修士論文

動画像処理ライブラリ RaVioli における処理精度の領域別変動手法

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学大学院 工学研究科
修士課程 創成シミュレーション工学専攻
平成 22 年度入学　22413537 番

近藤 勝彦

平成 24 年 2 月 3 日

動画像処理ライブラリ RaVioli における
処理精度の領域別変動手法

近藤 勝彦

内容梗概

侵入者検知システムや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，汎用計算機の性能向上や価格低下により，高性能な計算機環境を容易に入手可能になってきている．そのため，今後，汎用計算機上でリアルタイム動画像処理システムが盛んに開発されると予想される．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 リソース量を常に確保することは困難である．

この問題を解決するため，動画像処理ライブラリ RaVioli が提案されている．RaVioli では利用可能な CPU リソース量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，自動的に解像度を低減させることで処理量を調整し，リアルタイム性を保証している．しかし，解像度の低減により，動画像処理の処理精度が低下する問題がある．これに対して，RaVioli プログラムを並列化することで，処理精度低下を抑制する手法が提案されている．しかしこの手法を用いた場合，並列化により処理時間の短縮が見込まれるが，その効果は実行環境に依存し，限定的である．

そこで，入力の重要度によって処理精度を変動させる新しい処理量調整手法を提案する．この提案手法は重要な入力に対する処理精度の低下を抑制するために，リアルタイム動画像処理の入力に注目する．リアルタイム動画像処理には全ての入力を詳細に処理しなくてもよい場合があり，そのような入力を粗く処理することで処理量を削減することができる．そこで，動画像ストリームを部分ストリームに分割し，各ストリーム毎に解像度を保持，変動できるように RaVioli を拡張する．

なお，この提案手法は処理量そのものを削減するため，処理時間を短縮する並列化の手法とは別のアプローチである．そのため，これらの手法を組み合わせることで，処理精度低下の更なる抑制を実現する．

実際にサンプルプログラムを用いて提案手法を評価した．既存の RaVioli と提案手法を実装した RaVioli でそれぞれサンプルプログラムを実行し，解像度の変化と出力画像を比較した結果，提案手法を用いた際に解像度の低減が抑えられることを確認した．

動画像処理ライブラリ RaVioli における
処理精度の領域別変動手法

目次

1　はじめに

空港や工場などで侵入者や不審物を検出するシステムや，炎や煙などを認識して火災を検知するシステム，自動車走行時の衝突回避システムなど動画像処理のリアルタイム性が重要となるシステムが盛んに開発されている．また，このようなシステムへの需要も高まり，普及し続けている．一方で，計算機の高性能化により，顔認識アルゴリズムなどの処理量の多い動画像処理アプリケーションを汎用 PC 上で動作させることが可能になってきた．また，計算機の価格が低下しているため，高性能な計算機環境を容易に入手可能になってきている．これらのことから，今後，汎用 PC 上でリアルタイム動画像処理システムが盛んに開発されると予想される．

しかし，Linux に代表される汎用 OS 上で，動画像処理アプリケーションのリアルタイム性を保証することはいまだに困難である．その主な理由として，1 フレームあたりの処理量が入力によって変動することや，利用可能な CPU リソース量が他の並行実行プロセスによって変動することが挙げられる．これらは 1 フレームあたりにかかる処理時間に影響し，この処理時間の増減がリアルタイム性の保証を難しくしている．これに対して，Linux をリアルタイム OS に拡張するプロジェクトが存在する．しかし，元来 Linux はリアルタイム処理であってもカーネル実行中は割り込みができない非リアルタイム OS である．そのため，Linux をリアルタイム OS に拡張しても，リアルタイム性を常に保証できるわけではない．

この問題を解決するために，解像度の変動による処理量調整機能を備える動画像処理ライブラリ **RaVioli（Resolution-Adaptable Video and Image Operating Library）**[1, 2] が提案されている．RaVioli は利用可能な CPU リソース量に応じて，空間解像度（1 フレーム上の画素数）または時間解像度（フレームレート）を変動させて処理量を調整する．この方法では，処理精度を犠牲にして処理の大幅な遅れを回避し，リアルタイム性を擬似的に保証する．

一般に，このように動的に解像度を変動させる場合，処理フレームや処理画素にアクセスする際の，イテレーション幅やイテレーション回数の変動に対応したプログラムを記述する必要がある．しかしプログラマが，これらの処理量の変動を意識して動画像処理アプリケーションを開発することは困難である．そこで，RaVioli はプログラマから画像の幅や高さ，動画像のフレームレートを隠蔽する．これにより，ライブラリ内で解像度を制御可能になるだけでなく，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．

しかし，解像度を変動させて処理量を調整することには限界がある．解像度が大幅に低減してしまうと，動画像処理の処理精度が下がってしまい，プログラマが期待する処理結果を得られなくなる可能性がある．RaVioli は処理のリアルタイム性を保証するために解像度を低減させているため，処理精度の低下は避けられないが，できる限りそれを抑制することが求められる．この RaVioli の問題に対して，処理の並列化により処理時間を短縮することで処理精度を維持する手法が提案されている．この手法は画像処理プログラムのデータ並列性に注目し，空間分割した画像の各部分を複数のスレッドにより同時に実行することで処理時間を短縮する．また，並列プログラムの作成にはデータアクセスの競合解決やスレッドの管理などが必要になる．これらはプログラマにとって煩雑であるため，RaVioli の逐次プログラムを並列プログラムへと変換するプリプロセッサを提供している．これにより，プログラマは並列プログラミングの知識がなくとも，並列化の恩恵を受けることが可能である．

しかし，並列化による効果は限定的である．まず，複数のコアを備えた実行環境でなければ処理時間を短縮することはできない．また，たとえ複数のコアを備えていても，汎用 OS 上では複数のプロセスが並行に実行されているため，常に複数のコアを有効に活用できるとは限らない．

そこで，入力の重要度によって処理精度を変動させる新しい処理量調整手法を提案する．この提案手法では重要な入力に対する処理精度を低下させずに処理量を削減するために，リアルタイム動画像処理の入力に注目する．リアルタイム動画像処理には全ての入力を詳細に処理しなくてもよい場合があり，そのような入力を粗く処理することで処理量を削減することができる．しかし，現在の RaVioli では，フレーム全体を同じ精度でしか処理できない．これは RaVioli が画像全体に対してひとつの空間解像度パラメータを，また動画像全体に対してひとつの時間解像度パラメータを保持し，変動させているからである．そこで，動画像ストリームを分割し，各ストリーム毎に両解像度を保持，変動できるように RaVioli を拡張する．これにより，詳細に処理する必要がない領域に対する処理量を削減し，詳細に処理すべき領域の処理精度の低下を抑制することが可能になる．また，この提案手法は処理時間を短縮する並列化とは別のアプローチである．そのため，提案手法を並列化と組み合わせることで，更に処理精度の低下を抑制できる．

本論文では以下，2 章でリアルタイム動画像処理や動画像処理ライブラリの関連研究について説明する． 3 章では，本提案手法の基盤である動画像処理ライブラリ RaVioli

の基本機能や自動空間分割並列化機能の詳細，問題点について述べる． 4 章では，領域別に処理精度を変動させる新しい処理量調整手法を提案し， 5 章でその実装方法について説明する． 6 章では，提案手法を実現することによって発生する問題と，それらを解決するための RaVioli の機能拡張について述べる． 7 章は提案手法の評価結果を示し， 8 章で本論文全体をまとめる．

2 関連研究

本章では RaVioli が対象とするリアルタイム動画像処理について述べ，動画像処理ライブラリや言語の関連研究について説明する．

2.1 リアルタイム動画像処理

リアルタイム動画像処理アプリケーションが普及し，盛んに研究されている．例えば，Garcia-Martin ら [3] は動画像監視システムで用いられる動物体を検出するアルゴリズムを提案している．また，Kim ら [4] は火災の早期検出手法を提案し，Lin ら [5] は目の位置のリアルタイム検出アルゴリズムを提案している．

これらのリアルタイム動画像処理を実現するためには，画像処理のスケジューリングと処理負荷の調整が重要となる．画像処理のスケジューリング手法として，Lee ら [6] は複数のステージ，ループ，データ依存な演算を含む画像処理を並列実行する際の各処理の静的なスケジューリング手法を提案している．この手法は並列処理の実行時間を大幅に削減できるが，ある一定の時間内に処理が完了することを保証するものではない．また，Kywe ら [7] は anytime アルゴリズムを用いた適応的なスケジューリング手法を提案している．このスケジューリング手法では，限られた時間内に最良な画像処理結果を得られるが，従来の画像処理アルゴリズムを anytime アルゴリズムに基づいたものに変更する必要がある．これはプログラマにとって大きな負担となる．

一方で，処理負荷の調整手法もいくつか提案されている．トレードオフの関係にある処理精度と処理時間を動的に調整する方法として，複数アルゴリズムの切り換え手法がある．例えば，指定した計算時間を超過すると，その時点までに得られている不完全な中間結果を計算結果として採用するといった，計算時間の長さに応じて精度が向上するモデル（Imprecise Computation Model）[8] が提案されている．またこのモデルに基づき，処理精度および処理時間に関して経験的に得た知識を利用することで，プログラマがあらかじめ記述した複数のアルゴリズムから，状況に応じて適したアルゴリズムを動的に選択する信頼度駆動アーキテクチャも提案されている [9, 10]．しか

しこの方法では，計算負荷の異なる複数のアルゴリズムで処理を実装する必要があり，依然プログラマに対する負担は大きい．

別の処理負荷の調整手法として，動画像中のフレームをスキップする手法が提案されている [11]．これは動画像全体の品質に対して，全てのフレームが等しく重要ではないという事実に基づく手法である．入力動画像の全てのフレームを処理するのに必要な計算資源を確保できないときには，動画像全体の品質を考慮した基準に従って，重要度の低いフレームを飛ばして動画像を処理する．また，別の負荷調整手法が目の位置の検出処理 [5] で使用されている．この検出アルゴリズムでは，処理対象のフレームの大きさを調整することで，検出速度を向上させている．具体的には，目の検出処理後に，検出された目の領域の大きさによって次の処理フレームの大きさを調整する．例えば，検出された目の領域がフレームに対して十分な大きさである場合，次の処理フレームを縮小する．これにより，検出精度を落とすことなく，検出にかかる処理時間を短縮することが可能である．これらの 2 つの研究は RaVioli の処理量調整手法に似ているが，RaVioli を用いた場合，これらの手法を自動的に任意の動画像処理アプリケーションに適用できる．

2.2 動画像処理ライブラリや言語

一方で，多くの動画像処理や画像処理のライブラリがこれまでに提案されている．例えば，VIGRA[12]，OpenCV[13, 14]，OpenIP[15] 及び Pandore[16] はよく知られた画像処理のライブラリである．VIGRA は STL（Standard Template Library）を基本とするテンプレートを用いて一般的な画像処理パターンを抽象化したライブラリである．画像の反転や回転，エッジ処理などの基本的な処理から，ガウスやガボールに代表されるフィルタ処理，画像の分析処理などを抽象化して提供している．OpenCV は，画像処理の一般的なアルゴリズムを C の関数や C++のメソッドとしてユーザに提供している．OpenIP は教育，研究，及び産業分野の画像処理やコンピュータビジョンに対する要求を満たす，C++のライブラリ一式を提供する．各ライブラリは画像処理のためのデータ構造や，フィルタリングなどの画像処理アルゴリズム，画像の入出力のインタフェースなども提供している．Pandore は様々な画像処理を実行可能なプログラムとして提供している．プログラマはそれらの処理を組み合わせることで画像処理アプリケーションを作成することが可能である．これらのライブラリはプログラマが画像処理を容易に記述できるようにするが，これらを用いて処理負荷を調整する機能を実装することは困難である．

また，画像処理向けのプログラミング言語も提案されている．金井らは数式エディタを用いて記述できる独自の言語を提案し，画像処理プログラミングを抽象化している [17, 18]．処理単位となる画素配列の大きさを定義し，その配列の要素に対して処理を記述することでループレスな記述ができるという点は RaVioli と似ているが，この記述言語は構成画素数を明示的に指定する必要があるため，動的な構成画素数の変動には対応していない．他にも，PPL（Picture Processing Language）[19] という画像処理プログラミング言語が提案されている．PPL の基本方針は画像処理アルゴリズムをカプセル化することである．これによって，画像処理アルゴリズムを実装する際に個々の画素を扱う必要がなくなり，デジタル画像処理についてあまり詳しくないプログラマでも画像処理プログラムを記述できるようになる．また，PPL は一般的な画像処理アルゴリズムを提供しているが，PPL を拡張することで新しい画像処理アルゴリズムを使用することもできる．プログラマは PPL が提供する API を使用し，よりカスタマイズされた関数やメソッドを PPL に追加することができる．しかし，これはプログラマにとって容易ではない．

これに対し，動画像処理ライブラリ RaVioli[1, 2] のアプローチは，これまでに説明した画像処理や動画像処理のライブラリや言語とは全く異なる．RaVioli はプログラマから解像度という概念を隠蔽することで，より直感的な画像処理プログラミングを実現する．また，プログラマから解像度を隠蔽しているため，動的に解像度を変動させて，処理負荷を自動調整することが可能である．これにより，RaVioli は処理のリアルタイム性を保証している．

3　動画像処理ライブラリ RaVioli

本提案の対象となる動画像処理ライブラリ RaVioli の基本機能を述べ，RaVioli が抱える問題点と既存の解決手法を説明する．

3.1　基本機能

RaVioli はプログラマから解像度という概念を隠蔽する．これにより直感的なプログラミングを実現すると共に，処理のリアルタイム性を保証するための動的な処理量調整を実現している．この節では，このような RaVioli の基本機能について説明する．また，RaVioli が抱える問題点についても説明する．

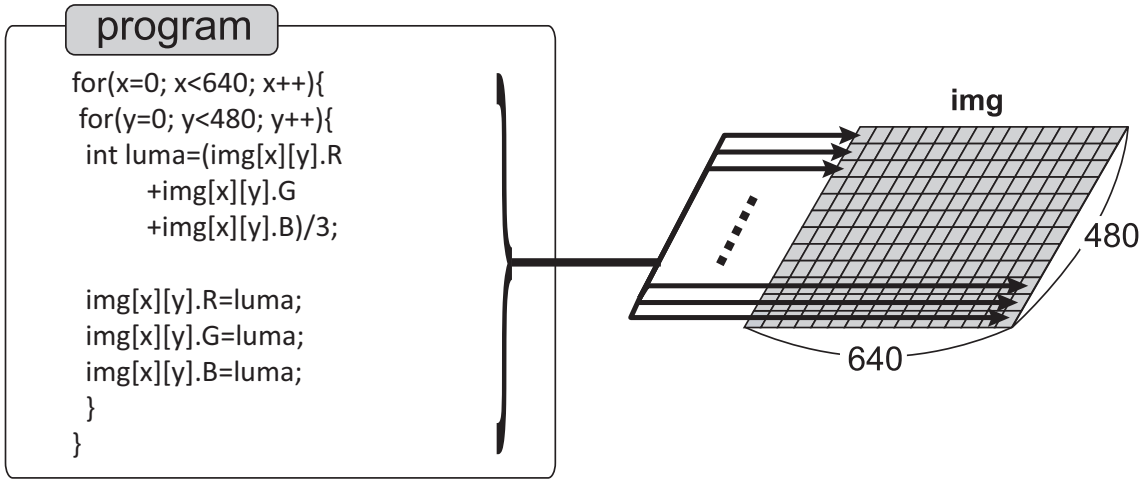

図 1: 一般的な画像処理プログラム

3.1.1 動画像処理の抽象化

動画像を構成する要素である「画素」や「フレーム」は画像や動画像を計算機上で扱うために導入された概念であり，そもそも人間の脳内における視覚情報の認識過程には存在しない．しかし量子的に情報を扱う必要のある計算機上では，画像を画素の集合として，動画像をフレームの集合として扱わなければならない．またこのように量子化されているが故に，動画像処理プログラムを記述する際は for 文などのループ文を用いて，これらの構成要素に対して繰り返し処理を施す必要があるが，この繰り返し処理もまた動画像処理の本質ではない．

これらの問題に対し RaVioli は，プログラマから解像度の概念を隠蔽するプログラミングパラダイムを提供している．ここで解像度とは**空間解像度**と**時間解像度**の 2 つを意味しており，空間解像度は 1 フレームを構成する画素数を，また時間解像度はフレームレートを意味している．RaVioli は，1 フレーム中の画素配列や画像の幅・高さ，フレームレート等をプログラマから隠蔽し， RaVioli 側でこれら全てを管理することで，プログラマは解像度を意識せずに動画像処理を記述できる．

一般に画像処理では，画像の構成要素に対する処理を，画像全体または任意の範囲に繰り返し適用するものが多い．例えばカラー画像からモノクロ画像への変換や色の反転などの処理では処理単位は画素であり，ぼかしやエッジ強調などの近傍処理では，処理単位は画素およびその近傍画素である．また，テンプレートマッチング等の処理では処理単位は小さなウィンドウである．そしてこれらの処理は，一般的に**図 1** のようにループイテレーションを用いて記述される．例えば，カラー画像をグレースケールに変換する場合，各画素を変換する処理は最も内側のループ内に記述され，この処理

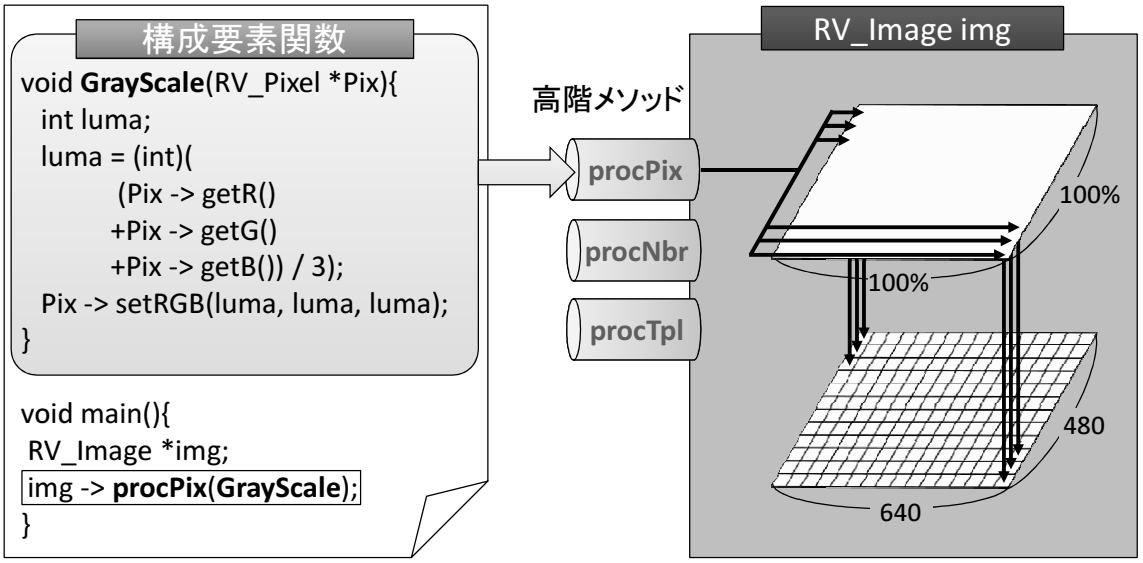

図 2: RaVioli の画像処理プログラム

が画像中の全ての画素に繰り返し適用される．このように，ループを用いる場合，プログラマは画像の幅と高さを意識してプログラムを記述しなければいけない．

一方，RaVioli では画像の構成要素である画素，または動画像の構成要素である単一フレームに対する処理のみを関数として定義し，その関数を RaVioli が提供しているメソッドに渡すことで，画像中の全ての画素に対して処理を施すことが可能である．RaVioli ではこの構成要素に対する処理を記述した関数を**構成要素関数**といい，その構成要素関数を引数にとるメソッドを**高階メソッド**と呼ぶ．ここで，RaVioli を用いてカラー画像をグレースケールに変換する処理の様子を**図 2** に示す．RaVioli では画像情報を持つクラス RV_Image のインスタンス img の高階メソッド procPix() に構成要素関数 GrayScale() を渡すのみでよい．この高階メソッド procPix() は img が持つ画像の全ての画素に，GrayScale() を繰り返し適用する．このような処理構造を用いることで，プログラマは解像度や繰り返し処理を意識することなく画像処理プログラムが記述できる．

次に，RaVioli の動画像処理プログラムとその処理の様子を**図 3** に示す．画像情報を RV_Image クラスのインスタンスにカプセル化したのと同様に，動画像中のフレームやフレーム数，フレームレートといった動画像に関する情報を RV_Streaming クラスのインスタンスにカプセル化している．プログラマは動画像の構成要素である単一フレームに対する処理のみを関数として定義する．ここで，フレームに対する処理とは先ほどの画像に対する処理と同義である．そのため， 図 3 に示すように，構成要素関数

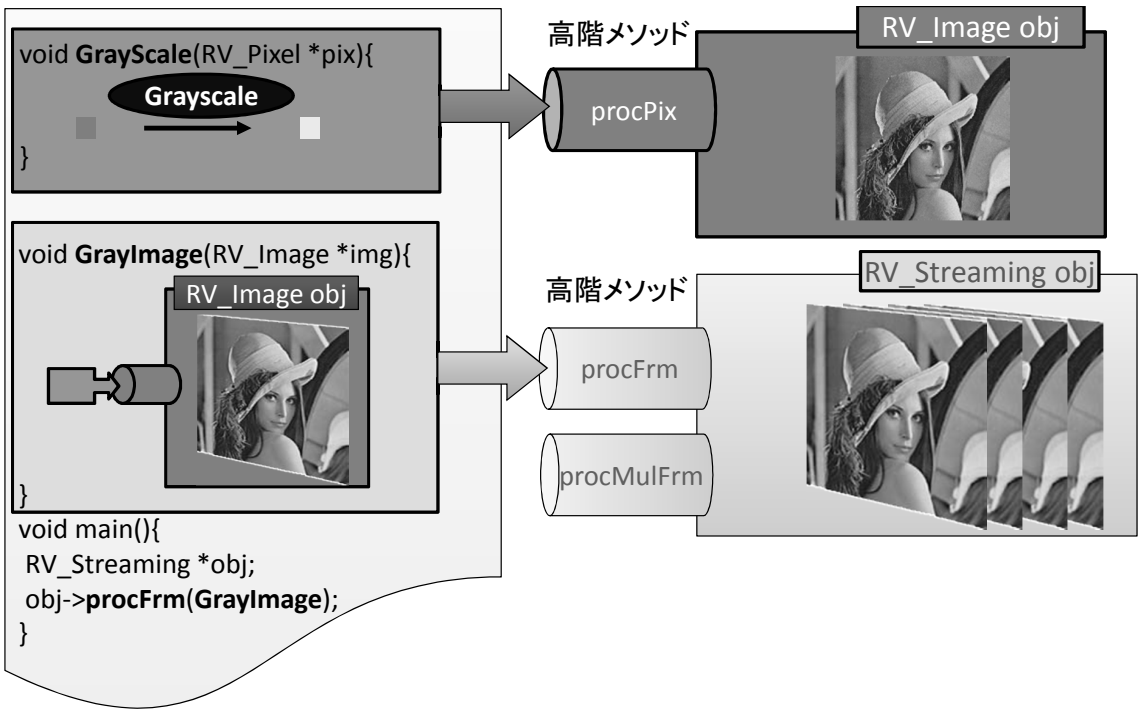

図 3: RaVioli の動画像処理プログラム

GrayImage() 内には， GrayScale() などの構成要素関数を引数にとる RV_Image クラスの高階メソッド呼び出しが含まれる．そして，その関数 GrayImage() を RV_Streaming クラスの高階メソッドに渡すことで，動画像中の全てのフレーム対して処理を適用することが可能である．このような処理構造を用いることで，プログラマは動画像の構成要素であるフレームの幅や高さ，フレーム数，フレームレートなどを意識することなく動画像処理プログラムを記述可能である．

3.1.2 自動処理量調整

複数のプロセスが並行に実行される汎用 OS 上では，動画像処理に必要な CPU リソース量を常に確保できる保証はない．そのため，汎用 OS 上でリアルタイム動画像処理システムを実現することはいまだに困難である．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て処理量を減らすことが考えられる．RaVioli はプログラマから解像度を隠蔽することで，負荷に応じて処理解像度を動的に変動させることを可能にした．

RaVioli は空間解像度と時間解像度を制御するために，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（S_S）**と，処理対象フレームの間隔を示す**時間解像度ストライド（S_T）**を持っている．これらのストライドを増減させることにより空

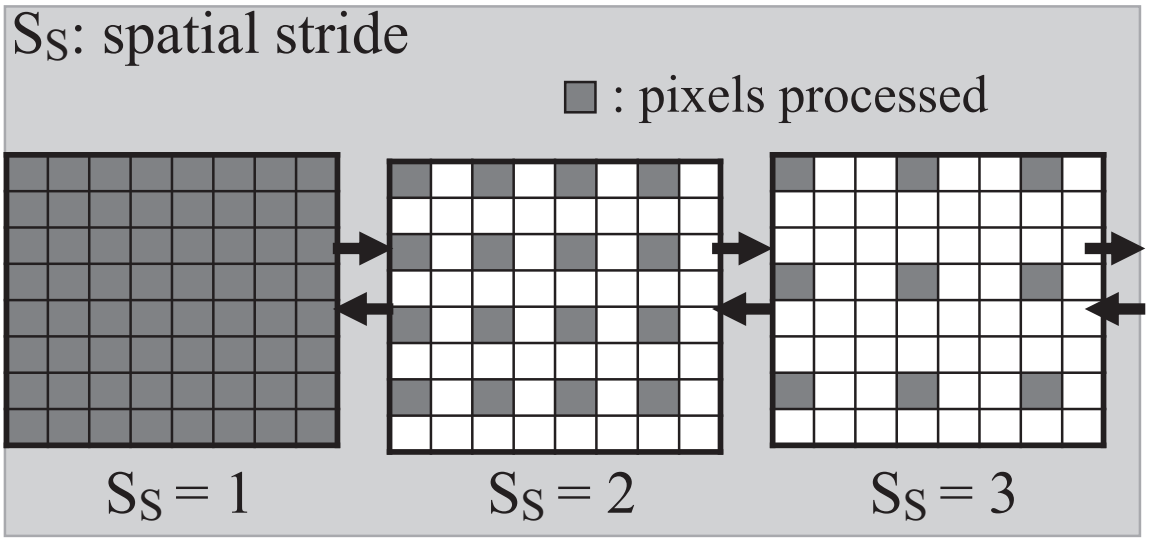

図 4: 空間解像度ストライドの変更

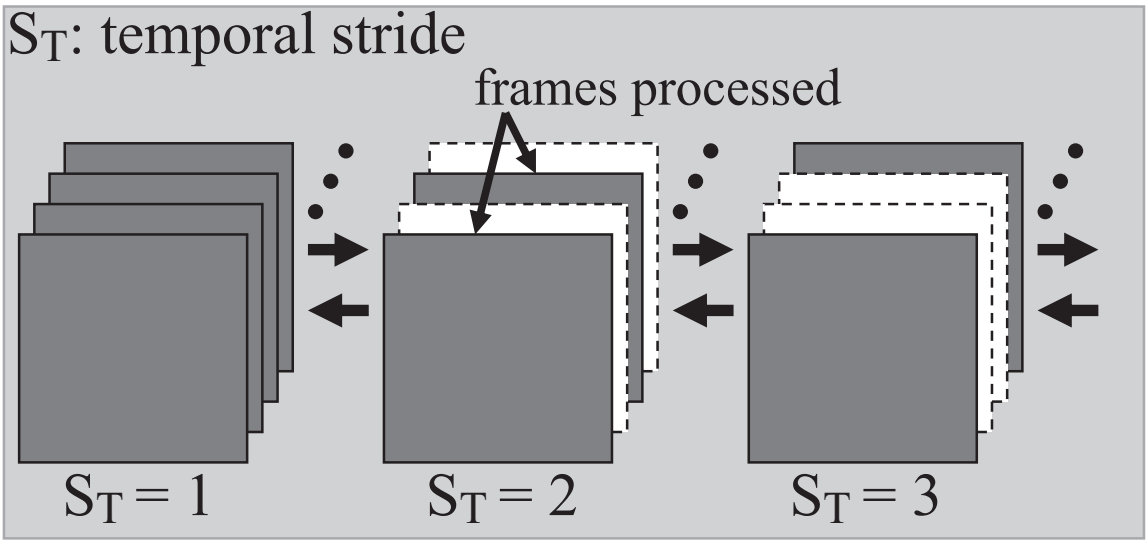

図 5: 時間解像度ストライドの変更

間解像度と時間解像度を変動させている．ここで，空間解像度を変動させるときの処理方法を**図 4** に示す．空間解像度ストライド $S_S = 1$ のとき，画像中の全ての画素が処理される．空間解像度ストライドを増加させ $S_S = 2$ となると，処理対象画素は 1 つおきとなり，空間解像度が低減する．このとき，全体の処理画素数は $S_S = 1$ のときの 1/4 となる．さらに空間解像度ストライドを増加させ $S_S = 3$ とすると，処理画素数は 1/9 となる．

一方で，時間解像度を変動させるときの処理方法を**図 5** に示す．時間解像度ストラ

イド $S_T = 1$ のとき，入力フレーム全てを処理する．時間解像度ストライドを増加させ $S_T = 2$ となると，処理対象フレームは 1 つおきとなり，時間解像度が低減する．このとき，全体の処理フレーム数は $S_T = 1$ のときの 1/2 となる．さらに時間解像度ストライドを増加させ $S_T = 3$ となると，処理フレーム数は 1/3 となる．

また，プログラマは空間解像度および時間解像度に対する優先度を指定することができ，RaVioli は指定された優先度の比に応じて解像度を維持する．これにより，プログラマは処理内容に応じて優先度を設定するだけで，目的のプラットフォームに適したアプリケーションの作成が可能となる．例えば，動物体検出などの時間分解能の重要な処理では，時間解像度が優先されるように設定することで，空間解像度が重点的に低減され，厳密なリアルタイム処理を実現することができる．一方，顔認証などの空間分解能の重要な処理では，空間解像度が優先されるように設定することで，時間解像度が重点的に低減され，処理精度を確保しつつリアルタイム性を実現することができる．

解像度の優先度は 2 つの値（P_S, P_T）の組である**優先度セット**を指定することで設定可能である．P_S は空間解像度に対する優先度を表し，P_T は時間解像度に対する優先度を表す．例えば，$(P_S, P_T) = (1, 3)$ と設定した場合，時間解像度を空間解像度よりも 3 倍優先したいということを表し，RaVioli は空間解像度ストライドと時間解像度ストライドを 3 : 1 の割合で維持しようとする．

3.1.3 問題点

RaVioli は処理負荷に応じて，解像度を低減させて処理量を適切な量に調整する．このとき，処理する画素数やフレーム数が低減するので，動画像処理の精度も低下する．これは，リアルタイム性を保証する際には避けられないことであるが，2 つの解像度をできるだけ高く維持することが望まれる．この問題に対して，プログラマは優先度を設定することで，どちらかの解像度の低減を抑えることが可能である．しかし，処理が間に合わないとき，優先度を低く設定された解像度が大幅に低減することにより，プログラマが期待した処理結果を得られない場合が存在すると考えられる．

そこで，まず空間解像度の優先度が低く設定される場合を，侵入者検知システムを例に説明する．ここで想定する侵入者検知システムとは，入力画像から侵入者を検知し，侵入者が検知された時刻の画像をユーザに提示するシステムである．このシステムの目的は素早く行動する侵入者を見逃すことなく，かつ侵入者の顔をできるだけ詳細な画像で検出することである．このシステムでは，侵入者を見逃すことだけは避けるために，全てのフレームを処理するように優先度を設定する．そのため，RaVioli は

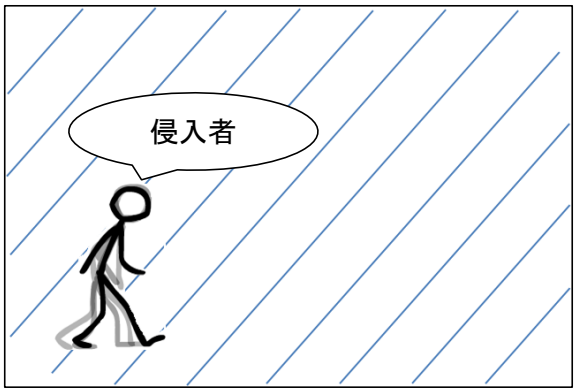

図 6: 入力フレーム

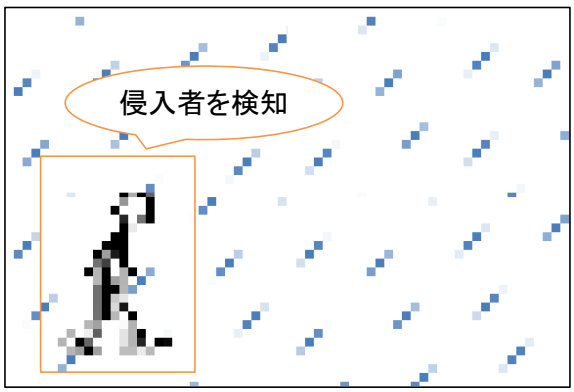

図 7: 出力フレーム

空間解像度ストライドを増加させることで処理量を調整する．ここで，このシステムを実際に動作させた場合を考える．このシステムへの入力を**図 6** に示す．図 6 は左下の領域に侵入者が現れた時のフレームである．また，左下の領域以外には侵入者はいないものとする．この入力フレームを空間解像度が大幅に低減した状態で処理すると，**図 7** のような出力が得られる．侵入者を検知することはできているが，侵入者の顔を詳細に検出することは困難であり，システムの目的を果たしていない．

一方，時間解像度の優先度が低く設定される場合を，携帯電話などに搭載されている QR（Quick Response）コード読み取りシステムを例に考える．まず，このシステムの動作を説明する．利用者は携帯電話のカメラを使って，QR コードを読み取るが，カメラ撮影の様にシャッターを押して画像を取り込むのではなく，ビデオ撮影のように読み取りたい QR コードにカメラを向けて一定間隔ごとに画像を取り込む．これには，シャッターを押すことによる画像のぶれを解消する目的がある．このようにして取り込まれる画像から QR コードを捉えた時に，QR コードから情報を取り出す．このシステムに求められるのは，QR コードを正確に読み取り，なおかつ読み取りにかかる時間をできるだけ短くすることである．そこで，読み取りの失敗だけは避けるために全ての画素を処理するように優先度を設定する．そのため，RaVioli は時間解像度ストライドを増加させることで処理量を調整する．しかし，時間解像度を大幅に低減させて処理を行うことで，QR コードから情報を取り出すまでに時間がかかり，リアルタイムに処理されていないようにユーザが感じる可能性がある．

このように，空間解像度や時間解像度を低減させることで，プログラマが期待した処理結果が得られなくなる場合が存在する．この RaVioli の処理精度低下の問題に対する既存の解決手法として，RaVioli の画像処理プログラムを並列化することで処理時

間を短縮し，処理精度の低下を抑制する手法が提案されている [2]．次節でその並列化手法について詳細に説明する．

3.2 自動空間分割並列化

RaVioli の問題である処理精度低下を抑制するために，RaVioli プログラムの並列化により，処理時間を短縮する既存手法について説明する．また，この既存手法は RaVioli の逐次プログラムを並列プログラムへと変換するプリプロセッサを提供しているため，このプリプロセッサによる，リダクション処理の必要性の検出とリダクション処理プログラムの自動生成についても説明する．

3.2.1 概要

チップ上に複数のコアを搭載するプロセッサが一般的になり，これらは研究開発分野で利用される高価なサーバなどだけではなく，安価な汎用 PC にも搭載されるようになってきている．そのため，複数のコアを有効に活用できるようにアプリケーションを改良することが重要になってきている．

それを実現する手法の一つに処理の並列化がある．一般的な画像処理は 3.1.1 項で述べたように，1 画素や近傍画素集合などに対する処理がループ文による繰り返し処理により画像全体に適用される．この繰り返し処理にはデータ並列性があるため，並列に処理することが可能である．例えば，グレースケール化処理は**図 8** に示すように，画像を均等な大きさの 4 つに分割し，各スレッドがその部分画像の開始座標（`xStart`, `yStart`）と終了座標（`xEnd`, `yEnd`）を指定して，グレースケール化処理を記述した関数 func を実行することで並列化できる．しかし，以前の結果を利用して次のイテレーション部分を計算するような，データの処理順に依存する処理の場合は，並列化すると処理結果の正当性を保証できない．そのため，このようなデータ並列化はループ内のイテレーションに互いに依存がないことが保証されている場合のみ可能である．また，データの処理順に依存していない場合でも，ループ外で共有している変数に対するデータの読み出しや書き込みがあると，その共有変数へのアクセスに競合が発生する可能性がある．このアクセス競合の解決方法の一つとして，並列数分用意した一時的な格納領域に対してデータを読み書きし，処理終了時にそれらのデータを逐次的に統合するリダクション処理がある．リダクション処理を用いることで，ループ内のイテレーションが完全に独立で動作し，ループが終わるまで他スレッドに影響を与えないため，高い並列度を保つことができる．

以上で述べたことを意識して画像処理の並列化プログラムを作成するためには，ス

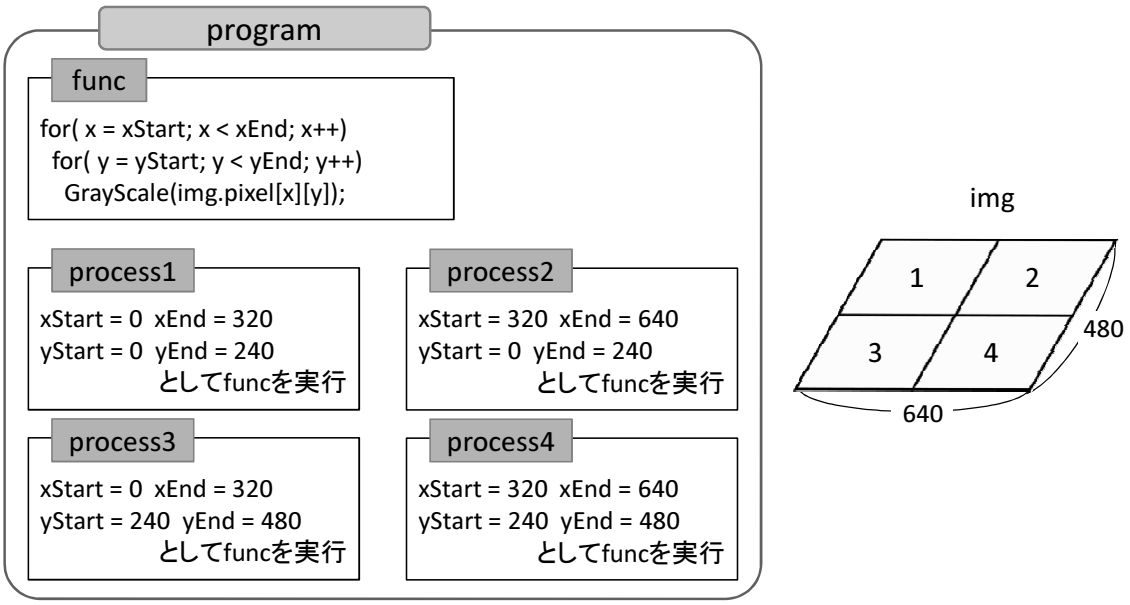

図 8: 一般的な画像処理プログラムの並列化

レッドの生成，管理のために pthread のような並列処理ライブラリの利用方法の習得や，上記に示した処理順序依存や競合といった問題を起こさないプログラミングスキルが必要となる．これはプログラマにとって大きな負担である．

そのため，既存手法 [2] は RaVioli の逐次プログラムを並列プログラムに変換するプリプロセッサを提供している． 3.1.1 項で述べたように RaVioli では，プログラマは構成要素に対する処理のみを関数として定義するため，並列化箇所の抽出が容易である．また，RaVioli は繰り返し処理をライブラリ内で制御可能なため，**図 9** に示すように並列化ができる．この例では，プログラマが記述した構成要素関数 GrayScale を高階メソッド procPix() を通じて受け取り，高階メソッド内部で画像を 4 つに分割し，複数のスレッドを用いてその部分画像に構成要素関数を適用している．このように RaVioli では，高階メソッド内部で処理を繰り返す範囲を決定できるため，プログラマに意識させずに並列化が可能である．しかし，先ほど説明した共有変数へのアクセス競合は RaVioli を用いた場合でも起こりえるため，プリプロセッサはリダクション処理が必要となる変数を検出し，リダクション処理のコードを自動生成する機能を備えている．次項で，このプリプロセッサによるリダクション処理の自動生成について詳細に説明する．

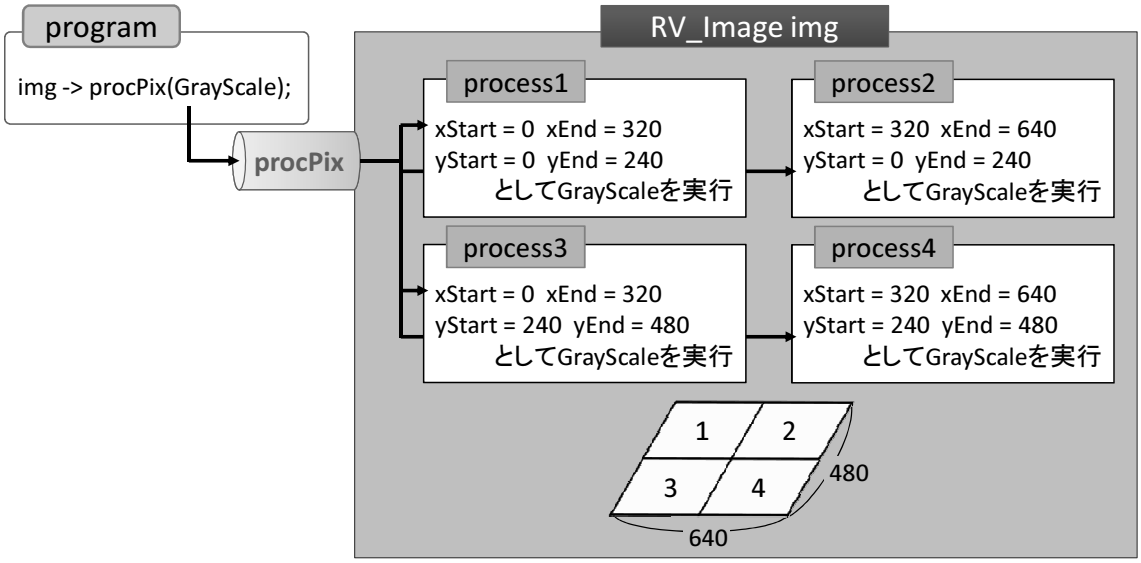

図 9: RaVioli プログラムの並列化

3.2.2 プリプロセッサによるリダクション処理の自動生成

RaVioli では，高階メソッドを用いて，各要素に構成要素関数を繰り返し適用する．そのため，この繰り返し処理間でデータを共有する場合，必ず構成要素関数内に大域変数へのアクセスが存在する．したがって，高階メソッド内の繰り返し処理を並列に実行する場合，この大域変数への競合を解決しなければいけない．そのため，リダクション処理を用いて競合を解決する．プリプロセッサはリダクション処理が必要な変数かどうかの判定に以下の条件を用いる．

条件 (1) 大域変数に対して読み出しおよび書き込みを行っている

競合が発生する条件として，1 変数に対する読み書きが挙げられる．

条件 (2) 構成要素関数の適用順序に依存関係がない

構成要素に対する処理の順序によって，処理結果が変わってしまう場合，リダクション演算を用いても競合を回避することはできない．そのため，構成要素関数の適用順序に依存関係がある（以降，処理順依存がある）かどうかを判定する．

上記の条件を満たしている場合，その大域変数をリダクション処理の対象とする．

条件 (1) 大域変数に対して読み出しおよび書き込みが行われているかの判定方法を，**図 10** に示すプログラムを例に，説明する．図中の関数 `bar1` は 1 画素に対する処理を記述した構成要素関数である．例えば，(左辺) = (右辺); といった代入文において，図 10 の 5 行目のように (左辺) だけに大域変数があれば書き込みのみが行われていると判定し，6 行目のように (右辺) にも大域変数があれば読み出しおよび書き込みが行なわ

```
1 int foo=0, foo1=0, foo2=0, foo3=0;
2
3 void bar1(RV_Pixel* p){
4   if(foo > 5){         // 読み出しのみ
5     foo1 = 5;          // 書き込みのみ
6     foo2 = foo2 + 5;   // 読み出しおよび書き込み
7     foo3 += 1;         // 読み出しおよび書き込み
8   }
9 }
```

図 10: 大域変数に対して読み出しおよび書き込みを行っているかの判定

れていると判定する．なお 7 行目のように，+=や-=といった複合代入演算子を用いた演算では，読み出しと書き込みの両方が行われている．また 4 行目のように if 文などの条件文に大域変数が使われている場合は，その大域変数への読み出しが行なわれていると判定する．

次に，処理順依存に関しては，現段階では次の 4 つの条件を全て満たさない場合に処理順依存がないと判定する．

条件 (2-A) 大域変数に対して加減算と乗除算を混在して使用している
条件 (2-B) if 文の条件文で使われている大域変数に対して，
比較した値と異なる値が if 文ブロック内で代入されている
条件 (2-C) 値が書き換えられた大域変数の値を画素へ書き込んでいる
条件 (2-D) ライブラリ内で定義されている関数の引数に大域変数を使用している
（RaVioli の関数は除く）

以下，それぞれの条件について具体例を用いて説明する．まず，**図 11** を例に条件 (2-A) について説明する．図 11 中の getR() や getG() は画素の RGB 色空間の R 値や G 値を返すメソッドである．そのため，構成要素関数の適用毎に値が異なる．ここで，5 行目は，foo1 に対して画素 p の R 値を足してから 2 をかけているが，この式を展開すると foo1 に対して+と*の演算を適用している．6 行目は，foo2 に-と/の両方の演算を適用している．このように +か-と*か/の両方を大域変数に適用した場合は処理順序に依存ができてしまう．一方 7 行目は foo1 に対して+のみを適用しており，8 行目は，foo2 に対して*と/のみを適用している．7，8 行目のような式は大域変数に対して適用する順序を入れ替えても，最終的な結果は変わらない．そのため，処理順依存がない．

次に，**図 12** を例に条件 (2-B) について説明する．図 12 の 4 行目は，条件式に使わ

```
1 int foo1=0;
2 double foo2=0;
3
4 void bar2(RV_Pixel* p){
5   foo1=(foo1+p.getR())*2;            //NG1
6   foo2=foo2/p.getR()-2;              //NG2
7   foo1=foo1+p->getR();               //OK1
8   foo2=foo2/p->getR()*(p->getG()+2); //OK2
9 }
```

図 11: 加減算と乗除算を混在して使用しているかどうかの判定

```
1 int foo=0;
2
3 void bar3(RV_Pixel* p){
4   if(foo > p->getR()) foo=foo+p->getR(); //NG1
5   if(foo > p->getR()) foo=p->getG();     //NG2
6   if(foo > p->getR()) foo=p->getR();     //OK
7 }
```

図 12: 比較した値と異なる値が代入されているかどうかの判定

れている foo に対して加算をしている．この式では，構成要素関数の適用順によって条件式で比較する foo の値が変化するため，最終的な foo の値が異なってくる．また 5 行目は，条件式で比較した p->getR() とは異なる値 p->getG() を foo に代入している．このとき，画素 p の G 値によって条件式の判定が変わるため，最終的な foo の値が異なってくる．そのため，処理順依存がある．一方 6 行目は，p->getR() の最小値を求めており，構成要素関数の適用順に依らず，最終的な foo の値は同じである．そのため，処理順依存がない．

次に，**図 13** を例に条件 (2-C) について説明する．図 13 の 4 行目は，大域変数 foo に対して画素 p の R 値を加算している．5 行目の setR() メソッドは引数にとった値を画素の R 値に設定するメソッドであり，値が書き換えられた大域変数 foo を画素 p の R 値に書き込んでいる．四則演算などで値が書き換えられた大域変数を出力画素に書き込むと，構成要素関数の適用順によって最終的な出力画像が変わってくるため，処理順依存がある．

```
1 int foo=0;
2
3 void bar4(RV_Pixel* p){
4   foo+=p->getR();
5   p->setR(foo);       //NG
6 }
```

図 13: 値が書き換えられた大域変数の値を画素へ書き込んでいるかどうかの判定

最後に，条件 (2-D) について説明する．リンクしたライブラリ内に定義されている関数は，関数の処理内容が詳細に分からないため，ライブラリ関数の引数に大域変数をとっている場合は解析不能であるとする．ただし RaVioli で定義されている関数については，プリプロセッサが各関数の処理順依存の有無に関する情報を持っているため，その情報を用いて処理順依存があるかどうかが判定できる．

このように検出された対象変数にリダクション処理を適用する．リダクション処理は，前項で述べた通り，並列数分用意した一時的な格納領域に対してデータを読み書きをし，処理終了時にそれらのデータを逐次的に統合する．そのため，リダクション処理の対象変数には自身の他に，各スレッドが読み書きの対象とする代替変数を定義する必要がある．そこで，プリプロセッサは，スレッド外部の変数をスレッド固有のものとして宣言する__thread 指定子を用いる．これは，変数の宣言時に__thread と指定することにより，その変数のコピーを各スレッド毎に保持できるようにする．全スレッドの処理終了時に，この代替変数が保持している値を対象変数に統合することでリダクション処理を実現する．

図 14 に構成要素関数からリダクション処理を生成する過程を示す．関数 average は画像中の全画素値の和，画素数，最小値を計算する構成要素関数である．また，pSum，pCnt および pMin はリダクション処理の対象となる大域変数である．まず，図 14 右側のプログラム（2 行目）に示すように，プリプロセッサはリダクション処理の対象変数である大域変数に対する代替変数__pSum，__pCnt および__pMin を__thread 指定子を用いて宣言する．次に，リダクション処理の対象変数を用いた式を含むコード（図 14 の左側，8-12 行目）を代替変数に対して読み書きするコード（図 14 の右側，8-12 行目）と，最後に代替変数を用いて結果を統合するコード（図 14 の右側，16-20 行目）に分割する．統合用のコードは関数として切り出され，処理終了時に各スレッドが一度だけ呼び出す．この統合用関数は高階メソッド内で実行するために，引数としてメソッド

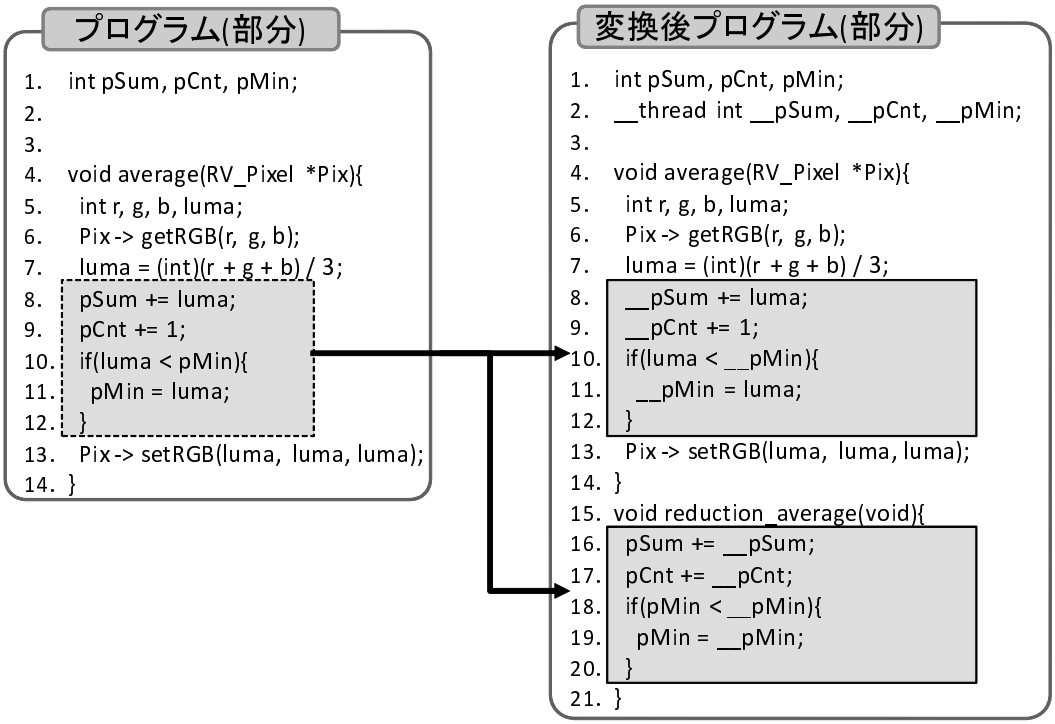

図 14: リダクション処理の自動生成

に渡す必要がある．そのため，プリプロセッサは高階メソッドの呼び出し部分のコードも変換する．以上のようなリダクション処理の生成を RaVioli プログラムに適用することで，プログラマが意識しなくても RaVioli プログラムを並列プログラムとして動作させることができる．

3.2.3 並列化の効果と問題

既存手法は，RaVioli の逐次プログラムを並列プログラムに変換することで，プログラマにプログラムの並列化を意識させることなく，画像処理を並列化することを可能にした．ここで，並列化によりどれくらい処理時間を短縮できるかを評価した結果を**図 15** に示す．評価には 8 コアで 32 スレッドを並行実行可能なプロセッサ UltraSPARC T1 を使用し，評価プログラムには次の 4 つのプログラムを使用した．

- *voronoi*: 複数個の母点に対して各画素がどの母点に一番近いかを計算し領域ごとに分けるボロノイ図の作成
- *laplacian*: ラプラシアンフィルタを用いたエッジ抽出
- *pixAverage*: 画素の平均値の算出
- *hough*: ハフ変換による直線検出

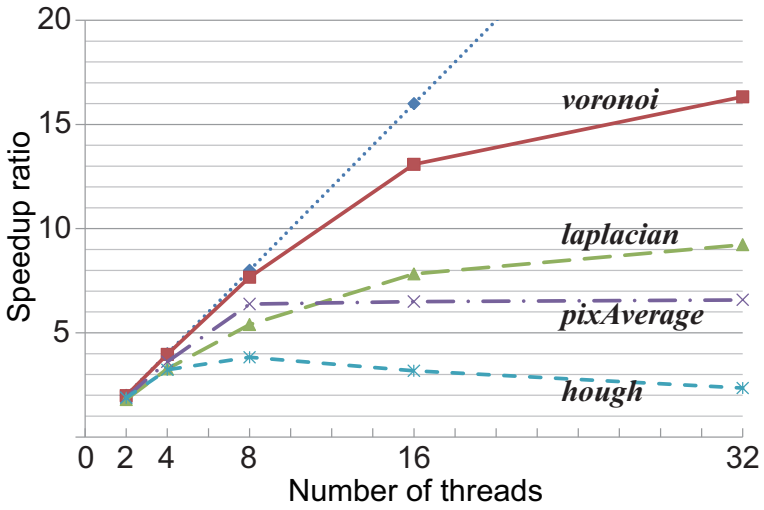

図 15: 並列化の性能向上

図 15 の横軸は並列数，縦軸は逐次プログラムの実行時間を 1 として正規化した高速化率を示している．図 15 からどのプログラムもコア数と同数の並列数までは，並列化により処理時間を短縮できたことが確認できる．この結果から，RaVioli の既存並列化機能により，処理時間を短縮し，処理精度の低下を抑制できると考えられる．

しかし，並列化による高速化の効果は限定的である．まず，実行環境が複数のコアを備えていなければ並列化の効果は得られない．また，たとえマルチコア環境でも他並行実行プロセスによっては全てのコアを有効活用できるとは限らない．そこで，入力の重要度によって処理精度を変動させる新しい処理量調整手法を提案する．これは並列化とは別のアプローチであるため，並列化と組み合わせることが可能である．この組み合わせにより，処理精度低下の抑制を目指す．

4　領域別処理量調整手法の提案

この章では，重要な入力に対する処理精度の低下を抑制する新しい処理量調整手法について説明する．まず，提案手法の着眼点について述べる．そして，提案手法の実現方法として，動画像ストリームの分割，分割された部分ストリーム毎のストライド変動について説明し，最後に動作モデルについて説明する．

4.1　提案手法の着眼点

3.1 節で説明したように，RaVioli はプログラマから解像度を隠蔽し，動画像処理時に解像度を変動させることで処理量を適切な量に調整する．これにより，動画像処理

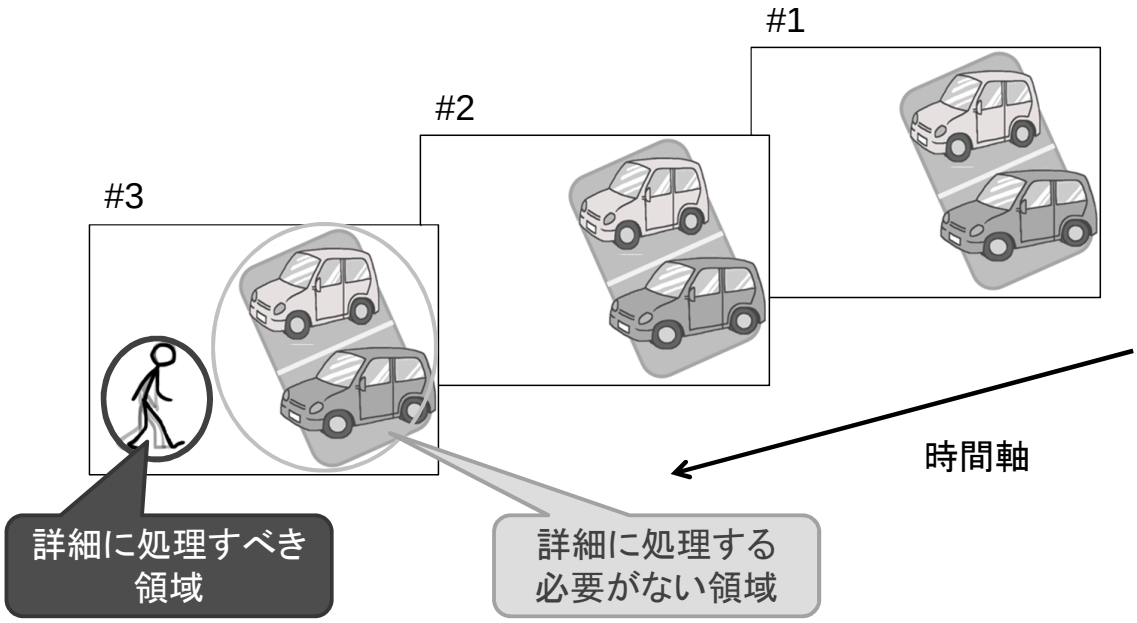

図 16: 侵入者検知システムへの入力とその特徴

のリアルタイム性を保証している．しかし，解像度を低減させることには限界があり，解像度の大幅な低減により，プログラマが期待した処理結果を得られなくなる可能性がある．そこで，重要な入力に対する処理精度の低下を抑制するために，リアルタイム動画像処理の入力の特徴に注目した，新しい処理量調整手法を提案する．

リアルタイム動画像処理には，侵入者検知システムや衝突回避システムのような空間解像度より時間解像度を重要とする処理と，顔認識システムや QR コード読み取りシステムのような時間解像度より空間解像度を重要とする処理が存在する．これらの動画像処理システムは 1/30 秒や 1/60 秒など一定の間隔で，入力画像をキャプチャし処理するが，入力によっては詳細に処理する必要がない領域が存在する．例えば，**図 16** のような入力を侵入者検知システムが処理する場合を考える．図中の 2 フレーム目のように侵入者が存在せず，前フレームからの変化がないとき，そのフレームを詳細に処理する必要はない．また，3 フレーム目のようにフレーム内に侵入者が存在する場合でも，侵入者が存在する領域以外の大半の領域は変化がなく，それらの領域は詳細に処理する必要がない．これと同様にその他のリアルタイム動画像処理にも，詳細に処理する必要がない領域が存在すると考えられる．そのため，このような領域に対する処理量を削減することにより，動画像処理のリアルタイム性を維持しつつ，重要な領域に対する処理精度の低下を抑制することが可能である．

しかし，RaVioli は入力フレーム内の全ての領域を同じ精度でしか処理できない構造

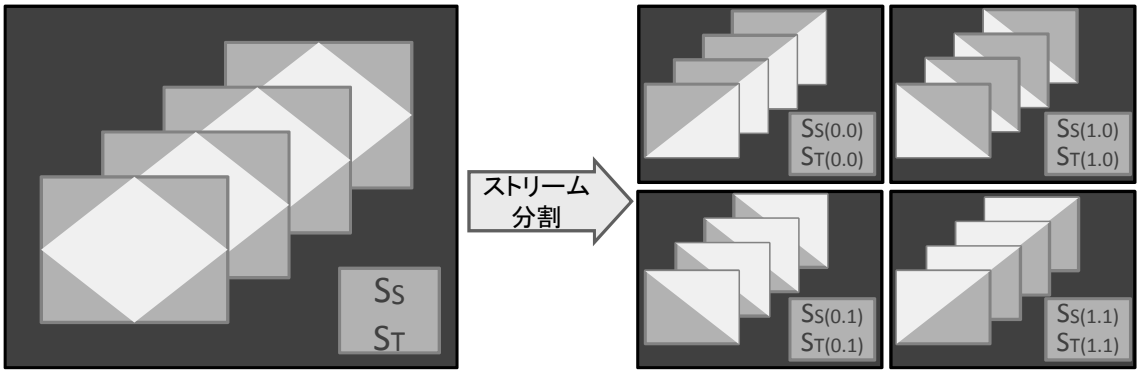

図 17: 動画像ストリームの分割

をとっている． 3.1 節で述べたように，RaVioli は動画像ストリームに対して，時間解像度ストライドと空間解像度ストライドをそれぞれ一つのみ設定し，それらのストライドに基づいて各解像度を変動させ，処理を適用している．そのため，動画像処理に必要な CPU リソース量を確保できない場合，全領域の解像度が低減してしまい，各領域の重要度に関わらず等しく処理精度が低下してしまう．

そこで，詳細に処理すべき重要な領域をできるだけ高い精度で処理するために，領域別に処理精度を変動させることを提案する．これを実現するためには，動画像ストリームをいくつかの部分ストリームに分割し，各ストリーム毎に解像度を変動できるようにする必要がある．以降， 4.2 節では，この動画像ストリームの分割について説明し， 4.3 節では，分割された部分ストリーム毎のストライド変動方法について述べ，4.4 節で提案手法の動作モデルを説明する．

4.2 動画像ストリームの分割

提案手法では，領域別に処理精度を変動させるために，動画像ストリームをいくつかの部分ストリームに分割する．ここで，動画像ストリームを 4 つの部分ストリームに分割する様子を**図 17** に示し，提案手法の動画像ストリームの分割方法について説明する．図 17 の左側は分割前の動画像ストリーム，右側は分割後の各部分ストリームを表している．提案手法では，動画像ストリームを均等な大きさの部分ストリームに分割する．そのため，各フレームは均等な大きさの部分フレームに分割される．このように分割することで，各部分フレームにかかる処理量を見積もりやすくなる．これは，複数のスレッドを用いて並列処理する際に重要となる，各スレッドの処理負荷の均衡化を容易にする．なお，この例では動画像ストリームを 2×2 に分割しているが，

分割数は $N \times M$ の行列の形でプログラマが指定できるようにする．

また，図 17 中の S_S，S_T は空間解像度ストライド，時間解像度ストライドをそれぞれ表しており，提案手法では，部分ストリーム毎に空間解像度ストライドおよび時間解像度ストライドをそれぞれ保持する．これにより，従来の処理量調整手法が画像や動画像全体に対してしか設定できなかった解像度ストライドを動画像の領域別に設定し，変動できるようになる．この各部分ストリームの解像度ストライドを増減させることで，領域別に処理精度を変動可能にする．

4.3 部分ストリームのストライド変動方法

動画像処理のリアルタイム性を維持しつつ，詳細に処理すべき領域に対する処理精度の低下を抑制するためには，各部分ストリームが保持している両解像度ストライドに複数種類の値を設定する必要がある．本提案手法では，リアルタイム動画像処理の入力に存在する，詳細に処理すべき領域と詳細に処理する必要がない領域の 2 つの領域に注目するため，2 種類の解像度ストライド値を各部分ストリームに設定する．

詳細に処理すべき領域には，その時点で設定可能なできるだけ小さい解像度ストライド値を設定する．また，RaVioli はそれらの領域に設定する各解像度ストライドを優先度セットに従って，変動させる．これは従来の処理量調整手法とまったく同じである．提案手法では，この詳細に処理すべき領域に設定されるストライドを**ベースストライド**と呼ぶ．一方の詳細に処理する必要がない領域には，ベースストライド値よりも一定の値だけ大きな値を設定する．この詳細に処理する必要がない領域に設定されるストライドを**ラフストライド**と呼ぶ．

ここで，空間解像度および時間解像度に対するベースストライドとラフストライドの設定について，それぞれ**図 18**，**図 19** を例に説明する． 二つの図は共に，動画像ストリームを 2×2 に分割した時の例を示しており，各フレーム内の破線は分割領域の境界線を表している．各部分領域は左側の 2 つの領域が詳細に処理すべき領域，右側の 2 つの領域が詳細に処理する必要がない領域とする．また，ラフストライドはベースストライドの 2 倍とする．まず，フレーム全体の空間解像度ストライド $S_S = 1$ のとき，図 18 の左側に示す通り，詳細に処理すべき領域にはベースストライドとして 1 を設定し，詳細に処理する必要がない領域にはラフストライドとして 2 を設定する．このとき，フレーム全体にかかる処理量は，全体を $S_S = 1$ で処理する場合と比べて，5/8 となる．空間解像度ストライド $S_S = 2$ の場合は，図の右側に示す通り，ベースストライドは 2，ラフストライドは 4 であり，それらのストライド値をそれぞれの領域に設定す

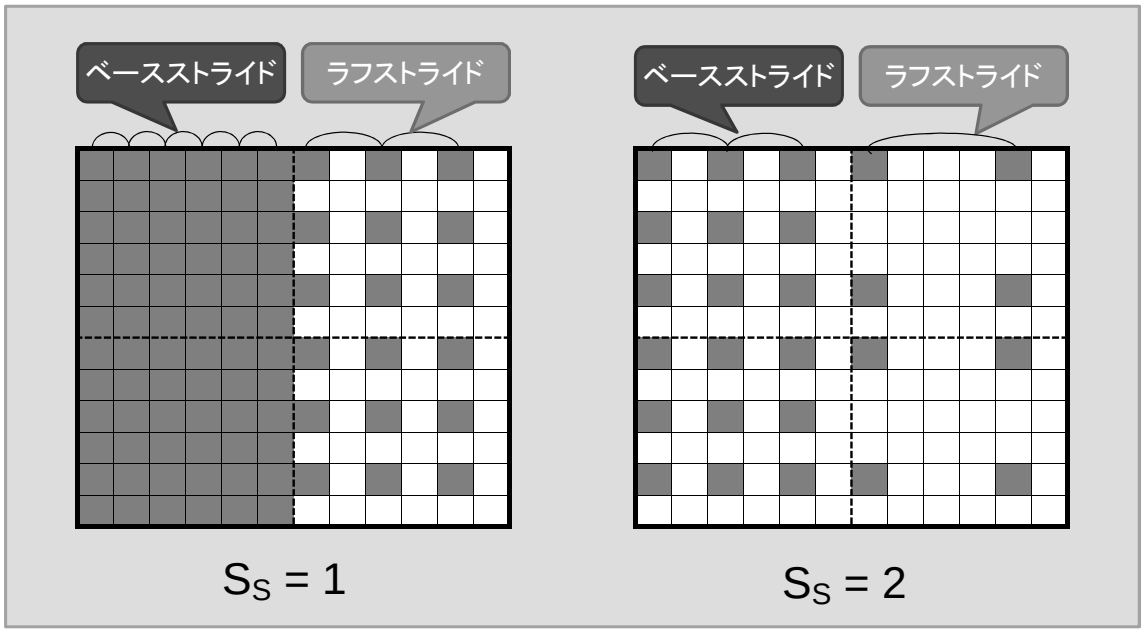

図 18: 空間解像度ストライドに対するベーススストライドとラフストライドの設定

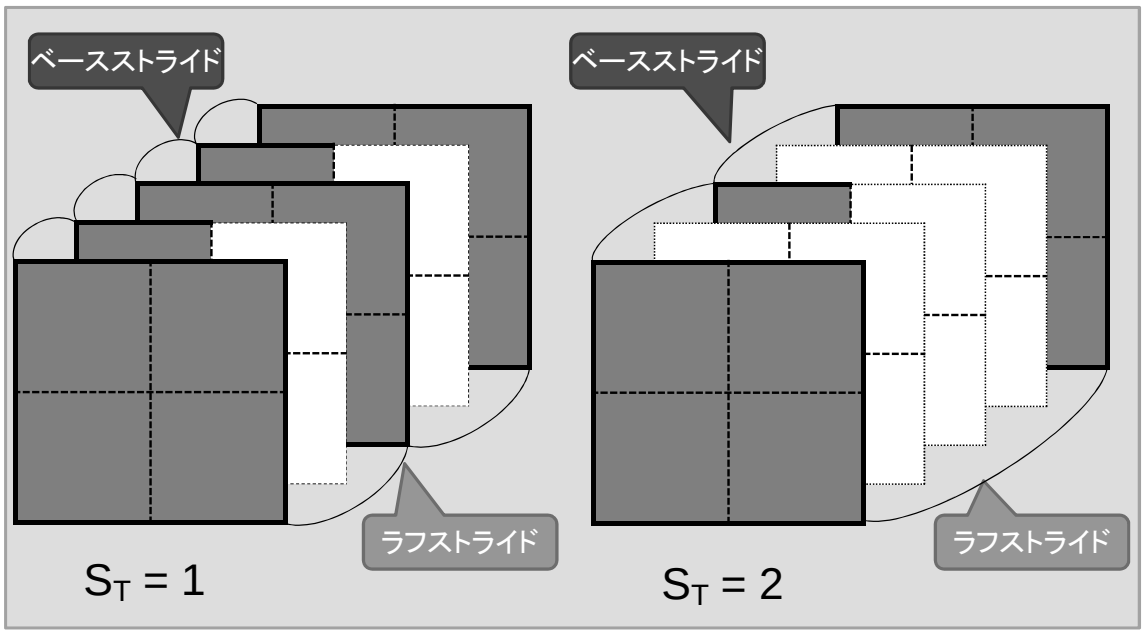

図 19: 時間解像度ストライドに対するベーススストライドとラフストライドの設定

る．このとき，フレーム全体にかかる処理量は，全体を $S_S = 1$，$S_S = 2$ で処理する場合と比べて，それぞれ 5/32，5/8 となる．

一方，動画像全体の時間解像度ストライド $S_T = 1$ のとき，図 19 の左側に示す通り，詳細に処理すべき領域を含む部分ストリームにベーススストライドとして 1 を設定し，詳

細に処理する必要がない部分ストリームにはラフストライドとして 2 を設定する．このとき，動画像全体にかかる処理量は，全体を $S_T = 1$ で処理する場合と比べて，3/4 となる．時間解像度ストライド $S_T = 2$ の場合，図の右側に示す通り，ベースストライドは 2，ラフストライドは 4 であり，それらのストライド値を各部分ストリームに設定する．このとき，動画像全体にかかる処理量は，全体を $S_T = 1$，$S_T = 2$ で処理する場合と比べて，それぞれ 3/8，3/4 となる．

以上のように，提案手法では時間解像度ストライド，空間解像度ストライドに対して，2 種類のストライド値を設定することで，詳細に処理する必要がない領域に対する処理量を削減する．これにより，詳細に処理すべき領域に対する処理精度の低下を抑制できる．また提案手法では，先述の通り，空間解像度ストライドと時間解像度ストライドのベースストライドにできるだけ小さい値を設定し，優先度セットに基づいてそれらの値を変動させる．これにより，従来の RaVioli の機能である動的な処理量調整によるリアルタイム性の維持機能を損なうことなく，提案手法を実現できる．そこで，次節では従来の処理量調整手法と提案する処理量調整手法の動作を比較し，提案手法の動作モデルを説明する．

4.4 動作モデル

前節までに，動画像処理時の無駄な処理を削減するために，動画像ストリームを分割して，領域ごとに解像度を変動させて処理量を調整する方法を提案した．そこで，侵入者検知システムを例として，従来の処理量調整手法を用いた RaVioli と提案手法を用いた RaVioli の処理を比較することにより，提案手法の動作モデルを説明する．なお，この侵入者検知システムでは，侵入者を見逃さないために全てのフレームを処理するように優先度を設定すると仮定する．そのため，RaVioli は処理量調整のために空間解像度を変動させる．

まず，既存の RaVioli を用いて侵入者検知を行う様子を**図 20** に示す．図 20 の上段はフレーム 1，フレーム 2，フレーム 3，フレーム 4 の順にキャプチャされた入力フレームを示している．この例では，数フレーム前から変化がない状態が続き，フレーム 1 で初めて侵入者が現れたとする．この侵入者はフレームの左下から現れ，フレーム 4 で示す位置まで移動すると仮定する．また，図 20 下段は既存の RaVioli を用いて得られる出力を示している．この図では，出力画像に被せるような形で処理された画素を描くことで，そのフレームの処理解像度ストライドを表現している．

既存の RaVioli は，全ての入力に対して，できるだけ高い解像度で処理しようとす

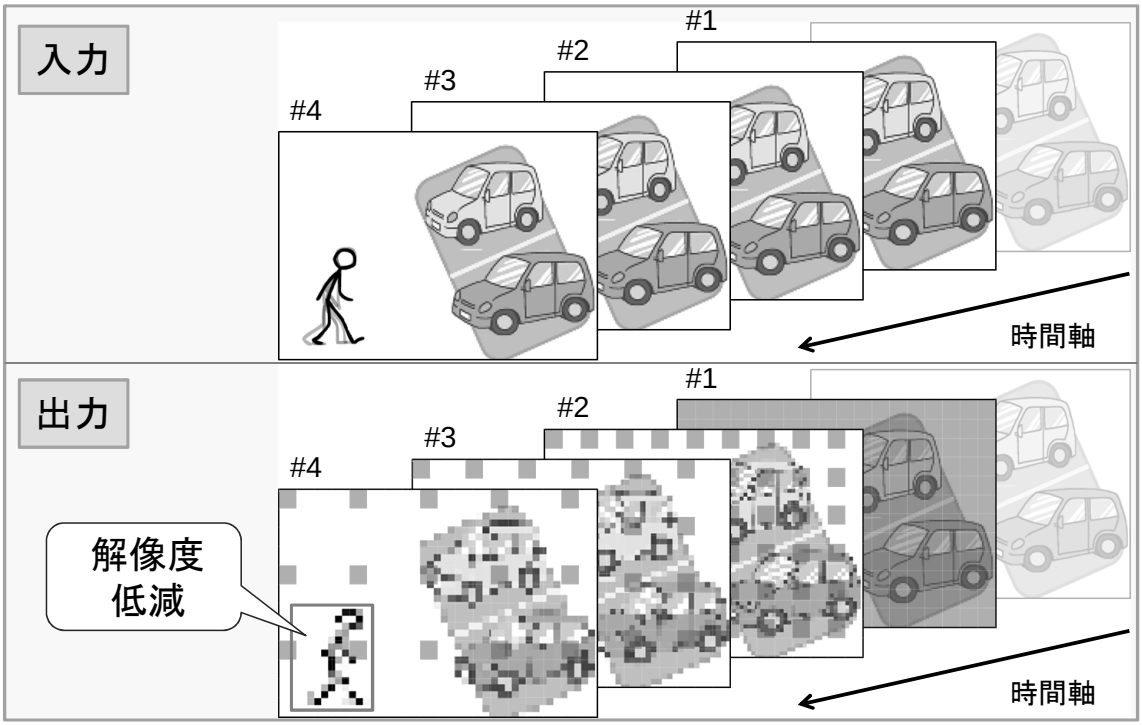

図 20: RaVioli を用いた侵入者検知

るため，侵入者が現れ始めたフレーム 1 のような，大半の領域に前入力からの変化がないフレームに対しても通常通り処理する．ここで，利用可能な CPU リソース量が減少し処理が間に合わなかったとすると，次のフレーム 2 は空間解像度を低減させて処理される．RaVioli は処理が間に合うまで，解像度ストライドを徐々に大きくするため，解像度低下が数フレームに渡って続くことがある．この例でもフレーム 3，フレーム 4 で解像度低下が起こったとする．ここで，フレーム 4 の処理では，出力フレームの空間解像度が低減しているため侵入者を詳細に検出することは難しい．

一方提案手法では，まず動画像ストリームを構成する各入力フレームをプログラマが指定した分割数に分割する．そして，その各領域で詳細に処理すべきかどうかを自動的に判定する．その判定の結果，詳細に処理すべき領域やその領域を含む部分ストリームにはベースストライドを設定し，詳細に処理する必要がない領域やその部分ストリームにはラフストライドを設定する．そして，設定された解像度ストライドに基づいて各領域を処理する．この侵入者検知システムでは，動物体を含む領域を詳細に処理すべき領域と判断し，その領域にベースストライドを設定する．

提案手法を用いた侵入者検知の様子を**図 21** に示す．図 21 は先ほどの図 20 と同じ入力に対して，画像処理を施した様子を示している．入力フレーム内の垂直，水平方向

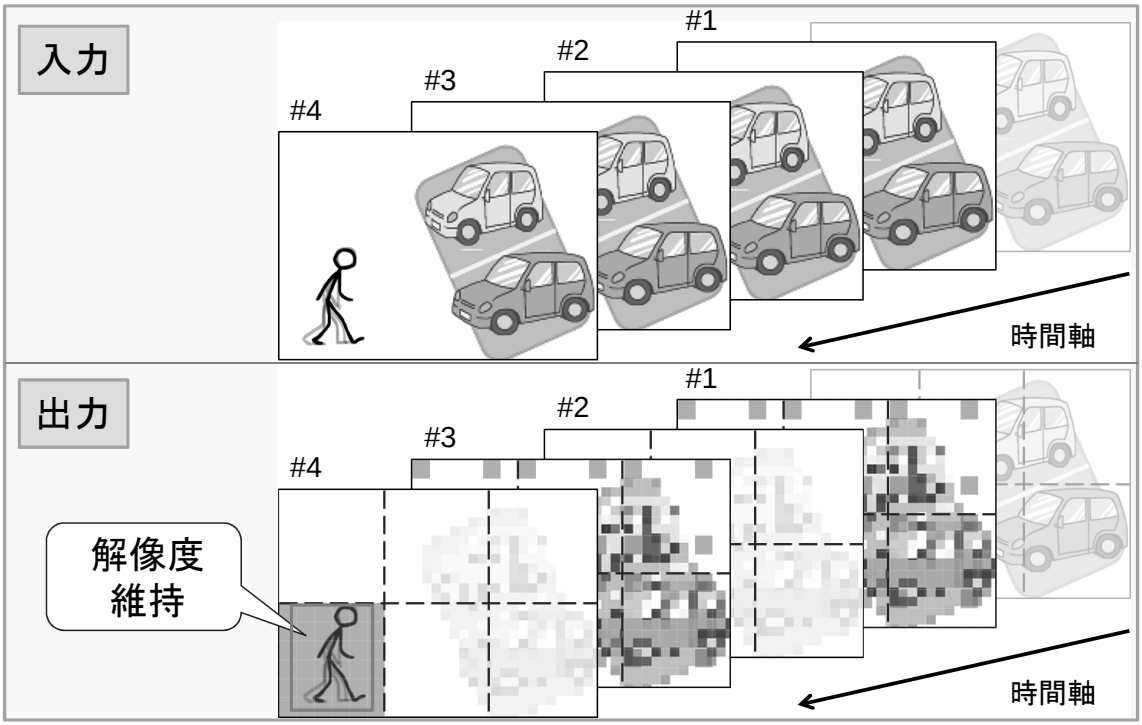

図 21: 提案手法を用いた侵入者検知

の破線は分割領域の境界線を表している．この例では，2 × 3 に領域を分割したとする．フレーム 1，フレーム 2，フレーム 3 に対して，既存の RaVioli では常に詳細な処理を施していた．一方，提案手法では，詳細に処理する必要がない領域や部分ストリームにラフストライドを設定し，その値を用いて処理する．例えば，フレーム 1 やフレーム 3 は詳細に処理する必要がない領域を粗く処理しており，一方で，フレーム 2 はそれらの領域を処理していない．これにより処理量を削減できるため，利用可能な CPU リソース量が減少しても，ベースストライドを増大させることなく詳細に処理すべき領域を処理できる．ここで，フレーム 4 の処理では，前フレームからの変化量が大きい左下の領域を詳細に処理すべき領域であると判定し，ベースストライドをその領域の解像度ストライドに設定する．フレーム 1 から 3 の処理時にフレーム全体にかかる処理量を削減できるため，このとき設定されるストライド値は既存の RaVioli を用いた時より小さい．そのため侵入者を詳細に検出することが可能である．

このように，提案手法は詳細に処理する必要がない領域やその領域を含む部分ストリームに対する処理量を削減することを可能にする．これにより，従来よりも処理量を削減することが可能になるため，詳細に処理すべき領域に対する処理精度の低下を抑制できる．この提案手法の実現には，入力フレーム内の各部分領域を異なるストラ

イドで処理することや，詳細に処理すべき領域を判断することが必要になる．次章でこれらの課題に対する実現方法を述べる．

5　提案手法の実装

この章ではまず，動画像ストリーム中の各フレームを領域別に処理するために必要な実装について説明する．提案手法では，分割領域を管理する新しいクラス RV_TileImage を RaVioli に追加し，RV_TileImage インスタンスを用いてフレーム全体を処理するために RV_Image クラスの高階メソッドを拡張する．次に，各領域を詳細に処理すべきかどうかを判定する方法について述べる．提案手法では，領域を詳細に処理すべきかどうかを判定する関数（以後，判定関数と呼ぶ）を RaVioli に導入する．最後に，領域別に解像度ストライドを変動させて処理する方法について説明する．

5.1　フレームの領域別処理

この節では，フレームを領域別に処理するために必要な RV_TileImage クラスの追加，高階メソッドの拡張について説明する．

5.1.1　領域クラス RV_TileImage の追加

提案手法では，領域別に処理精度を変動させるために，動画像ストリームを部分ストリームに分割する．このとき各フレームは均等な大きさの部分領域に分割される．そこで，この部分領域を管理するための新しいクラス RV_TileImage を RaVioli に追加する．RV_TileImage クラスの概要を**図 22** に示す． RV_TileImage クラスは，部分領域が含まれる画像を保持する RV_Image クラスと関係がある．図 22 に示す通り，RV_Image クラスのインスタンスは画像の幅や高さ，空間解像度ストライドと時間解像度ストライド，画素配列などの画像情報の領域，および高階メソッドをメンバとして持っている．一方，RV_TileImage クラスのインスタンスは領域の幅や高さ，両解像度ストライド，高階メソッド，領域の左上を表す開始座標，および判定関数へのポインタを持つ．しかし，RV_Image インスタンスとは違い，RV_TileImage インスタンスは画像情報のための領域を確保する代わりに，RV_Image インスタンスが確保した画像情報の領域へのポインタを持つ．

提案手法を用いて，動画像のフレームを処理する場合，RV_TileImage インスタンスが部分領域と同じ数だけ生成される．各 RV_TileImage インスタンスがフレーム内の担当範囲を処理することで，フレーム全体を処理できる．ここで，RV_TileImage インスタンスの処理の担当範囲は開始座標，領域の幅と高さによって決まる．そして，適

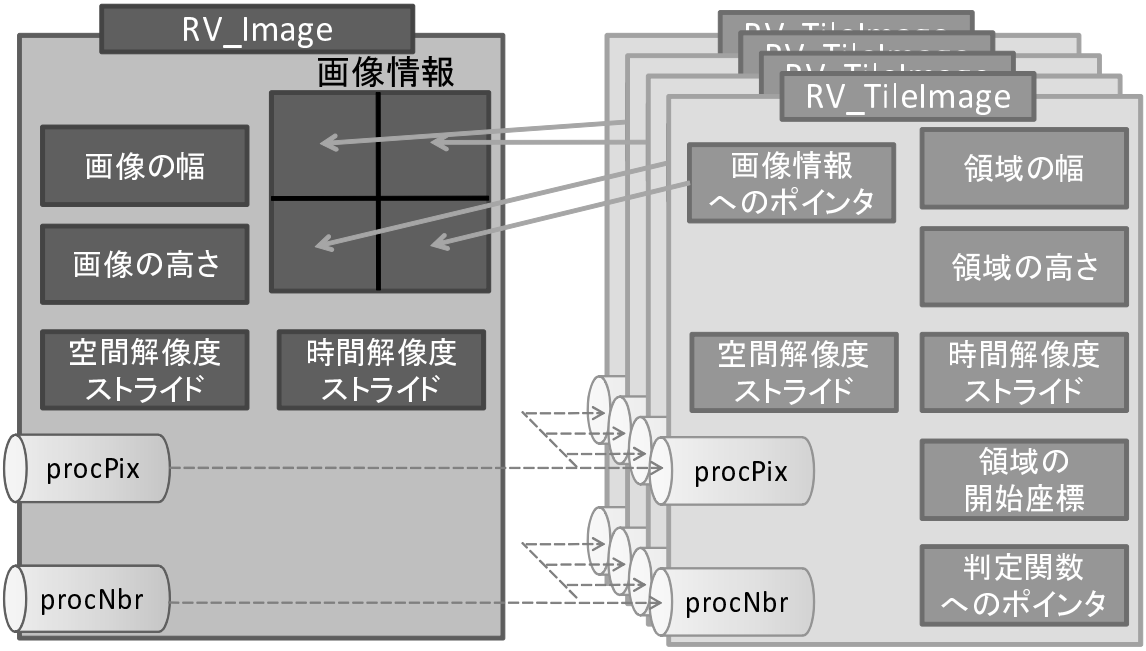

図 22: RV_TileImage クラスの概要

切な高階メソッドを使って，その担当範囲に構成要素関数を適用することで処理する．このようにすることで，RV_TileImage インスタンスは RV_Image インスタンスの持つ画像の対応する領域を処理することができる．

5.1.2 高階メソッドの拡張

本提案ではプログラマが既に記述した構成要素関数を変更しなくても領域別に処理精度を変動できるようにする．そのために，RV_Image インスタンスは，高階メソッドを使って動画像のフレームに対して構成要素関数を適用する際に，自身の高階メソッドを呼び出す代わりに RV_TileImage クラスの高階メソッドを呼び出す必要がある．そこで，RV_Image クラスの高階メソッドの動作を変更する．領域別に処理精度を変動させる際の高階メソッドの動作を**図 23** に示す．高階メソッド procPix() を通じて構成要素関数 GrayScale() を受け取った RV_Image インスタンスは，分割数に応じて RV_TileImage インスタンスを生成する．そして，その構成要素関数を各 RV_TileImage インスタンスの同名の高階メソッドに渡す．そして，前項で述べたように各 RV_TileImage インスタンスが各領域を処理することにより，フレーム全体が処理される．

さらに，複数のコアを備えた実行環境であるときには，処理が並列化されるようにする．そこで，各 RV_TileImage インスタンスの高階メソッドを呼び出す処理を並列実行できるようにする．今回，その並列化の実装には openMP[20] の parallel と for 指

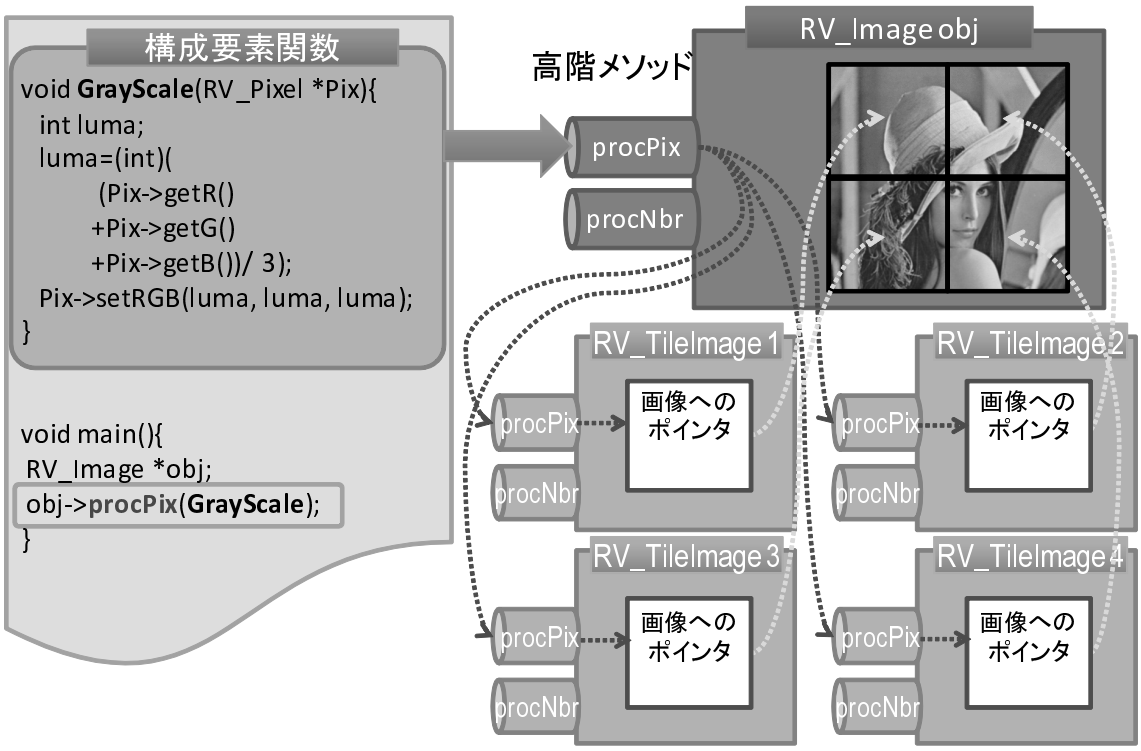

図 23: RV_Image クラスの高階メソッドの動作

```
1 void RV_Image::procPix(void(*CF)(RV_Pixel*)){
2   /*/ RV_TileImage インスタンス配列 Tile の生成 */
3 #pragma omp parallel for
4   for(int i=0; i<tile_num; i++){
5     Tile[i].procPix(CF);
6   }
7 }
```

図 24: openMP を用いた呼び出しの並列化

示子を用いる．これらの指示子は**図 24** の 3 行目のように，for 文の直前の行に記述する．これにより，その for 文の各イテレーションの処理は複数スレッドを用いて実行される．その結果，5 行目の RV_TileImage インスタンスの高階メソッド呼び出しは並列に実行可能になる．

また，高階メソッドの繰り返し処理の単位には 1 画素や近傍画素集合，テンプレート画像などがあるため，各高階メソッドごとに 1 つの RV_TileImage インスタンスが担当する処理範囲が異なる．そのため，メソッドごとに領域の分割方法を変える必要が

ある．ここで，代表的な高階メソッドの引数，処理内容，用途などを示し，各メソッドを実行する際の領域の分割方法について説明する．なお，以降の説明ではフレームを 4 つ（2×2）に分割する場合を想定する．

procPix(void(CF*)(RV_Pixel **P*))**

この高階メソッドは 1 画素 P を処理する構成要素関数 CF へのポインタを受け取り，その関数を全ての画素に対して繰り返し適用する．そのため，RV_TileImage を用いてフレームを領域別に処理する時，フレームは**図 25** に示すように 1，2，3，4 の領域に単純に分割される．ここで，図 25 中の C は開始座標，W は領域の幅，H は領域の高さを表しており，これ以降の他の高階メソッドの説明においても同じ意味で用いる．

procImgComp(void(CF*)(RV_Pixel **P*, $\boldsymbol{P_c}$), RV_Image $\boldsymbol{I_c}$)**

この高階メソッドは比較画像 I_c 内の画素 P_c を参照し，その画素と同じ位置の画素 P を処理する構成要素関数 CF へのポインタを受け取り，その関数を全ての画素に対して繰り返し適用する．なお，この高階メソッドはフレーム間差分の計算や 2 フレーム間の類似度の計算などに使用される．procImgComp() の繰り返し処理の単位は procPix() と同じなので，2 つのフレームとも図 25 に示すように分割される．

procNeighbor(void(CF*)(RV_Pixel **P*, **$\boldsymbol{P_{nbr}}$*, int *k*))**

この高階メソッドは 1 画素 P とその k 近傍画素集合 $P_{nbr}(k{=}8)$ に対する処理を記述した構成要素関数 CF を受け取り，その関数を全ての画素に繰り返し適用する．フレームは**図 26** に示すように，図 26 の 1 の領域は図 25 の 1 の領域の右端と下端の近傍画素を含むように分割される．同様に 2 の領域は左端と下端，3 の領域は右端と上端，4 の領域は左端と上端の近傍画素を含むように分割される．

procBox(void(CF*)(RV_DpImage $\boldsymbol{I_s}$,RV_Coord $\mathbf{C}_s$,$\mathbf{C}_e$),RV_Length *W*,*H*)**

この高階メソッドは幅 W，高さ H の部分画像 I_s に対する処理を記述した構成要素関数 CF を受け取り，その関数を画像全体に繰り返し適用する．C_s および C_e はその部分画像の始点座標および終点座標であり，画像全体に対する位置を示している．このメソッドは，テンプレートマッチング処理などに用いられる．フレームは**図 27** に示すように，図 27 の 1 の領域は図 25 の 1 の領域に対して右に W，下に H 大きな領域に分割される．同様に 2 の領域は下に H，3 の領域は右に W 大きな領域として分割される．

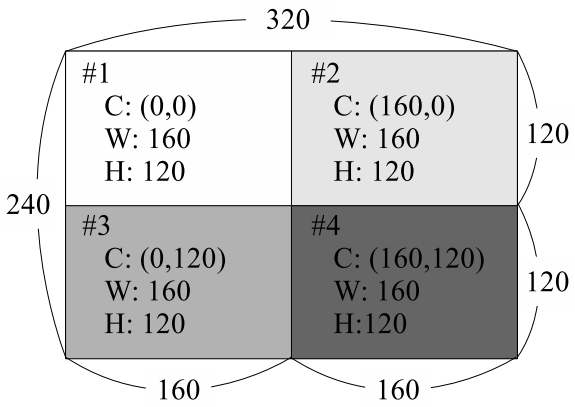

図 25: procPix，procImgComp の分割領域

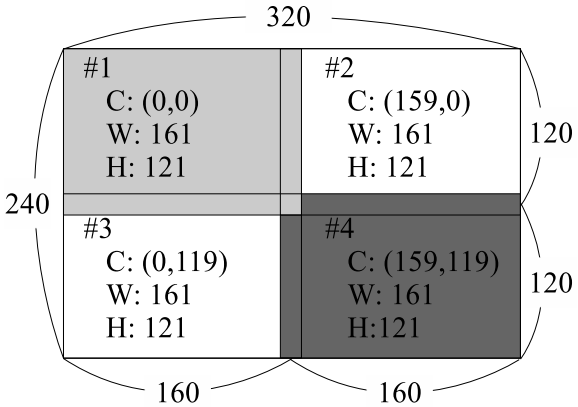

図 26: procNeighbor 分割領域

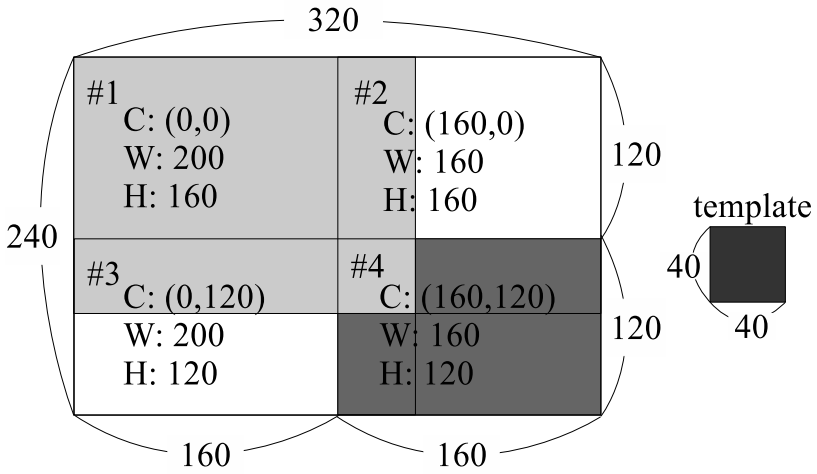

図 27: procBox の分割領域

5.2　領域を詳細に処理すべきかの判定

この節では，領域を詳細に処理すべきかを判定する関数について説明し，その判定結果の効果的な利用方法についても述べる．

5.2.1　判定関数

領域別に処理量を調整するために，各領域を詳細に処理すべきかどうかを判定しなければいけない．そこで，領域を詳細に処理すべきかどうかを判定する関数を RaVioli に導入する．本提案では，その判定関数に構成要素関数と同じ処理構造を用いる．プログラマは入力の 1 フレームまたは 2 フレームを用いて，領域を詳細に処理するかどうかを判定する関数を定義し，高階メソッドを使ってライブラリに渡す．また，使用頻度が高いと思われるいくつかの判定関数を予め定義し，これをユーザに提供する．これにより，プログラマが判定関数を記述しなくても領域別に処理量を調整できる．判

```
1  void pixF(RV_Pixel *pix){...}
2  void imgF(RV_Image *img){
3      img->procPix(pixF);              // 高階メソッド呼び出し
4  }
5  int FleshDetect(RV_Image *Curr, RV_Image *Prev){
6      /* 詳細に処理すべきかどうかを判定する */
7  }
8  int main(int argc, char* argv[]){
9      RV_Streaming video;
10     video.setPriority(7, 3);         // 優先度の設定
11     video.setCondFunc(FleshDetect);  // 判定関数の設定
12     // video.setCondFunc(FrameDiff); // 判定関数の選択
13     video.setTileNum(3, 4);          // 分割数の指定
14     video.procStream(imgF);          // 動画像処理用の高階メソッド
15 }
```

図 28: 提案手法を用いた RaVioli のプログラム記述例

定関数は，領域を詳細に処理すべきときは 1，そうでないときは 0 を返す関数とする．

ここで，提案方式を用いた RaVioli の動画像処理プログラムの記述例を**図 28** に示し，それを用いて判定関数の記述や設定の方法を説明する．プログラマは main 関数と構成要素関数（pixF，imgF），判定関数（FleshDetect）を記述する．main 関数では，まず動画像を管理する RV_Streaming クラスのインスタンス `video` を生成し（9 行目），そのインスタンスの持つメソッド setPriority を用いて優先度を設定する（10 行目）．設定された優先度に応じて，`video` は時間解像度と空間解像度を調整する．次に，プログラマは判定関数を高階メソッド setCondFunc に渡す．ここで，その高階メソッド setCondFunc の仕様について述べる．

void setCondFunc(int(CdF*)(RV_Image **F*$_c$))**

void setCondFunc(int(CdF2*)(RV_Image **F*$_c$,*F*$_p$))**

このメソッドは，現在の処理フレーム F_c を用いる判定関数 *CdF* へのポインタ，または F_c と 1 つ前の処理フレーム F_p を用いる判定関数 *CdF2* へのポインタを引数として受け取る．そして，プログラマが記述した判定関数へのポインタを図 22 に示した RV_TileImage の持つ判定関数用のポインタ変数に設定する．

この例の場合，プログラマは記述した判定関数 FleshDetect を setCondFunc に渡している（11 行目）．また，プログラマはライブラリに予め定義されている関数を指定

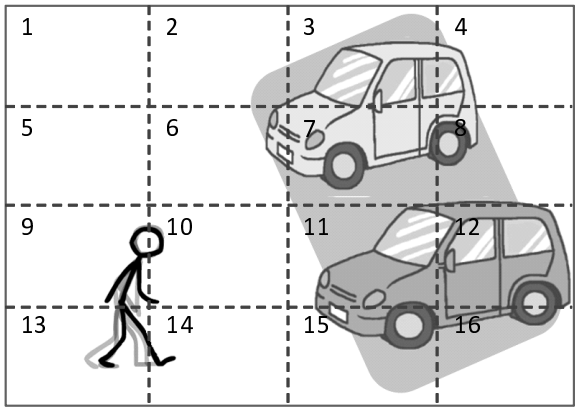

図 29: 動物体が領域を跨いでいる入力

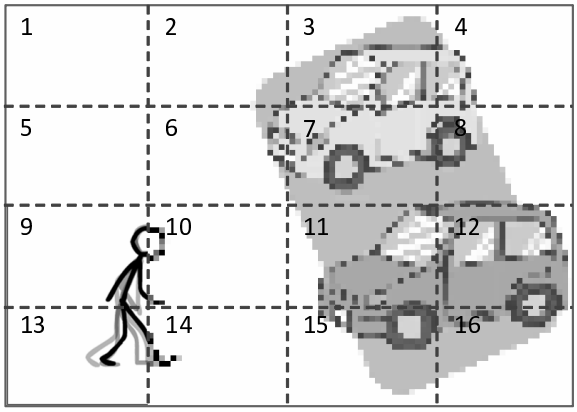

図 30: その入力の出力結果

することも可能である（12 行目）．なお，FrameDiff は 2 フレームの差分を取り，変化がある領域を詳細に処理する必要があると判定する関数である．そして，動画像ストリームを何分割するかを指定し（13 行目），動画像処理の高階メソッドに 1 フレームを処理する構成要素関数 imgF を渡す（14 行目）．構成要素関数 imgF 内では， 5.1.2 項で説明したように変更した RV_Image クラスの高階メソッド procPix に構成要素関数 pixF を渡す．procPix() 内では，各 RV_TileImage インスタンスの高階メソッドが呼び出され，各領域ごとに詳細に処理すべきかどうかを判定関数を用いて判定する．そして，その結果に応じて各ストライドを増減させて，そのストライドに基づいて領域を処理する（3 行目）．このように記述することで，動画像中のすべてのフレームに対して処理を施すことが可能である．

5.2.2 隣接領域の判定結果の利用

前項で述べた通り，提案手法は判定関数を各領域毎に適用し，その領域を詳細に処理すべきかどうかを決定する．そのため，判定する際に考慮される情報はその領域が持つ情報だけである．しかし，その領域を詳細に処理すべきかどうかを判定する際に，他の領域が持つ情報を考慮しなければ正確に判定できない場合がある．例えば，フレーム内の動物体が存在する領域を詳細に処理したい場合，前フレームとの差分を判定に用いることが考えられる．これは，領域内の差分が大きいとき，その領域に動物体が含まれる可能性が高いと考えられるからである．しかし，**図 29** のように動物体が部分領域を跨いでいる場合，差分が少ない領域内にも動物体の一部が含まれることがある．これらの領域は詳細に処理すべき領域であるが，判定関数により詳細に処理する必要がないと判定されてしまう．その結果，**図 30** のように詳細に処理すべき動物体が含まれる領域（領域 10，14）を詳細に処理できていない出力が得られる．この場合，隣接

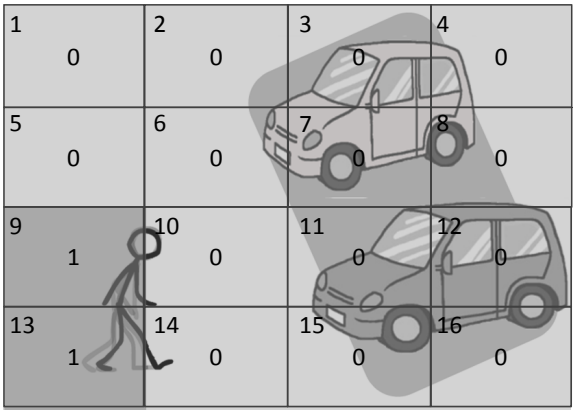

図 31: 図 29 の入力に対する判定結果

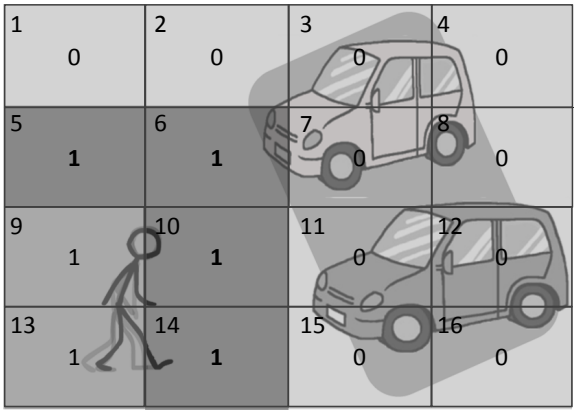

図 32: 隣接領域の判定結果を反映

する領域の差分を考慮すれば，その領域を詳細に処理すべきと判定できる．このように，他の領域の持つ情報を考慮することで，各領域の判定精度を向上できる．

しかし，各領域の判定時にどの領域の情報を考慮すべきなのかを決めるのは困難である．そのため，考慮する領域の範囲を大きくすることが考えられるが，考慮する範囲が大きくなると，処理量が増大する．これは，判定関数のそもそもの目的である，処理量の削減に反してしまうため，無意味である．そこで，本提案手法では，各領域の判定処理時に考慮する領域の範囲は変えずに，判定結果のみを利用して，判定精度の向上を図る．具体的には，全ての領域に判定関数を適用後，フレーム全体の情報を持つ RV_Image インスタンスが各領域とその隣接領域の判定結果によって，その領域の判定結果を変更する．各領域の判定結果を保持するために，RV_Image クラスに各領域数分の要素を持つ配列を追加する．

ここで，判定結果の変更動作を先ほどの図 29 の入力を例に説明する．まず，図 29 の入力に判定関数を適用した結果を，**図 31** に示す．図中の各領域内の中心には判定結果（判定関数の返り値）を示している．この例では，先ほど述べた通り，領域 9，13 が詳細に処理すべきと判定されている．この時，RV_Image インスタンスは詳細に処理する必要がないと判定されている各領域に対して，その領域の隣接領域に詳細に処理すべき領域がいくつ存在するかを計算する．その結果を用いて，判定結果を変更する．例えば，隣接 8 近傍領域の内ひとつでも詳細に処理すべき領域が含まれている場合に，その領域の判定結果を変更すると仮定する．その場合，**図 32** に示すように，領域 5，6，10，14 の判定結果が変更され，重要な領域をラフストライドで処理してしまう場合を減らすことができる．

提案手法では，判定結果を変更する条件に隣接領域の判定結果を用いるため，いく

つの隣接領域が詳細に処理すべきと判定されている時に判定結果を変更するかや，どの範囲の隣接領域の判定結果を考慮するかを決めなければいけない．そこで，本提案手法では，判定結果を変更するこれらの条件をプログラマが指定できるように，動画像を管理する RV_Streaming クラスにメソッド setNbrCondition を追加する．以下にそのメソッドの仕様を示す．

void setNbrCondition(int condtype, int threshold)

このメソッドは，近傍領域の範囲を決めるための `condtype`，いくつの領域が詳細に処理すべきと判定されている時に結果を変更するかを示す `threshold` を引数にとる．現時点では，以下の 4 つの範囲を `condtype` で指定することができる．

- `condtype` = 0 のとき，各領域の上下の領域を範囲指定．
- `condtype` = 1 のとき，各領域の左右の領域を範囲指定．
- `condtype` = 2 のとき，各領域の 4 近傍領域を範囲指定．
- `condtype` = 3 のとき，各領域の 8 近傍領域を範囲指定．

プログラマはこれらから近傍領域の範囲を選択し，変更の閾値とともにこのメソッドに渡す．これにより，プログラマが判定結果を変更する条件を指定することを可能にする．

また，本提案手法では，このように条件を指定しなくても，プログラマが隣接領域の判定結果を利用できるようにする．そこで，条件が指定されていないときには，各領域の 4 近傍領域のうち詳細に処理すべき領域が 2 つ以上ある場合に，判定結果を変更する条件を用いる．これにより，プログラマは条件の指定を省略可能になる．

5.3 領域別のストライド変動

5.1 節で述べたように，分割領域を管理する RV_TileImage クラスを追加し，RV_Image クラスの高階メソッドを拡張することで，分割領域毎に処理することを可能にした．また， 5.2 節で述べたように，判定関数を導入し，詳細に処理すべきかどうか決めることを可能にした．この節では，これらの実装を用いて，領域別に解像度ストライドを変動させて処理する方法を示す．

まず，空間解像度ストライドを領域別に変動させて処理する方法を示す．判定関数の結果に基づいて，図 22 に示した各 RV_TileImage インスタンスが持つ空間解像度ストライドにベースストライドまたはラフストライドを設定する． 4.3 節で述べた通り，これらのストライドは RV_Image インスタンスが持つフレーム全体に設定されている空間解像度ストライドによって決まり，ベースストライドにはその値を代入し，ラフスト

ライドにはベースストライドよりも大きな値を設定する．このように各RV_TileImageインスタンスの持つ空間解像度ストライドを設定し， 5.1 節で述べたように，各インスタンスの高階メソッドを呼び出すことで，担当範囲を自身が持つ空間解像度ストライドで処理する．このとき，ラフストライドで処理された領域には，本来処理されるはずであった画素が未処理のまま残る．この未処理の画素に対して，本提案手法では，ファイルなどに出力する時に処理された画素を補完することで解決する．具体的には，未処理の画素から見て，左，上，左上のいずれかの方向に存在する最も近い，処理済みの画素を補完に用いる．これは従来の RaVioli が空間解像度ストライドを大きくした際に，未処理の画素を処理済の画素を用いて補完する方法と同じである．以上のようにして，空間解像度を領域別に変動させて，その領域を処理することができる．

一方，時間解像度ストライドを領域別に変動させて処理する方法を示す．空間解像度ストライドと同様に，判定関数の結果に基づいて，RV_TileImage インスタンスが持つ時間解像度ストライドにどちらかのストライドを設定する．時間ストライドを領域別に変動させて処理する場合，ラフストライドを設定された領域は処理自体を飛ばすことになる．しかし，そのフレームは処理フレームであるため，ラフストライドの領域も出力されてしまう．そこで，前の処理フレームの結果をそのまま現在の処理フレームに出力できるように，RV_TileImage クラスを拡張する．また，各領域により詳細に処理する必要がないと判定されるタイミングが違うため，同じラフストライド領域だとしても，あと何フレーム処理を飛ばせばいいのかは領域毎に異なる．そこで，RV_TileImage クラスを拡張し，その値を保持，管理できるようにする．これらの拡張について**図 33** に示す．拡張後の RV_TileImage クラスのインスタンスは前の処理フレームへのポインタと処理を飛ばした数を数えるカウンタをメンバとして持つ．そのカウンタは初期値として時間解像度ストライド値を保持しており，詳細に処理する必要がないと判定された時に 0 を代入する．領域の処理を飛ばす場合 RV_TileImage は高階メソッド内の処理を実行する代わりに，前の処理フレームの処理結果を現在の処理フレームにコピーし，そのカウンタをインクリメントする．カウンタの値と自身が保持している時間解像度ストライドの値が一致している時，次のフレームを処理する．

以上のように空間解像度ストライドと時間解像度ストライドを領域別に変動させ，処理する．これにより，提案手法は詳細に処理すべき領域に対する処理精度の低下を抑制する．

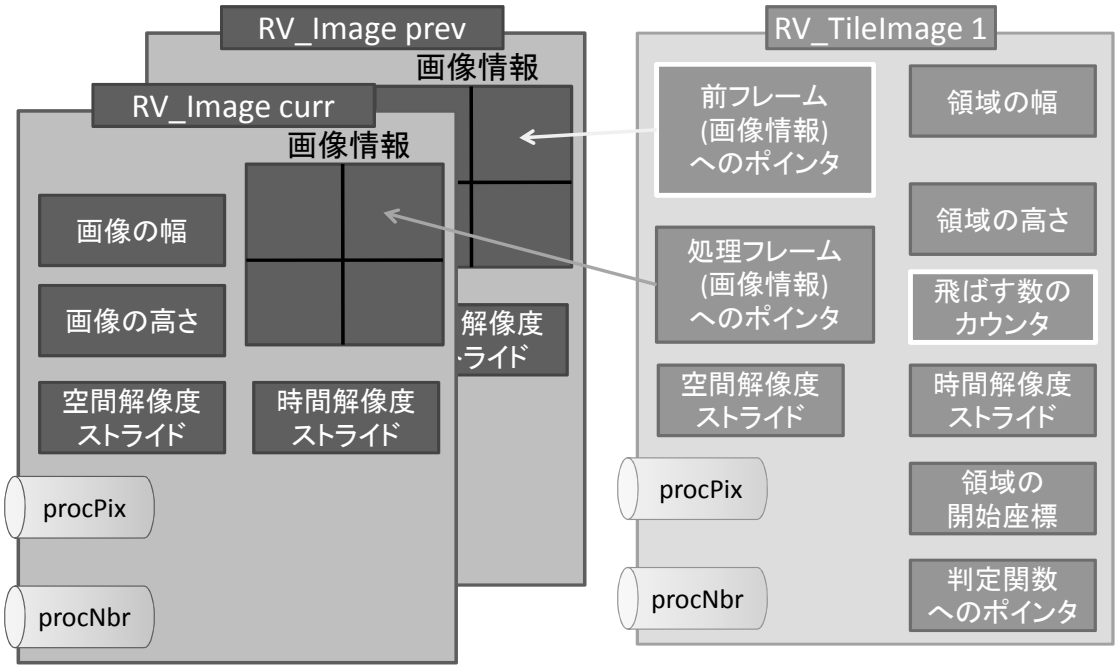

図 33: RV_TileImage クラスの拡張

6 実装上の問題とその対応

本章では，領域別に処理精度を変動させる提案手法を RaVioli に追加する際に発生する問題点について触れ，それらの具体的な解決策について述べる．

6.1 プリプロセッサの拡張

この節では，一つ目の問題点について説明し，その問題点を解決するためのプリプロセッサの拡張について説明する．

6.1.1 解決すべき問題

提案手法を用いる場合，1 フレームを領域別に複数の空間解像度ストライドを用いて処理する．RaVioli はプログラマから処理画素数を隠しているため，基本的に各領域を任意のストライドで処理することが可能である．しかし， RaVioli を用いて正しく記述した処理でも，複数の空間解像度ストライドを用いることで，正しい結果を得られないことがある．

例えば，入力画像の部分画像とテンプレート画像との類似度を計算し，その値が最小となるような位置を求めるテンプレートマッチング処理はこの問題が発生するプログラムの一例である．そこで，このテンプレートマッチング処理に対して，単一の空

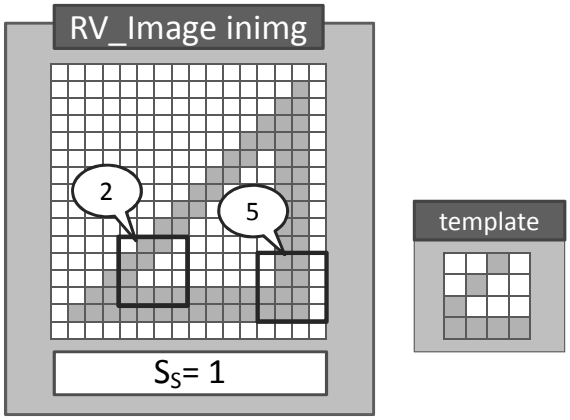

図 34: 従来の RaVioli の処理

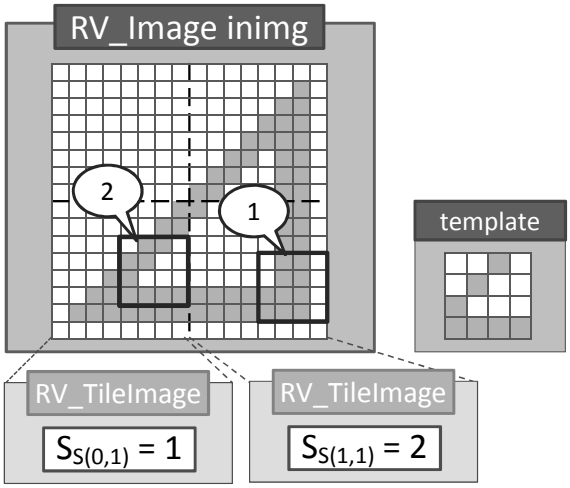

図 35: 提案手法の処理

間解像度ストライドを用いる従来の RaVioli の処理と，複数のストライドを用いる提案手法の処理を示し，正しい結果が得られない状況を示す．まず，従来の RaVioli を用いた処理の様子を**図 34** に示す．図 34 の左側の `inimg` は入力画像，右側の `template` はテンプレート画像を表している．この入力画像に空間解像度 $S_S = 1$ が設定されている場合に，`inimg` 内の 2 箇所の類似度はそれぞれ 2 と 5 となる．この類似度が 2 となる領域が最も類似度が小さくなる箇所である．

一方で，提案手法を用いた処理の様子を**図 35** に示す．入力画像 `inimg` を 2×2 に分割し，左下の領域を $S_{S(0,1)} = 1$，右下の領域を $S_{S(1,1)} = 2$ で処理すると仮定する．先ほどの例と同じ箇所の類似度はそれぞれ 2 と 1 となり，全ての領域を同じ空間ストライドで処理した際に最小でなかった，右下の部分画像とテンプレート画像との類似度が，最小であるかのように処理されてしまう．このように，各領域を複数のストライドで処理することで，正しい結果が得られなくなる場合が存在する．この例の場合，類似度の値に各領域のストライド値の 2 乗を掛けることで，処理画素数の差をある程度埋めることができる．そこで本研究では，この問題を解決するために，領域のストライド値の差が結果に現れる処理を検出して，その差が結果に現れないように処理を変換することで解決する．そのために， 3.2 節で述べたプリプロセッサを拡張する．

6.1.2 領域間のストライド値の差が結果に現れる処理の検出と変換

前項で述べた問題を解決するために，領域間のストライド値の差が結果に現れる処理を検出できるようにプリプロセッサを拡張する．そこで，先ほどのテンプレートマッチング処理を RaVioli を用いて記述したプログラムを示し，領域間のストライド値の

```
1  RV_Image* tpImg;
2  int min=INT_MAX;
3  int sum=0;
4  RV_Coord coord;
5
6  void Count(RV_Pixel *p1,RV_Pixel p2){
7    sum+=abs(p1-p2);
8  }
9
10 void Comp(RV_DpImage* dImg, RV_Coord Cstart){
11   dImg->procImgComp(Count, tpImg); // 高階メソッド呼び出し
12   if(min > sum){
13     min=sum;
14     coord=Cstart;
15   }
16   sum=0;
17 }
18
19 int main(){
20   RV_Image* Img;
21   //画像ファイル読み出し
22   Img->procBox(Comp, tpImg->size);//高階メソッド呼び出し
23 }
```

図 36: RaVioli のテンプレートマッチングプログラム

差が結果に現れる処理の特徴を挙げ，検出条件について説明する．

RaVioli を使用してテンプレートマッチングを記述する場合， **図 36** に示すように 2 つの構成要素関数を定義する（6∼8，10∼17 行目）．構成要素関数 Comp（10∼17 行目）はテンプレート画像と同じ大きさの領域 dImg を処理対象とし，procBox() の引数として渡すことで画像全体に処理を適用することができる（22 行目）．ここで，dImg の各画素に対する処理を記述するには， 1 画素に対する処理を記述した構成要素関数 Count（6∼8 行目）を定義して， RV_DpImage インスタンスの高階メソッドに渡す必要がある（11 行目）．この RV_DpImage は，処理対象画像の部分画像を表すクラスである．このクラスは RV_Image クラスを継承しており，RV_Image クラスと同様の高階メソッドを利用することができる．さらに，処理を開始する位置の座標をクラス独自のメンバ変数として持っている．この変数を用いて処理する範囲をずらすことで，構

成要素関数内に記述された処理を画像の全領域に対して適用することができる．なお，この開始位置を示す変数はライブラリ内で制御するため，プログラマは部分画像中の1画素に対する処理を記述するのみでよい．このように，RaVioliを用いてテンプレートマッチングプログラムを記述する場合，プログラマは2つの構成要素関数を定義する必要がある．

ここで，構成要素関数Comp内で使用されている大域変数sumは，構成要素関数Count内で計算される画像間の類似度を保持している．そして，構成要素関数Comp内でsumの最小値を求めることで，類似度が最小となる位置を求めている．このとき，sumの値は，構成要素関数Countが呼び出される回数，つまり空間解像度ストライド値に依存している．従来のRaVioliでは，画像全体を同じ空間解像度ストライドで処理するため，このストライド値に依存する値を処理に使用しても，正しい処理結果が得られた．しかし，提案手法のように，領域別に複数のストライドを使用する場合，ストライドに依存した値を処理に使用することで，正しい結果を得られなくなる．

そこで，プリプロセッサは現段階では以下の条件を用いて，領域間のストライドの差が結果に現れる処理を検出する．

条件 (1) 別の構成要素関数内で読み書きされた大域変数が，構成要素関数内で他の大域変数へ代入されている

構成要素関数内で読み出しおよび書き込まれた大域変数は構成要素関数の呼び出し回数，つまり空間解像度ストライドに依存する．

条件 (2) 構成要素関数内に高階メソッド呼び出しが存在する

構成要素関数は高階メソッドによって処理単位に適用されるため，高階メソッド呼び出しを検出することで，構成要素関数内の別の構成要素関数の呼び出しを検出できる．

以上の条件を共に満たす場合，プリプロセッサは，構成要素関数内の処理で使われる大域変数が領域のストライドに依存していることをプログラマに通知する．通知を受けたプログラマは，全ての領域を同じストライドで処理するように指定するか，ストライド値の差が結果に現れないようにプログラムを変換する．そこで，このプログラムの変換例を**図37**に示す．この図はテンプレートマッチングのプログラムの変換部分を示している．構成要素関数Comp内の高階メソッドprocImgCompの呼び出し（4行目）後，その高階メソッドによって計算されたsumの値をストライド値に依存しない値に変換している（5行目）．この例では，空間解像度ストライドを返すgetGrain()メソッドを用いて，その2乗の値をsumに掛け合わせている．この処理により，スト

```
1 int sum=0;
2 ...
3 void Comp(RV_DpImage* dImg, RV_Coord Cstart){
4   dImg->procImgComp(Count, tpImg); // 高階メソッド呼び出し
5   sum *= pow(dImg->getGrain(),2);
6   ...
7 }
```

図 37: ストライド値の差が結果に現れない処理例

ライド値の違いによる処理画素数の差をある程度埋めることできる.

6.2 並列化時の処理の割り当てスケジューリングの拡張

この節では，二つ目の問題点として，並列化時の処理の割り当てについて触れ，領域別に処理精度を変動させる本提案手法に適した，処理の割り当て方法を示す.

6.2.1 一般的な割り当て方法を提案モデルに適用した際の問題

前章までに述べたように，本提案手法は，動画像ストリームを部分ストリームに分割し，領域別に処理量を調整可能にすることで，詳細に処理すべき領域に対する処理精度の低下を抑制する．さらに，この各領域の処理を並列に実行することで，処理時間を短縮し，処理精度低下の更なる抑制を図る．このように処理を並列化して処理時間の短縮を図る場合，各スレッド間の処理負荷を均衡化することが重要となる．そこで，領域別に処理量を調整する手法を並列化と組み合わせる際に，処理の割り当て手法によってどのような問題が発生するかについて考える.

3.2 節で述べた従来の RaVioli が提供している空間分割並列化では，画像を均等な大きさに分割して，その部分画像の処理を各スレッドに静的に割り当てる．この単純な静的割り当て手法は，予め処理範囲や処理量を見積もりやすい画像処理の並列化として一般的である．そのため，提案手法を並列化する際にも，部分領域の処理を各スレッドに均等に割り当てることが考えられる．しかし，提案手法では，領域を詳細に処理すべきかどうかによって，各領域に設定されるストライド値が異なるため，同じ大きさの領域でも処理量が異なる場合がある．例えば，**図 38** のように入力フレーム内の領域 6，9，10，13，14 にベースストライドが，その他の領域にはラフストライドが設定されている場合に，各領域の処理を均等に各スレッド（4 スレッド）に割り当てる様子を**図 39** に示す．図 39 の各バーの長さはその領域を処理するためにかかる処理量

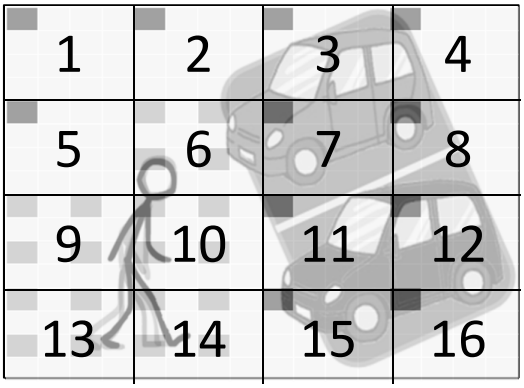

図 38: 入力フレーム内の各領域の空間ストライド

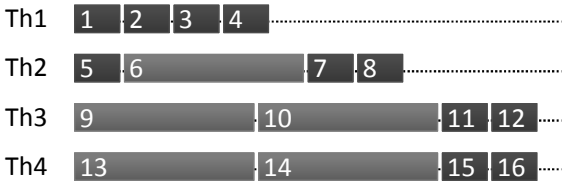

図 39: 静的な割り当て

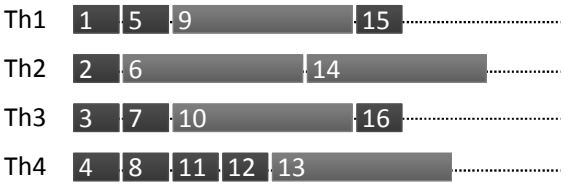

図 40: 動的な割り当て

を示している．図 39 に示した通り，領域の処理を均等にスレッドに割り当てると，スレッド間の処理負荷に偏りが生じてしまい，並列化による得られる効果が小さくなる．

一方で，並列処理の実行中に各スレッドに処理を割り当てていく動的な手法も広く用いられている．これは，各スレッドに割り当てられる処理にかかる負荷が一定でなく，なおかつその負荷を実行前に判断できない場合に適した手法である．ここで，先ほどの図 38 の例の場合に，動的に処理を割り当てる様子を**図 40** に示す．

動的な割り当て手法では一般的に，まず各スレッドに一定の処理を割り当て，残りの処理は実行を終えたスレッドに順次割り当てる．この例では，各領域の処理を 1 領域ずつ各スレッドに割り当てていく．そのため，始めに領域 1，2，3，4 がスレッド 1，2，3，4 に割り当てられ，その処理が終了すると同様に領域 5，6，7，8 が各スレッドに割り当てられる．ここで，領域 6 は処理量が多く実行中であるとすると，スレッド 2 を除いたスレッド 1，3，4 に処理が割り当てられる．以降同様に，処理を終えたスレッドに割り当てていくと，図 40 に示した通り，各スレッドの処理負荷をある程度均衡化でき，図 39 の静的な割り当て手法に対して，フレーム全体を処理するのにかかる時間を短縮できる．しかし，並列処理時に動的に各スレッドに処理を割り当てるためのコストは小さくなく，各スレッド間の負荷の均衡化も最適ではない．そこで，本研究では，領域別に処理精度を変動させる手法に適した，静的な割り当て手法を提案し，

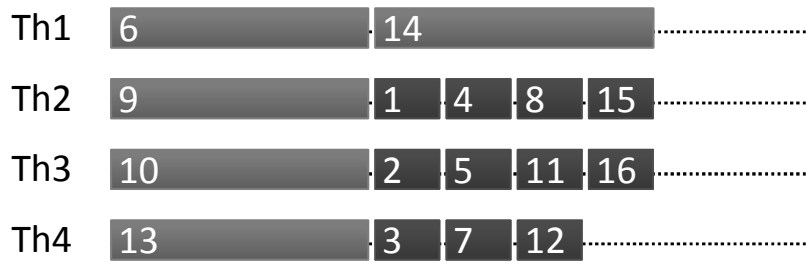

図 41: ストライドを考慮する割り当て

RaVioli に実装する．

6.2.2 領域の空間ストライドを考慮した処理の割り当て

提案手法では，解像度ストライドにベースストライドとラフストライドの 2 種類の値を設定する．そのため，この 2 つの値から詳細に処理する領域と詳細に処理しない領域の処理量の比を容易に求めることができる．例えば，ラフストライドがベースストライドの 2 倍であるとき，ラフストライドが設定されている領域（以降，ラフストライド領域）にかかる処理量はベースストライドが設定されている領域（以降，ベーススストライド領域）のおよそ 1/4 である．この処理量比を考慮して各スレッドに処理を割り当てるように， 5.1.2 項で述べた RV_Image クラスの高階メソッドをさらに拡張する．

そこで，まず具体的な処理の割り当ての例を**図 41** に示し，処理量比を考慮する割り当て手法の動作について説明する．この割り当て手法では，まず処理量の多いベーストライド領域の処理を各スレッドに割り当てる．そして，ベーストライド領域の処理の割り当て数に各スレッド間で差がある場合，割り当ての少ないスレッドに処理量比に基づいて優先的にラフストライド領域の処理を割り当てる．この例の場合，スレッド 1 に対して，スレッド 2，3，4 はベーストライド領域の処理の割り当て数が少ない．そこで，処理量比に基づいて最大 4 つのラフストライド領域を各スレッドに割り当てる．これにより，図 41 に示した通り，各スレッド間の処理負荷を均衡化できる．

このストライド値を考慮した割り当て手法を実現するため，RV_Image クラスの高階メソッド内部の RV_TileImage クラスの高階メソッド呼び出し部分をさらに拡張する．まず，各領域の処理をどのスレッドに割り当てるかを保持するための配列を用意する．この配列のサイズは，分割領域の数と，処理量比にスレッド数を掛けた値の和に等しい．図 38 の例の場合，分割領域の数 16，処理量比 4，スレッド数 4 なので，32 要素を持つ配列が生成される．次に，この配列の使用例を**図 42** に示す．図 42 はスレッ

Th1	6	14	0	0	0	0	0	0
Th2	9	0	0	0	0	0	0	0
Th3	10	0	0	0	0	0	0	0
Th4	13	0	0	0	0	0	0	0

Th1	6	14	0	0	0	0	0	0
Th2	9	1	4	8	15	0	0	0
Th3	10	2	5	11	16	0	0	0
Th4	13	3	7	12	0	0	0	0

図 42: 各スレッドに対する処理の割り当てを保持する配列の使用例

```
1  void RV_Image::procPix(void(*CF)(RV_Pixel*)){
2    /* RV_TileImage インスタンス配列 Tile の生成 */
3    /* 割り当てを示した配列 Array の生成          */
4  #pragma omp parallel for schedule(static,N)
5    for(int j=0; j<N*thread_num; j++){
6      int i = Array[j];
7      if(j>=0&&j<tile_num)
8        Tile[i].procPix(CF);
9    }
10 }
```

図 43: RV_TileImage クラスの高階メソッド呼び出し部分のコード

ド 1 への割り当てを保持する要素を 1 行目に示し，同様にスレッド 2，スレッド 3，スレッド 4 への割り当てを保持する要素を 2，3，4 行目に示している．この配列に対してまず始めに，ラフストライド領域の番号を先頭から順に書き込んでいく（図 42 の左側）．そして，処理量比に応じて，ラフストライド領域の番号を配列に書き込む（図 42 の右側）．この例では，スレッド 1 に対してラフストライド領域の処理を割り当てずに，他のスレッドに最大 4 つの領域の処理を割り当てる．このように値が設定された配列を用いて，図 41 の割り当てを実現する，RV_TileImage クラスの高階メソッド呼び出し部分のコードを**図 43** に示す．まず，複数スレッドで並列処理される for ループの処理（5~9 行目）について説明する．5 行目の変数 `N` は割り当てを管理する配列の 1 スレッド辺りの要素数，変数 `thread_num` は実行スレッド数を保持している．つまり，このループは割り当てを保持する配列の要素数と同じ回数の繰り返し処理である．そして，ループのボディ部では，配列から値を取り出して（6 行目），その値が領域の番号であるかを判定し（7 行目），領域の番号である場合は，対応する RV_TileImage インスタンスの高階メソッドを呼び出している（8 行目）．そうでない場合は，呼び出

表 1: 評価環境

OS	Fedora15
CPU	Core2Quad
Frequency	2.83GHz
Memory	3GB
Compiler	gcc 4.6.1
Compile options	-O3，-fopenmp
OpenMP version	3.1

しをしない．次に，この for ループの並列実行を指示している（4 行目）．4 行目には，for 指示子の後に schedule と記述されている．これは openMP の指定節の一つであり，各スレッドへの処理の割り当て方法を指定する．この例では，N 個の要素を静的にスレッドに割り当てるように指定している．こうすることで，各スレッドは自身への処理の割り当てが格納されている部分を配列から読み出すことができ，ストライドを考慮した処理の割り当て手法が実現できる．

7 評価

提案手法を追加した RaVioli と従来の処理量調整手法を用いる RaVioli で，動画像処理時の解像度ストライドの変動と出力画像を比較した．また，スレッドへの処理の割り当て手法も画像処理プログラムを用いて評価した．

7.1 評価環境

評価環境を**表 1** に示す．CPU には 4 コア構成である Core2Quad を用い，並列プログラムの並列数は 4 とした．また，コンパイラオプションには最適化オプションの`-O3`の他，openMP を使用するために`-fopenmp`を指定した．

処理量調整手法の評価には，サンプルプログラムとして，テンプレートマッチングプログラムを使用した．入力には解像度が 320×240 画素の 100 フレームで構成される動画像を使用した．また，入力動画像の 30 から 60 フレームの間，処理負荷の高い別のアプリケーションを実行することで，利用可能な CPU リソース量の減少を再現し，処理量調整が必要な状況が発生するようにした．なお，領域の分割数は 16×16 とし，判定関数には 5.2.1 項で述べた 2 フレームの差分を判定基準にする FrameDiff を用い，

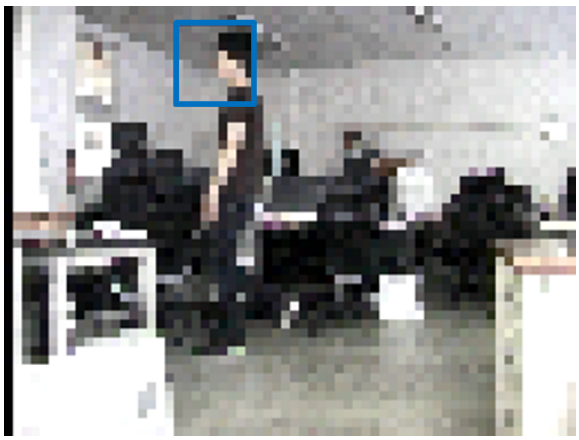

図 44: 従来の RaVioli の出力

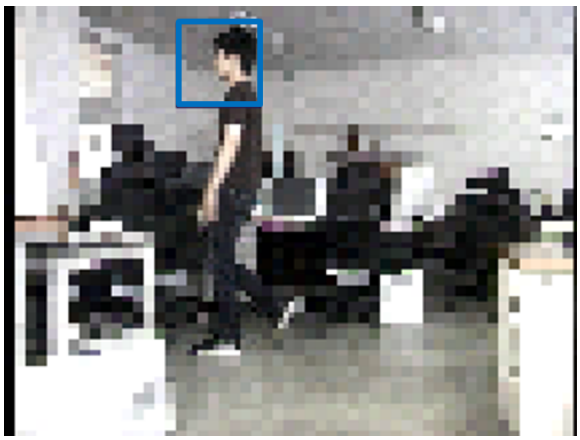

図 45: 提案手法の出力

優先度セットは $(P_S, P_T) = (1, 1)$ とした．

また，スレッドへの処理の割り当て手法の評価にはグレースケール化，エンボスフィルタ，テンプレートマッチングのプログラムを使用した．入力画像には 320×300 画素の画像を使用した．また，テンプレートマッチングではテンプレート画像として 103×145 画素の画像を使用した．また，入力フレーム内の各領域のストライド値には図 38 と同様に，5 つの領域にベースストライド，11 の領域にラフストライドを設定した．これらの入力に対して処理を施し，実行時間を比較した．比較対象は 6.2.1 項に示した，均等な処理範囲をスレッドに割り当てる単純な静的割り当て手法と，動的な割り当て手法とした．これらの割り当て手法は OpenMP の schedule 指定節により実現した．

7.2 領域別処理量調整手法のみの評価

領域別に処理量を調整することによって，重要な領域に対する処理精度低下の抑制を確認するために，並列化を用いない RaVioli と提案手法を追加した RaVioli を評価した．**図 44**，**図 45** に最も負荷の高かった時刻におけるそれぞれの出力画像を示す．各図に出力されている枠線はテンプレートマッチング処理の結果を示している．従来の RaVioli が画像全体を同じストライドで処理し，動物体が存在する重要な領域に対する処理精度が低減してしまっているのに対して，提案手法では前フレームからの変化が大きい領域を詳細に処理できていることが確認できる．

次に，時間経過に伴う空間ベーストライドの変化を**図 46** に，時間ベーストライドの変化を**図 47** に示す． グラフの横軸は何番目に処理されたかを表すフレーム番号，縦軸はベーストライド値である．提案手法では両ベーストライドを，従来の

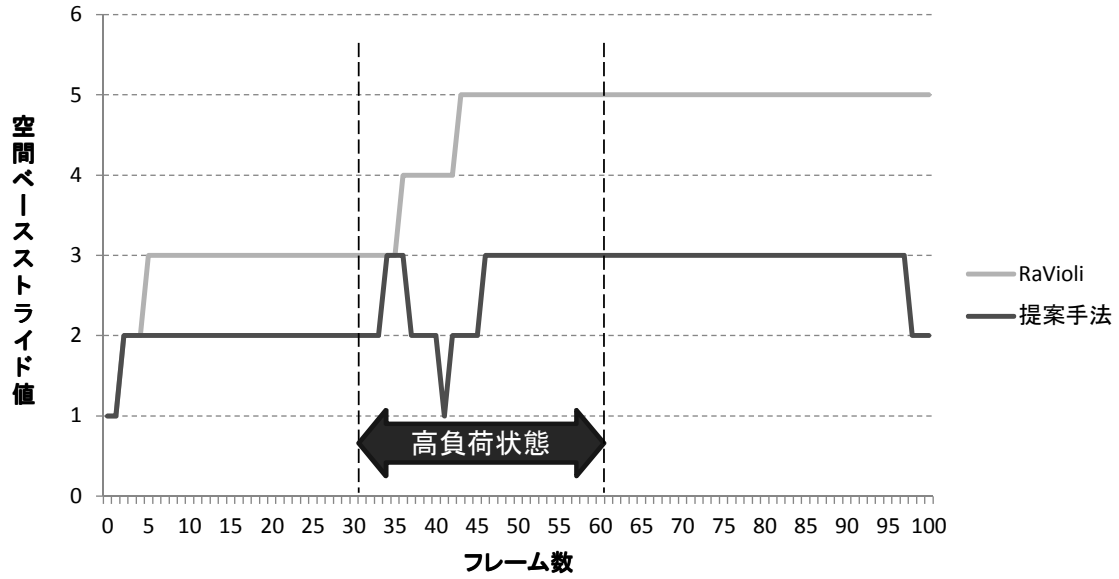

図 46: 空間ベーススストライド値の変動 (逐次プログラム)

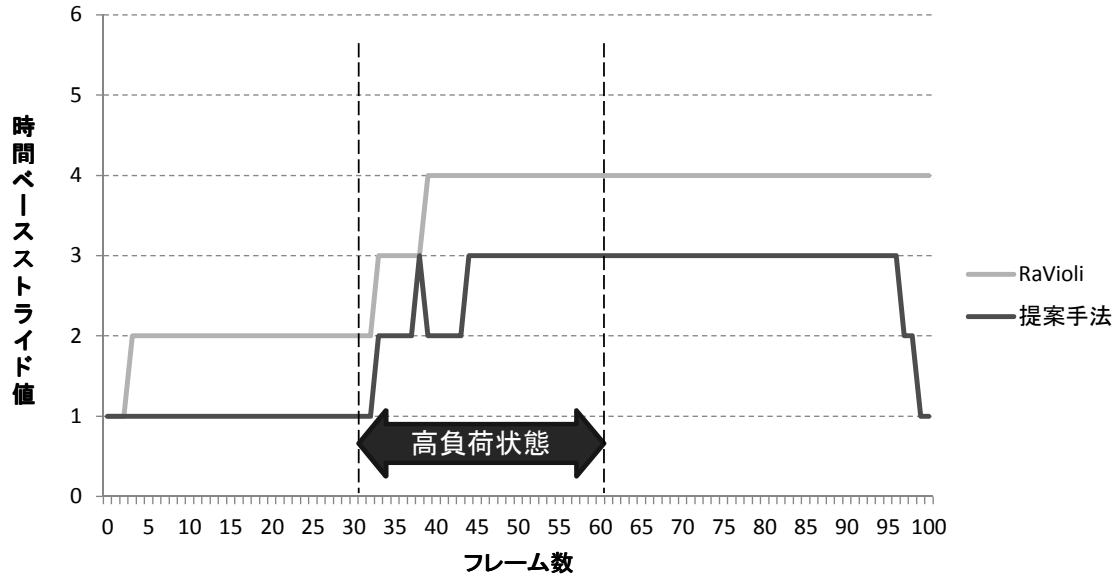

図 47: 時間ベーススストライド値の変動 (逐次プログラム)

RaVioli の両解像度ストライド値に対して低く保つことができている．具体的には，空間ベーススストライドの平均は 1.63 低く，時間ベーススストライドの平均は 1.07 低く保つことができており，提案手法によって重要な領域に対する処理精度の低下を抑制することを確認できた．

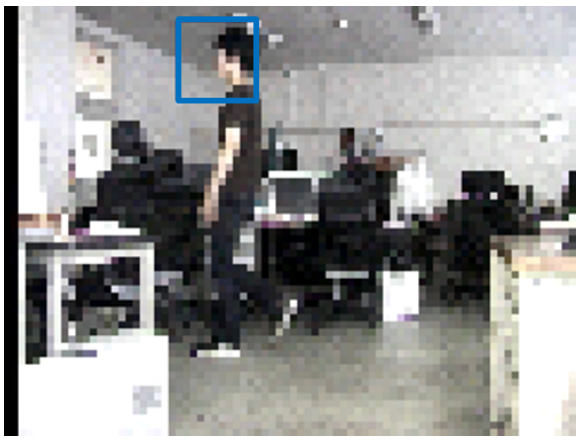

図 48: “RaVioli+並列化” の出力

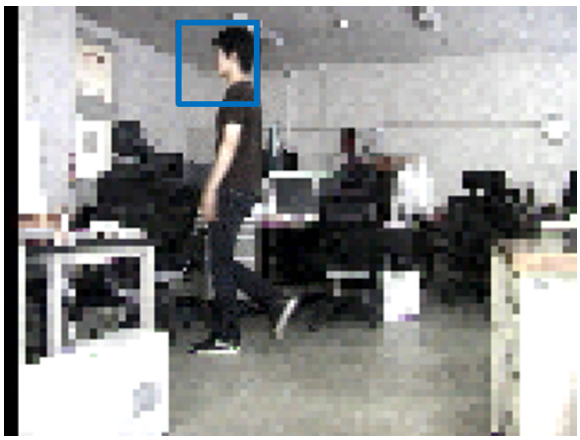

図 49: “提案手法+並列化” の出力

7.3　領域別処理量調整手法と並列化を組み合わせた評価

次に，提案手法と並列化手法を組み合わせることにより得られる，処理精度の低下の抑制を確認するため，並列化手法を用いた RaVioli（以降，“RaVioli+並列化”）と比較し，並列化を組み合わせた提案手法を追加した RaVioli（以降，“提案手法+並列化”）を評価した．**図 48**，**図 49** に最も負荷の高かった時刻における出力画像を示す．図 48，図 49 はそれぞれ “RaVioli+並列化”，“提案手法+並列化” で得られた出力画像である．前項同様に，“RaVioli+並列化” が画像全体を同じストライドで処理し，動物体が存在する重要な領域に対する処理精度が低減してしまっているのに対して，“提案手法+並列化” では前フレームからの変化が大きい領域を詳細に処理できていることが確認できる．また，並列化を組み合わせることで，前項の逐次動作の場合と比べて，重要な領域をより詳細に処理できたことが確認できる．

次に，時間経過に伴う空間ベースストライドの変化を**図 50** に，時間ベースストライドの変化を**図 51** に示す． 図には，前項で得られた並列化を用いない手法の結果も再掲している．“提案手法+並列化” は “RaVioli+並列化” に対して，両ベースストライド値を低く保てたことが確認できる．具体的には，空間ベースストライドの平均を 0.83 低く，時間ベースストライドの平均を 1.07 低く保つことができていた．また，並列化を用いない手法に対してもベースストライドを低く保つことができた．これらの結果より，提案手法を並列化と組み合わせることで，重要な領域に対する処理精度の低下を更に抑制することが確認できた．

最後に，6.2 節で説明したスレッドへの処理の割り当て手法を評価した．評価結果を**表 2** に示す．“静的手法” は均等な処理範囲を静的に割り当てる手法を用いた場合，“動

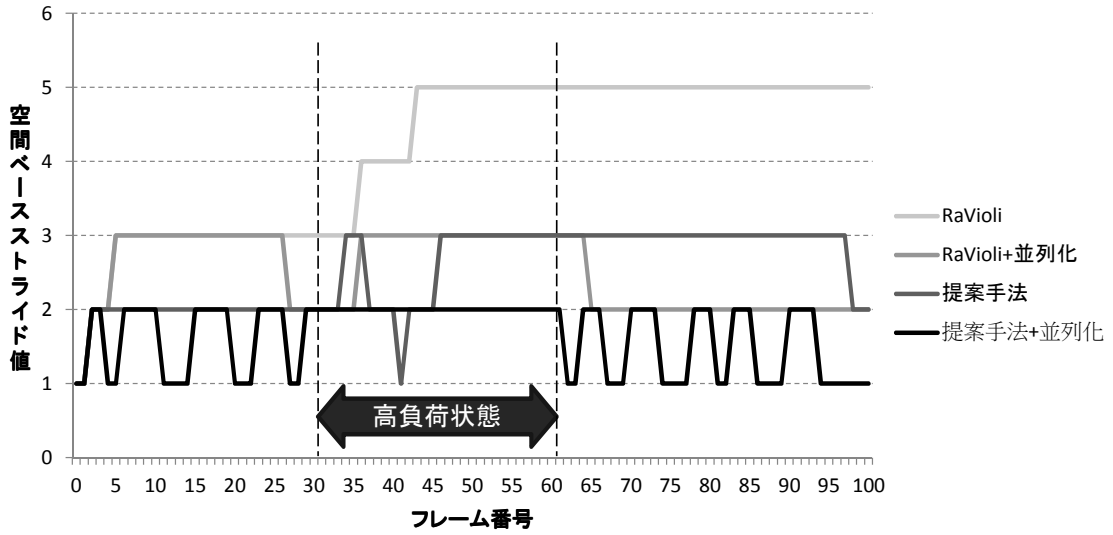

図 50: 空間ベーススストライド値の変動 (逐次+並列プログラム)

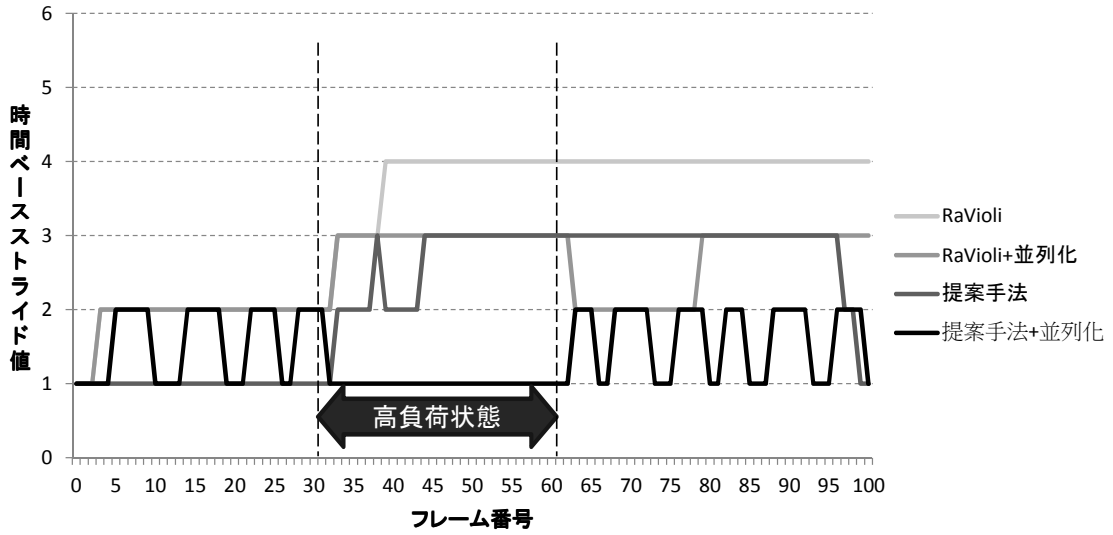

図 51: 時間ベーススストライド値の変動 (逐次+並列プログラム)

的手法” は並列処理の実行中に動的に処理を割り当てる手法を用いた場合，“提案手法” は領域のストライド値を考慮して処理を割り当てる手法を用いた場合をそれぞれ表している．

結果から，“提案手法” は “静的手法” に対して，全てのプログラムで高速に実行できていた．この結果から，複数のストライドで 1 フレームを処理する場合には，単純に均等な範囲の処理を各スレッドに割り当ててしまうと，並列化の効果を薄めることが確認できた．また，“動的手法” に対しても，テンプレートマッチングプログラムで

表 2: 各割り当て手法を用いた際の実行時間 (ms)

プログラム名	静的手法	動的手法	提案手法
グレースケール化	1.061	0.993	0.992
エンボスフィルタ	2.177	1.882	1.958
テンプレートマッチング	1216.441	1129.385	1093.711

3.16% 高速に実行できていた．これはテンプレートマッチング処理にかかる処理量が多く，ストライド値の差による各領域にかかる処理時間に大きな差があったためだと考えられる．一方，グレースケール化やエンボスフィルタでは効果が得られなかった．これらのプログラムは処理量が少なく，ストライド値の差による各領域にかかる処理時間の差がほぼなかったためだと考えられる．

8 おわりに

本論文では，まず，利用可能な CPU リソース量が変動する汎用計算機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，リアルタイム性を擬似的に保証する動画像処理ライブラリ RaVioli について述べた．また，RaVioli には解像度低減による処理精度低下の問題があることを示した．

その問題を解決するために，動画像ストリームを分割し，その部分ストリーム毎に処理量を調整することを提案し，RaVioli に実装した．分割領域を処理するためのクラス RV_TileImage を新たに追加し，RV_TileImage クラスのインスタンスを用いて処理するために RV_Image クラスの高階メソッドを拡張した．また，プログラマが既に記述した RaVioli プログラムを変更することなく，提案手法をその RaVioli プログラムに適用することを可能にした．

さらに，本提案手法を RaVioli に適用するにあたって考慮する必要がある問題点について述べ，これを解決するための，プリプロセッサの拡張やスレッドへの処理の割り当て手法の実装も行った．これにより，適用可能なプログラム数の増大や，並列化性能の向上を実現した．

最後に，提案手法を評価し，領域別に処理量を調整することによって画像処理にかかる処理時間を削減できることを確認した．また，動画像処理中に領域別に処理量を調整することにより，詳細な処理が必要な領域の空間解像度および時間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，プログラム実行時に，分割して処理すべきかどうかを自動的に決定することや分割領域数を動的に変更することが考えられる．動画像処理の内容によっては画像を領域に分割して処理しない方が高速に処理できる場合があるので，プログラム実行中に処理に適した方法に切り替えることで最良の結果を得ることを可能にする．また，本提案手法をより具体的な動画像処理アプリケーションへ適用することも考えなければいけない．これを実現するためには，新たな処理パターンに対応する高階メソッドが必要になる可能性があるため，その処理パターンの調査を進める．

また最近は，家電機器などに実装されている組み込み OS を対象とした高速化もしくは省エネルギー化への要求が高まっていることから [21]，高速化および省エネルギー化を目指す手法も考えられる．画像処理関数の入力値が RGB 色情報という限られた範囲の値であることに着目し，例えばグレースケール化された画像において関数の入力数は，ビット深度次第では 256 にとどまり，2 値化された画像では黒と白のみとなる．このことから，関数の入力値に対応した出力値を表にそれぞれ記憶しておき，再度同じ関数が同じ入力値で呼び出された場合に，出力値を表から呼び出すことにより計算を省略するメモ化手法 [22] を RaVioli に追加実装することを検討している．これにより，計算の省略による更なる高速化や省エネルギー化が見込める．

さらに，現在 RaVioli は CUDA[23] や Cell/B.E.[24]，TBB[25] など様々なプラットフォーム上の並列化に対応するように改良が進められている．そこで，これらの並列化に対応した RaVioli[26, 27] に，本論文で提案した新しい処理量調整方法を統合することが考えられる．また，RaVioli に適した新しい動画像処理向けの言語の開発や，その言語を各プラットフォームに対応した RaVioli プログラムに変換するトランスレータの開発などにも今後取り組んでいきたい．

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，松井俊浩准教授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，齋藤研究室および松井研究室の方々に深く感謝致します．特に，研究に関して貴重な意見を下さった稲葉崇文氏，今井満寿巳氏，小野亜実氏，大平真司氏，内山寛章氏に感謝致します．

著者発表論文

論文

1. Katsuhiko KONDO, Ami ONO, Takafumi INABA, Tomoaki TSUMURA, Hiroshi MATSUO: “Tiling with Different Spatial Resolutions for Pseudo Real-Time Video Processing Library RaVioli”, Proc. of The 7th Int’l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011), Dijon, France, pp.253-260 (Nov. 2011)
2. Ami ONO, Katsuhiko KONDO, Takafumi INABA, Tomoaki TSUMURA, Hiroshi MATSUO: “A GPU-supported High-Level Programming Language for Image Processing”, Proc. of The 7th Int’l Conf. on Signal-Image Technology and Internet-Based Systems (SITIS2011), Dijon, France, pp.245-252 (Nov. 2011)
3. Katsuhiko KONDO, Takafumi INABA, Hiroko SAKURAI, Masaomi OHNO, Tomoaki TSUMURA, Hiroshi MATSUO: “RaVioli: a GPU Supported High-Level Pseudo Real-time Video Processing Library”, Communication Papers Proc. 19th Int’l Conf. on Computer Graphics, Visualization and Computer Vision (WSCG2011), Plzen, Czech, pp.39-48 (Jan. 2011)

報文

1. 近藤 勝彦, 大野 将臣, 津邑 公暁, 松尾 啓志: “動画像処理ライブラリ RaVioli における領域別処理量調整の実現”, 信学技報 (SWoPP2010), Vol.IEICE-110, No.IEICE-CPSY-167, pp.13-18 (Aug. 2010)

参考文献

[1] 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. 2, No. 1, pp. 63–74 (2009).

[2] Sakurai, H., Ohno, M., Tsumura, T. and Matsuo, H.: RaVioli: a Parallel Video Processing Library with Auto Resolution Adjustability, *Proc. IADIS Int’l. Conf. Applied Computing 2009*, Vol. 1, pp. 321–329 (2009).

[3] Garcia-Martin, A. and Martinez, J. M.: Robust Real Time Moving People Detection in Surveillance Scenarios, *Proc. 7th IEEE International Conference on*

Advanced Video and Signal Based Surveillance, AVSS '10, IEEE Computer Society, pp. 241–247 (2010).

[4] Kim, C., Han, Y., Seo, Y. and il Kang, H.: Statistical Pattern Based Real-time Smoke Detection Using DWT Energy, *Proc. International Conference on Information Science and Applications*, IEEE Computer Society, pp. 1–7 (2011).

[5] Lin, K., Huang, J., Chen, J. and Zhou, C.: Real-time Eye Detection in Video Streams, *Proc. Fourth International Conference on Natural Computation - Volume 06*, IEEE Computer Society, pp. 193–197 (2008).

[6] Lee, C., Wang, Y. and Yang, T.: Static global scheduling for optimal computer vision and image processing operations on distributed-memory multiprocessors, *Computer Analysis of Images and Patterns*, Lecture Notes in Computer Science, Vol. 970, Springer Berlin / Heidelberg, pp. 920–925 (1995).

[7] Kywe, W. W., Fujiwara, D. and Murakami, K.: Scheduling of Image Processing Using Anytime Algorithm for Real-time System, *Proc. the 18th International Conference on Pattern Recognition - Volume 03*, ICPR '06, IEEE Computer Society, pp. 1095–1098 (2006).

[8] Liu, J., Shih, W.-K., Lin, K.-J., Bettati, R. and Chung, J.-Y.: Imprecise Computations, *Proceedings of the IEEE*, Vol. 82, pp. 83–94 (1994).

[9] 吉本廣雅, 有田大作, 谷口倫一郎: 実時間分散画像処理システムのための信頼度駆動アーキテクチャ, 情処研報 2004-ARC-149 (HOKKE 2004), pp. 85–90 (2004).

[10] Yoshimoto, H., Date, N., Arita, D. and Taniguchi, R.: Confidence-Driven Architecture for Real-time Vision Processing and Its Application to Efficient Vision-based Human Motion Sensing, *Proc. of the 17th Int'l. Conf. on Pattern Recognition (ICPR'04)*, Vol. 1, pp. 736–740 (2004).

[11] Isovic, D.: Flexible Media Processing in Resource Constrained Real-Time Systems, *Proc. Eighth IEEE International Symposium on Multimedia*, ISM '06, IEEE Computer Society, pp. 363–370 (2006).

[12] Köthe, U.: Generic Programming for Computer Vision: The VIGRA Computer Vision Library, http://hci.iwr.uni-heidelberg.de/vigra/ (2011).

[13] Bradski, G. and Kaehler, A.: *Learning OpenCV: Computer Vision With the OpenCV Library*, O'Reilly & Associates Inc (2008).

[14] Intel Corp.: *Open Source Computer Vision Library* (2001).

[15] Kovács, G., Iván, J. I., Pányik, A. and Fazekas, A.: The openIP Open Source Image Processing Library, *Proc. ACM Multimedia 2010*, MM '10, ACM, pp. 1489–1492 (2010).

[16] : Pandore: A library of image processing operators (Version 6.4). [Software]. Greyc Laboratory, http://www.greyc.ensicaen.fr/~regis/Pandore (2011).

[17] 金井達徳, 瀬川淳一, 武田奈穂美: 組み込みプロセッサのメモリアーキテクチャに依存しない画像処理プログラムの記述と実行方式, 情報処理学会論文誌：コンピューティングシステム, Vol. 48, No. SIG 13(ACS 19), pp. 287–301 (2007).

[18] Segawa, J. and Kanai, T.: The Array Processing Language and the Parallel Execution Method for Multicore Platforms, *The First International Symposium on Information and Computer Elements* (2007).

[19] Wang, S., Dong, Z., Chen, J. X. and Ledley, R. S.: PPL: A whole-image processing language, *Comput. Lang. Syst. Struct.*, Vol. 34, pp. 18–24 (2008).

[20] Dagum, L. and Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*, Vol. 5 (1998).

[21] 金井遵, 佐々木広, 近藤正章, 中村宏, 天野英晴, 宇佐美公良, 並木美太郎: 性能予測モデルの学習と実行時性能最適化機構を有する省電力化スケジューラ, 情報処理学会論文誌：コンピューティングシステム, Vol. 49, No. SIG 2(ACS 21), pp. 20–36 (2008).

[22] Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).

[23] NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 2.0 edition (2008).

[24] Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).

[25] Reinders, J.: *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, O'Reilly (2007).

[26] Kondo, K., Inaba, T., Sakurai, H., Ohno, M., Tsumura, T. and Matsuo, H.: RaVioli: a GPU Supported High-Level Pseudo Real-time Video Processing Library, *Communication Papers Proc. 19th Int'l Conf. on Computer Graphics, Visualization and Computer Vision (WSCG2011)*, pp. 39–48 (2011).

[27] 稲葉崇文, 大野将臣, 桜井寛子, 津邑公暁, 松尾啓志: GPU 及び Cell/B.E. に対

応した解像度非依存型動画像処理ライブラリ RaVioli の提案と実装, 信学技報, Vol. IEICE-110 (SWoPP2010), No. IEICE-CPSY-167, 電子情報通信学会, pp. 7–12 (2010).