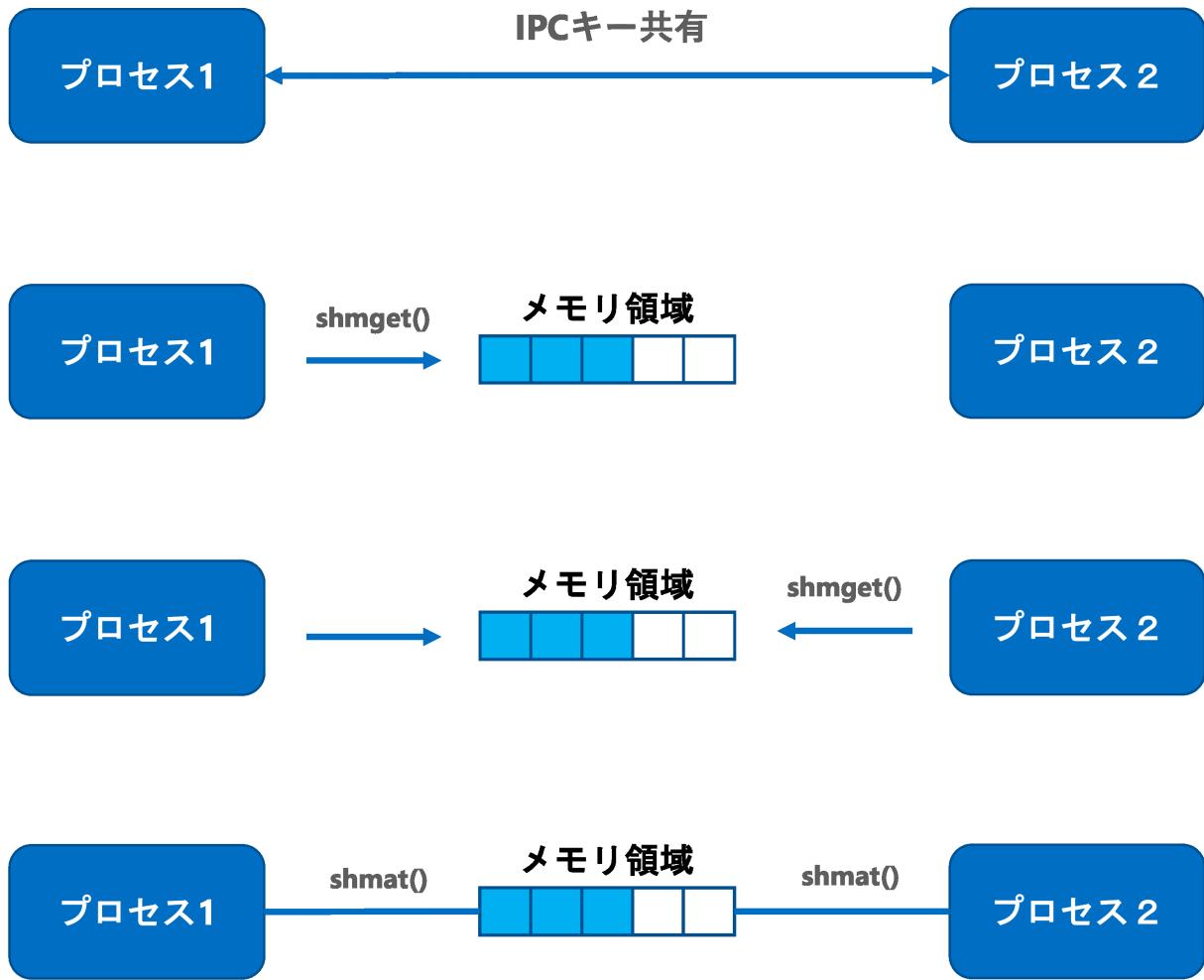


図 3.5: 共有メモリ使用の流れ

### 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。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。このとき、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として保持している。Mat 構造体の内容を図 3.6 に示す。

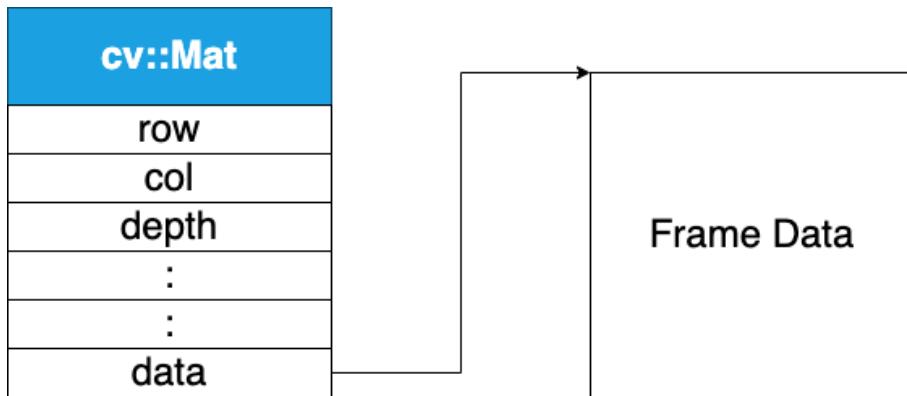


図 3.6: OpenCV における Mat 構造体

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実態ではなくデータ格納場所を指すポインタのみを保持しており、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そのプロセスに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 型の配列として定義し、Mat 構造体のもつフレームデータ格納場所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 バイト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で生じるオーバーヘッドは小さくなる。

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分割して割り当てるという手法を採用する。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 形式を文字型の配列に変換してからフレームデータを格納する(図 3.7)。

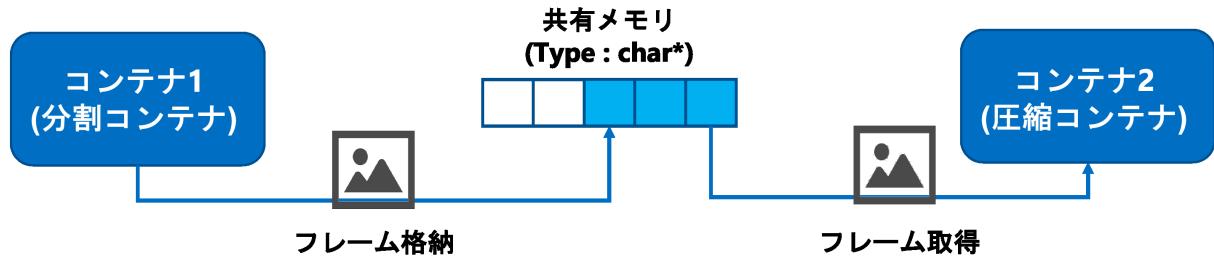


図 3.7: 共有メモリの使用例 2

### フレーム受け渡し処理の詳細

続いて、共有メモリ領域を用いたフレームデータの受け渡しについて詳細な動作を説明する。フレームデータ受け渡しに用いる共有メモリ領域は、はじめに分割ノードによってホストマシン内に1つだけ作成される。その1つの共有メモリ領域に対して分割メモリはフレームデータの格納を行う。接続ディスプレイと同数だけ用意された全ての圧縮コンテナは1つの共有メモリに対してデータの取り出しを行う。また、作成したメモリの共有を行うには共有メモリを作成した際に使用されたIPCキーを分割ノードと圧縮ノードとの間で共有する事が必要になる。このIPCキーの共有にはホストマシンのメモリ領域を利用している。まず、分割ノードにより共有メモリの作成に使用されたIPCはあらかじめ設定されたパスに存在する物理マシン内のテキストファイルに文字列として保存される。その後起動された圧縮コンテナが物理マシン内の該当パスに存在するフォルダを認識し、その中に記述されているIPCキーを引数としてコンテナ内でプログラムを実行する。この仕組みによって分割コンテナと圧縮コンテナとの間でIPCキーを共有することができ、同一のメモリ領域にアクセスする事が可能になる。

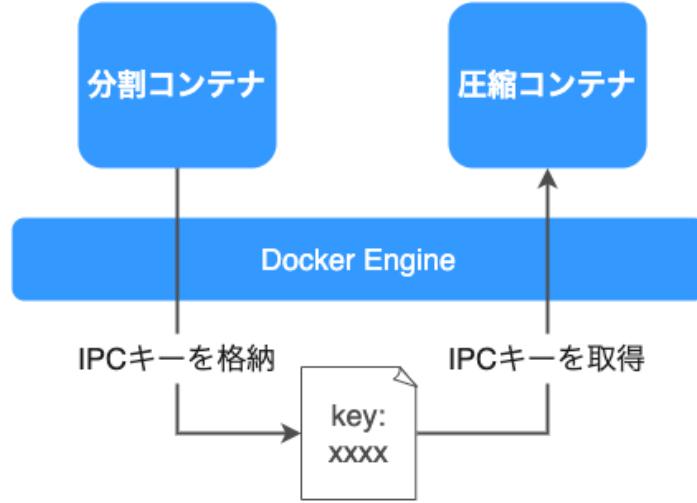


図 3.8: 共有メモリの使用例 2

共有メモリ領域では、フレームの格納および取り出しを高速で行うために、分割コンテナによって格納された分割済みの画像フレームデータをバッファリングする仕組みを用意する。共有メモリに格納される画像フレームデータは、ディスプレイノードが接続しているディスプレイと同じ解像度となる。本研究では、動画の表示に使用されるディスプレイモニタの解像度として Full HD (1920 x 1080 ピクセル) のものを想定する。また、色深度は赤、緑、青の 3 深度とする。この想定下では、共有メモリに格納される分割後の画像フレームは 1920 x 1080 ピクセル、色深度 3、ビット深度 (1 ピクセル、1 色のデータを保持するのに必要なビット数) 8 となり、そのデータ量はおよそ 5.9MB となる。よって、各圧縮コンテナに対して 8 枚の分割済みフレームをバッファリングするとすると、必要な共有メモリの領域は ( $\text{圧縮コンテナ数} \times 5.9 \times 8 (\text{MB})$ ) で算出することができる。例えば、4 面構成のマルチディスプレイを構築する場合には約 189MB、9 面構成のマルチディスプレイを構築する場合には約 425MB の共有メモリ領域が必要となる。このメモリ領域を、分割コンテナは起動と同時に獲得する処理を行う。

分割コンテナは、新たなフレームの切り出しを行いそれに対する分割処理を完了すると、共有メモリ領域に用意されたバッファに順次フレームを格納していく。バッファにはそれぞれの圧縮コンテナに対して領域を定めておき、圧縮コンテナはその領域から順番にフレームの取り出しを行い後続の処理を開始する。共有メモリ領域に作成されたバッファを用いてコンテナ間で画像フレームの受け渡しを行う様子を、図 3.9 に示す。

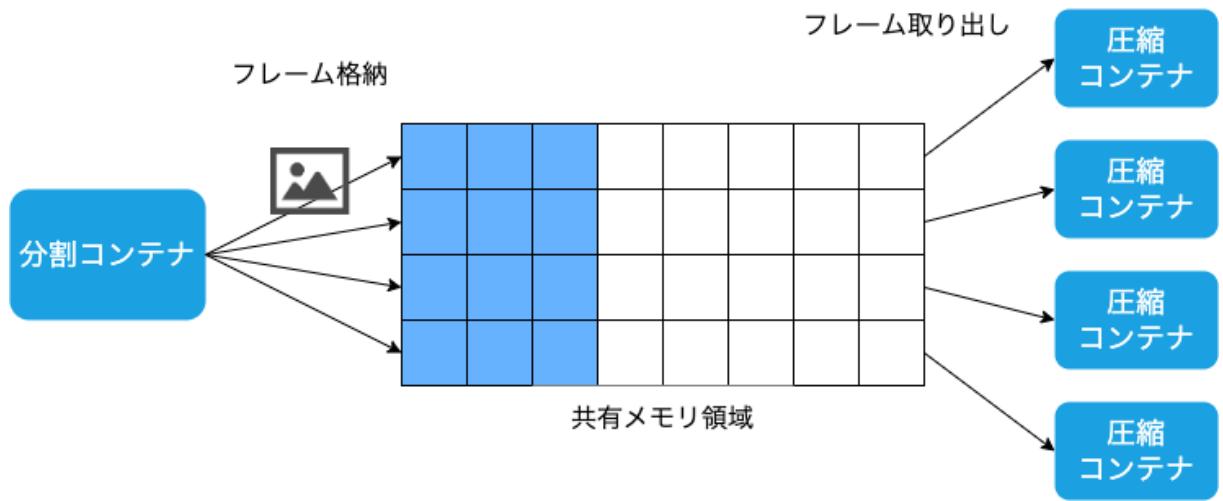


図 3.9: 共有メモリ内におけるフレームのバッファリング

### 各コンテナでの処理

ここでは、各コンテナ内での処理について順に述べる、図 3.10 および図 3.11 に提案手法における分割コンテナと圧縮コンテナの処理フローチャートを示す。

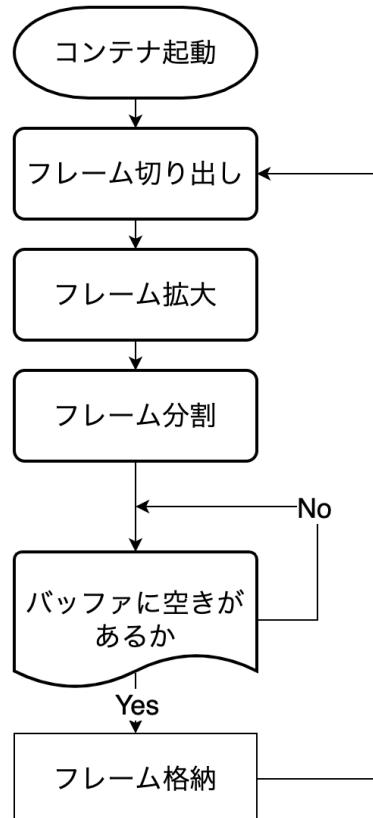


図 3.10: 分割コンテナの動作フロー

分割コンテナは、コンテナが起動されるとすぐに動画ファイルからのフレーム切り出し

処理を開始する。その後、切り出したフレームに対して、構築したマルチディスプレイの解像度に応じたサイズにフレームの拡大処理を行う。次に、ディスプレイの数に応じた分割処理を行い、共有メモリに作成されたバッファに順に格納していく。このとき、共有メモリ内のバッファにはあらかじめ用意したバッファの数以上のフレームを格納する事ができない。そのため、フレームの格納処理を行う前にバッファに格納されているフレーム数を取得し、バッファ上にあるフレーム数が上限に達していれば格納処理を行わずに待機状態になる。その後、圧縮フレームがバッファから画像フレームを取り出すことによってバッファに空きができたことを認識すると待機状態を終了し、現在処理中のフレームデータを格納して次のフレームの処理へと移る。

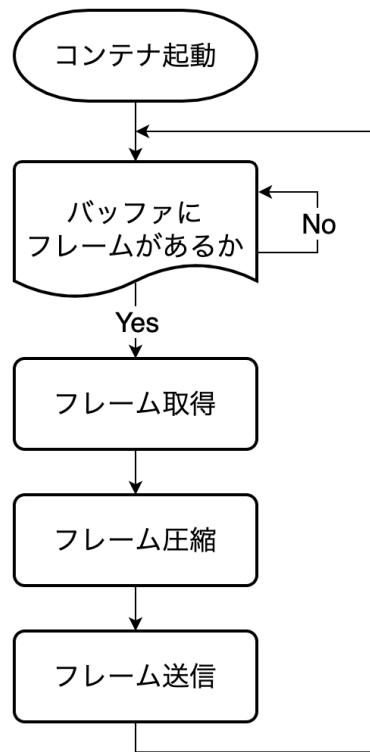


図 3.11: 圧縮コンテナの動作フロー

圧縮コンテナは起動と同時に分割コンテナとの共有メモリを取得し、バッファにフレームが格納されているかどうかの監視を開始する。分割コンテナによってバッファに分割済みの画像フレームが格納されたことを確認すると監視を終了し、フレーム処理を開始する。圧縮コンテナで行われるフレーム処理は、フレームの圧縮と送信である。バッファから画像フレームを取り出して圧縮処理を行うと、OpenCV の Mat 形式から文字列オブジェクト (C++ の string 型) への変換が行われる。この文字列オブジェクトを接続したディスプレイノードへと送信するまでが各フレームに対する一連の処理となる。

## 提案手法を用いたマルチディスプレイシステムの構築

最後に、提案手法を用いて構築したマルチディスプレイの全体像について述べる。図 3.12 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。処理の流れとしては、まず分割コンテナが動画ファイルから 1 フレームずつ切り出し、構成ディスプレイ数に応じて分割処理が行われた後、作成した共有メモリ領域に格納する。圧縮コンテナは、共有メモリから分割済みの画像フレームを取得する。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の送信処理を行う。圧縮コンテナからフレームを受け取ったディスプレイノードではフレーム展開処理を行い、展開後の画像フレームをホストマシンの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。この処理を各フレームに対して行うことで接続されたディスプレイ上への動画の再生が実現される。また、ヘッドノードの内部では分割コンテナと複数の圧縮コンテナの他に同期制御コンテナも動作する。同期制御コンテナの役割はディスプレイノードで動作している表示制御スレッドにフレームの表示命令を送信することである。同期制御スレッドはディスプレイノードから各画像フレームが表示バッファに格納され、ディスプレイの表示準備が完了した旨の通知を受け取る。この通知が全てのディスプレイノードから送信されたことを確認し、同期制御スレッドが各ディスプレイノードに対してフレームの表示命令を送信する。さらに、同期制御スレッドは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。

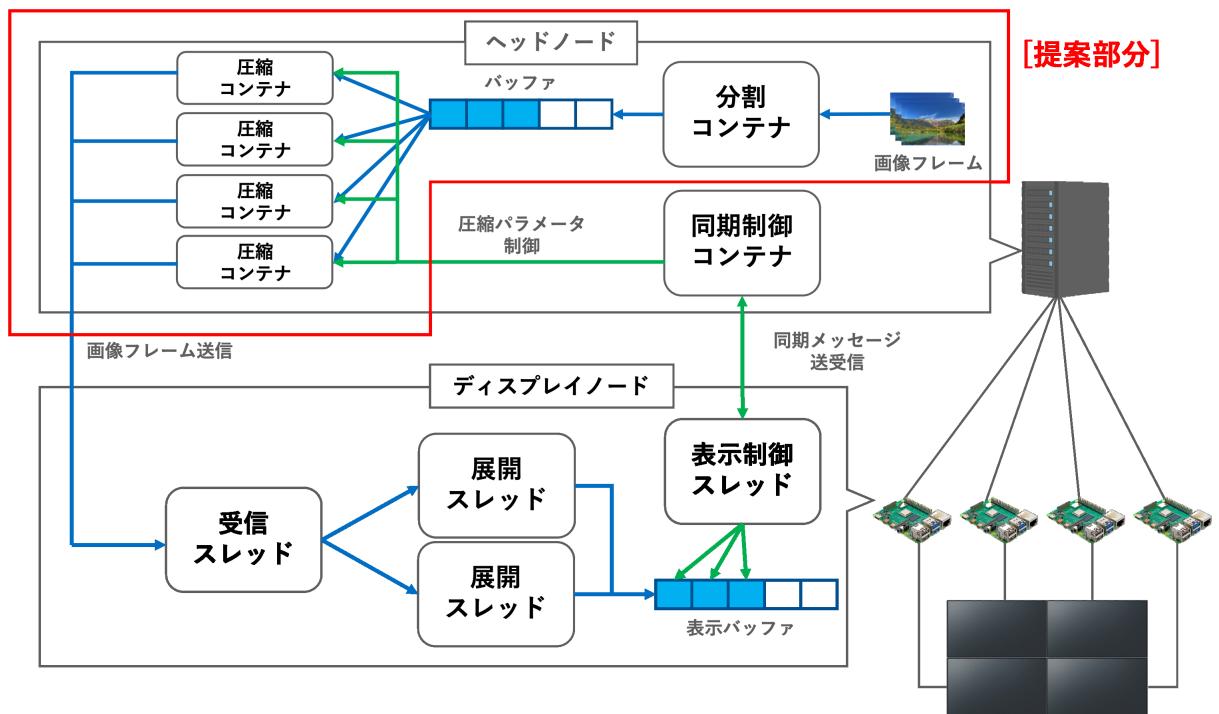


図 3.12: 提案手法を用いた MD の動作フロー

## 4 評価

本章では、3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実験を行う。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、提案手法を用いたマルチディスプレイシステムに対して接続ディスプレイ数を変化させた場合のフレーム処理時間の変化を確認する。さらに、実験3では、ヘッドノード内で分割コンテナ・圧縮コンテナを動かす場合のコンテナの起動と終了に要する時間の測定を行い、コンテナ環境でプロセスを動作させる際のオーバーヘッドについて調べる。また、実験4では既存手法を用いて構築したマルチディスプレイシステムにおいて全てのノードでコンテナ環境を用意し、完全にコンテナ化されたマルチディスプレイシステムを構築した際のフレームレートについて調査する。本章では、まず4.1節で、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、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。さらに、4.4章では実験3における実験方法と実験結果について述べ、最後に4.5章では実験4に用いた環境と実験方法および実験結果を示す。

### 4.1 実験環境

本節では、実験1で用いた実験環境とその方法・結果について述べる。実験1では、4つのディスプレイを接続して構築した4面構成のマルチディスプレイシステムを想定した実験を行った。本実験でヘッドノードとして用いたマシンの使用を表4.1に、ヘッドノード内のコンテナ構成を図4.1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。今回の実験では4面構成のディスプレイを想定しているため、ヘッドノード内では1つの分割コンテナと4つの圧縮コンテナを動作させた。

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

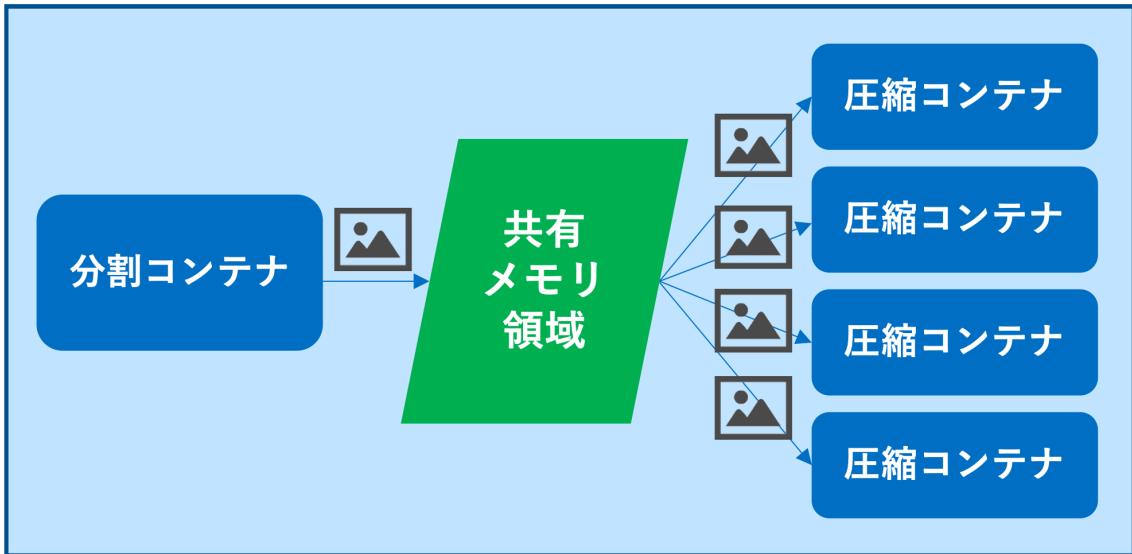


図 4.1: 実験環境のコンテナ構成

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 [40]を使用した。表4.2に、Big Buck Bunnyのプロパティを示す。さらに、図4.2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 [41]
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 4.2: 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 [42], libjpeg-turbo [43], Boost.Aasio [44], gstreamer [45] , FFMPEG [46] という 5 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.Aasio はソケット通信を行うライブラリである. gstreamer および FFMPEG はオープンソースのマルチメディアフレームワークであり, 動画と音声の記録・変換・再生が可能である. また, ディスプレイノードの処理の実装には libjpeg-turbo, Boost.Aasio, 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) [47] でコンパイルを行った. 評価で利用したソフトウェアのバージョンを表 4.3 に示す.

表 4.3: 評価で利用したソフトウェア

ソフトウェア名	バージョン
OpenCV	3.4.5
Boost.Aasio	1.53
FFMPEG	2.8.17
libjpeg-turbo	2.0.1
Gstreamer	1.16.1
GCC	8.5.0

## 4.2 実験 1: フレーム処理時間に関する評価

### 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。

### 実験結果

図4.3に先行研究で提案された手法と提案手法でのフレーム処理時間を示す。既存手法においてフレームの処理に要する時間の平均を計測すると、それぞれ拡大・分割は5.04ms、圧縮は47.31msであった。結果より、フレームの拡大と分割に比べてフレームの圧縮に長い時間がかかっていることがわかる。また、1フレームあたりに要する処理時間は拡大・分割にかかる時間と圧縮にかかる時間との和で表すことができ、その時間は52.35msとなる。1フレームあたりの処理時間を用いることで1秒あたりに何フレームを表示する事ができるかを示す動作時フレームレートを計算することができる。この場合には、動作時フレームレートは $1000[\text{ms}]/52.35[\text{ms}] \approx 19.1[\text{fps}]$ となり、この付近に動作時フレームレートの限界が存在する。

一方、提案手法においてフレームの処理に要する時間の平均を計測すると、それぞれ拡大・分割は4.82ms、共有メモリへの格納は0.77ms、共有メモリからの取り出しが0.02ms、圧縮は16.11msであった。フレームの拡大・分割については既存手法と提案手法とで処理に差はないため、既存手法と同等の処理時間を要している。フレームの圧縮処理においてはOS仮想化技術を用いた並列化の効果が顕著に現れており、既存手法と比較しておよそ3分の1程度の時間で行うことが可能となっている。さらに、フレーム圧縮処理を並列化することによって生じたオーバーヘッドである共有メモリを用いたプロセス間でのフレーム受け渡し処理においても、1ms未満の非常に短い時間で行えていることがわかる。これは、ホストマシン内のメモリ領域に対してはコンテナ内から memcpy() 関数を用いた高速なデータ書き込み/読み出しが可能であることを表している。また、全体のフレーム処理時間について比較すると、提案手法では既存手法から60%程度短縮することに成功している。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ことが確認できる。

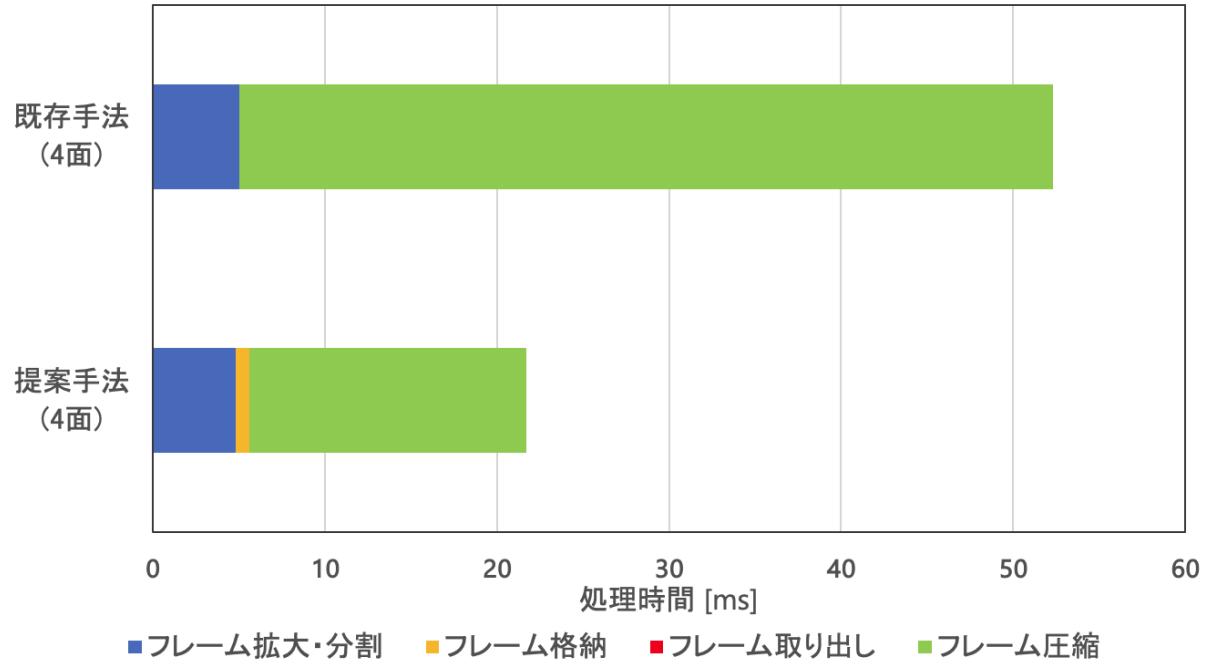


図 4.3: フレーム処理時間の比較 (4面構成時)

#### 4.3 実験 2:スケーラビリティに関する評価

実験 2 では、構成ディスプレイ数を増加させた場合におけるフレーム処理時間の変化を調べる。本実験では、ディスプレイ 9 面構成のマルチディスプレイを構築することを想定し、実験 1 の結果との比較を行う。本実験のヘッドノード内では、1 つの分割コンテナと 9 つの圧縮コンテナを動作させる。

#### 実験結果

図 4.4 に先行研究で提案されたシステムと提案手法のそれぞれにおける、9面構成のマルチディスプレイを想定した場合のフレーム処理時間を示す。9面構成を想定した場合におけるヘッドノードでのフレーム処理時間の合計は既存手法で 117ms、提案手法では 52ms であった。このことから、提案手法はディスプレイ数を増加させた場合においても既存手法より高速なフレーム処理が可能であることが確認できる。

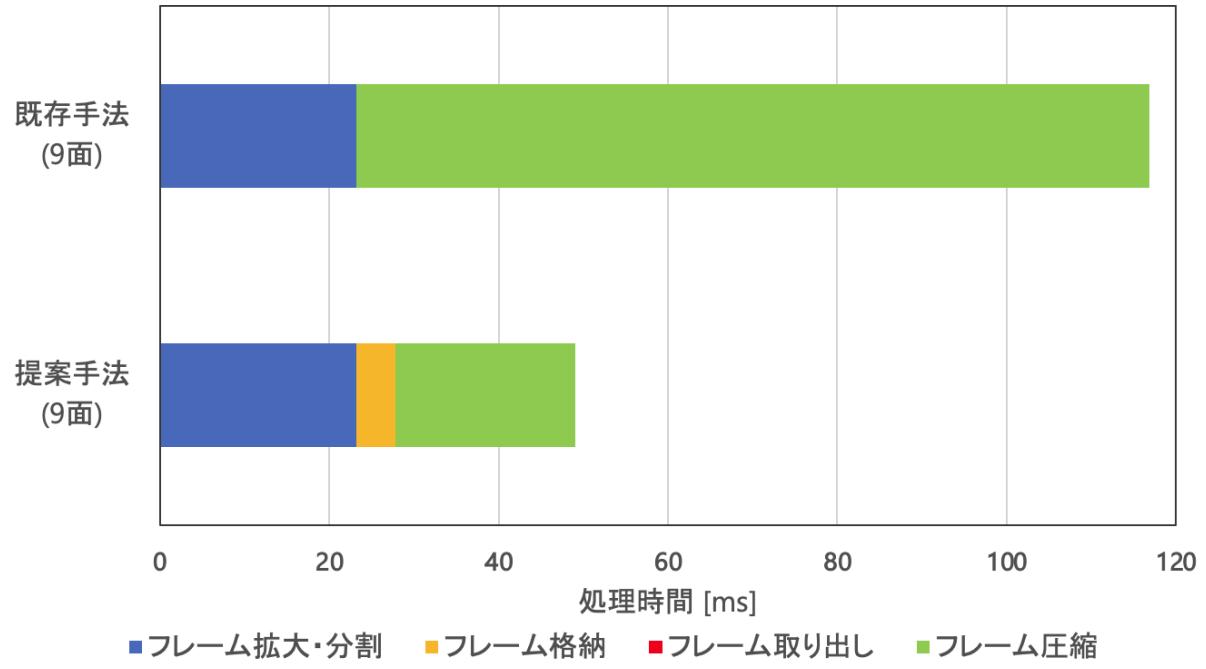


図 4.4: フレーム処理時間の比較 (9面構成時)

フレーム処理に含まれるそれぞれの操作ごとに見ると、フレームの拡大・分割時間については既存手法と提案手法の間で大きな変化は見られない。これは両手法ともに1つのプロセスで画像フレームの拡大・分割処理を行っているためであると考えられる。しかし、フレームの圧縮時間においては既存手法と提案手法の間で大きな差が生じていることが確認できる。既存手法ではフレームの圧縮処理を单一のプロセスで行っているため、処理に要する時間はマルチディスプレイを構成するディスプレイモニタの数に比例して増加する。実験結果では、ディスプレイ数を4面から9面に増加させたために処理時間が2倍以上になっていることがわかる。一方提案手法ではフレームの圧縮処理をディスプレイ数と同じ数だけ用意した圧縮コンテナ内で並列して行うため、ディスプレイ数の増加による影響は少なく抑えられていることが確認できる。実際、4面構成の場合には16msであった処理時間は、9面構成の場合でも21msに抑えられている。少し処理時間が増加しているのは、ヘッドノード内で動作するコンテナ数がホストマシンのCPUコア数を上回ったため、処理にCPUの割り当て待ち時間が生じたためであると考えられる。

また、提案手法におけるフレームの格納と取り出しにかかる時間については、ディスプレイ数の増加に比例して処理時間も増加する傾向にあるが、フレームの拡大・分割や圧縮といった他の処理に要する時間と比較して短い時間であることも確認できる。実験の結果から、提案手法ではマルチディスプレイシステムを構成するディスプレイ数の増加に対するフレーム処理時間の増加割合は小さくなってしまっており、提案手法の実装によってマルチディスプレイシステムのスケーラビリティが改善されたことが確認できた。

#### 4.4 実験 3: コンテナの起動時間に関する評価

ヘッドノード上でコンテナ環境を動作させる際には、コンテナの起動時間および起動したコンテナを停止し後処理を行うための処理時間が必要である。本節では、ヘッドノードのプロセスをコンテナ化したことによって生じるオーバーヘッドを調べるために、分割コンテナと圧縮コンテナそれぞれの起動および停止にかかる時間の計測を行った。実験環境は実験 1 および実験 2 と同一のものを用い、実験にはコンテナの起動と停止命令を記述したシェルスクリプトを用いた。また、処理時間の計測には time コマンドを用い、シェルスクリプトに記述されたコンテナの起動と停止処理の呼び出しから終了までにかかった実時間を計測した。実験結果を表 4.4 に示す。

表 4.4: コンテナの起動と停止に要する時間

コンテナ	起動時間と停止時間の和 [s]
分割コンテナ	0.71
圧縮コンテナ	0.62

実験結果より、分割コンテナ・圧縮コンテナとともに 1 秒未満でコンテナの起動と停止が可能なことが確認できた。この結果より、ヘッドノード内のプロセスをコンテナ化したことによるプロセス起動および停止にかかるオーバーヘッドは 1 秒未満であり、提案手法を用いて構築したマルチディスプレイシステムの動作開始時、終了時にもほとんど遅延は発生しないことがわかる。

#### 4.5 実験 4: マルチディスプレイシステムのコンテナ化に関する評価

本節では、マルチディスプレイシステムのコンテナ化を行った実験について述べる。本実験は、先行研究において提案された SBC マルチディスプレイシステムの完全なコンテナ仮想化についての可能性を探るためのものである。マルチディスプレイシステムを構成する各ノードに仮想化環境を導入し、その環境上で各ノードのプロセスを動作させることにより、異なる SBC をディスプレイノードとして使用した場合にもアーキテクチャ・仕様の差異を吸収して同一の環境で動作させることが可能となる。本実験では、SBC の一種であり、Docker が標準的にサポートされている Jetson Nano [48] をディスプレイノードとして使用し、仮想環境上でノードプロセスを動作させた場合におけるマルチディスプレイシステム全体のパフォーマンスを計測した。

##### 実験環境

実験 4 に用いた実験兼環境について説明する。ヘッドノードは実験 1 で用いたもののと同様のものを使用している。ディスプレイノードには SBC の一種である Jetson Nano を

用いた。図 4.5 に Jetson Nano の画像を、表 4.5 にその仕様を示す。



図 4.5: Jetson Nano

表 4.5: Jetson Nano の仕様

要素	仕様
CPU	ARM Cortex-A57(4 コア/1.43GHz)
GPU	CUDA コア 128 基/Maxwell アーキテクチャ/472GFLOPS
メモリ	4GB/64bit LPDDR4
OS	Jetson Nano Developer Kit(Ubuntu 18.04 LTS)

実験の際には Jetson Nano 4 台をディスプレイノードとして使用し、ディスプレイ 4 面構成のマルチディスプレイシステムを構築した。ヘッドノードとディスプレイノード群の間には 1 台のイーサネットスイッチ (1Gbps) が接続されている。本実験で用いた評価環境を図 4.6 に示す。

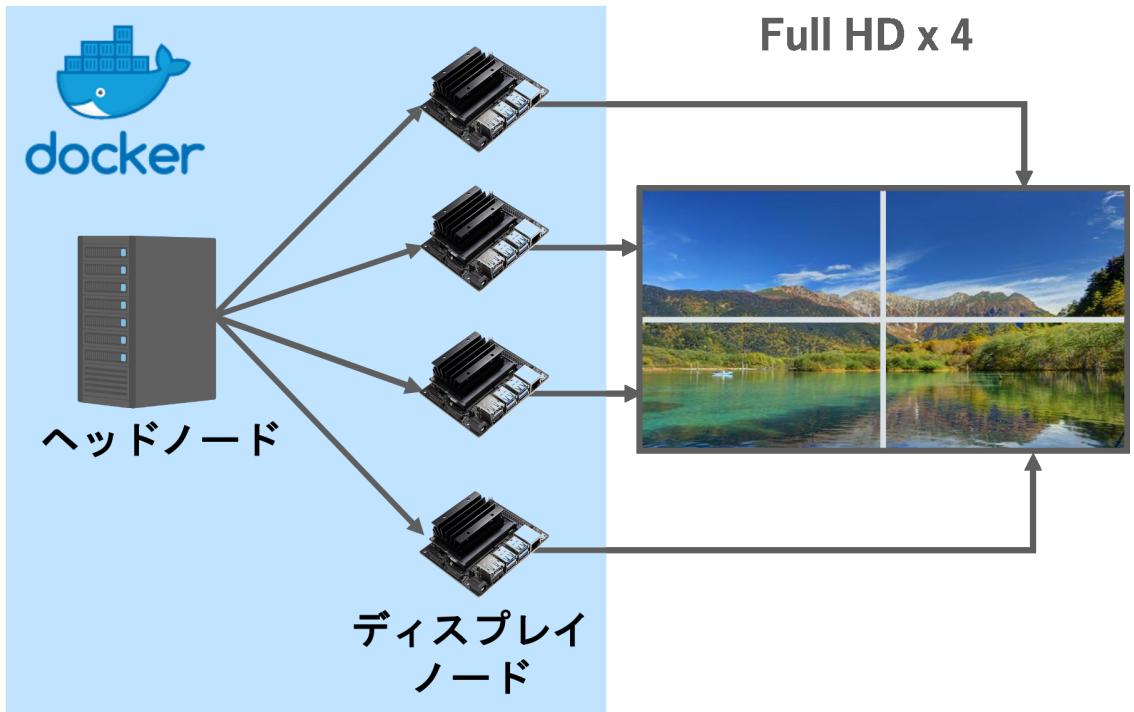


図 4.6: 実験 4 で用いた実験環境

なお、評価に用いた映像、ソフトウェアなどは全て実験 1 と同様のものである。

### 実験方法

4面構成のマルチディスプレイにおいて、先行研究で提案されたミドルウェアをヘッドノード、ディスプレイノードそれぞれに構築した仮想環境上で動作させ、動画表示のフレームレートを計測した。また、動作時の目標フレームレートは動画本来のフレームレートである 30fps に設定した。

### 実験結果

図 4.7 に、仮想環境上において先行研究で提案されたミドルウェアを動作させた場合のフレームレートの時間変化を示す。

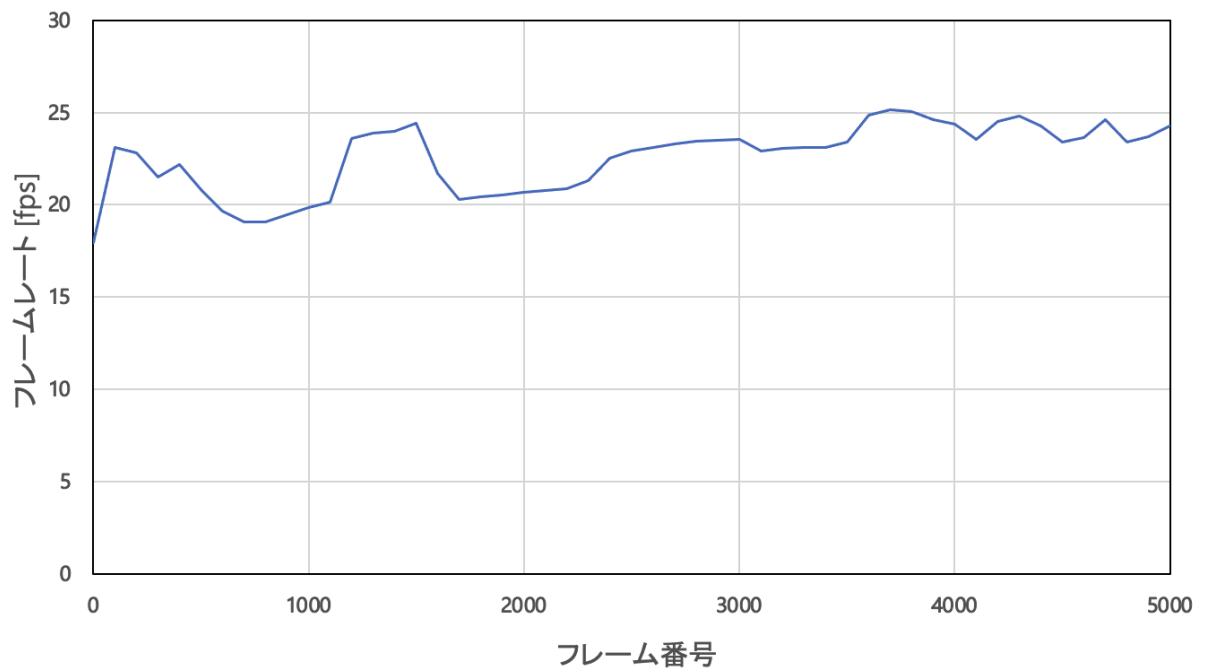


図 4.7: フレームレートの時間変化

実験の結果、マルチディスプレイ上に 4K 解像度の映像を表示した場合におけるフレームレートの平均値は 22.55fps であった。この結果から、各ノードを仮想化環境上で構築した際ににおいても動作時のフレームレートは大きく変化していないことが確認できた。

## 5 関連研究

本章では、本研究と同様に、仮想化技術を用いてマルチディスプレイシステムを構築した関連研究をいくつか紹介し、その特徴について説明する。

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 [49] がある。Bundulis らは、以前に Infiniviz [50] 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 [51] 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、仮想化を利用してことで、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。

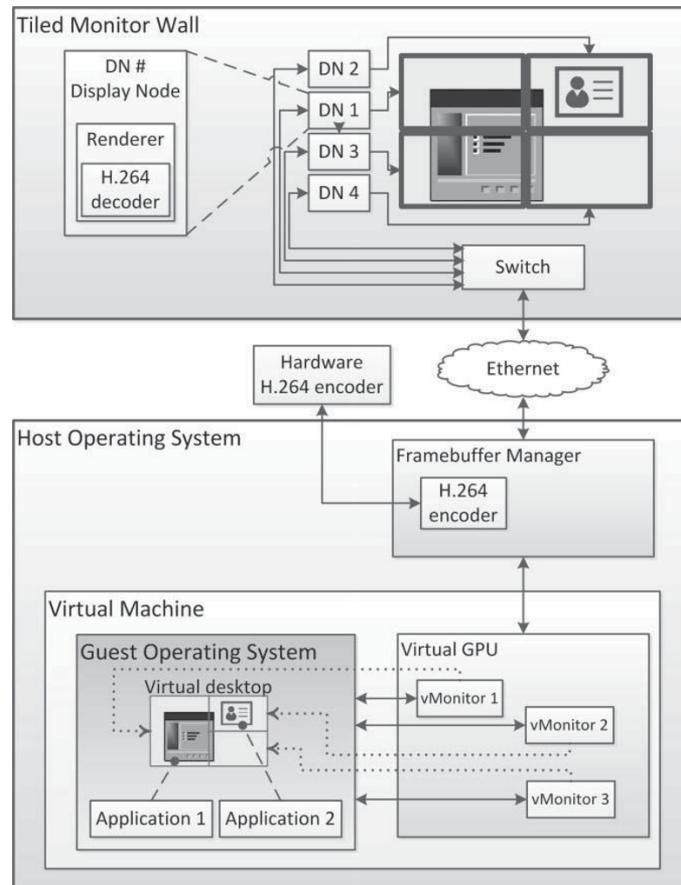


図 5.1: Infiniviz のアーキテクチャ図

Liu らは、Docker と GPU を利用して、画像、動画、 WebGL アプリケーションなどを表示する大画面ビジュアライゼーションアプリケーションをデプロイする手法を提案している [52]. この研究では、Docker クラスタを構築して 28 個のコンテナを含むマイクロサービスを起動し、 Unix/Linux ユーザにグラフィカルなインターフェースを提供するために広く使用されている X11 Unix ソケットをマッピングすることで大画面可視化システムをデプロイした。同時に、アプリケーションをローカルホストで動作させた場合と比較し、性能を検証し、彼らの手法を大画面ビジュアライゼーションに適用した場合、許容できる性能劣化で良好な結果を得ることができると結論付けている。

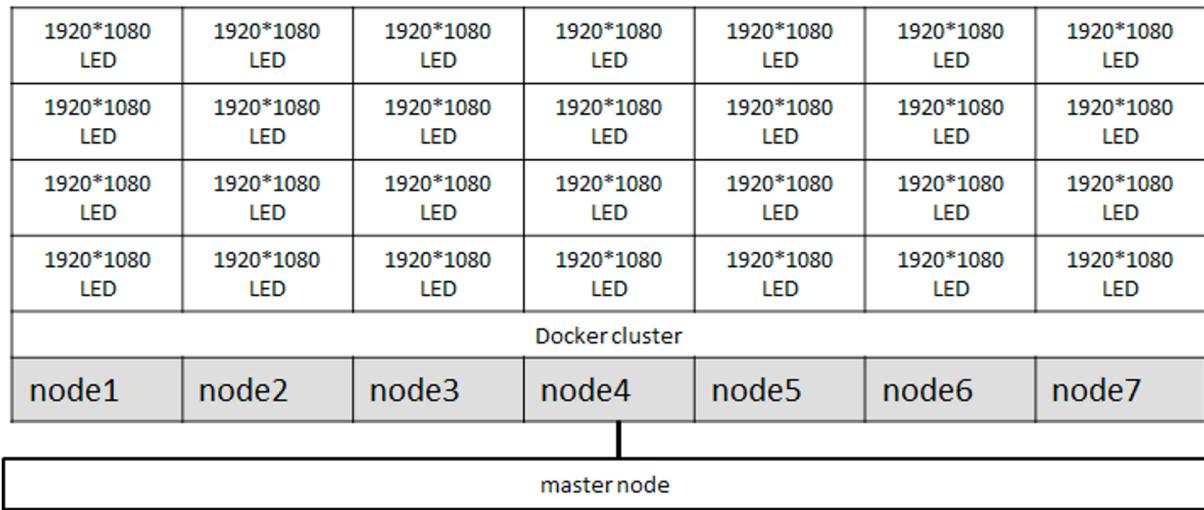


図 5.2: Docker クラスタを用いて構築したマルチディスプレイのトポロジ

## 6 結論

### 6.1 本研究の総括

本研究では、先行研究で提案されたSBCマルチディスプレイシステムにおける動作時のパフォーマンスとスケーラビリティの向上を目的とした。提案では、OS仮想化技術を用いてヘッドノード内のフレーム圧縮処理を独立したプロセスとして並列化するとともに、コンテナ間でのフレームデータの受け渡しを共有メモリで行う機能を実装した。提案手法におけるヘッドノードの機能は、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種類のコンテナとして設計した。圧縮コンテナはディスプレイ数と同数を動作させ、ヘッドノード内の単一スレッドで行われていたフレームの圧縮処理を複数の圧縮コンテナで行うことで1フレームあたりに要する処理時間を短縮している。また、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Linuxの機能である共有メモリ(SYSTEM V IPC)を使用し、カーネルの機能を使用せずに、プロセス間のデータ受け渡しを可能にする実装を行った。共有メモリを利用することで、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ことができた。

本研究の評価では、提案手法の効果を検証するために、提案手法を用いて実装したヘッドノード上のフレーム処理時間を計測し、既存システムのヘッドノード内でのフレーム処理時間との比較を行った。実験では4面構成および9面構成のマルチディスプレイを構築することを想定し、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じ数の圧縮コンテナを動作させた。結果、既存手法のヘッドノードではフレーム処理に4面構成の場合は49ms、9面構成の場合は117msを要していたのに対し、提案手法ではそれぞれ21ms、52msとなっており、提案手法によるフレーム処理並列化の効果によって処理時間が短縮されている事が確認できた。また、ディスプレイ数を増加させた場合においてはフレームの拡大・分割に要する時間は既存手法と同様に増加しているものの、圧縮処理では既存手法と比較して提案手法では処理時間の増加は小さくなっていること、システムのスケーラビリティが改善されていることも確認できた。

### 6.2 将来課題

今後、本研究の研究成果をさらに発展させるためには、以下の2点が将来課題となる。

- 提案手法を用いたマルチディスプレイシステムにおける同期制御処理の実装
- 圧縮パラメータ変更処理の実装

## 提案手法を用いた MD における同期制御処理の実装

本研究の実装範囲はヘッドノード内のフレーム処理プロセスのみにとどまっており、共有メモリから受け取ったフレームデータの描画処理までは実装できていない。ディスプレイ上に動画を表示する場合にはヘッドノードとディスプレイノードの間での同期制御処理が不可欠である。そのため、提案手法を用いたシステム内で同期制御処理を行う機能の実装が将来課題としてあげられる。

## 圧縮パラメータ変更処理の実装

既存手法を用いたマルチディスプレイシステムでは、各ディスプレイノード上のフレーム受信・展開処理がボトルネック化するのを防ぐために、同期メッセージを利用した JPEG パラメータ (YCbCr サンプル比・品質係数) のフィードバック制御機能が実装されている。本研究での提案手法を用いたヘッドノードでは、複数のコンテナでフレーム圧縮処理を行うため、1つのプロセス内でフレームの圧縮を行っていた既存手法と比較してフレームレートを一定に保つためのフィードバック機能の実装は複雑になると考えられる。そのため、複数のコンテナで行われるフレームの圧縮処理を統合的に管理し、制御できるような手法の考案が必要であると考えられる。

## 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応用情報システム研究部門 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。

日頃より、丁寧かつ熱心な御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の意を表します。

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機構サービス創出・支援部門 春本要教授に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並びに事務補佐員片岡小百合氏に心より感謝致します。

## 参考文献

- [1] 産学官連携による共同研究強化のためのガイドライン【追補版】.  
[https://www.meti.go.jp/policy/innovation\\_corp/200630\\_guideline\\_tsuiho\\_r2.pdf](https://www.meti.go.jp/policy/innovation_corp/200630_guideline_tsuiho_r2.pdf),  
Jun. 2020.
- [2] 科学技術・学術政策研究所. <https://www.nistep.go.jp/>, Feb. 2022.
- [3] 2021 年（令和 3 年）科学技術研究調査結果の要約.  
<https://www.stat.go.jp/data/kagaku/kekka/youyaku/pdf/2021youyak.pdf>, Dec. 2021.
- [4] 特定非営利活動法人バイオグリッドセンター関西スパコン創薬プロジェクト.  
<http://www.biogrid.jp/kbdd/>, Feb. 2022.
- [5] J.B. Brown, Masahiko Nakatsui, and Yasushi Okuno. Constructing a Foundational Platform Driven by Japan's K Supercomputer for Next-Generation Drug Design. *Molecular Informatics*, Vol. 33, pp. 732–741, Jul. 2014.
- [6] スーパーコンピュータ「富岳」： 富士通. <https://www.fujitsu.com/jp/about/businesspolicy/tech/fugaku/>, Feb. 2022.
- [7] ポスト「京」で重点的に取り組むべき社会的・科学的課題に関するアプリケーション開発・研究開発. [https://www.eri.u-tokyo.ac.jp/LsETD/Post\\_K/bosaitop/index.html](https://www.eri.u-tokyo.ac.jp/LsETD/Post_K/bosaitop/index.html), Feb. 2022.
- [8] Kohei Fujita, Kentaro Koyama, Kazuo Minami, Hikaru Inoue, Seiya Nishizawa, Miwako Tsuji, Tatsuo Nishiki, Tsuyoshi Ichimura, Muneo Hori, and Lalith Maddegedara. High-fidelity nonlinear low-order unstructured implicit finite-element seismic simulation of important structures by accelerated element-by-element method. *Journal of Computational Science*, Vol. 49, , Feb. 2021.
- [9] Rahul Bale, Chung-Gang Li, Masashi Yamakawa, Akiyoshi Iida, Ryoichi Kurose, and Makoto Tsubokura. *Simulation of Droplet Dispersion in COVID-19 Type Pandemics on Fugaku*. Association for Computing Machinery, New York, NY, USA, Jul. 2021.
- [10] Kazuto Ando, Rahul Bale, ChungGang Li, Satoshi Matsuoka, Keiji Onishi, and Makoto Tsubokura. Digital transformation of droplet/aerosol infection risk assessment realized on "Fugaku" for the fight against COVID-19, Oct. 2021.
- [11] 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>, Feb. 2022.

- [12] 名古屋大学情報連携統括本部. <http://www.icts.nagoya-u.ac.jp/en/center/jhpcn/suppl/>, Feb. 2022.
- [13] 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.
- [14] 木戸善之, 石田和也, 伊達進, 下條真司. 低性能計算機を用いたマルチディスプレイシステムの試作. In *IPSJ Computer System Symposium 2019 (ComSys2019)*, Dec. 2019.
- [15] Luciano Soares, Bruno Raffin, and Joaquim Jorge. PC Clusters for Virtual Reality. *IJVR*, Vol. 7, pp. 67–80, Jul. 2008.
- [16] Haeyong Chung, Christopher Andrews, and Chris 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.
- [17] 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.
- [18] 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.
- [19] Luc Renambot, Arun Rao, Rajvikram Singh, Byungil Jeong, Naveen Krishnaprasad, Venkatram Vishwanath, Vaidya Chandrasekhar, Nicholas Schwarz, Allan Spale, Charles Zhang, Gideon Goldman, Jason Leigh, and Andrew Johnson. SAGE : the Scalable Adaptive Graphics Environment. *Proceedings of the 4th Workshop on Advanced Collaborative Environments (WACE2004)*, Sep. 2004.
- [20] Gregory P. Johnson, Gregory D. Abram, Brandt Westing, Paul Navr'til, and Kelly Gaither. DisplayCluster: an Interactive Visualization Environment for Tiled Displays. In *Proceedings of the 16th International Conference on Cluster Computing*, pp. 239–247, Sep. 2012.
- [21] Petr Holub, Martin Srom, Martin Pulec, Jiri Matela, and Martin Jirman. GPU-Accelerated DXT and JPEG Compression Schemes for Low-Latency Network Transmissions of HD, 2K, and 4K Video. *Future Generation Computer Systems*, Vol. 29, pp. 1991–2006, Oct. 2013.

- [22] G. K. Wallace. The JPEG Still Picture Compression Standard. *IEEE Transactions on Consumer Electronics*, Vol. 38, No. 1, pp. 18–34, Feb. 1992.
- [23] 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.
- [24] 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.
- [25] M. Surya Deekshith Gupta, Vamsikrishna Patchava, and Virginia Menezes. Health-care 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.
- [26] Raspberry Pi 4 Model B. <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>, Feb. 2022.
- [27] Orange Pi Zero Plus2. <http://www.orangepi.org/OrangePiZeroPlus2/>, Feb. 2022.
- [28] Banana Pi オープンソースプロジェクト. <http://www.banana-pi.org/m64.html>, Feb. 2022.
- [29] Raspbian. <https://www.raspbian.org/>, Feb. 2022.
- [30] 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.
- [31] Noor Ibraheem, Mokhtar Hasan, Rafiqul Zaman Khan, and Pramod Mishra. Understanding Color Models: A Review. *ARPJ Journal of Science and Technology*, Vol. 2, pp. 265–275, Jan. 2012.
- [32] Paul T. Chiou, Yu 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.
- [33] Empowering App Development for Developers — Docker. <https://www.docker.com/>, Feb. 2022.
- [34] Adrian Mouat. *Using Docker.* ” O'Reilly Media, Inc.”, Dec. 2015.
- [35] Kubernetes. <https://kubernetes.io/>, Feb 2022.
- [36] The Single Board Computer Database. <https://hackerboards.com/>, Feb. 2022.

- [37] GitHub — GitHub is a development platform inspired by the way you work. From open source to business, you can host and review code, manage projects, and build software alongside millions of other developers. <https://github.co.jp/>, Feb 2022.
- [38] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, Oct. 2010.
- [39] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management.* ” O'Reilly Media, Inc.”, Nov. 2005.
- [40] Big Buck Bunny. <https://peach.blender.org/>, Feb. 2022.
- [41] 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.
- [42] OpenCV. <https://opencv.org/>, Feb. 2022.
- [43] libjpeg-turbo. <https://libjpeg-turbo.org/>, Feb. 2022.
- [44] boost C++ LIBRARIES. <https://www.boost.org/>, Feb. 2022.
- [45] gstreamer: open source framework. <https://gstreamer.freedesktop.org/>, Feb. 2022.
- [46] FFmpeg. <https://www.ffmpeg.org/>, Feb. 2022.
- [47] GCC,GNU コンパイラコレクション. <https://gcc.gnu.org/>, Feb. 2022.
- [48] Jetson Nano Developer Kits and Module. <https://www.nvidia.com/ja-jp/autonomous-machines/embedded-systems/jetson-nano/>, Feb. 2022.
- [49] 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, Nov 2018.
- [50] Rudolfs Bundulis and Guntis Arnicans. Infiniviz-Virtual Machine Based High-Resolution Display Wall System. In *DB&IS (Selected Papers)*, pp. 225–238, Dec. 2017.
- [51] Steam : Quake III Arena. [https://store.steampowered.com/app/2200/Quake\\_III\\_Arena/](https://store.steampowered.com/app/2200/Quake_III_Arena/), Feb. 2022.

- [52] Xin Liu, Wenfeng Shen, Baohua Liu, Qin Li, Rong Deng, and Xuehai Ding. Research on Large Screen Visualization Based on Docker. *Journal of Physics: Conference Series*, Vol. 1169, , Feb 2019.