卒業研究論文

解像度非依存型動画像処理ライブラリ RaVioli におけるパイプライン記法の実現

指導教員　　松尾 啓志 教授
　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 17 年度入学　17115037 番

大野 将臣

平成 21 年 2 月 10 日

解像度非依存型動画像処理ライブラリ RaVioli におけるパイプライン記法の実現

大野 将臣

内容梗概

近年，侵入者検知システムなどリアルタイム性の重要なシステムの開発が盛んに行われている．また，ビデオカメラなどの入力装置からリアルタイムに画像をキャプチャ可能な環境が整ってきたことや，汎用計算機の高性能化により高度な画像処理が実行可能となってきた．そのため汎用システム上でリアルタイム動画像処理を行うことが多くなると予想される．しかし，汎用システムでは並行実行プロセスなどの外乱により，リアルタイム動画像処理に必要な CPU リソースの確保が困難である．

そこで，擬似的なリアルタイム性を保証する動画像処理ライブラリ RaVioli がある．RaVioli では確保可能な CPU リソースの減少によりリアルタイム動画像処理が困難になった場合，解像度をさせることで処理量を調節する．こうすることで，擬似的なリアルタイム性を保証している．

一方，動画像処理の高速化に，処理のパイプライン化がある．一般に動画像処理には，処理を複数のステージに分割してパイプライン的に実行可能なものが多い．

そこで本研究では，RaVioli に容易にパイプライン処理を実現できる記法を提案し，実装した．これにより，プログラマは動画像処理をパイプライン化する煩雑な処理を省略することが可能となる．さらに，各パイプラインステージの処理量に応じたステージの統合・並列化を提案，実装した．これにより，各ステージ間のスループットを平均化することが可能となり，効率の良い並列化を実現することが可能となる．

サンプルプログラムを用いて評価を行った．パイプラインステージの統合・並列化を行った場合と，行わなかった場合で解像度の変化を比較し，解像度の低下を抑えられることを確認する．

解像度非依存型動画像処理ライブラリ RaVioli におけるパイプライン記法の実現

目次

1　はじめに

近年，侵入者検知システムや自動車の衝突回避システムなどリアルタイム性の重要なシステムの開発が盛んに行われている．また，汎用計算機の高性能化と価格低下により，高性能な計算機を容易に手に入れることが可能である．そのため今後，汎用 PC および汎用 OS 上でリアルタイム動画像処理を行うことが多くなると予想される．

しかし，Linux に代表される汎用 OS 上で，1/30 または 1/60 秒毎に処理を行うリアルタイム動画像処理の実現は困難である．その主な理由として，1 フレームあたりの処理量の変動や，他のプロセスによる使用可能な CPU リソースの変動があげられる．

そこで，汎用システム上で擬似的なリアルタイム性を保証する動画像処理ライブラリ RaVioli がある．RaVioli では処理対象画像の解像度を CPU 使用率や平行実行プロセスによる負荷に応じて，自動的に変動させる．これによって処理量を調節し，擬似的なリアルタイム動画像処理を実現している．

このように動的に解像度を変動させる場合，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムの記述が必要になってくる．そこで，RaVioli ではプログラマから 1 フレームあたりの幅および高さを隠蔽し，解像度をライブラリ内で制御している．こうすることで人間の映像認識過程に存在しない画素およびフレームといった概念を排除することが可能となり，より直感的な動画像処理プログラミングが実現できる．

一方，動画像処理の高速化手法として，並列化が挙げられる．この方法には，SIMD 命令を用いたベクトル並列化や，ブロック分割・パイプライン化によるスレッド並列化があげられるが，本稿ではパイプライン化に着目して並列化を行った．一般的に動画像処理には，処理を複数のステージに分割してパイプライン的に実行することが可能なものが多い．たとえば顔検出では，背景差分，エッジ抽出，ハフ変換の各処理をステージとして分割し，パイプライン的に並列実行することが可能である．近年では計算機の低価格化によりマルチコア CPU が比較的安価で容易に手に入るようになってきた．そのためマルチコア環境でマルチスレッドを用いたパイプライン処理が有効である．

一般的に効率のよい並列化を行うには，各ステージの処理量を均等にする必要がある．しかし，例えば顔検出の場合では，背景差分，エッジ抽出の処理量に対しハフ変換の処理量が多い．このとき，ハフ変換を行うステージがボトルネックとなり，パイプライン全体のスループットが低下する．このように効率の良い並列化を行うことは

困難である．また，各ステージの処理量を意識し，処理量を均等にするようなプログラミングは困難であり，動画像処理の本質ではない部分でプログラマの負担となる．

そこで本研究では，動画像処理ライブラリ RaVioli にパイプライン記法を追加し，容易にパイプライン処理を実現可能な環境をユーザに提供する．さらに，負荷に応じ自動的にパイプラインステージの統合・並列化を行う機能を追加する．現時刻での各ステージの処理時間に応じてパイプラインステージの統合または並列化を行うことによって，各ステージのスループットを平均化し効率のよい並列化を実現する．

本論文では，2 章で本研究の背景と動画像処理ライブラリ RaVioli について述べ，3 章でパイプライン記法とステージの統合・並列化を提案し，4 章でその実装について述べる．次に 5 章で提案の評価とそれに対する考察を述べる．最後に 6 章で本論文全体をまとめる．

2 研究背景

2.1 動画像処理

近年，ビデオカメラなどの入力装置からリアルタイムに画像をキャプチャ可能な環境が整ってきた．また，汎用計算機の高性能化によって，従来では不可能であった顔認証などの処理量の多い高度な動画像処理が可能になった．そのため今後汎用計算機および汎用 OS 上でリアルタイム動画像処理を行うことが多くなると予想される．

しかし汎用システム上でリアルタイム動画像処理を行う場合，処理に必要な CPU リソースの確保が困難である．その原因として 1 フレームあたりの処理量が変動することや，平行実行されている他のプロセスによる使用可能な CPU リソースの減少などが挙げられる．たとえば，顔検出を行うプログラムでは，背景画像とキャプチャした画像との差分をとり，エッジ抽出を行い，その結果に対しハフ変換を行う．このとき背景画像とキャプチャした画像に差がない場合とある場合とで，ハフ変換の処理量が変動する．また，汎用 OS 上では複数のプロセスが並行実行されている．それらのプロセスによって使用可能な CPU リソースの変動がおこるため，リアルタイム動画像処理に必要な CPU リソースが常に確保可能だという保証はない．

そこで，分散並列計算機の 1 種である PC クラスタを利用した，実時間並列画像処理アプリケーション構築環境 RPV[1] がある．しかし，これは高速ネットワークに接続された PC クラスタを利用したものであり，汎用システム上でリアルタイム動画像処理を実現するものではない．

一方で，擬似的なリアルタイム性を保証する動画像処理ライブラリ RaVioli[2] が提案

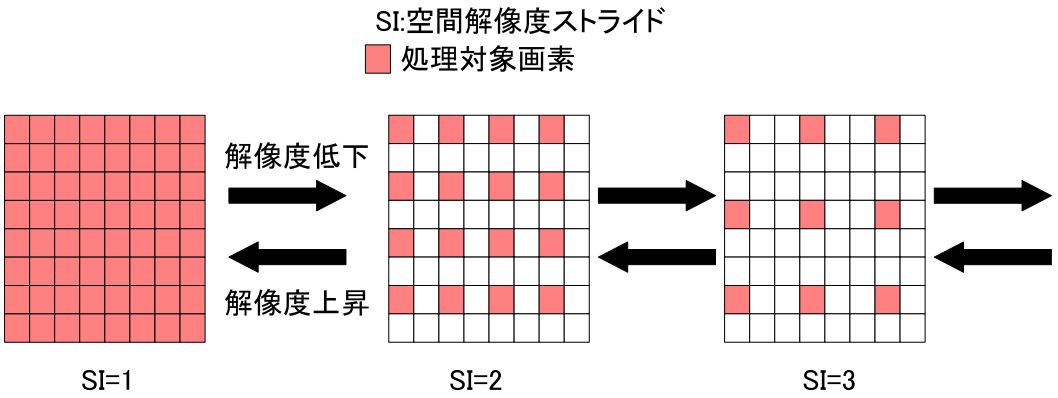


図 2.1: 空間解像度ストライドの変更

されている．RaVioli では CPU リソースの変動によりリアルタイム処理が困難になった場合，解像度を自動調節することで処理量を減らしリアルタイム性の保証を行う．次節で RaVioli の詳細と問題点について述べる．

2.2 RaVioli

2.2.1 処理量の自動調節

一般的に汎用 PC および汎用 OS では，1 フレームあたりの処理量の変動や，並列実行されている他のプロセスによる CPU リソースの減少などによって，リアルタイム動画像処理を行うことは困難である．そこでこれを解決する方法として，動画像の解像度を低減させ処理量を減らす方法が考えられる．

動画像における解像度には**空間解像度**および**時間解像度**の 2 種類がある．空間解像度とは 1 フレームを構成する画素数である．一方，時間解像度とはフレームレートである．RaVioli は各解像度を制御する解像度ストライドを持ち，CPU リソース量に応じてこれを変更することで処理量の低減を実現している．

RaVioli ではユーザが指定した優先度に応じて空間解像度および時間解像度を自動的に変動させることが可能である．たとえば，高いフレームレートを維持し，厳密にリアルタイム性を保証したい場合，空間解像度を低減させ，時間解像度を維持したままリアルタイム処理をする．また，顔認証などのように厳密なリアルタイム処理が必要ではなく，画像の精度が必要な処理の場合には，時間解像度を低減させ，空間解像度を維持する．このようにユーザは処理内容に応じて優先度を設定することで目的の解像度を維持したリアルタイム処理が可能である．

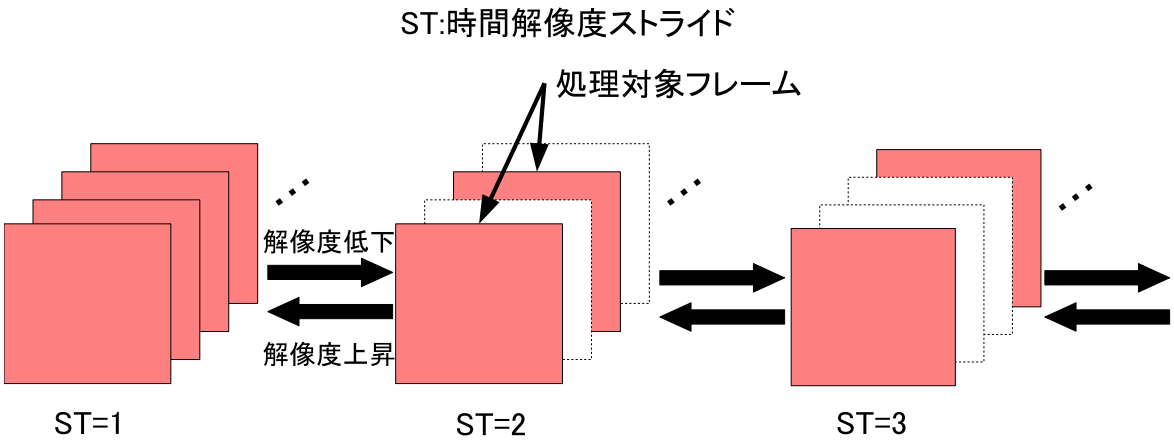


図 2.2: 時間解像度ストライドの変更

空間解像度　空間解像度低下時の処理方法を**図 2.1** に示す．空間解像度とは 1 フレームにおける画素数である．RaVioli で空間解像度の変更を行う場合，1 フレーム上で処理する画素の間隔を示す**空間解像度ストライド**を大きくするかまたは小さくすることで空間解像度の変更を行っている．たとえば，空間解像度を低減させる場合には，空間解像度ストライドを大きくし，処理対象画素の間隔を大きくする． 図 2.1 の場合，空間解像度ストライド $SI = 1$ のとき，画像中の全ての画素を処理する．解像度低下が発生し空間解像度ストライドが $SI = 2$ に増加すると，処理対象画素は 1 つおきとなり，全体の処理画素数は $SI = 1$ のときの 1/4 となる．さらに解像度低下が発生し $SI = 3$ になると，処理が素数は 1/9 となる．

時間解像度　時間解像度低下時の処理方法を**図 2.2** に示す．時間解像度とはフレームレートである．RaVioli で時間解像度の変更を行う場合，処理するフレームの間隔を示す**時間解像度ストライド**を大きくするかまたは小さくすることで時間解像度の変更を行っている．たとえば，時間解像度を低減させる場合には，処理するフレームの間隔を大きくする．図 2.2 の場合，時間解像度ストライド $ST = 1$ のとき，全てのフレームを処理する．解像度低下が発生し時間解像度ストライドが $ST = 2$ に増加すると，フレームを 1 つ飛ばしで処理することになり，フレームレートは 1/2 になる．さらに解像度低下が発生し $ST = 3$ となると，フレームレートは 1/3 となる．

このように解像度を動的に変動させる場合，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムの記述が必要になってくる．そこでこの問題に対する RaVioli の解決策を次節で示す．

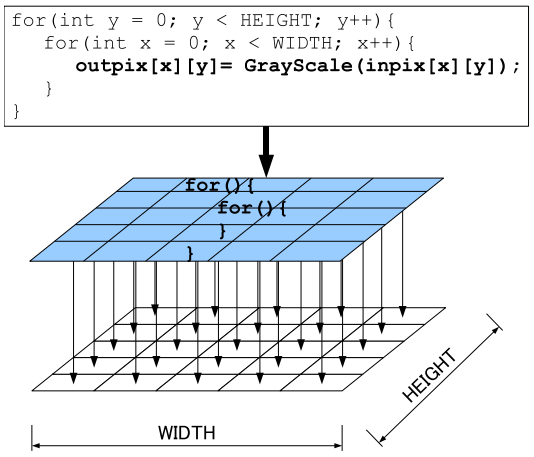


図 2.3: RaVioli 不使用時プログラム記述例

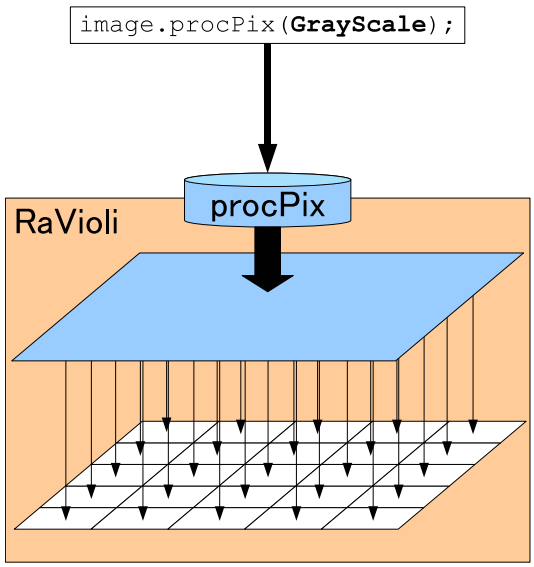


図 2.4: RaVioli 使用時プログラム記述例

2.2.2 動画像処理の抽象化

解像度を動的に変動させる場合，1 フレームあたりの画素数やフレームレートの変動に対応したプログラムの記述が必要になってくる．そこで RaVioli ではプログラマから，1 フレーム中の高さと幅の画素数やフレームレートを隠蔽し，空間解像度および時間解像度をライブラリ内で制御している．そうすることでプログラマは 1 フレームあたりの画素数やフレームレートを意識した記述を省略できる．

RaVioli 不使用の場合では，動画像のフレーム数や画像の高さまたは幅のループイタレーションを記述することになる．ここで空間解像度および時間解像度が動的に変動させたい場合，動画像のフレーム数や画像の高さや幅もしくはインクリメント幅が動的に変動させることになる．そのためイタレーション回数やインクリメント幅の変動に対応したループイタレーションの記述が必要になる．これは動画像処理の本質とは異なった部分でプログラマの負担となる．

RaVioli では動画像の構成要素である画素またはフレームに対する処理のみを記述し，それをライブラリで提供しているメソッドに渡すことで，画像中の全ての構成画素に処理を施すことが可能である．たとえば，カラー画像をグレースケール画像へ変換する処理では，通常**図 2.3** のような，高さと幅の二重ループ内で対象画素をグレースケールにする処理を呼び出すようなループイタレーションで記述される．動的に解像度が変動する場合，HEIGHT および WIDTH の値の変動に対応したプログラムの記述

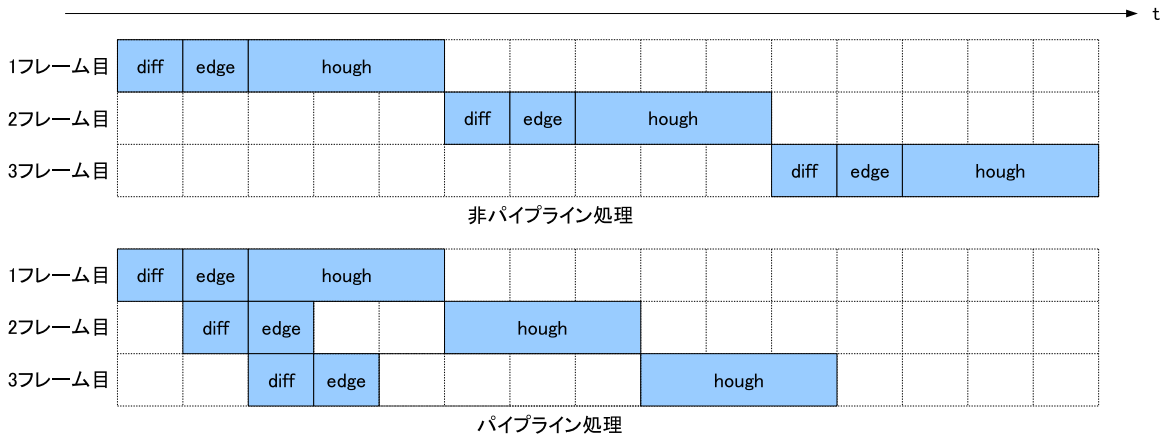


図 2.5: パイプライン処理

が必要になる．一方，RaVioli を使用した場合では，**図 2.4** のように記述される．ユーザは対象画素をグレースケール化する関数 GrayScale を記述し，すべての構成画素に処理を行うメソッド procPix に渡す．こうすることで，procPix はすべての構成画素に GrayScale の処理を施す．

このようにプログラマから画像の幅や高さを隠蔽することで，プログラマに解像度の変動を意識させずに処理量を変動させることが可能となる．さらに，本来人間の動物体認識過程に存在しない画素やフレームといった概念を意識しない直感的なプログラミングを可能にさせる．また，動画像処理中の繰り返し単位が明確となりデータ並列性の抽出が容易となる．

2.3 問題点

動画像処理には，処理を複数のステージに分割してパイプライン的に実行可能なものが多い．そのため，動画像処理にはパイプライン処理が有効である．しかし，動画像処理をパイプライン化するに当たって問題となる点が存在する．次節以降で一般的に動画像処理をパイプライン処理化するときの問題点と RaVioli でパイプライン処理を実現するときの問題点を示す．

2.3.1 パイプライン化の問題点

一般的に動画像処理には，パイプライン処理が有効である．たとえば，顔検出プログラムでパイプライン化を行わなかった場合では，**図 2.5** 上段のように，1 フレーム目の背景差分 diff，エッジ抽出 edge，ハフ変換 hough のすべての処理が終了するまで 2 フレーム目の処理を開始することができない．しかし，パイプライン化を行った場合

では，図 2.5 下段のように，各処理をステージとして分割し，実行する．このとき，各ステージの処理は並列に実行することが可能であるため，フレームの処理が終了するのを待つことなく次のフレームの処理を開始することが可能になる．そのため，図 2.5 の場合で無限のストリームが流れるとき，パイプライン化した場合，しなかった場合と比べ 40%高速化される．

しかし効率の良い並列化を行うには各ステージの処理量を平均化する必要がある．図 2.5 下段の 2 フレーム目と 3 フレーム目の edge 処理が終了した後，hough 処理が開始されるまで待ちが発生している．これは，前のフレームの hough 処理に時間が掛かっているためである．このように，各ステージの処理量が不均衡な場合，このようなパイプラインストールをなくすことは難しい．しかし，動画像処理ではフレームや処理内容によって処理量が変動する可能性があるため，プログラマが各ステージの処理量を意識し，平均化するようなプログラミングは困難であり，大きな負担となる．

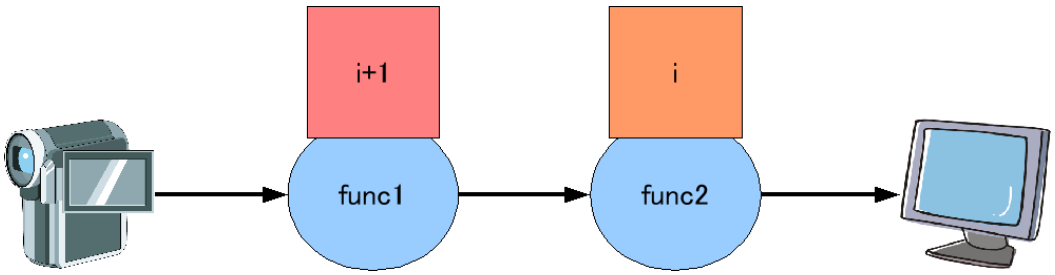
2.3.2 RaVioli の問題点

動画像処理は，動画像の構成要素であるフレームに対する処理をすべてのフレームまたは任意の時間のフレームに繰り返し適用するものである．しかし，人間の動物体認識過程においてフレームといった概念は存在しない．たとえば，動物体検出では，前のフレームと現在のフレームの差分をとり変化のある場合，そこに動物体があると検出する．しかし，人間の場合では前のフレームや現在のフレームといったフレームの処理順やフレームその物が存在しない．このように，人間の認識過程と計算機の動画像処理過程に差があるため，画像処理プログラムは人間にとって煩雑である．

そこで，RaVioli では，人間の動画像認識過程存在しないフレームおよびフレーム処理順の概念を隠蔽することでユーザにより直感的なプログラミングパラダイムを提供している．しかし，フレームやフレームの処理順を隠蔽したため，プログラマはフレームの処理順を指定することができない．そのため，パイプライン処理を記述することができない．

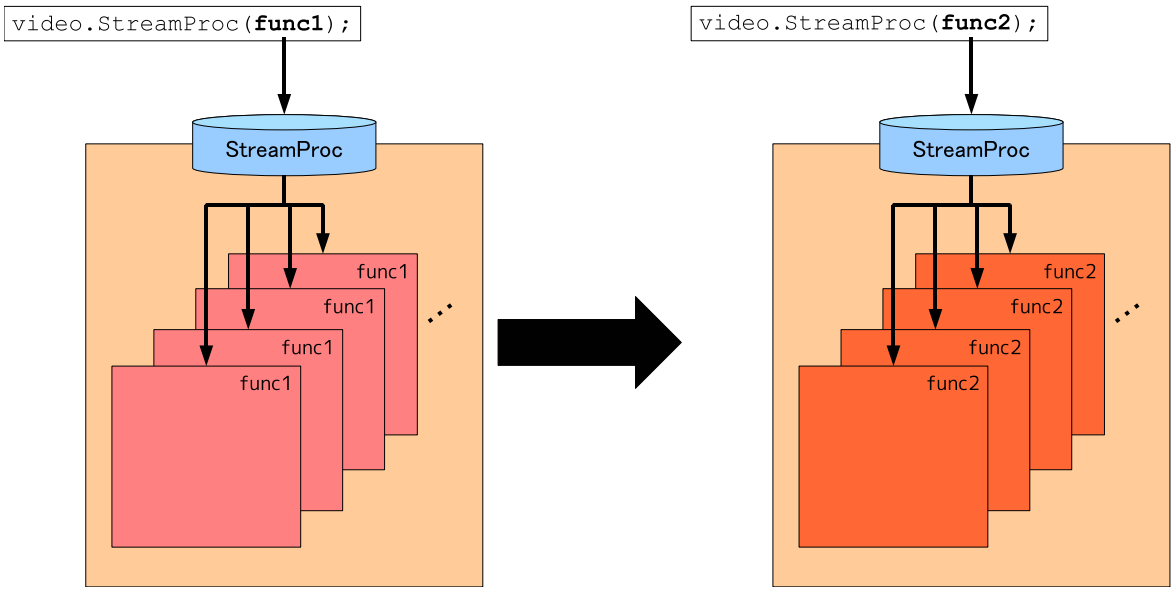
一般的に動画像処理をパイプライン的に処理する場合，**図 2.6** のように i+1 番目のフレームに func1 の処理を行っているイタレーションで，i 番目のフレームに func2 の処理を行う．このように 1 イタレーションないで処理するフレームを指定することでパイプライン処理を実現可能である．

RaVioli を用いた動画像処理プログラムは，**図 2.7** のようになる．プログラマは 1 フレームに対する処理を記述した関数を定義する．全てのフレームに指定された処理を施すメソッド StreamProc にその関数を渡すことで動画像に処理を施す．動画像に対し




```
for(int i = 0; i < MAX_FRAME; i++){
  tmpframe[i+1] = func1(inframe[i+1]);
  outframe[i] = func2(tmpframe[i]);
}
```

図 2.6: RaVioli 不使用時のパイプライン処理の記述




```
video.StreamProc(func1);
video.StreamProc(func2);
```

図 2.7: RaVioli を使用した場合

func1 および func2 の処理を適用したい場合，func1 を引数にし StreamProc を呼ぶことで全てのフレームに func1 の処理を適用する．同様に func2 を引数にし StreamProc を呼ぶことで全てのフレームに func2 の処理を適用する．このように RaVioli では StreamProc が呼ばれたときに，指定された処理を全てのフレームに施すため，func1 と func2 のよう

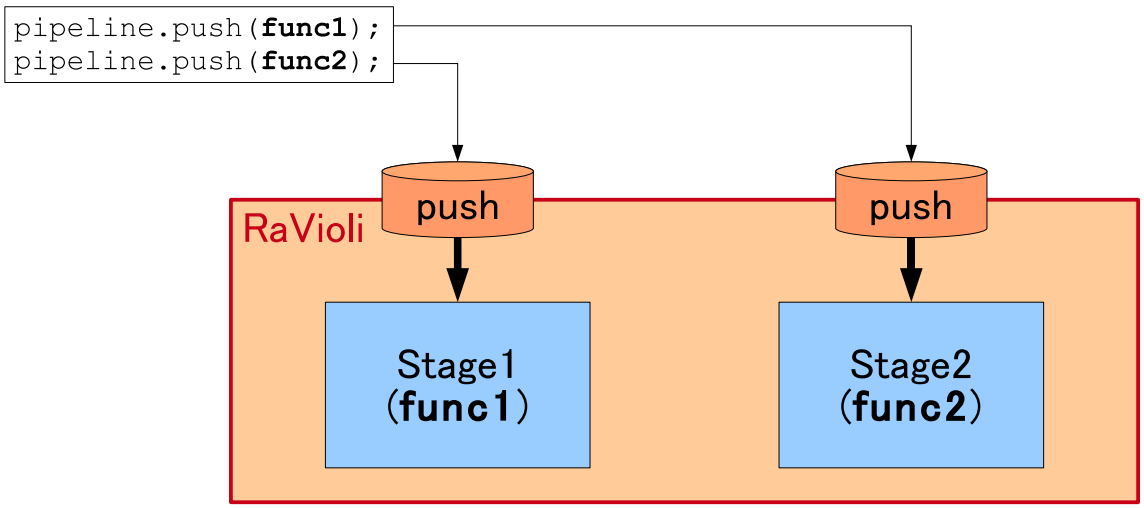


図 3.1: パイプライン処理記法概念図

な複数の処理を並列に異なるフレームに適用することができない．そのため，RaVioli の提供する機能だけではパイプライン処理の記述が不可能であった．この問題の解決策として動画像処理ライブラリ RaVioli におけるパイプライン記法を提案する．

3 提案

3.1 パイプライン処理記法

一般的に動画像処理には，処理を複数のステージに分割してパイプライン実行可能なものが多い．そのため動画像処理にはパイプライン処理が有効である．しかし，既存の RaVioli ではパイプライン処理を記述することができない．そこで，各ステージの処理を記述し，ステージを作成するメソッドに渡すだけでいいようなインターフェースを提案する．

通常，顔検出などの動画像処理をパイプライン的に処理したい場合，プログラマは背景差分処理をするステージスレッド，エッジ抽出処理をするステージスレッドおよびハフ変換処理をするステージスレッドを作成する．各ステージスレッドは前のステージからの入力を待つ，入力されたフレームに対しそれぞれのステージの処理を施し，処理したフレームを次のステージへ出力するようなプログラムをステージ数分記述する必要がある．しかしステージスレッドの作成やフレームの入力待ちおよびフレームを次のステージへ出力する処理は，本来プログラマが行いたかった動画像処理の本質ではない．さらにこれは，プログラムの可読性の低下や，バグの温床となりプログラマの負担となる．

一方，提案手法では，プログラマは 1 フレームに対する処理を記述し，ステージを作成するメソッドに記述した処理を渡すだけでよく，その他の処理はライブラリ側で自動的に行われる．たとえば，**図 3.1** の場合では，プログラマは 1 フレームに対する処 a 理を記述した関数 func1 および func2 を定義する．つぎにその各関数をステージの作成を行うメソッド push に渡すのみでよい．ライブラリ側では，func1 および func2 の処理を行う各ステージが作成され，各ステージはそれぞれを動作させるスレッドに割り当てられる．ステージ間でデータを受け渡しするためのバッファであるパイプラインレジスタの作成や，各ステージ間での同期などが自動的に行われる．これらの処理をライブラリ側で自動的に行い，プログラマから隠蔽することで，プログラマはパイプライン処理を容易に記述することが可能となる．

3.2 ステージ統合・並列化

3.2.1 ステージ統合・並列化モデル

一般的に効率の良い並列化を行うには各パイプラインステージの処理量が均等である必要がある．しかし，一般に顔検出などの動画像処理をパイプライン処理化した場合，背景差分を行うステージとエッジ抽出を行うステージに対し，ハフ変換を行うステージの処理量が多い．このように，各パイプラインステージの処理量は不均衡である．このとき，ハフ変換を行うステージがボトルネックとなりパイプライン全体のスループットが低下する．

そこで各ステージのスループットを平均化するために，ステージ統合・並列化を提案する．処理量の少ないステージを同一スレッドに割り当て，処理量の多いステージを複数のスレッドに割り当てることで，各パイプラインステージのスループットを平均化する．たとえば，顔検出プログラムの場合では，背景差分，エッジ抽出およびハフ変換を行うステージが作成される．初期状態では，1 つのスレッドに対し 1 つのステージが割り当てられている．各スレッドの処理量を平均化するためにステージ統合・並列化を行う．処理量の少ない背景差分およびエッジ抽出ステージを同一のスレッドに割り当て，処理量の多いハフ変換ステージを 2 つのスレッドに割り当てる．こうすることで，各ステージの各ステージのスループットを平均化し，パイプライン全体のスループット低下を抑制する．

今回は以下の 2 つをステージ統合・並列化の条件とした．

- 隣接ステージのみが統合可能
- 統合を行ったステージは並列化を行わない

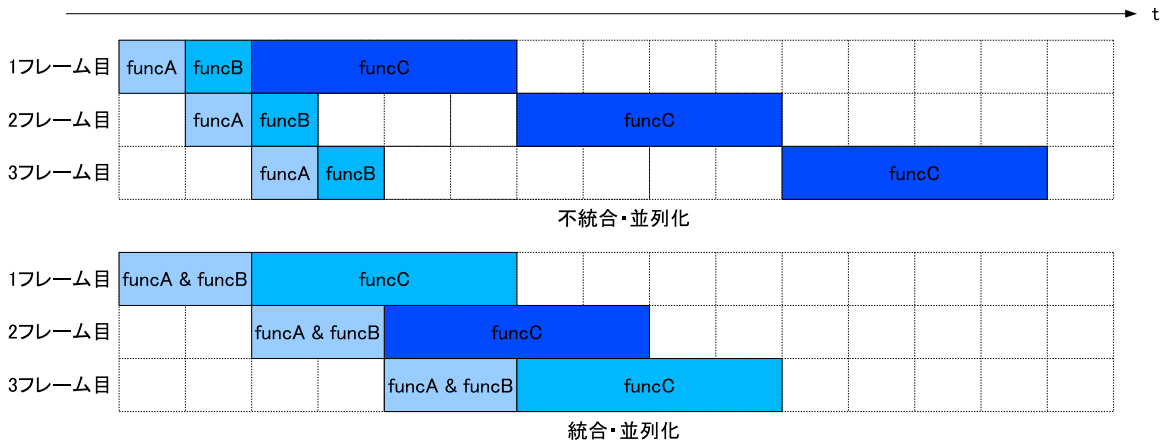


図 3.2: ステージ統合・並列化による解像度維持

上記の 2 つの条件がない場合，統合および並列化パターンが増加し，ステージ統合・並列化のオーバーヘッドが増加すると考えられる．そのためこれら 2 つの条件を設けることとした．

3.2.2 ステージ統合・並列化による解像度維持

RaVioli では 1 フレームあたりの処理量の変動や，他の平行実行されているプロセスによって，リアルタイム動画像処理に必要な CPU リソースの確保が困難になった場合，解像度を低下させることでリアルタイム性の保証を行っている．

ここでは，ビデオカメラなどから画像を取り込むキャプチャ間隔に対し，各ステージがフレームを処理する間隔が大きいとき，そのステージの処理が間に合っていないと判断する．そのとき，リアルタイム性の保証を行うために解像度の低下が発生し，各ステージの処理量が減少する．処理が間に合っていなかったステージのフレーム処理間隔がキャプチャ間隔より短くなるまで解像度の低下が発生する．

たとえば，**図 3.2** の不統合・並列化時のように，funcA，funcB および funcC の処理を行うステージがそれぞれスレッドに割り当てられている．funcA，funcB，funcC の各ステージの処理量の比は 1:1:4 となっている．このとき，各ステージの処理は前のフレームを処理するまで開始できないため，funcC のステージの処理間隔が大きくなり，解像度の低下が発生する．このように，ステージ間で処理量が均等ではないような図 3.2 の場合では，funcC のステージの為だけに解像度の低下が発生する．

図 3.2 のような状況でステージ統合・並列化を行った場合，処理量の少ない funcA および funcB のステージを統合し同一スレッドに割り当てる．次に処理量の多い funcC のステージを並列化し，複数のスレッドに割り当てる．ここでは 2 つのスレッドに funcC

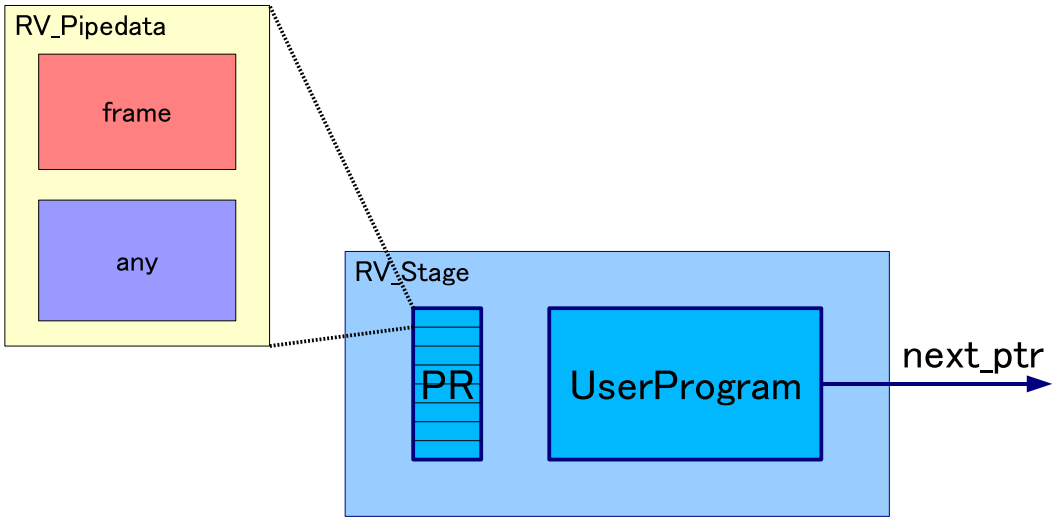


図 4.1: RV_Stage クラスおよび RV_Pipedata クラス

を割り当てる．この場合，funcA および funcB の割り当てられたスレッドは funcA と funcB の各ステージの処理を終了するまで，次のフレームを処理することはできないため，フレームの処理間隔は大きくなる．しかし，funcC は 2 つのスレッドに割り当てられているため 1 フレーム目の処理の終了を待つことなく 2 フレーム目を処理することが可能となる．そのため funcC のフレーム処理間隔は小さくなり，解像度の低下を抑えることが可能となる．

4　実装

これらの提案を実現するために，RV_Stage クラス，RV_Pipedata クラス，RV_Thread クラスおよび RV_Pipeline クラスを RaVioli に追加実装をした．以降，これらのクラスとステージの統合・並列化を行うマネージャの動作を示す．

4.1　ライブラリ仕様

4.1.1　RV_Stage クラス

RV_Stage クラスの概念図を**図 4.1** に示す．RV_Stage クラスはパイプラインステージの情報を持つクラスで，入力フレームを一時的に格納しておくパイプラインレジスタ PR，フレームに対する処理 UserProgram，次のステージ next_ptr を持っている．

パイプラインレジスタはキュー構造となっており，前のステージからの入力フレームがキューイングされている．UserProgram はプログラマがフレームに対する処理を記述した関数へのポインタである．next_ptr は次のステージである RV_Stage オブジェ

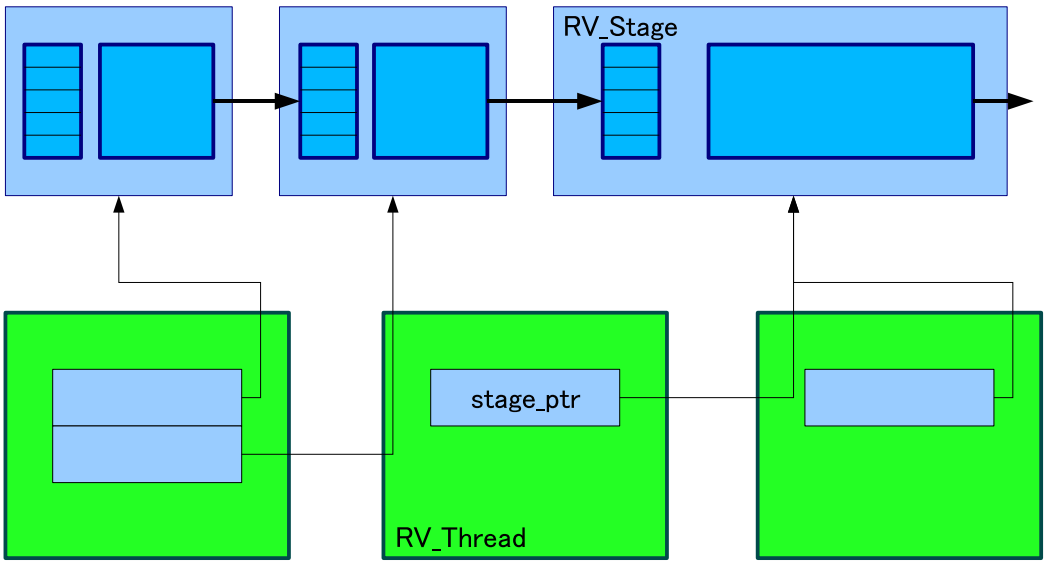


図 4.2: RV_Thread クラス

クトへのポインタである．各ステージの処理の流れはパイプラインレジスタから 1 フレーム dequeue し，そのフレームに対し UserProgram に指定された処理を施し，出力先ステージ next_ptr のパイプラインレジスタへ enqueue する．

RV_Stage クラスを設計するときに，パイプラインレジスタを UserProgram の入力側に置く場合と，出力側に置く場合とが考えられる．仮に出力側にパイプラインレジスタを置いた場合では，次のステージが前のステージのパイプラインレジスタにアクセスすることになる．ここでパイプラインレジスタが空であった場合，ステージ間を跨いだ空アクセスが発生し，無駄な通信となる．そこで UserProgram の入力側にパイプラインレジスタを置くことで，ステージ間を跨いだ無駄なアクセスの発生を防いでいる．

各 RV_Stage 間を流れるデータ RV_Pipedata は図 4.1 に示す通りである．RV_Pipedata はビデオカメラなどからキャプチャした画像 frame とその他のデータである any からなる．動画像処理では，ハフ変換などのように画像以外のデータが必要な場合がある．そこで RV_Pipedata では boost ライブラリ [3] の boost::any クラスを用い any を実装している．boost::any は C++の int 型などの通常の型の他にユーザ定義の型を格納することができるクラスである．これを用いることで，プログラマは画像データ以外のデータを扱うことができる．

4.1.2 RV_Thread クラス

RV_Thread クラスの概念図を**図 4.2** に示す．RV_Thread クラスはスレッドの情報を持つクラスで，スレッドで動作しているステージへのポインタを配列の形で持っている．RV_Thread はステージ配列に存在するステージを実行する．

また，1 つの RV_Thread に対し複数の RV_Stage を割り当てることで統合を実現し，複数の RV_Thread に対し 1 つの RV_Stage を割り当てることで並列化を実現している．このようにスレッドで動作するステージを配列として管理している．こうすることで，スレッドの作成，破棄などの無駄な処理を行わずにステージ配列を変更するだけでステージの統合・並列化を行うことができる．

ステージが統合されたとき，RV_Thread のステージ配列には複数の RV_Stage へのポインタが存在する．このとき，RV_Thread はステージ配列の 1 つ目の RV_Stage を実行したあと，2 つ目の RV_Stage の動作を行う．

ステージが並列化されたとき，1 つの RV_Stage へのポインタが複数の RV_Thread のステージ配列に存在する．つまり，複数のスレッドで同一のステージが動作していることになる．このとき，ある RV_Thread が 1 フレーム目を処理している間に，別の RV_Thread が 2 フレーム目を処理する．このように，フレーム単位で処理を並列化することでステージの並列化を実現している．

4.1.3 RV_Pipeline クラス

RV_Pipeline クラスの概念図を**図 4.3** に示す．RV_Pipeline クラスはパイプライン全体の情報を持つクラスで，パイプライン中のすべてのステージの配列，すべてのスレッドの配列およびスレッドにステージの割り当てを行うマネージャを持っている．次にメンバメソッドについて説明する．RV_Pipeline クラスのメンバメソッドには，解像度の優先度を設定する setParam メソッド，ステージの作成を行う push メソッドおよびスレッドの実行を行う run メソッドがある．各メソッドについて説明する．

void setParam(int $prio\_fra$**, int** $prio\_res$**)**

setParam メソッドは時間解像度の優先度である $prio\_fra$ と空間解像度の優先度である $prio\_res$ を引数にとるメンバメソッドである．ここで指定された優先度に応じて Manager は時間解像度または空間解像度どちらを低下させるか決定する．指定された値に応じ，$prio\_fra : prio\_res$ の割合で値の大きい解像度が優先される． **図 4.4** のように $prio\_fra = 7$，$prio\_res = 3$ と設定した場合，時間解像度が 7:3 の割合で優先される．また $prio\_fra = 1$，$prio\_res = 0$ と設定した場合，時間解像度は低下せず，空間解像度のみ低下が発生する．setParam によって優先度が

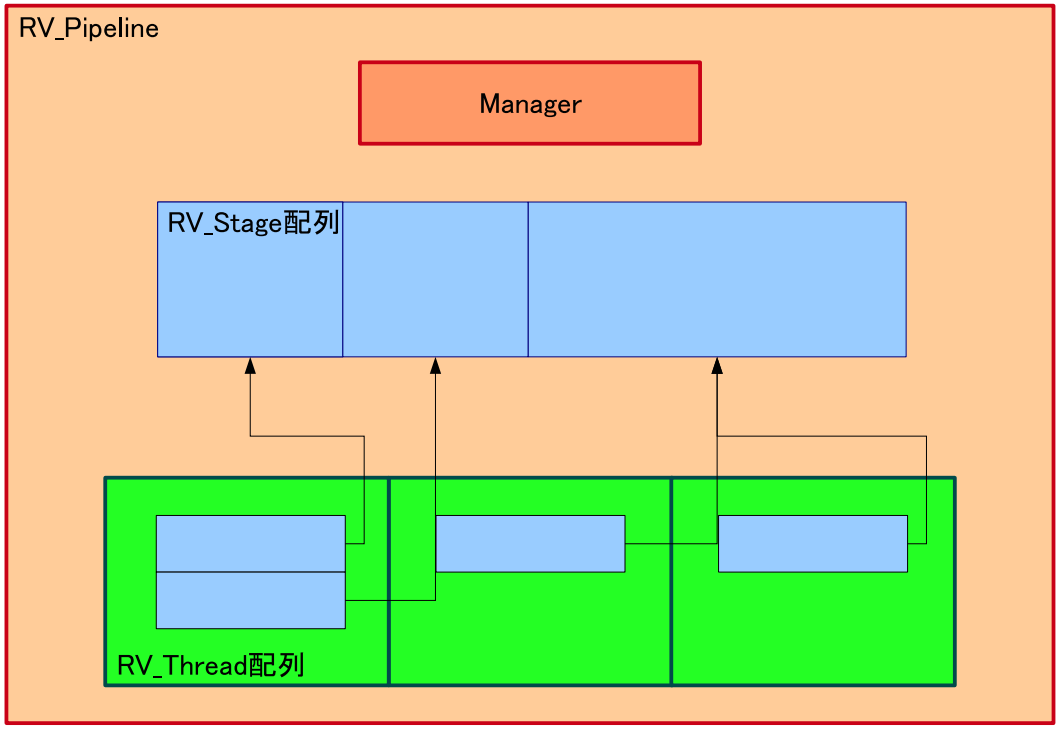


図 4.3: RV_Pipeline クラス

設定されなかった場合は，$prio\_fra = 1$，$prio\_res = 1$ として設定される．

void push(RV_Pipedata* (*UserProgram)(RV_Pipedata*))

push メソッドは関数ポインタ UserProgram を引数にとるメンバメソッドで，入力フレームに対し UserProgram 処理を適用するステージの作成を行う．次に，スレッドを作成し，そのスレッドに対し作成したステージを割り当てる．UserProgram は，プログラマがステージでフレームに対して行う処理を記述した関数へのポインタである．グレースケール化した後にラプラシアンフィルタ処理を行うプログラムをパイプライン化したい場合には，図 4.4 のようにフレームに対しグレースケール化を行う関数へのポインタ GrayScale を引数として push を呼び出す．同様にフレームに対しラプラシアンフィルタ処理を行う関数へのポインタ Laplacian を引数として push メソッドを呼び出すと入力フレームに対し各処理を行うステージの作成が行われる．

void run()

run メソッドはパイプライン処理を開始するメソッドである．このメソッドが呼ばれると，各スレッドは割り当てられているステージの動作を開始する．同時に，マネージャも起動する．

```
RV_Pipedata* GrayScale(RV_Pipedata* data){
  //frame をグレースケール化する処理
  return data;
}
RV_Pipedata* Laplacian(RV_Pipedata* data){
  //frame にラプラシアンフィルタを適用する処理
  return data;
}

int main(){
  RV_Pipeline pipe;
  pipe.setParam(7,3);      //優先度を設定
  pipe.push(GrayScale);    //GrayScale ステージ作成
  pipe.push(Laplacian);    //Laplacian ステージ作成
  pipe.run();              //パイプライン実行
  return 0;
}
```

図 4.4: setParam，push および run メソッド記述例

図 4.4 に setParam メソッド push メソッドおよび run メソッドを用いたプログラム記述例を示す．ユーザは各ステージでのフレームに対する処理を記述した関数 GrayScale および Laplacian を記述する．GrayScale および Laplacian は RV_Pipedata へのポインタ data を引数として RV_Pipedata へのポインタを返す関数である．プログラマは引数として受け取った data に対し処理を施し，処理を施した data を返す関数を記述する．次にユーザはそれらの関数へのポインタを引数としてステージの作成を行う push メソッドを呼ぶ．最後に，パイプライン実行を行うための run メソッドを呼ぶ．優先度の設定を行いたい場合は，時間解像度の優先度および空間解像度の優先度を引数とし setParam メソッドを呼ぶ．以上がパイプライン記法を用いたプログラムの流れである．

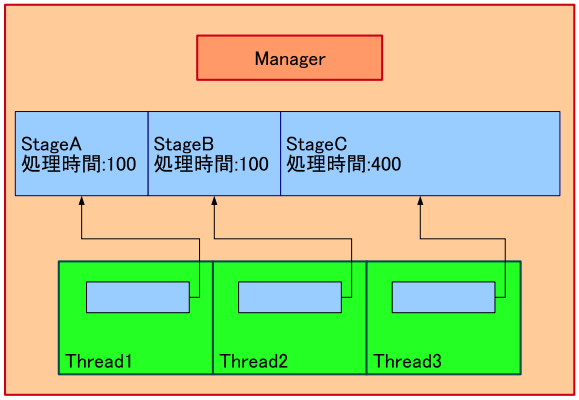


図 4.5: ステージ統合・並列化前

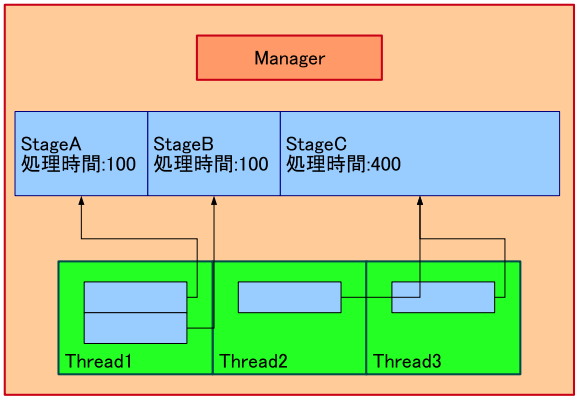


図 4.6: ステージ統合・並列化後

4.2 マネージャの動作

ステージの統合・並列化を行うためにスレッドに対しステージを割り当てるマネージャの動作を示す.

マネージャの動作は大きく分けて処理量収集，平均値計算，統合，並列化，割り当ての 5 フェーズに分かれる．以降，各フェーズでのマネージャの動作を説明する.

情報収集フェーズ

フレームの入出力を行うステージを除くすべてのステージから 1 フレーム処理するのにかかる時間を収集する.

平均値計算フェーズ

収集した各ステージの処理時間の平均値をもとめる.

統合フェーズ

どのステージとどのステージを統合するのかを決定する．マネージャは処理時間の最も小さいステージの処理時間とそのとなりのステージを統合した場合の処理時間のうち，平均値に近い方を選択し統合する.

並列化フェーズ

統合フェーズで統合を行ったステージ数分，空きスレッドが発生することから，その空きスレッドの数だけ並列化を行う．並列化は統合フェーズで統合されていないステージのうち処理時間の最も大きいステージから行う.

割り当てフェーズ

統合フェーズと並列化フェーズで決められた統合・並列化方法にしたがってスレッドにステージの割り当てを行う.

マネージャの動作を例を用いて説明する．パイプラインステージの統合・並列化を

行う前の状態が**図 4.5** のように，StageA の処理時間が 100 ミリ秒，StageB の処理時間が 100 ミリ秒，StageC の処理時間が 400 ミリ秒となっており，Thread1 に StageA が割り当てられ，Thread2 に StageB が割り当てられ，Thread3 に StageC が割り当てられているような状態になっている．

初めにマネージャは情報収集フェーズに移り，各ステージが 1 フレーム処理するのにかかる時間を収集する．次にマネージャは平均値計算フェーズに移り，処理時間の平均値を求める．ここでは $(100 + 100 + 400) \div 3 = 200$ となり，平均処理時間は 200 となる．

次に統合フェーズに移り，最も処理時間の小さいステージ StageA の処理時間 100 と，StageA とその隣のステージ StageB を統合 (StageA+StageB) した処理時間 $100+100 = 200$ のうちで，平均処理時間 200 に近い StageA と StageB を統合した場合を選択し，StageA および StageB を統合し処理時間 $100+100 = 200$ の StageAB とする．同様に最も処理時間の小さい StageAB の処理時間 200 と，StageAB とその隣のステージ StageC を統合 (StageAB+StageC) した場合の処理時間 $200 + 400 = 600$ のうち，平均処理時間 200 に近い StageAB と StageC を統合しない場合を選択する．同様に次に処理時間の小さいステージ StageC の統合動作を行うが，隣接ステージに統合対象ステージが存在しないため StageC の統合は起こらない．すべてのステージに対し統合動作を行った後，並列化フェーズに移る．

統合フェーズで StageA と StageB の統合を行ったので，全体のステージ数は StageAB と StageC の 2 ステージになり，初期状態図 4.5 から 1 ステージ減ったことになる．そこで並列化フェーズでは統合フェーズで減ったステージ数分，ここでは 1 ステージ分だけ並列化を行う．統合フェーズで統合を行っていないステージの中で処理時間の最も大きいステージの並列度を 1 増やす．ここで並列度とは 1 つのステージがいくつのスレッドに割り当てられているかを示す値である．StageAB は統合フェーズで統合を行っているため並列化を行わず，StageC に対し並列化処理を行う．初期状態図 4.5 で StageC は Thread3 にのみ割り当てられていたため並列度は 1 である．この値を増やし 2 とする．統合フェーズで減ったステージ数は 1 ステージであったので並列化動作を 1 度行い並列化フェーズを終了する．

次に割り当てフェーズに移る．割り当てフェーズでは，統合フェーズと並列化フェーズの結果をもとに，スレッドに対しステージの割り当てを行う．統合フェーズの結果をもとに StageA と StageB を Thread1 に割り当てる．次に並列化フェーズの結果をもとに StageC を Thread2 および Thread3 に割り当てる．このようにステージをスレッドに割

表 5.1: 評価環境

	評価環境 1	評価環境 2
OS	Solaris10	Solaris10
CPU	Core 2 Extream	UltraSPARC T1
クロック周波数	3.00GHz	1.00GHz
コア数	4	8
Memory	8GB	16GB

り当てた結果が**図 4.6** となる．このとき，Thread1 は 2 つのステージ StageA と StageB の処理を行う．そのため，各ステージが 1 フレーム処理し，次のフレームの処理を開始するまでの時間は 100+100 = 200 となり，スループットは $1frame \div 200ms = 5fps$(frame par second) となる．また，StageC は Thread2 および Thread3 に割り当てられ並列実行されているため，スループットが 2 倍の $1frame \div 400ms \times 2 = 5fps$ となり，各ステージのスループットが平均化された．

5 評価

表 5.1 に示す評価環境 1 および評価環境 2 でパイプラインステージの統合・並列化による解像度維持の評価を行った．サンプルプログラムを評価環境 1 および評価環境 2 で実行し，空間解像度ストライドおよび時間解像度ストライドの推移を示しステージ統合・並列化による解像度維持の評価を行う．評価プログラムには入力フレームを処理するステージが 3 ステージを 3 スレッドで動作させた場合と 7 ステージを 7 スレッドで動作させた場合の 2 つのプログラムを用い，空間解像度優先度および時間解像度優先度を (1, 0) と設定したときの空間解像度ストライドの推移と，各優先度を (0, 1) と設定したときの時間解像度ストライドの推移を示す．また，マネージャは 1000 フレーム処理したときにステージ統合・並列化を行い，各ステージのスループットの平均化を行う．ストライドの推移を示すグラフの横軸は処理したフレーム数，縦軸は各ストライドを表す．

表 5.1 評価環境 1 で 7 ステージのプログラムを実行した場合と，3 ステージのプログラムを実行した場合の空間解像度ストライドおよび時間解像度ストライドを**図 5.1**, **図 5.2**, **図 5.3** および**図 5.4** に示す．図 5.2，図 5.3 および図 5.4 では 1000 フレーム以降でストライドの低減が発生している．これはステージ統合・並列化により，各ステー

評価環境 1 でのストライドの推移

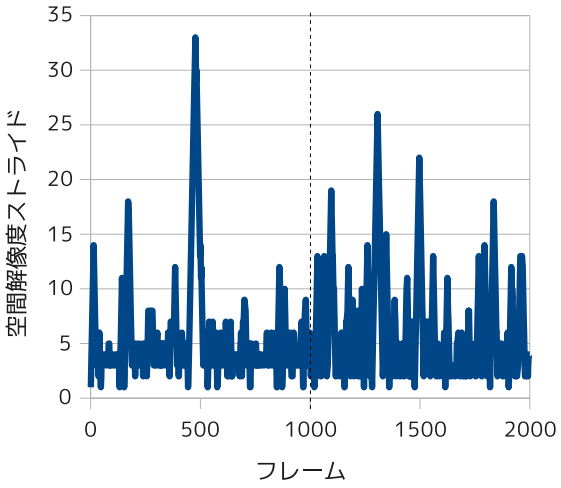


図 5.1: 7 ステージでの空間解像度の推移

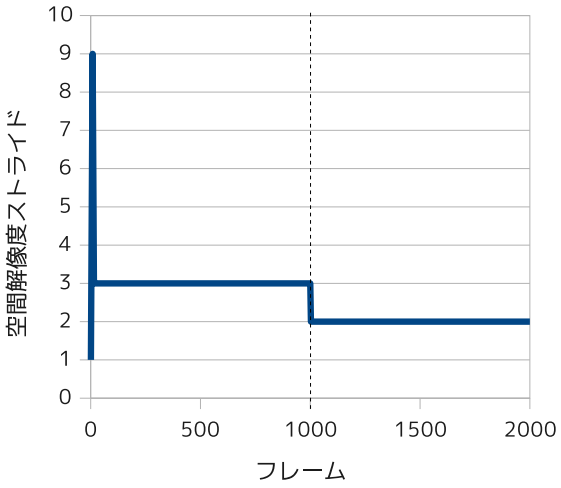


図 5.2: 3 ステージでの空間解像度の推移

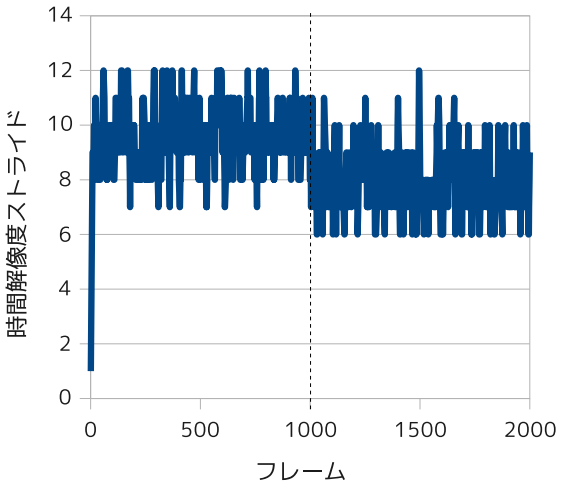


図 5.3: 7 ステージでの時間解像度の推移

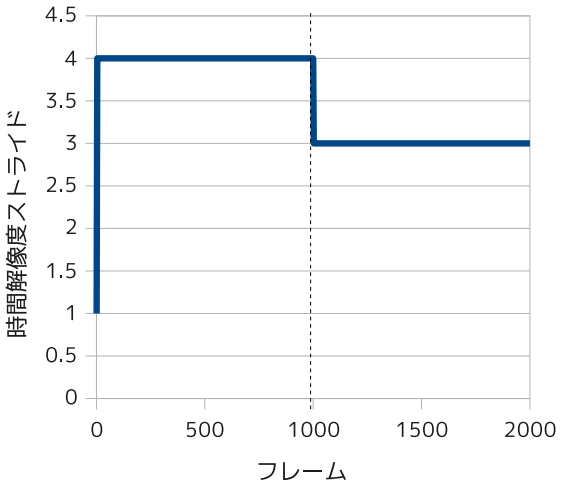


図 5.4: 3 ステージでの時間解像度の推移

ジのスループットが平均化され処理間隔の大きかったステージが並列に実行されたためである．

図 5.1 および図 5.3 で各ストライドが発振している．これは評価環境 1 が 4 コアであるため，7 ステージ全てが並列に動作することができない．そのため各ステージが 1 フレーム処理する間隔にぶれが発生し，間隔が大きい時ではストライドが上昇し，小さい時ではストライドが低減する．

次に，評価環境 2 での実行結果を**図 5.5**，**図 5.6**，**図 5.7** および**図 5.8** に示す． 評

評価環境 2 でのストライドの推移

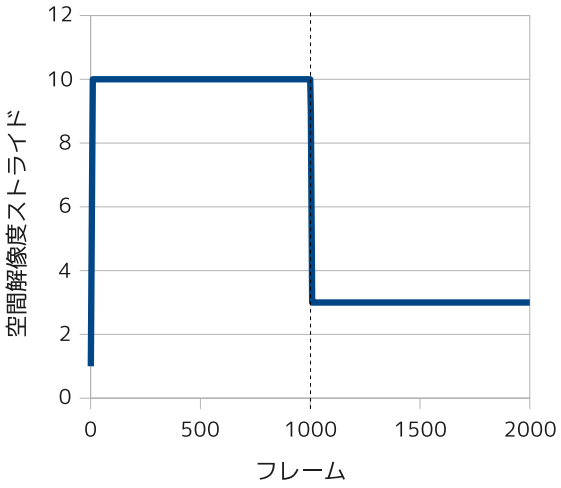


図 5.5: 7 ステージでの空間解像度の推移

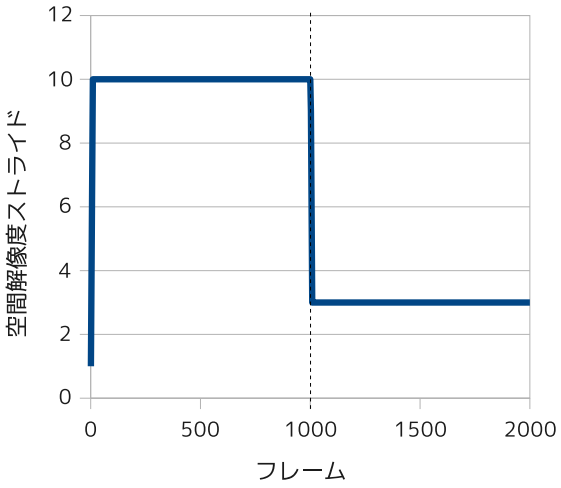


図 5.6: 3 ステージでの空間解像度の推移

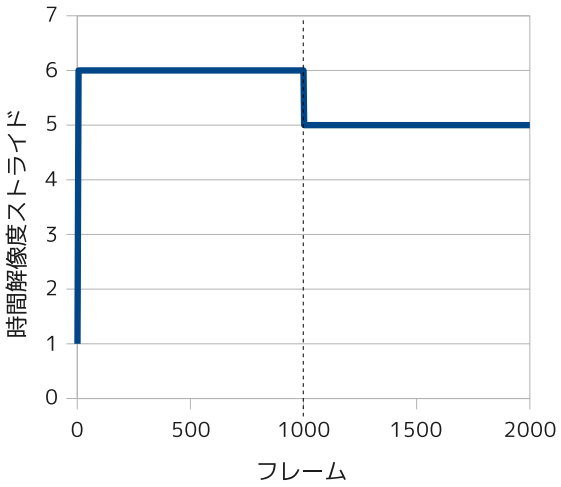


図 5.7: 7 ステージでの時間解像度の推移

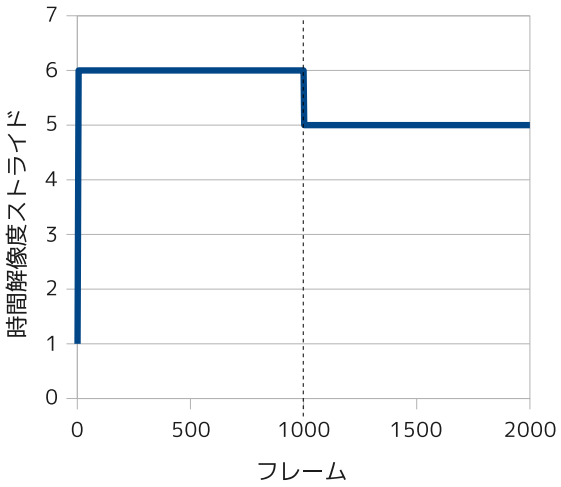


図 5.8: 3 ステージでの時間解像度の推移

価環境 2 では，7 ステージのプログラムと 3 ステージのプログラムとで同様の結果となった．これは評価環境 2 のコア数 8 に対し並列実行されているスレッドの数が少ないため，全てのスレッドが並列実行され，全ステージが並列動作しているためである．また，ステージ統合・並列化を行うことで，空間解像度ストライドは 10 から 3 へ 7 ストライド低減され，時間解像度ストライドは 6 から 5 へ 1 ストライド低減された．ステージ統合・並列化を行うことで行う前に比べ空間解像度および時間解像度の低下を抑えたままリアルタイム処理が可能であることを確認した．

6　おわりに

本論文ではまず，動的に使用可能なリソースが変動するような環境において動的に解像度を変動させることで処理量を減らしリアルタイム性の保証を行う解像度非依存型動画像処理ライブラリ RaVioli について述べた．また動画像処理におけるパイプライン処理の有用性を述べた．

そして RaVioli を用いてパイプライン処理を実現する際の問題点を示し，これに対しパイプライン記法を提案し実現した．パイプライン記法を実現したことによって，RaVioli を用いたパイプライン処理が可能となった．また，一般に各パイプラインステージの処理量の不均衡という問題を示し，これに対しパイプラインステージの統合・並列化を提案し実現した．パイプラインステージの統合・並列化を実現したことによって各ステージのスループットの均等化を可能とした．ステージ統合・並列化によるステージの処理量の均等化によって解像度の低下を抑えられることを確認した．

今後の課題として，マルチグレイン並列化が挙げられる．動画像処理は，たとえば SIMD 命令を用いた並列処理や，画像を複数のブロックに分割し処理を行うブロック並列処理や，画像処理に特化した GPU を用いた並列処理や，複数の計算機を用いた並列処理など，さまざまな粒度で並列化が可能である．そこで処理内容などによって，これらの並列化方法を自動的に適用するような環境が考えられる．こうすることでプログラマは既存のプログラムに変更を加えることなく様々な並列処理を行うことが可能になる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室およびの齋藤研究室の方々に深く感謝致します．

参考文献

[1] 有田大作, 濱田義雄, 米元聡, 谷口倫一郎: PC クラスタを利用した実時間並列画像処理環境 RPV, 電子情報通信学会論文誌 D-II, Vol. J84-D-II, No. 6, pp. 965–975 (2001).

[2] 岡田慎太郎, 桜井寛子, 津邑公暁, 松尾啓志: 解像度非依存型動画像処理ライブラリ

RaVioli の提案と実装, 情報処理学会論文誌：コンピュータビジョンとイメージメディア（CVIM）, Vol. 1, No. 4 (2009).

[3] Sutter, H., Alexandrescu, A. and C++ Coding Standards: The Boost C++ Library, http://www.boost.org.