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

高畑 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

大学院博士前期課程修士学位論文

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

高畑 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　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記す．

1.1　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 [1] 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可能であり，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．

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 1.1 に示す．

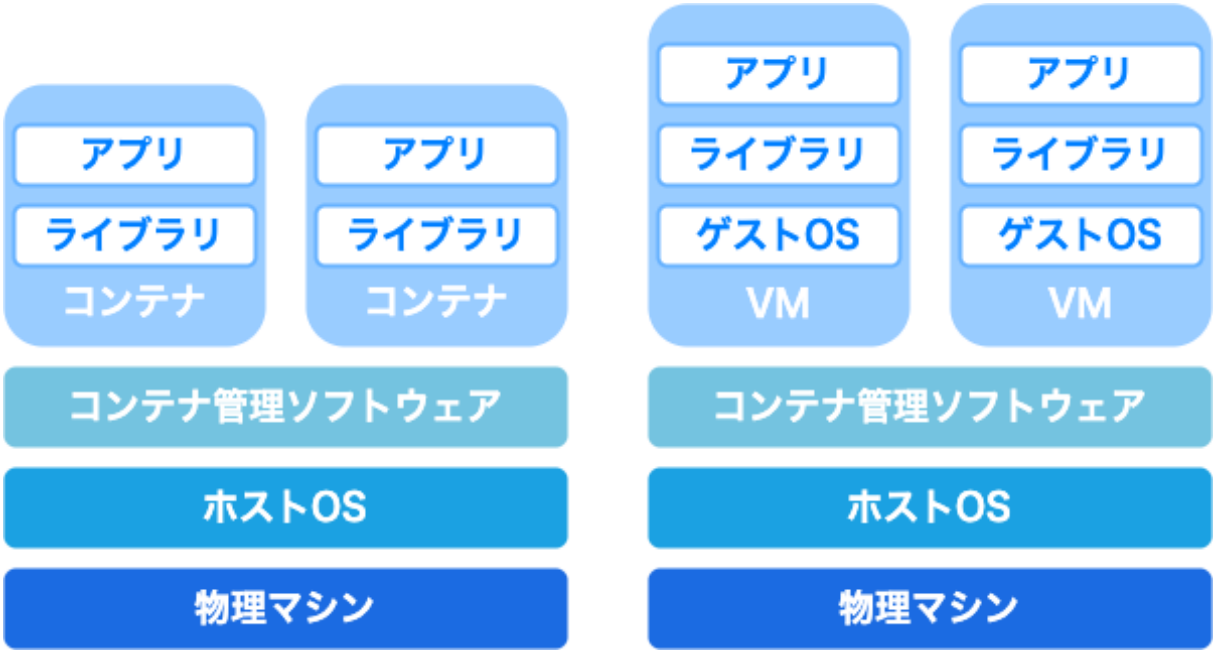

図 1.1: ハイパーバイザ型仮想化とコンテナを用いた OS 仮想化

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 [2] 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．

1.2 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 [3]．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，様々な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．

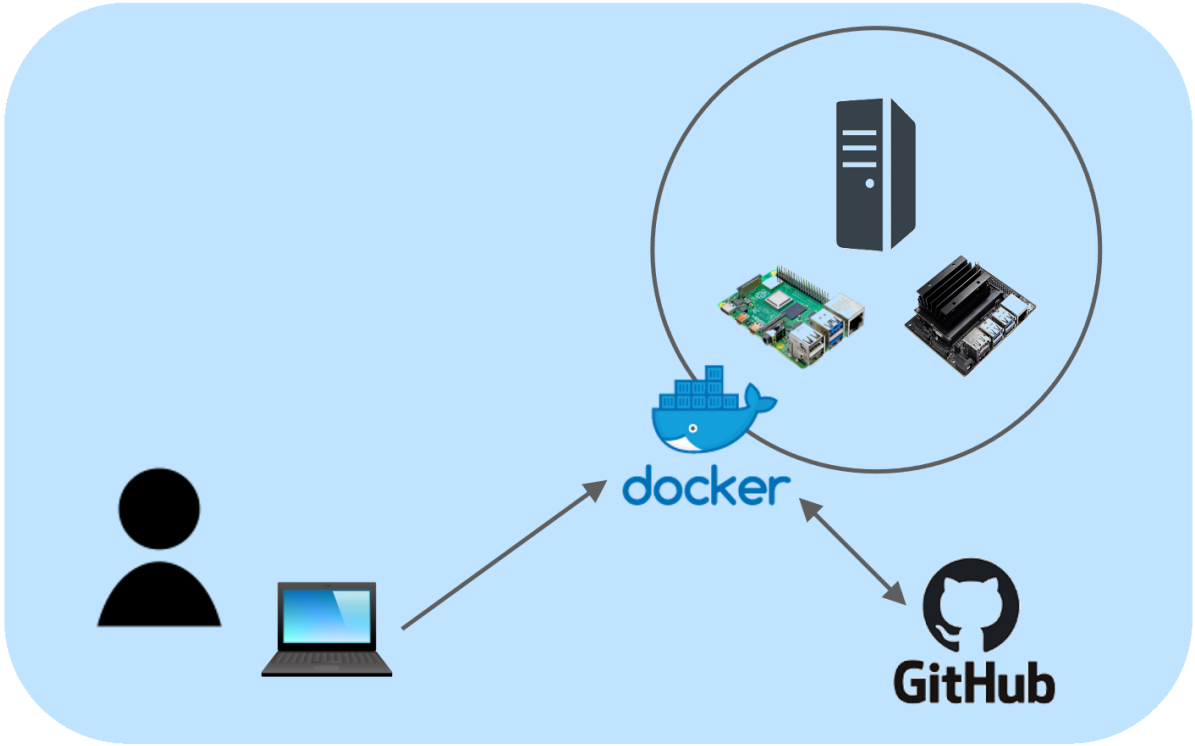

図 1.2: Dockerを用いた環境構築

Dockerを用いた環境構築の概要図を図1.2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 [4] 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．

1.3　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．

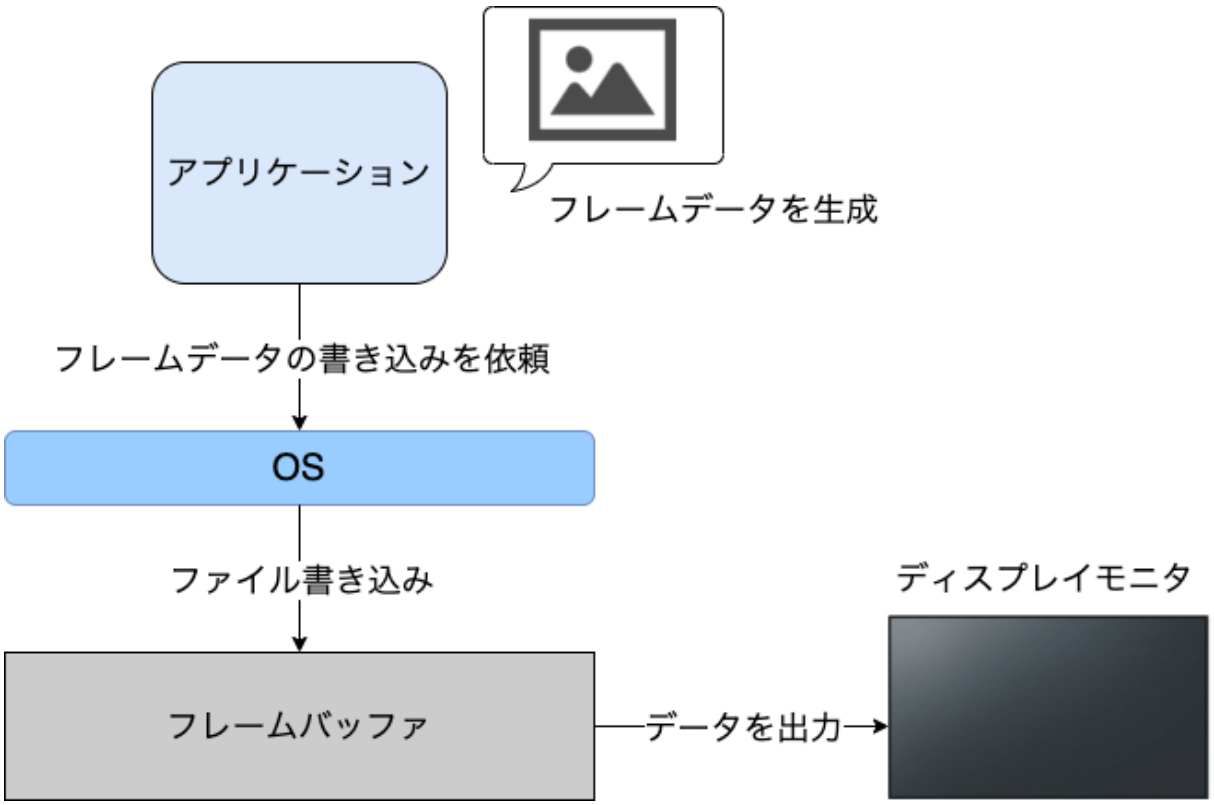

図 1.3: フレームバッファを用いたディスプレイ表示

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 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 priviledge 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．

1.4　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，ヘッドノード内で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．

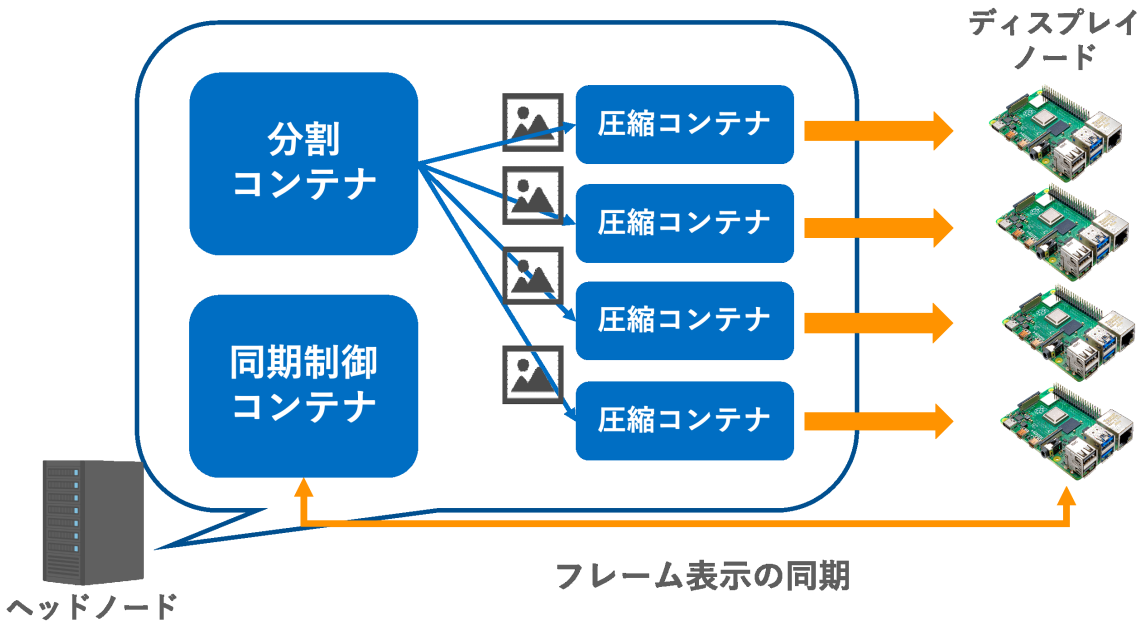

図 1.4: フレーム処理の並列化

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．

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) [5, 6] 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．

以上の方式を検討した結果，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 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)．

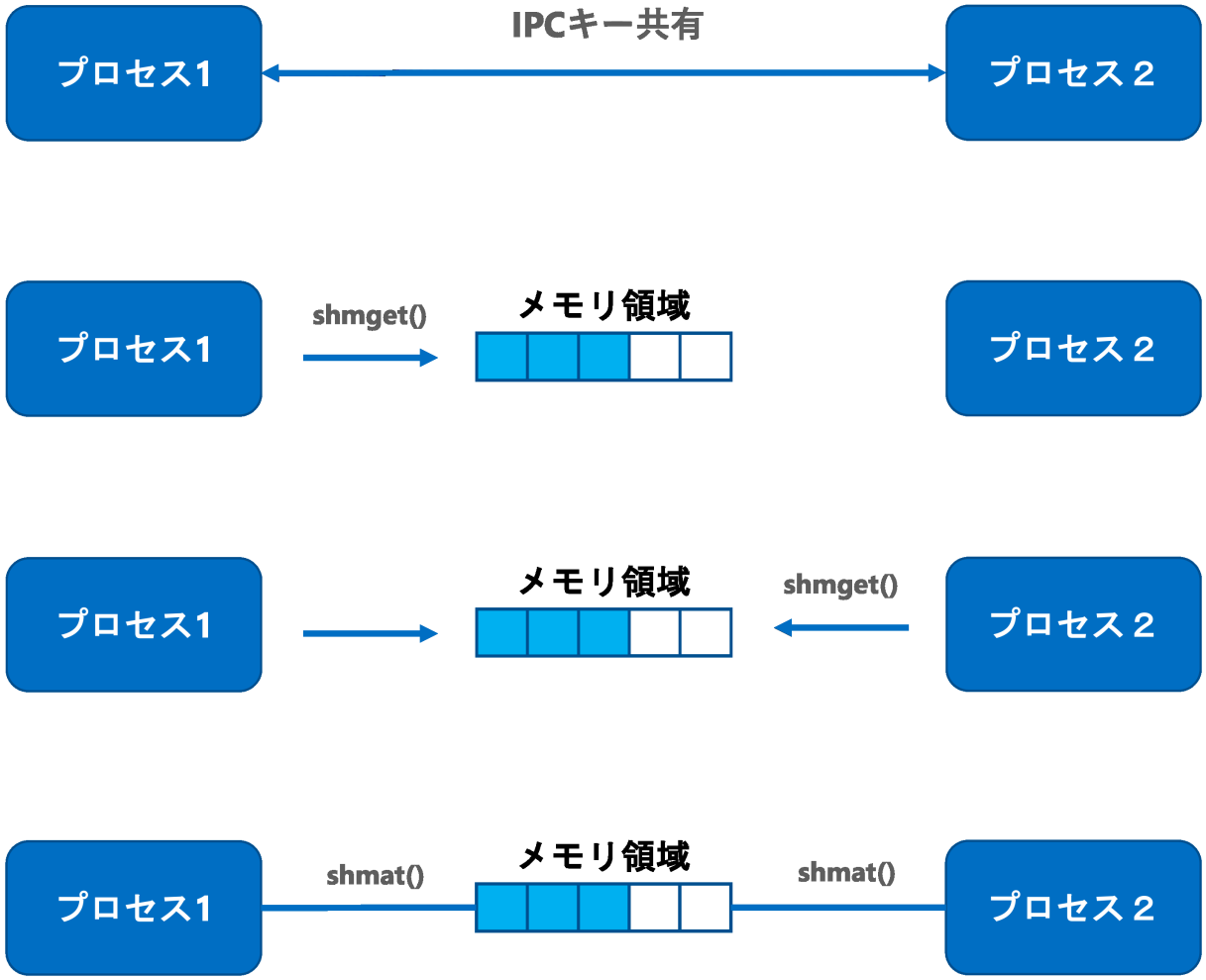

図 1.5: 共有メモリ使用の流れ

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に持つ．Mat 構造体の内容を図 3.6 に示す．

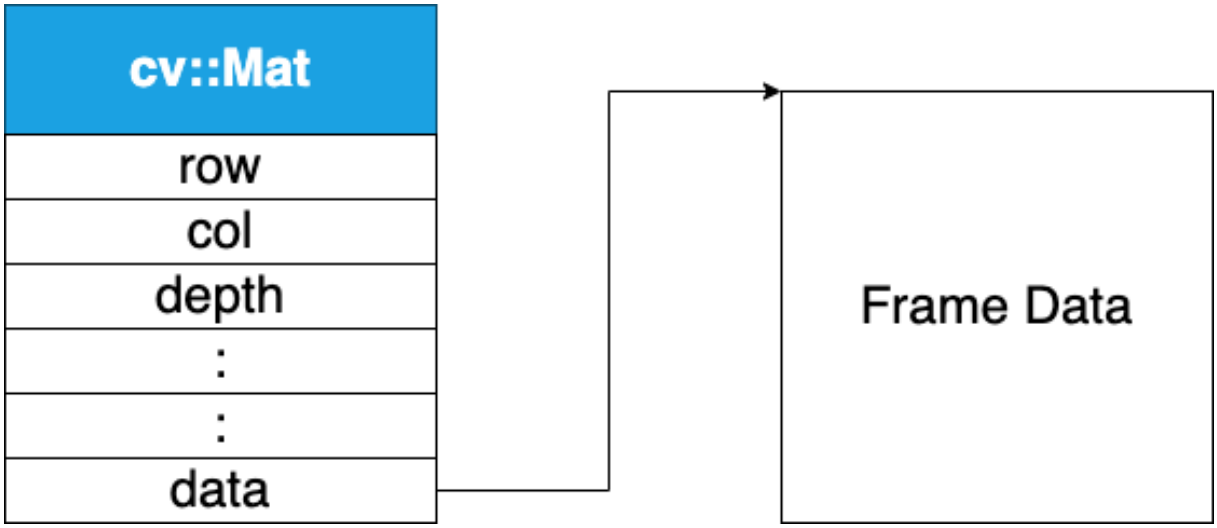

図 1.6: OpenCV における Mat 構造体

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 3byte 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．

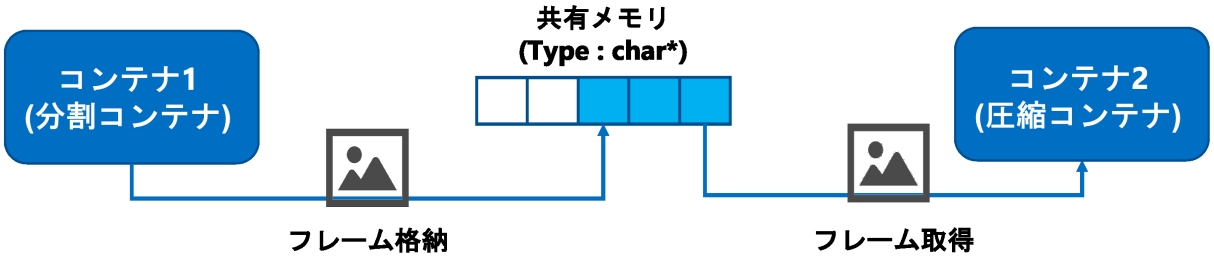

図 1.7: 共有メモリの使用例 2

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 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圧縮コンテナは全て 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．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ファイルにデータとして保存される．その後起動された圧縮コンテナが該当パスに存在するフォルダを獲得し，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 キーを共有し，同一のメモリ領域にアクセスする事が可能になる．

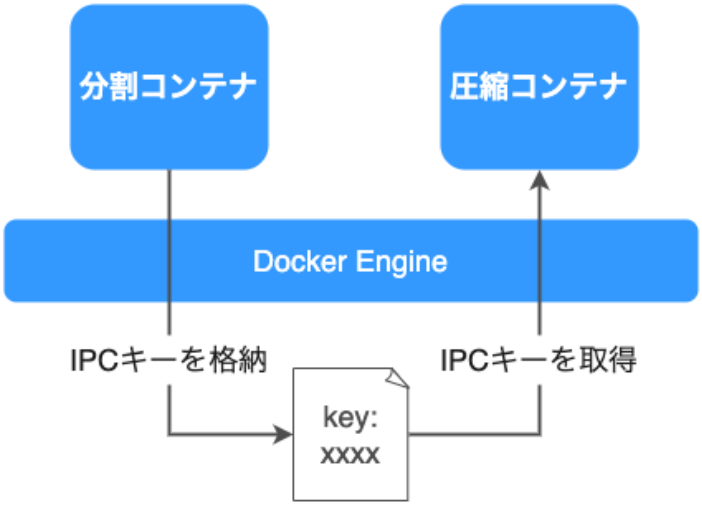

図 1.8: 共有メモリの使用例 2

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ず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える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顔像度として，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 (圧縮コンテナ数) x 5.9 x 8 (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．

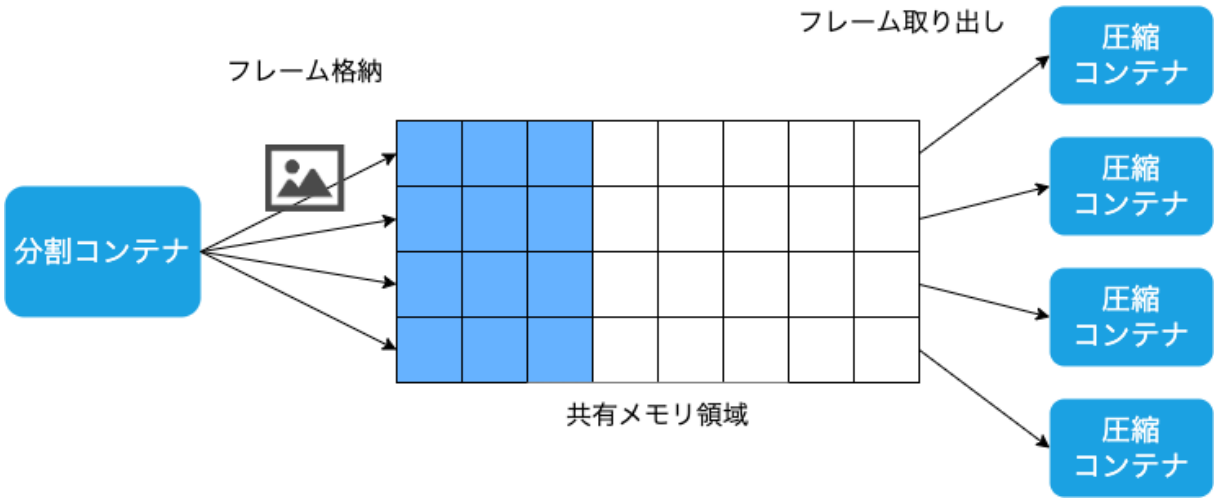

図 1.9: 共有メモリ内におけるフレームのバッファリング

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述べる，図 3.9 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．

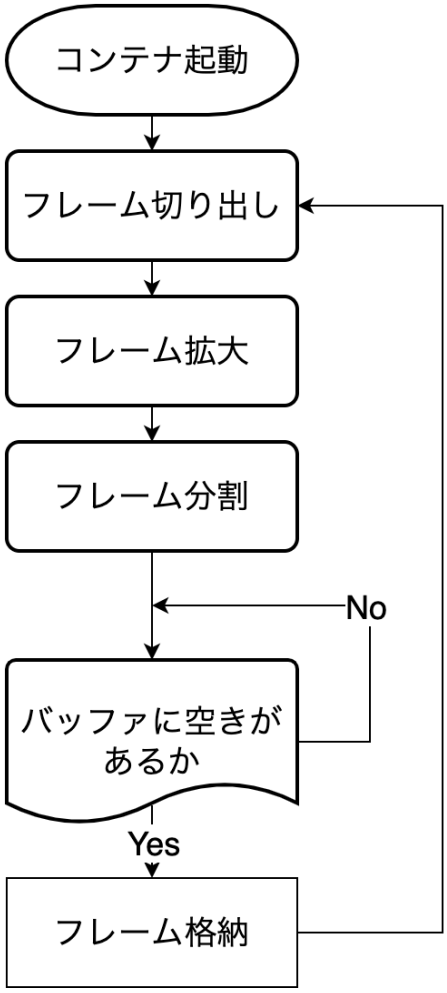

図 1.10: 分割コンテナの動作フロー

圧縮コンテナは，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．この時，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ると待機状態を終了し，フレームを格納して次のフレームの処理へと移る．

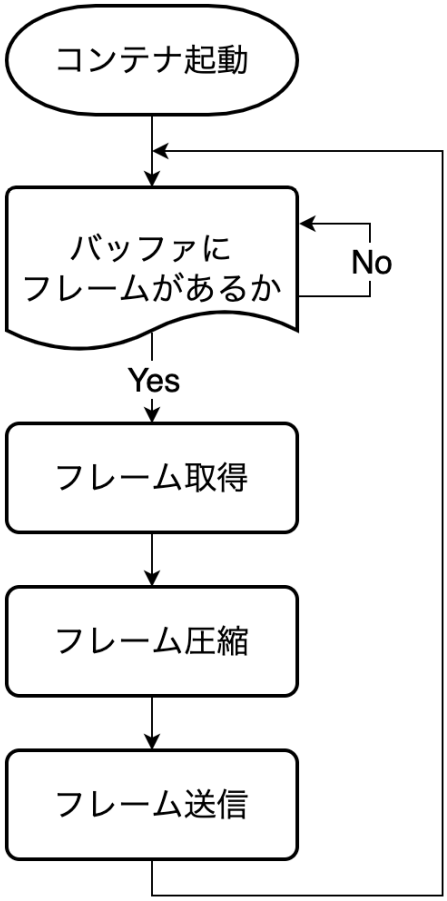

図 1.11: 圧縮コンテナの動作フロー

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 3.8 および図 3.9 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，図 3.10 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．

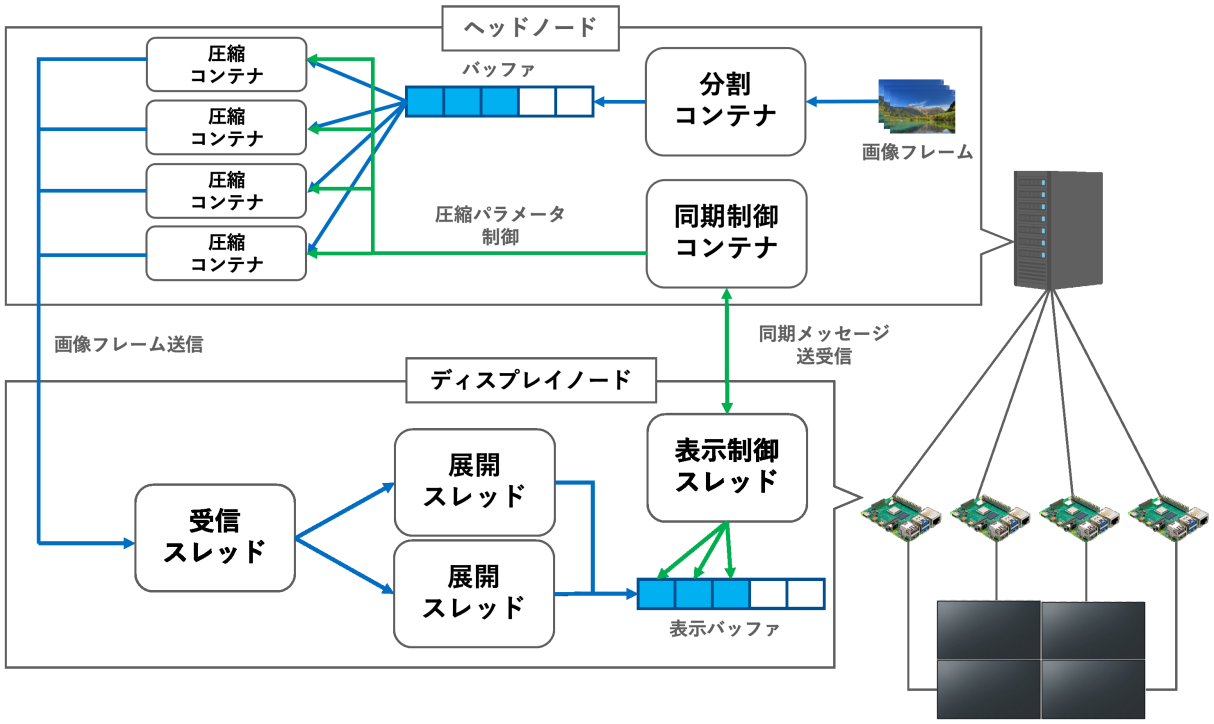

図 1.12: 提案手法を用いた MD の動作フロー

参考文献

[1] Empowering App Development for Developers — Docker. https://www.docker.com/, Feb. 2022.

[2] Kubernetes. https://kubernetes.io/, Feb 2022.

[3] The Single Board Computer Database. https://hackerbords.com/, Feb. 2022.

[4] 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.

[5] Michael Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[6] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.