修士論文

自動解像度調整および自動並列化機能を備える動画像処理ライブラリRaVioliの開発

指導教員　　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学大学院工学研究科
修士課程情報工学専攻
平成 19 年度入学　19417525 番

岡田 慎太郎

平成 21 年 2 月 5 日

自動解像度調整および自動並列化機能を備える 動画像処理ライブラリ RaVioli の開発

岡田 慎太郎

内容梗概

リアルタイム動画像処理システムへの要求から，これまでさまざまな専用チップが開発されてきた．しかし近年の汎用 PC の高性能化に伴い，今後汎用 PC および汎用 OS を用いたリアルタイム動画像処理の需要が高まっていくと予想される．本稿では，使用可能な演算リソース量の変動によりリアルタイム性の保証が難しい汎用 OS 上でも，解像度 (時間解像度及び空間解像度) を自動的に変動させることによりこれを実現する動画像処理ライブラリ RaVioli を提案する．また，変動する解像度を意識したプログラムをユーザに記述させるのは記述容易性に欠けることから，RaVioli は動画像処理における解像度をプログラマから完全に隠蔽するプログラミングパラダイムを提供することでこの問題を解消する．

この一方で，汎用 PC の低価格化に伴いマルチコア CPU を搭載した汎用 PC が広く市場に出回ってきたことから，コアを有効利用するようなシステム開発が求められる．ループ処理における独立したイテレーションを並列に動作させるデータ並列化の考え方は，コアを有効に活用でき，また画像処理の適用に向いているが，処理結果の正当性の保証が困難である．本稿では，まず正当性が損なわれる原因を明らかにし，これが解消されるよう RaVioli の逐次プログラムを並列プログラムに書き換えるプリプロセッサを提供することで，プログラマが並列処理について一切意識しなくても画像処理を並列に動作させることを可能とする．

顔認識およびフレーム間差分検出プログラムを用いた検証の結果，RaVioli が十分な記述能力を持つこと，およびプログラムの変更なく解像度が変更可能であり，また汎用環境において，実時間で動画像が処理できるよう負荷が自動調整されることを確認した．またプリプロセッサを様々な画像処理に対して適用し，マルチコアプロセッサ上で速度を検証した結果，広く並列化の適用が可能であることを確認し，またコアの台数に応じた速度向上が見込めることを確認した．

自動解像度調整および自動並列化機能を備える動画像処理ライブラリ RaVioli の開発

目次

1 はじめに

動画像処理能力の高いチップが多く開発され，空港や工場などの侵入者検知システムや，自動車走行中の前方車両もしくは障害物の認識による衝突回避システム [1] など，処理のリアルタイム性を重要視した動画像処理システムの開発が盛んに行われている．一方で計算機の高性能化により，顔認識アルゴリズム等に代表される処理量の多い画像処理を汎用 PC 上で行うことが可能となりつつある．従って高度な認識アルゴリズムを実装したリアルタイム動画像処理を，比較的安価な汎用 PC および汎用 OS 上で行うことも今後多くなると予想される．

一方で，汎用 OS 上においてリアルタイム性を保証するためには，処理に必要な CPU のリソース量（以下 CPU リソース）を常に確保することが重要な課題となる．複数プロセスの並行実行によって使用可能な CPU リソースが時々刻々と変化したり，フレーム毎に処理量が異なるような画像処理も実在するため，1/30 もしくは 1/60 秒毎に 1 フレームを処理するための CPU リソースの確保を保証することは困難である．Linux をリアルタイム OS に拡張するプロジェクトも存在するが [2]，一般に毎フレームの演算量が変動する認識処理では，ワーストケースに基づく 1 フレーム単位の演算が可能な CPU リソースの確保が必要となる．

そこで我々は，汎用システム上でも擬似的にリアルタイム性が保証された動画像処理を動作させることを可能とするライブラリ **RaVioli** (Resolution-Adaptable Video and Image Operating Library) を提案する．一般に，動画像処理プログラムにおいては出力フレームレートと 1 フレームの処理に割り当てることのできる演算量はトレードオフであり，使用可能な CPU リソースが限られる場合には**空間解像度**（1 フレームにおける画素数）および**時間解像度**（フレームレート）を低減させる必要がある．これに対し RaVioli は，ユーザが指定した優先度に応じて処理対象の空間解像度および時間解像度を自動的に変動させ，演算量を軽減する機能を提供する．

また解像度を変動させる場合，画像の幅や高さの画素数や動画像におけるフレーム数，画素配列やフレーム配列にアクセスする際の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，動的にスキップするフレーム数や画素数の変動を意識したプログラムを記述するのは困難である．また複雑化したプログラムはバグの温床となる可能性があり，開発プロセスにおけるコストの増大にもつながる．そこで RaVioli では，動画像処理における解像度の変動，ひいては解像度そのものをプログラマから完全に隠蔽するプログラミング環境を提供する．この

環境では，動画像処理における繰り返し処理単位を任意に設定し，その単位に対する繰り返し方法を記述するのみでよいため，プログラマは構成画素数やフレームレート，繰り返し文など解像度を意識したプログラムの記述を省略でき，また RaVioli 側では解像度を自由に変動させることができる．

さらに最近市場に出回っている汎用計算機は，Intel Xeon，Core 2 や AMD Phenom，Sun UltraSPARC T1 といったマルチコア CPU が搭載されており，マルチコア資源を有効に活用するようなシステム開発が求められている．逐次的に作られてきたシステムを複数のコア上で効率良く動作させる一般的な方法として，ループ内で処理の順番を入れ替えても支障をきたさないイテレーションを，並列に動作するよう書き換えるデータ並列化がある．ここで一般的な画像処理は，画素に対して行う処理の順番に依存性がなくそれぞれが独立しているため，データ並列化が比較的容易であると考えられる．しかし実際に画像の並列処理プログラミングをするためには，次のことを考慮する必要がある．まず，そもそも画素毎の処理が独立であることが保証されている必要があり，ある画素の処理結果によって他の画素の処理結果が変わってくるような，処理順に依存がある場合は，並列化しても処理結果の正当性が保証できない．次に，たとえば画素の平均値を求める処理を並列化した場合には，共有領域の変数に対して読み書きを行うため，競合がおこる．これらの事を画像処理の並列化をする際には考慮する必要があるが，ここで RaVioli を用いて記述する画像処理プログラムをライブラリ側で並列化させることを考える．RaVioli では構成要素に対する処理のみを記述しており，単位処理が要素毎に独立しているため，処理順に依存した処理の発見が容易である．また前述したように画素配列への繰り返し処理は RaVioli 側で行うため，プログラマが並列処理を意識することなくライブラリ側でデータ並列化が可能となる．これらのことから，RaVioli で記述したプログラムは並列処理をするのに敵していると言える．本稿では，RaVioli を用いて記述した逐次プログラムの構成要素に対する処理部分をプリプロセッサで解析することにより処理結果の正当性を保証した並列プログラムに自動で変換する機能を提案する．これにより，プログラマが並列化を意識しなくてもマルチコア資源を有効活用できるようにする．

以下，2 章で既存の画像処理ライブラリおよび並列化ライブラリと RaVioli を比較する．3 章では RaVioli の基本仕様であるリアルタイム性の保持と記述手法の仕様について述べ，4 章では RaVioli を使って書かれたプログラムを並列化させるための仕様と実装方法について述べる．次に 5 章では動画像処理を通じたリアルタイム性および並列性の評価結果を示す．最後に 6 章で本稿全体をまとめる．

2 関連研究

2.1 画像処理の抽象化

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に VIGRA[3] がある．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．また MAlib[4] や OpenCV[5] は，画像処理の一般的なアルゴリズムを C の関数や C++のメソッドとして提供している．特に OpenCV は Video4Linux2 を使用することにより，IEEE1394 カメラ経由のデータに対するリアルタイム処理も提供している．

一方 RaVioli[6] は，処理量に直接関係する 1 フレームの構成画素数やフレームレートをプログラマから隠蔽することを目的としている．このため，従来の抽象化ライブラリのように単純に処理内容を抽象化してユーザライブラリの形で提供するものとは完全に異なっている．プログラマは画像に対する処理内容を，解像度を考慮することなく記述できることから，従来の抽象化ライブラリと比べて，動画像処理の詳細なアルゴリズムを簡単に実現することが可能になる．

また金井らは数式エディタを用いて記述できる独自の記述言語を用いることにより，画像処理プログラミングを抽象化している [7]．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し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，RaVioli で行っている動的な構成画素数の変動には対応していない．

2.2 処理時間の自動調整

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アプローチとして，複数アルゴリズムの切り換え手法がある．たとえば，指定した計算時間を超過すると，その時点で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）が提案されている [8]．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]．しかしこの方法では，処理を計算負荷の異なる複数のアルゴリズムで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．

一方 Streaming VIOS[10] は，動的に処理画素数を変更し，プログラマから指定された画素座標を現処理画素数の対応座標に読みかえることのできるライブラリである．これは RaVioli に近い考え方であるが，Streaming VIOS の提供する枠組ではほとんどの動画像処理において処理画素数を透過的に扱うことはできない．このため Streaming VIOS では，プログラマが現解像度に依存するパラメータを取得したうえでそのパラメータを用いた処理を行う必要がある．

これらに対し本稿で提案する RaVioli は，動画像処理の抽象化と処理精度・処理時間の自動調整を両立することのできるものである．行いたい処理に対してプログラマは複数のアルゴリズムを記述する必要はなく，RaVioli 側が処理対象となる画像の構成画素数およびフレームレートを状況に応じて自動的に変更することで処理精度・時間を調整する点で，既存手法とは完全に異なる．

2.3 並列プログラミング

並列処理の分野では，一般的に用いられる並列処理パターンをライブラリの形で抽象化して提供する並列スケルトン [11] の考え方が古くからある．それぞれのスケルトンは並列動作を内部に隠蔽しているので，プログラマは逐次プログラムを作成するような感覚で並列プログラムを作ることができる．Intel で開発された並列計算パターン用 C++ ライブラリ Intel Threading Building Blocks(以下 TBB) は，並列スケルトンの形で書かれたプログラムをマルチコア上で並列に動作させるライブラリであり，スレッド管理を隠蔽することで処理の並列化を抽象的に扱える機能を提供する．プログラマはスレッドを直接操作するといった記述はせず，スレッドに割り当てる仕事 (サブタスク) の内容を記述したものを，タスクを管理するクラスに渡す．ライブラリ側にタスクのスケジューリングを任せることから，プログラマはスレッドの生成や同期などを考慮する必要が無い．またライブラリは，並列に動作するサブタスク同士の相互作用を記述するためのデータ構造を提供しており，排他制御や reduction 演算を内部的に行う．

並列プログラミングの標準化されたモデルである OpenMP[12] は，C 言語の逐次プログラム内の並列化したい部分を簡単な注釈で指定し，それをプリプロセッサが半自動的に並列処理へ変換するディレクティブ形式を採用している．**図 1** は 1 から 256 を加算するプログラムを OpenMP を用いて記述したプログラムである．for 命令の前に `#pragma parallel` と記述することにより，イテレーション部分が並列に処理されるが，このままでは `sum` の書き込みが競合する．よって競合の対象となる変数 `sum` を並列

```
#define NUM_THREAD 4
int i; int sum=0;
#pragma parallel reduction(+:sum)
  for(i=1;i<=256;i++){
    sum+=i;
  }
```

図 1: OpenMP を用いた reduction 演算を伴う並列処理記述手法

数分複製して，そこに一時的に結果を書き出し，最後に複製された sum に対して加算を適用する．OpenMP がこの一連の処理をするよう，プログラマは reduction(+:sum) と記述する．なお始めの#define NUM_THREAD 4 は，4 並列でプログラムを動作させるという意味である．

Streaming VIOS の前身となる VIOS[13] は，マルチコアだけでなくマルチプロセッサやバス共有型の分散処理を想定した環境である．VIOS による並列プログラミングは，並列処理部を記述したモジュール部と，処理用計算機の登録，データの読み込みや初期化，モジュール部の制御を担当する実行フロー部との 2 つがある．並列処理をするためには各処理用計算機への画像データとモジュール処理の転送命令を実行フロー部に記述する．モジュール部では並列に処理するイテレーションの定義として，終了条件やインクリメントのない for 文を記述する．

これらは抽象化に関して非常に洗練された並列モデルであると言えるが，システム開発の際には実行順序に依存のない命令群同士をループ並列またはタスク並列の形で処理ができるよう，アルゴリズムの段階から考慮して開発する必要がある．また競合が起きるであろう処理をあらかじめ把握しておく必要があり，排他制御や reduction 演算などの対応をとらなければならない．

一方で，提案する RaVioli の記述方法は，画素を走査するループ処理をライブラリの関数内で制御しているため，この部分が並列化できることは明白である．このことからプログラマがあらためて並列化可能な部分を考慮してプログラムを作成しなくてもよい．さらに本稿で提案する自動並列化のためのプリプロセッサの働きにより，並列関係にある処理の間に依存関係があることを検出した場合は，並列処理プログラムに書き換えない．また競合が起こりうる処理を検出した場合は，プログラムから reduction 演算を自動生成する．以上のことから，RaVioli は全く並列処理を考慮しなくてもよい，逐次プログラムの自動並列化を実現する．

3　解像度非依存型動画像処理ライブラリ RaVioli

本章では，動画像処理のリアルタイム性を保証し，かつ解像度を隠蔽したプログラミング環境を提供する解像度非依存型動画像処理ライブラリ RaVioli について述べる．

3.1　設計思想

3.1.1　演算量の自動調整

一般に汎用 PC および汎用 OS 上では，1 フレームあたりの演算量の変動や他プロセスの動作による使用可能な CPU リソースの変動から，1/30 もしくは 1/60 秒毎にカメラ等から送られてくる画像のすべてをリアルタイムに処理することは困難である．この状況を擬似的に解決するために，動画像のフレームあたりの演算量に応じて，動的に処理する画素数を調整させることが考えられる．1 フレームの演算量が多く，演算にかかる時間が 1 フレームのキャプチャ間隔より長い場合は，処理する画素数を動的に少なくすることにより演算量を低減させる．逆に演算に使用できる CPU リソースが十分ある場合には，元画像に近づくように処理する画素数を増やしていくことにより，十分高い処理画素数を維持することができる．このようにフレームレートを維持しつつ処理できる最大の画素数となるように処理画素数を自動調整することで，使用可能な CPU リソースが変動する環境においても，常に実時間に近い形で処理結果を出力することが可能になる．

一方で詳細な顔認識のような，処理結果のフレームレートよりも 1 フレームの処理精度が重要である処理も存在する．そのような場合には，画素数を高い精度で維持しつつ処理するフレームをスキップして処理量を低減させることで，画像の精度を重視したリアルタイム処理を実現できる．

一般には，動画像処理の内容によって空間解像度か時間解像度のどちらがどれだけ重要かが異なる．これに対して，優先する割合を指定できるインターフェースをユーザに提供することで，より処理目的に対して価値のある結果が得られる．例えば自動車を検出する処理は，時間解像度を低減しすぎると，飛ばされたフレーム内にしか自動車が観測されなかった場合に検出されない．一方で空間解像度を低減しすぎると，観測された動物体が自動車であるかどうかが判別できない．ここで優先度をバランスよく設定することで，有限な CPU リソースをうまく活用した処理ができる．

演算量を調節するにあたり RaVioli は，1 フレーム内で処理対象となる画素数（処理画素数）を低減させるために画素を x 軸方向および y 軸方向に一定量づつスキップし

てアクセスする手法をとる．また処理フレームレートの低減は，フレームを一定量づつスキップする手法をとる．スキップする量は使用可能リソース量に依存し，処理結果をリアルタイムに近い時刻で出力できない場合にスキップする量を増やす．

以上提案した処理量の自動調整機能を，動画像処理プログラムに組み込むことによりリアルタイム性は保たれる．しかし一般にはこれを実現するために，変動する解像度を意識したプログラミングをする必要がある．この問題と問題に対する解決策を次節に提案する．

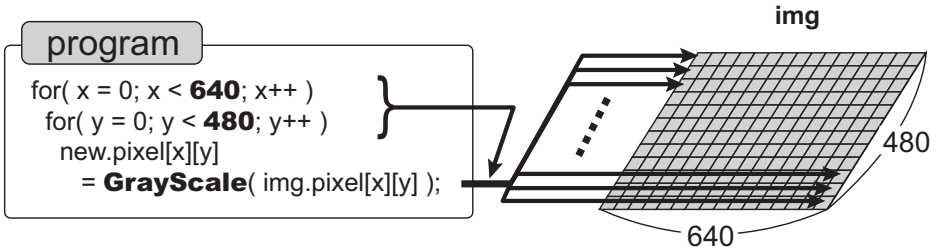
3.1.2 動画像処理の抽象化

前節では，リアルタイム性を保持するために解像度を自動的に調節するというアプローチを提案した．この方針に基づいて動画像処理プログラムを実装する際，プログラマは解像度の変動を考慮してプログラムを記述する必要がある．これはプログラムの可読性が低くなり，デバッグの際にバグ特定が困難になる等の問題につながる．

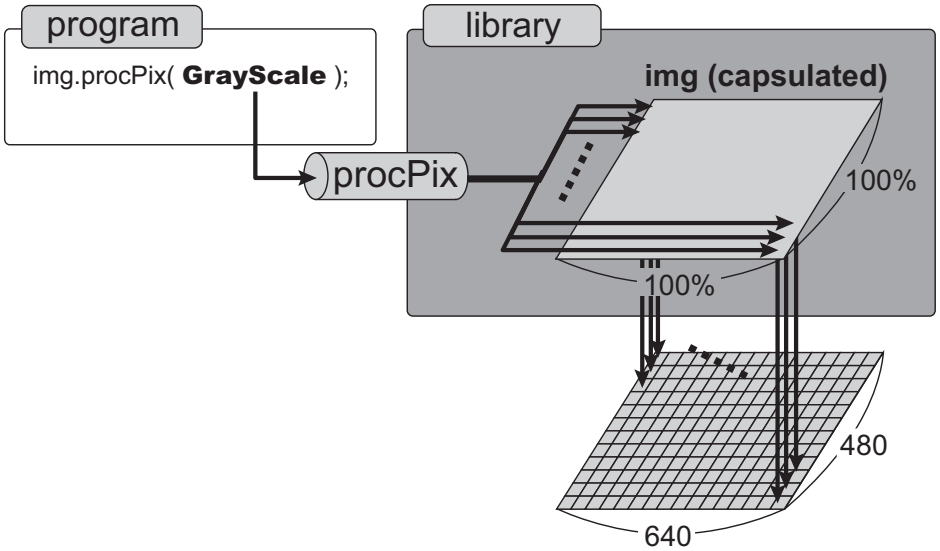
ここで，解像度の単位となる「画素」や「フレーム」といった動画像を構成するための要素に焦点をあてる．これら構成要素は，画像や動画像を計算機上で扱うために導入された概念であり，そもそも人間の脳内における認識過程には，視覚情報に対して「これは画素の集合である」という意識は存在しない．しかし量子的に情報を扱う計算機では，画像を点の集合として，動画像を平面の集合として扱わざるを得なかった．また人間が視覚情報から動く物体を認識する過程について考えてみよう．人間は物体が動いているかどうかを，時系列でいう「直前」と「今」の視覚情報の差から判断する．この時人間はフレームという区切られた単位のうち 2 枚の，一つ一つの画素に対して差分を取ろうとは思わない．しかし計算機上では for などのループ文を用いてすべての構成要素に対して繰り返し差分処理を施す必要がある．この繰り返し部分もまた動画像処理の本質ではなく，人間が動画像に対して持つ直感的な処理イメージとは異なる．

この問題に対し RaVioli は，プログラマから解像度ひいては構成要素の概念を隠蔽するプログラミングパラダイムを提供する．動画像処理プログラムから空間解像度と時間解像度を示す変数を排除し，RaVioli 側ですべて管理することで，プログラマは解像度を意識しなくても動画像処理を記述できる．また繰り返し部分も RaVioli 側で管理し，プログラマには構成要素に対する処理のみを記述させることにより，人間が動画像に対して持つ直感的な処理をほぼそのままの形でプログラムに記述できる．

一般に画像処理は，小さな単位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．たとえばカラーからモノクロへの変換や色の反転などの処理



(a) Traditional image processing.



(b) Image processing with RaVioli.

図 2: 画像処理の処理イメージ

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，**図 2**(a) のように通常ループイテレーションで記述され，画像全体もしくは特定の範囲に対して繰り返し適用する形で行われる．ここで空間解像度の変動の影響を受けるのは，640, 480 といったイテレーション回数や，イテレーション変数のインクリメント幅である．そこで RaVioli では，図 2(b) で示すようにイテレーション自体を隠蔽する．図中の `GrayScale` は 1 画素の RGB 値をグレースケールに変換する関数である．ライブラリが提供する高階関数 procPix() に画素単位の処理を引数として渡すことにより，ライブラリ側で自動的に画像中の全画素や特定範囲の画素に適用する．このようにイテレーションをライブラリ側で制御することにより空間解像度を隠蔽する．

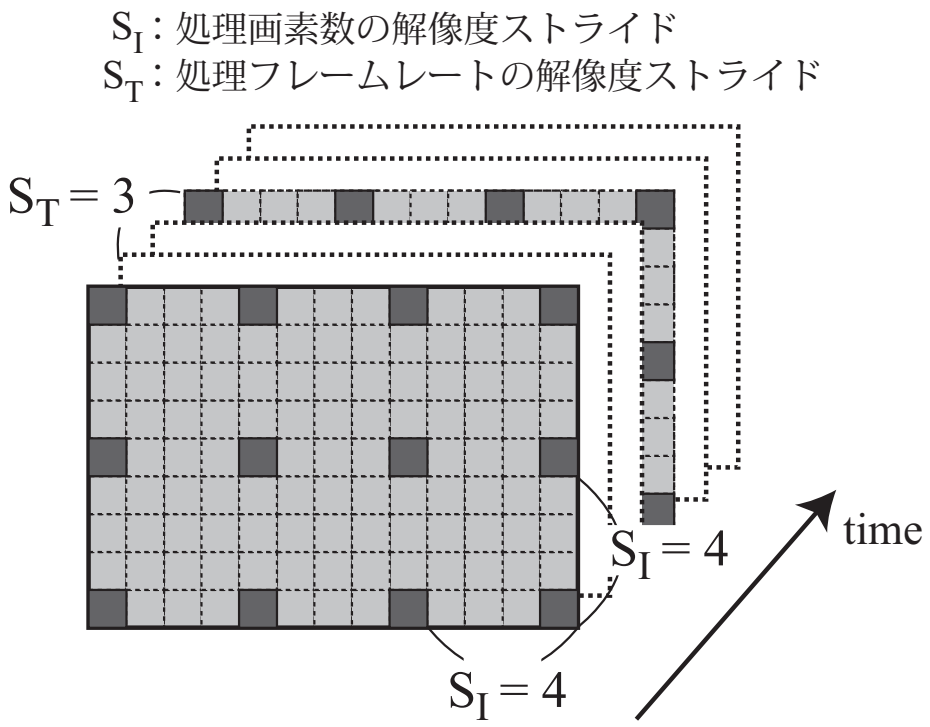


図 3: 解像度ストライドに基づいたアクセス位置の指定手法

また空間解像度と同じく，時間解像度の隠蔽も同様に扱う．プログラマは当該フレームまたは当該及び隣接フレームに対する処理のみを記述し，RaVioli が必要なフレームに処理を適用する．

3.2 仕様

3.2.1 パラメータに基づく処理量調整

汎用 PC が提供可能な CPU リソースより 1 フレームの処理に必要な演算量が多く，処理結果をリアルタイムに近い時間間隔で出力できない場合に，RaVioli は処理解像度を制御する**解像度ストライド**を変更させることで演算量を調整する．この解像度ストライドは処理画素数と処理フレームレートのそれぞれを制御する 2 つの値から成り，これらの値に応じて，動的に空間解像度及び時間解像度を変更する．**図 3** は，処理画素数の解像度ストライド S_I と処理フレームレートの解像度ストライド S_T がそれぞれ 4 と 3 の場合の処理対象となる部分を示した図である．薄いグレーのフレームが処理対象となるフレームであり，濃いグレーの画素が処理対象となる画素である．まず S_T が 3 であるため，キャプチャされたフレームを 2 枚づつスキップしたものを処理対象

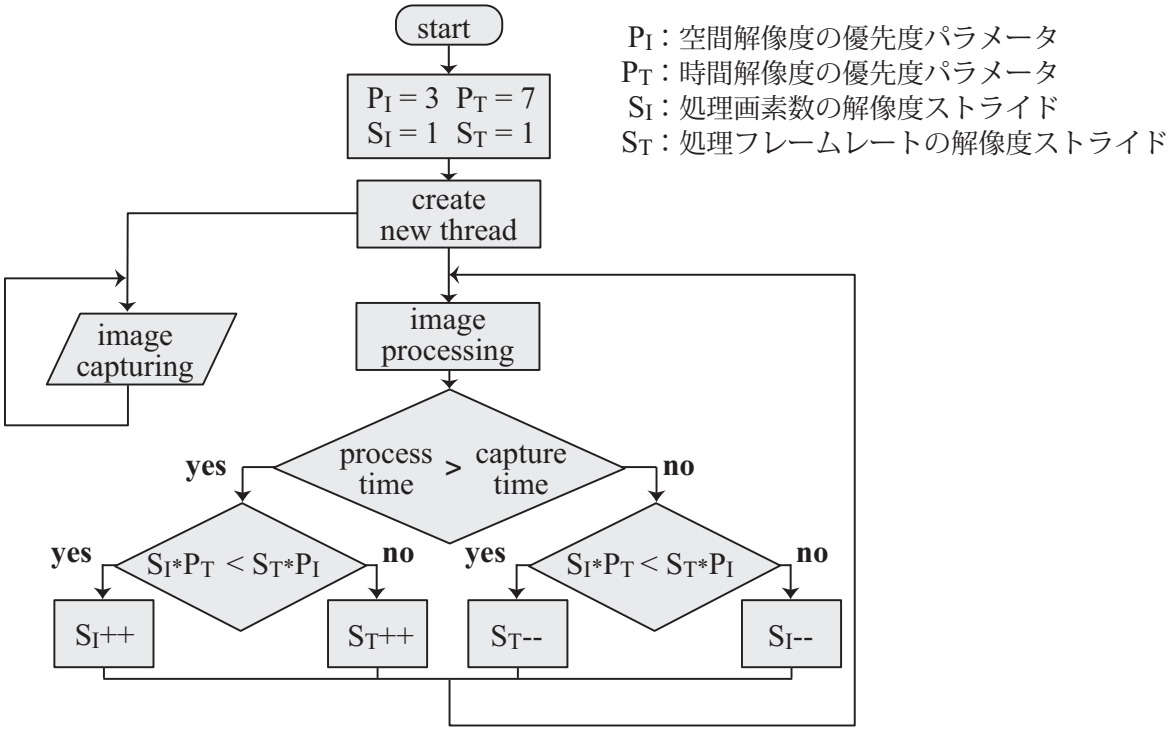


図 4: 処理量を制御するパラメータのフローチャート

フレームとする．さらに S_I が 4 であるため，処理対象フレーム内の画素を 3 つづつスキップしてアクセスする．つまり実際に処理されるのは濃いグレーの画素になる．このように解像度ストライドを変更し，通常の 1/3 のフレーム数と 1/16 の画素数のみに対し処理を適用する事により，演算量を低減させる．

またプログラマが処理画素数と処理フレームレートのどちらをどれだけ維持させるかを指定するために，維持させる割合を指示するための**優先度パラメータ**を導入する．この優先度パラメータには，処理画素数の優先度を表すパラメータ P_I と処理フレームレートの優先度を表すパラメータ P_T があり，パラメータの値が大きいほど優先する割合が高くなる．またどちらかのパラメータ値を 0 とした場合には，0 と設定された要素が一切優先されなくなる．例えば (P_I, P_T) を $(1, 0)$ とした場合は，S_I を常に 1 とするかわりに S_T を増加させて処理フレームレートを低減させる．逆に $(0, 1)$ とすれば，S_T を 1 で維持するかわりに S_I を増加させて処理画素数を低減させる．また (P_I, P_T) を $(3, 7)$ と設定した場合は，S_I と S_T が 7：3 に近い値で維持されるようにする．こうすることで優先度パラメータに基づいた割合で処理画素数及び処理フレームレートを削減する．

以上に述べた優先度パラメータに基づく演算量調整の流れを**図 4** に示す．図はそれ

ぞれの優先度パラメータ (P_I, P_T) を (3, 7) とした場合とし，処理開始後にそれぞれのパラメータの値を設定する．キャプチャしたフレーム 1 枚を処理した後にフレームあたりの処理時間とキャプチャ間隔を比較し，処理時間が長い場合は解像度ストライドを増加させる．この際，処理画素数の解像度ストライド S_I と処理フレームレートの解像度ストライド S_T のどちらを増加させるかは，優先度パラメータ (P_I, P_T) を元に決定する．また一方でフレームあたりの処理時間よりもキャプチャ間隔が長い場合は，(P_I, P_T) を元に解像度ストライドを減少させる．解像度ストライドの決定後は次のフレームの処理に移るが，その際図 3 で示すように処理フレームレートの解像度ストライドに応じた処理フレームを選択する．そして処理画素数の解像度ストライドに応じた処理画素に対して 1 フレーム分の処理を適用する．

3.2.2 主なクラスと高階メソッド

RaVioli では，画素数やフレームレートといった構成要素を隠蔽するために，それぞれの要素配列をカプセル化する．またカプセル化されたインスタンスに対して処理を適用するためのインターフェースとして**高階メソッド**を提供する．高階メソッドは，構成要素を処理単位とした関数を引数として受け取り，その関数を要素配列の指定された範囲内または全体に施すメタ関数である．

この機能を実現するにあたり，1 画素の情報を持つ RV_Pixel，1 枚の画像情報を持つ RV_Image，2 次元座標の情報を内部的に持つ RV_Coord，距離の情報を内部的に持つ RV_Length，画像配列以外の配列情報を持つ RV_Array，および RV_Streaming の 6 つのクラスを持つライブラリ RaVioli を C++で実装した．以下それぞれのクラスと，RV_Image クラスおよび RV_Streaming クラスが持つ高階メソッドについて示す．

RV_Pixel クラス

画像を構成する画素の情報を保持するクラスであり，RGB それぞれの値を，8 ビットのメンバ変数として持つ．また，今後ビット深度の異なる色情報も扱えるように，濃度の範囲を 0 から 1000 の整数値で表した値（以下，**抽象濃度**）を用いて操作するメソッドを持つ．

RV_Image クラス

画像の実体を表すクラスであり，メンバ変数として構成画素数分の RV_Pixel インスタンスを配列の形で保持する．また，空間解像度における解像度ストライド変数 stride を持ち，画素アクセスの際の現スキップ間隔を保持する．

表 1 に実装した高階メソッドの種類，返り値及び使用例の詳細を示す．なお，高階メソッドの引数として定義されている RV_Coord オブジェクト C_S および C_E は，画像

表 1: 高階メソッドの種類

メソッド名	処理単位	返り値	使用例
procPix	1 画素	同位置の画素	二値化
procNbr	1 画素，近傍	同位置の画素	畳み込み積分
procCoord	1 画素，座標	同位置の画素	ハフ変換
transCoord	1 座標	変換座標	拡大縮小
procBox	ウィンドウとその左上座標	なし	テンプレートマッチング
compImg	1 画素，別画像の画素	同位置の画素	差分検出

内における処理範囲の対角頂点を抽象的に指定するものであり，省略した場合は画像全体が処理対象の範囲となる．

procPix(RV_Pixel *Func(P), C_S, C_E)

プログラマは，RV_Pixel クラスである画素 P を引数とした関数 *Func* を定義する．procPix() は引数として受け取った関数 *Func* をすべての対象画素に対して適用し，処理結果の画像を返す．閾値判定による二値化などに利用する．

procNbr(RV_Pixel *Func(P, *P_{Nbr}, N_w, N_h), C_S, C_E)

プログラマは，画素 P，その近傍画素の集合である RV_Pixel 配列へのポインタ P_{Nbr}，および RV_Pixel 配列の幅と高さの要素数 N_w,N_h を引数とした関数 *Func* を定義する．procNbr() は引数として受け取った *Func* を，1 画素とその近傍を入力として全ての対象画素に適用し，処理結果の画像を返す．畳み込み演算などに利用する．

procCoord(RV_Pixel *Func(P, C), C_S, C_E)

プログラマは，画素 P とその座標である RV_Coord オブジェクト C を引数とした関数 *Func* を定義する．procCoord() は引数として受け取った *Func* を，1 画素およびその位置座標を入力として，全ての対象画素に適用し，処理結果の画像を返す．ハフ変換などに利用する．

transCoord(void *Func(C_{bfr}, *C_{aft}))

プログラマは，変換前，変換後の 2 つの RV_Coord C_{bfr} および C_{aft} を引数とした関数を定義する．transCoord() はこの定められた対応規則である *Func* を引数として受け取り，画像全体を処理範囲とした場合の全対象画素の座標を，*Func* に従って変換する．画像の回転、拡大縮小などに利用する．

procBox(void *Func(Img_B, C_{Bs}, C_{Be}), W, H)

プログラマは，マッチパターン等に用いる RV_Image インスタンス Img_B，およびウィンドウの始点・終点 RV_CoordC_{Bs}, C_{Be} を引数とした関数 *Func* を定義する．また RV_Coord を利用して，切り出すウィンドウの幅 W，高さ H を指定する．procBox() は引数として受け取った *Func* に，幅 W，高さ H を持つ RV_Image オブジェクトを全対象画素を始点として *Func* に渡す．主に画素から得た中間データを生成するために使われる．テンプレートマッチングなどに利用する．

compImg(RV_Pixel **Func(P1, P2)*, *Img*)

プログラマは，比較元と比較対象の画像それぞれに対応する 2 つの画素 P_1，P_2 を引数とした関数 *Func* を定義する．また compImg() を呼び出した RV_Image インスタンスの比較対象である *Img* を引数として定義する．compImg() は引数として受け取った *Func* に対して，画像全体を処理範囲とした場合の全対象画素 P_1 と，*Img* の対応する位置の画素 P_2 を適用し，処理結果の画像を返す．2 枚の画像間の差分検出などに利用する．

RV_Coord クラス

RV_Coord クラスは画像の任意の位置の x, y 座標を内部的に保持するクラスであり，空間解像度が隠蔽された画像中の位置情報を扱いたい場合に用いる．RV_Coord オブジェクトに対して具体的な値を代入することはできず，元画像の幅や高さを 1 とした場合の相対的な大きさとして座標を扱う．プログラマは具体的な値を使用できない代わりに，RV_Coord クラスが提供する RV_Coord クラス同士の四則演算や数値との乗除演算ができるオペレータを用いて，座標を抽象的に扱うことができる．

RV_Coord オブジェクトを扱う方法について具体的に説明する．例えば座標値 ($x1$,$y1$) を内部的に持つ座標オブジェクト A と ($x2$,$y2$) を持つ B に対して $A = A + B$ とすれば，A の内部的な値は ($x1 + x2$,$y1 + y2$) となり，$A = A/2$ とすれば，元の A に対して幅と高さが半分の座標を生成することができる．また RV_Image オブジェクトには，その画像の大きさを示す RV_Coord オブジェクトがメンバ変数として定義されている．画像の中心を指定する場合にはこれをコピーし，1/2 との積をとることで，解像度に影響されず常に中心を指定することができる．

RV_Length クラス

RV_Coord クラスは座標値を持つことから，常に原点からの位置を示していると言える．これは換言するとベクトルを表現しているといえるが，これに対し RV_Length クラスはスカラー値を表している．RV_Coord クラスと同様に内部的に値を持ち，プログラマからはその値にアクセスできない．RV_Length クラスを用いて RV_Coord イ

ンスタンスが示す点と原点の線分の長さを表したり，RV_Image クラスの幅や高さを別々に扱うことができる．

ここで，RV_Coord クラスおよび RV_Length クラスが内部で持つ座標や大きさは，画素を単位とした数量であるため，画素数を意識している様に感じられるかもしれない．しかし実際にユーザからは値が見えないことから RV_Coord や RV_Length のオブジェクトは，構成単位を規定しない相対的な大きさであるといえる．たとえば幅 30cm 高さ 40cm の絵の中心を左下を原点として表現する場合に，幅 15cm 高さ 20cm の位置というようにそのままメートル法で示すこともできるが，幅 50%高さ 50%の位置というように割合を用いて表現することもできる．ここで絵の大きさが変化した場合，メートル法は幅や高さを違う数値として指定しなおす必要があるが，割合では幅 50%高さ 50%という数字は常に一定である．RV_Coord, RV_Length クラスはこのような割合を用いた位置指定の方法を採用することで，画像の大きさや特定の位置を抽象的に扱っている．

RV_Array クラス

画像処理のうち，結果を画像以外のデータとして出力するシステムが多くあるが，画像の大きさを要素数とする配列は，RaVioli が画素数を隠蔽しているために定義できない．RV_Array クラスでは，画像の幅や高さが格納された RV_Coord, RV_Length インスタンスを取り込み，それを要素数とした 1 次元もしくは 2 次元の配列を作成する．なお配列要素の型は，RV_Array インスタンスの生成時にテンプレートを用いて指定する．またプログラマは全体の要素数が隠蔽されていることから，配列の位置を指定する場合にも RV_Coord, RV_Length を利用する．

利用例として，二値画像において縦方向の黒画素数の合計を表すヒストグラムを生成する場合は，画像の幅の大きさを持つ RV_Length インスタンスを読み込み，これを要素数とした 1 次元の RV_Array インスタンスを生成する．RV_Array インスタンスの配列要素へ値を格納する際には，位置指定に高階メソッド procCoord() から得られる RV_Coord インスタンスを用いる．

RV_Streaming クラス

動画像を処理するクラスであり，カメラから動画像をキャプチャするメソッドと，数枚のフレームを配列で格納するリングバッファおよびそのメソッドなどを持つ．概略を**図 5** に示す．カメラから動画をキャプチャする部分は，Video4Linux2 (V4L2) [1] ド

[1] http://linux.bytesex.org/v4l2/

ライバを用いて実装し，取り込んだフレームをリングバッファに一時的に保存する．次に，取り込んだ複数枚のフレームから，処理フレームレートの解像度ストライドに応じた間隔でフレームをリングバッファから取り出し，そのフレームのデータと，現段階での処理画素数の解像度ストライドを RV_Image クラスのインスタンスに格納する．そしてこの RV_Image インスタンスには，プログラマが記述した画像単位の処理が，高階メソッドを通じて適用される．1 フレームの処理が終了した後，優先度ストライドの値を元に各解像度ストライドを変更する．そしてリングバッファから処理フレームレートの解像度ストライドに応じた時刻のフレームを選択し，新たに生成された RV_Image インスタンスに処理画素数の解像度ストライドを設定する．

ここで，例として当該フレームと一つ前のフレームをそれぞれグレースケールに変換し，それぞれのフレーム間の差分をとる処理を考える．一つ前のフレームをグレースケールに変換した時点での処理画素数の解像度ストライドと，当該フレームを変換した時点での処理画素数の解像度ストライドは，それぞれが異なる場合が存在する．当該フレームの処理画素数が一つ前のフレームの処理画素数より多い場合は，位置によっては差分処理のための対応する処理画素が一つ前のフレームに存在しない．このように，対応した位置に画素が存在しない場合には，対応する位置の近傍にある画素を比較対象として代用する．具体的には，当該フレームと一つ前のフレームのそれぞれが持っている処理画素数の解像度ストライドを比較し，当該フレームにおける処理画素数の解像度ストライドが大きい場合は，前フレームで処理がされていない画素へのポインタを，その近傍で処理がされている画素に読み替える．このように実装することで，画素数の異なるフレーム同士の比較・差分処理をすることができる．

代表的な高階メソッドを以下に示す．

proc1Frm(*Func(Img_{Curr}))

全ての処理対象フレームに対し，指定された関数 *Func* を適用する．*Func* は 1 つの RV_Image インスタンス Img_{Curr} を引数にとる関数である．背景差分などに利用する．

procAdjFrm(*Func(Img_{Curr}, Img_{Prev}))

全ての処理対象フレームとその隣接フレームに対して指定された関数 *Func* を適用する．*Func* は 2 つの RV_Image インスタンス Img_{Curr}, Img_{Prev} を引数にとる関数である．RaVioli を使用した環境では，状況に応じて処理フレームをスキップすることで自動的に処理フレームレートが変動する．この際隣接フレームがどのキャプチャフレームにあたるかは RaVioli によって自動的に判断され，当該フレームと共に *Func* に渡される．動画のフレーム間差分などに利用する．

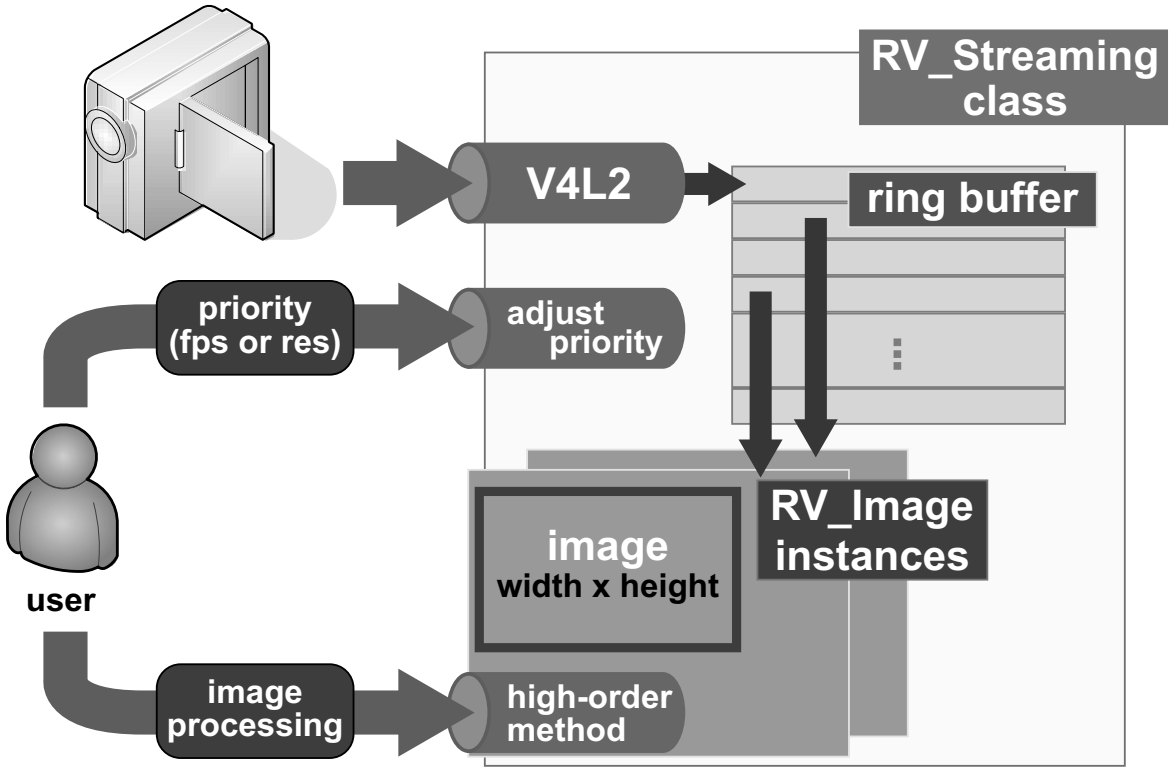


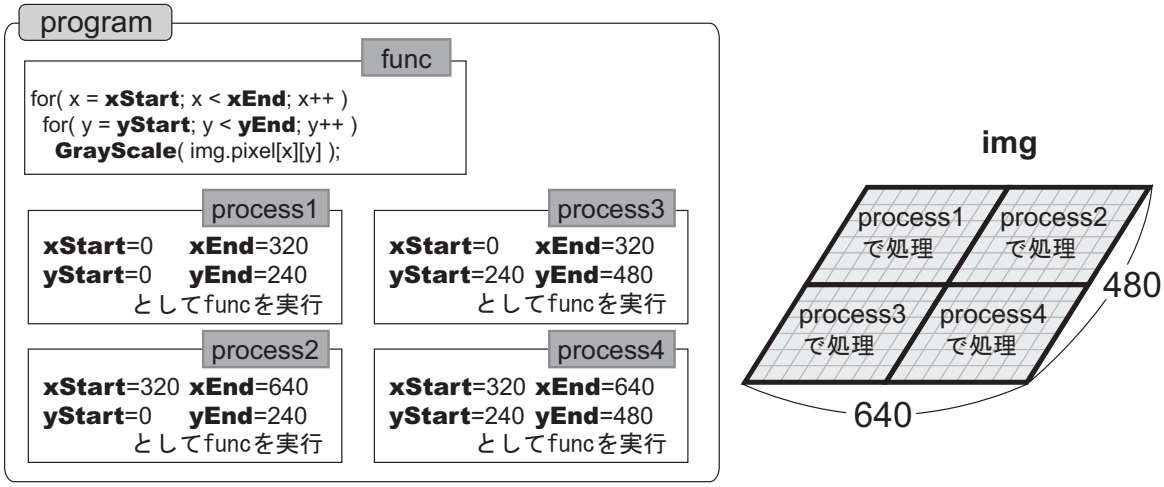
図 5: RV_Streaming クラスの構造

4 RaVioli 使用プログラムの自動並列化プリプロセッサ

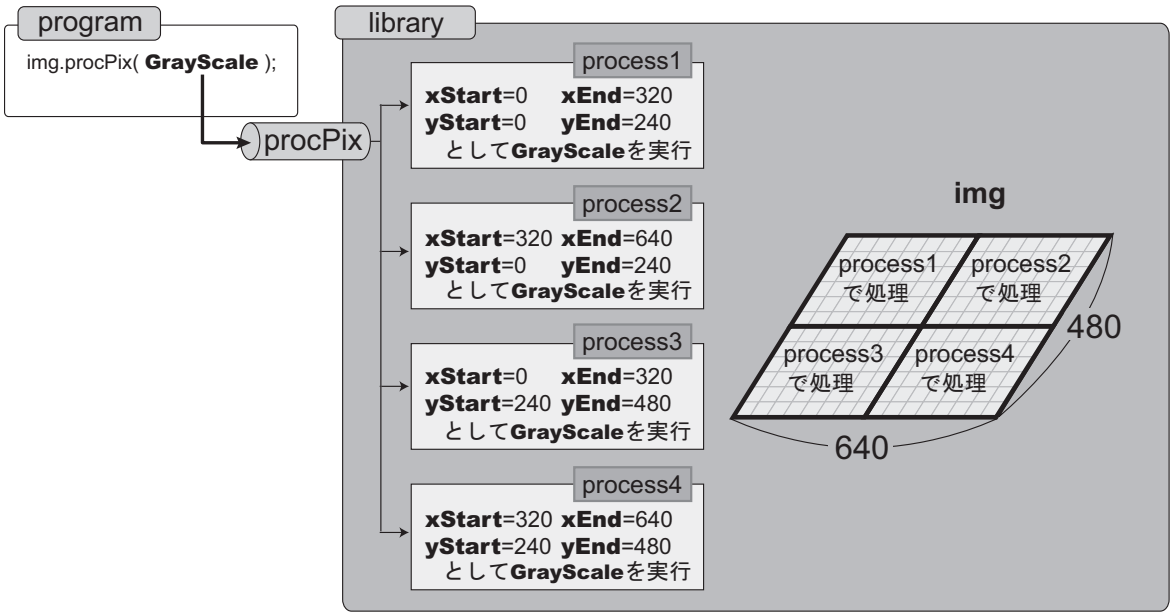
本章では，RaVioli を使用して記述した逐次プログラムを自動的に並列プログラムに変換する自動並列化プリプロセッサについて述べる．

4.1 自動並列化プリプロセッサの方針

これまでの情報システムの分野ではアプリケーションの実行に対して，限られた CPU リソースでいかに満足した結果が得られるかが重要視されて研究開発が進められてきた．しかし今日ダイの集積度が向上し，複数のコアを搭載するマルチコア，メニーコア環境や GPU が混載された環境が一般的になってくるに従い，コンピュータアーキテクチャやアプリケーションを改良することで，多数ある計算資源を如何に効率よく利用するかが重要な課題となってきている．さらにマルチコアや GPU は，研究や開発の分野で利用される高価なサーバだけでなく，個人で利用するような汎用 PC にも搭載されるようになってきた．このため今後は，幅広いプラットフォームでも CPU リソースを余すことなく稼働できるシステムが必要となってくる．そこで RaVioli では前章までに説明した機能に加えて，マルチコアプロセッサ上で余剰コアを有効利用で



(a) parallel image processing.



(b) parallel image processing with RaVioli.

図 6: 画像処理のデータ並列化のイメージ

きるような機能も提供する．

コアを有効利用し効率よく動作させる方法として複数コアによる処理の並列化がある．この並列化には，データ依存がなく独立した処理を並列化するタスク並列と，ループ処理のような多数のデータに対して同じ処理を並列に行うデータ並列がある．ここで画像処理は 1 画素単位の処理を，ループを用いて全画素に適用させるのが一般的で

あるため，データ並列に向いていると言える．グレースケール化はその典型例であり，データ並列化を適用した場合は**図 6**(a) のようになる．画像に対してグレースケール化を適用する際に画像データを 4 つに分割し，それぞれの分割画像を別スレッドで動作させることにより，4 並列化を実現している．しかしデータ並列はループ内のイテレーションが独立していることが保証されている必要があり，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に依存のある処理の場合は，並列化しても処理結果の正当性を保証できない．画像処理も例外ではなく，画素に対して行う処理の実行順序に依存のある (以降，順序依存のある) 場合は並列化できず，それぞれが独立でなければならない．またイテレーション演算の中に，ループ外変数に対するデータの読み出しや書き込みがあると，並列処理のタイミングによっては競合が起きる可能性がある．この一般的な解決方法に，ループ外変数に対する読み書きを相互排他的に行う排他制御がある．しかし排他制御を利用する場合，1 回のイテレーションの演算量が比較的少ない画像処理では，ループ外変数に対して読み書きする演算を排他的に処理すると，他のスレッドの待ち時間が増えてしまう．よって，並列数を上げていくにつれ並列度が見込めなくなる．もう一つの解決方法として，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をし，最後にデータを逐次的に統合する reduction 演算がある．ループ内のイタ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，画像処理の分野では一般的に reduction 演算を用いることが多い．reduction 演算を利用する場合，プログラマはループがどのように分割されるかをある程度知っておき，図 1 に示した OpenMP のプログラム例のように，一時的に定義した格納領域にあるデータをどのような演算で統合するのかを指定する必要がある．

以上で述べたように，画像処理のデータ並列処理システムを実装するためには逐次の画素処理ループを並列処理できるようにアルゴリズムを改良する必要があり，並列化の知識が必要となる．また，pthread のような並列処理ライブラリの基本的な利用方法の習得や，上記に示した実行の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となり，バグの混入等からも開発にかかる工数が増大すると考えられる．

そこで前章で示した RaVioli を用いて書かれた逐次画像処理プログラムを，並列プログラムに自動で変換するプリプロセッサを提案する．そもそも図 6(b) で示すように，プログラマは高階メソッドを用いて画像処理を適用するが，その際には画素単位の処理のみ書けばよく，ループはライブラリ内で制御されるため，プログラマから意識さ

れない部分で並列化ができる．このことから，高階メソッドにブロック分割によるデータ並列化機能を追加する．そしてプリプロセッサは，RaVioli によって書かれたプログラムの中で高階メソッドを呼び出している部分を見つけ，画像が並列に処理されるように書き換える．さらに，既存のライブラリにはなかった reduction 演算が必要となる処理の自動検出や reduction 演算ソースの生成及び適用を可能とする．これによりプログラマは並列処理に関する知識を持ち合わせていなくても，複数コアを有効利用したシステムが作成可能となる．

本プリプロセッサは変換を適用する前に，画像に対する処理毎に順序依存があるかどうかを確認し，検出されなければ RaVioli の仕様に沿った並列処理に書き換える．逆に処理の順序依存が検出された場合は，並列化した処理結果の正当性が無くなる旨をプログラマに通知し，逐次のまま処理するか並列に直すかを選択させる．そして順序依存の確認が終わった後に，reduction 演算を考慮した並列処理プログラムに変換する．

4.2 節では RaVioli が順序依存がある処理を検出しやすい特徴を持っていることを示し，順序依存の検出手法を説明する，4.3 節では既存のライブラリにはなかった reduction 演算を自動で検出可能とする手法を示し，それを用いた逐次プログラムの変換例を示す．

4.2 画素の処理順に依存した処理の検出

4.2.1 処理順に依存する画像処理と並列化

1 枚分の画像データになんらかの処理を施す際，通常は y 軸方向のループの中に x 軸方向のループを入れた 2 重ループの中に，1 画素に対する単位処理を入れることで，全画素に目的の処理を施す．RaVioli では処理対象となる画像データを並列数分にブロック分割し，それぞれのブロックを独立に処理させることにより，単位処理を並列に動作させている．しかし画素毎の処理に順序依存のある場合は，並列化をすることで処理結果が変わる可能性が高い．**図 7** は画素単位の処理に順序依存のある例である．このプログラムはモノクロ画像 `grayImg` を 2 値画像に変換し `binaryImg` に出力している．画素単位の処理として，`grayImg` の画素 1 つを閾値 `th` と比較して白 or 黒の判定をした後に，`th` を結果に応じて増減させる．画素を走査するループは画素配列の上端から始まるため，処理が下方に移るに従って全画素の平均に近い閾値で判定できるようになる．2 重の for を並列動作させるべく，ループの範囲をいくつかのブロックに分割させた場合，画素の処理順が逐次の場合と異なるために，下の方の画素に対して平均に近い閾値が用いられなくなる．これはある画素を処理する際に書き換えた閾値を，次の画素処理時に参照しているために起こる．

```
main(){
  int grayImg[WID][HEI];  //WID は画像の幅
  int binaryImg[WID][HEI];  //HEI は画像の高さ
  int x,y;
  int th=127;
  //グレースケール画像を grayImg へ格納
  for(y=0;y<HEI;y++){
    for(x=0;x<WID;x++){
      if(grayImg[x][y]>th){
        binaryImg[x][y]=255;
        th++;
      }else{
        binaryImg[x][y]=0;
        th--;
  }}}
}
```

図 7: 順序依存のある画像処理

ここで，そもそもなぜこのような順序依存のある処理が記述されるかについて説明する．画像処理におけるアルゴリズムとは本来，画素 1 つに対する処理を独立に定義するため，データ並列性を持っており，画素の処理順には依存しないはずである．しかし一般的に逐次手続き型言語で画像処理を記述する場合，これら画素の処理がループ等により全順序化されてしまう．しかしこれは記述言語の制約から来るものであり画像処理の本質とは異なる．一方 3.1.2 項で述べたように，RaVioli では画素 1 つに対する処理を独立に定義するのみでよいループレスな記述方法であり，全順序化がプログラム記述の段階で行われない．**図 8** の (a) はグレースケール画像の全画素値の和を求める RaVioli プログラムである．グレースケール画像 `grayImg` を読み出し，高階メソッド `procPix()` を用いて画素毎の処理を画像に適用する．画素値 `p` のクラスメソッドである `getR()` は RGB 色空間の R 値を返す関数であり，これを用いて R 値 (明度に相当) の和 `foo` を算出する．このように RaVioli を利用するプログラマは，順序依存のない 1 画素に対する処理のみを記述し，全順序化の過程を経ないため，結果的に並列性が失われず，自動並列化が比較的容易となる．

上記のように RaVioli では要素の処理順を規定しないため，プログラマは画素毎の処理を順序依存があるように書かないはずである．しかし，作成したアルゴリズムをプログラムに反映させる時に誤った記述をしてしまった場合や，従来の逐次手続き型言語の場合と同様に暗黙的に処理順を想定してしまった場合には，結果が処理順に依存するような記述をしてしまう場合が考えられる．たとえば図 8(b) では，高階メソッ

(a) 順序依存のない処理

```
int foo=0;

void bar1(RV_Pixel p){
  foo=foo+p.getR();
}

void main(){
  RV_Image grayImg;
  //入力画像を grayImg へ格納
  grayImg.procPix(bar1);
}
```

(b) 順序依存のある処理

```
int foo=0;

void bar2(RV_Pixel p){
  foo=(foo+p.getR())*2;
}

void main(){
  RV_Image grayImg;
  //入力画像を grayImg へ格納
  grayImg.procPix(bar2);
}
```

図 8: RaVioli プログラムでの順序依存のある処理と順序依存のない処理

ド procPix に渡される画素毎の処理 bar2 内で，大域変数 foo に画素値を足し 2 をかけた値を返しているが，このような処理は画素の処理順により最終的な foo の値が異なる．RaVioli ではこの順序依存のある処理を処理系で検出することで，バグ検出等の助けとする．

4.2.2 検出手法

画像処理は一般的に，2 値化の後にエッジ抽出をし，その画像データに対しハフ変換をするといったように複数の段階 (以下，これを画像処理の処理ステップとする) を経るが，RaVioli で書かれたプログラムが画像処理の各処理ステップで並列動作可能であるかを判断するために，プリプロセスの段階で順序依存のある処理を検出する．並列化の適用／不適用の判断は高階メソッド呼出しを単位として行う．たとえば画像全体に対してグレースケール化，ローパスフィルタ，二値化，エッジ検出処理を順に施すプログラムでは，それぞれ高階メソッドを使うことから，並列化の適用／不適用もそれぞれ個別に決定される．そして，図 8 のプログラムでの bar1 や bar2 のような，高階メソッドの引数である画素毎の処理関数 (以降この種の関数名を bar$n$ ： n は整数 とする) のプログラムを解析する．基本的に順序依存があるのは，関数の外側にある大域変数を利用している場合に限られるため，関数内で処理が閉じている場合は解析を行わない．また大域変数を利用している場合でも，それが読み出しのみであれば順序依存は無いとする．逆に画素毎の処理関数内で大域変数に対して読み出しかつ書き込みをしていれば，順序依存のある処理である可能性がある．よって，大域変数に対する代入や四則演算がある処理に対して順序依存の解析をする．現段階では，

(A) 大域変数に対して読み出しは無く書き込みのみである

(B) 読み書きのある大域変数に対して + もしくは – を使用し，なおかつ ∗ もしくは / を使用している

(C) if 文の条件式で使われている大域変数に対して，比較した変数と異なる値が代入されている

(D) 四則演算を施した大域変数を画素へ書き込んでいる

(E) ライブラリレベル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)

の 5 つのどれかを満たした場合に順序依存があるとしているため，これらの 5 つのうちのひとつでも検出されれば逐次と並列処理の結果に一貫性がないと判断する．以下，それぞれについて具体例を用いて説明する．

(A) について

```
int foo=0;

void bar3(RV_Pixel p){
  foo=p.getR()*2; //NG
}
```

このプログラムでは，大域変数 foo に対し書き込みのみがあり，読み出しはしていない．変数 p に最後に渡される画素値によって foo の最終的な値が変わってしまうため，順序依存があると言える．

(B) について

```
int foo1=0;
double foo2=0;

void bar4(RV_Pixel p){
  //foo1=(foo1+p.getR())*2; //NG1
  //foo2=foo2/p.getR()-2;   //NG2
  foo1=foo1+p.getR();              //OK1
  foo2=foo2/p.getR()*(p.getG()+2); //OK2
}
```

このプログラム上で NG1 と記述されている行は，foo1 に対して 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．NG2 の行は，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順序に依存ができてしまう．一方 OK1 と記述さ

れている行は foo1 に対して + のみを適用しており，OK2 の行は，foo2 に対して * と / のみを適用している．こ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．順序依存のない式は一般的に，大域変数に対して複合代入演算子の形で記述できる式が該当する．OK1 の行は foo1+=p.getR(); と書き換えることができ，OK2 の行は foo2*=(p.getG()+2)/p.getR(); と書き換えられる．なお * と / の順序によっては丸め誤差の影響から結果が多少変わってくることがあるが，今回は誤差の範囲内であるとしている．

(C) について

```
int foo=0;

void bar5(RV_Pixel p){
  //if(foo > p.getR()) foo=foo+p.getR(); //NG1
  //if(foo > p.getR()) foo=p.getG(); //NG2
  if(foo > p.getR()) foo=p.getR(); //OK
}
```

このプログラム上で NG1 と記述されている行は，条件式に使われている foo に対して加算をしている．画素の処理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 NG2 の行は，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．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．一方 OK と記述されている行は，いわゆる p.getR() の最小値を求めるものであり，p.getR() がどんな値であっても，最終的に foo に代入されている値は同じである．

(D) について

```
int foo=0;

RV_Pixel bar6(RV_Pixel p){
  foo+=p.getR();
  //return(p.setR(foo));  //NG
}
```

このプログラムでの bar6 は画素を返り値とする関数である．関数内では大域変数 foo に対して加算が行われており，その結果を画素 p の R 値に代入としている．そして p を返り値として出力している．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代入すると，画素の処理順によって最終的な出力画像が変わってくるため，処理

順に依存があるといえる．

(E) について

インクルードしたライブラリで定義された関数は，関数内でどんな処理がなされているか分からないため，ライブラリ関数の引数に大域変数が定義されていた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順序依存の有無に関する情報を持っているため，その情報を用いて判断する．

また画像処理には，少数ではあるが輪郭追跡など画素毎の処理に順序依存が必要になってくるものもある．たとえば RaVioli では輪郭追跡を procDirect() という高階メソッドを用いて記述するが，これは逐次で処理する必要があるため，プリプロセッサでは procDirect() 用いて作られた処理に対しては変換を施さない．このように，高階メソッドの種類によって並列化の適用／不適用を判断する．

4.3 並列処理プログラムへの変換

4.3.1 変換手法

プリプロセッサで RaVioli プログラムの画像処理の各処理ステップ に対して順序依存の有無を検出した後，順序依存の無い画像処理が並列に動作するよう変換する．**図 9** は，プリプロセッサにより変換を施す前と施した後のコードである．処理内容は，読み出したカラー画像に対してグレースケール化を適用するとともに画素値の平均値と最小値を求めている．変換前のコードでは 30 行目から始まる `main` 文中で `input_img` に画像データを格納し，その画像に対し高階メソッド `procPix` を用いて (45 行目)，画素毎の処理関数 `Ave` を画像全体に施す．8 行目で定義されている `Ave` では，画素 `p` の情報から輝度情報 `y` を算出して画素として返すという処理をする (11 行目) とともに，`pSum` に画素値を加算 (12 行目) し，`pCnt` を 1 画素カウントするためにインクリメント (13 行目) する．また画素の中での最小値を求めるため，`pMin` と `y` を if 文を用いて比較し，`y` の方が小さければ `pMin` に格納する (14∼16 行目)．そして高階メソッドの処理が終了した後，46 行目で画素値の総和と画素数から画素の平均値を求め，`pSum` に格納している．

プログラマが逐次プログラムを並列で動作させるようにするためには，プログラム中に`#parallel NUM` と記述する．`NUM` は並列数であり，図 9 の変換前の例では，1 行目で`#parallel 4` と記述し 4 並列となるように指定している．プリプロセッサはプログラムを読み出すと，`#parallel NUM` を探し，これを見つけると検出処理に移る．main

変換前

```
#parallel 4
int pSum;

int pCnt;

int pMin;

RV_Pix Ave(RV_Pix p){
  int r,g,b,y;
  p.getRGB(r,g,b);
  y=rint(0.3*r+0.59*g+0.11*b);
  pSum=pSum+y;
  pCnt+=1;
  if(y<pMin){
    pMin=y;
  }
  p.setRGB(y,y,y);
  return p;
}









int main(){
  RV_img* input_img;
  input_img=new RV_img();
  //画像ファイルを読み出し
  // input_img に格納
  pCnt=0;
  pSum=0;
  pMin=256;




  input_img=
     input_img->procPix(Ave);
  pSum=rint(PSum/pCnt);
  return 0;
}
```

変換後

```
#include<pthread>
int pSum,__initpSum;
__thread int __pSum;
int pCnt,__initpCnt;
__thread int __pCnt;
int pMin,__initpMin;
__thread int __pMin;
RV_Pix Ave(RV_Pix p,int __thN){
  int r,g,b,y;
  p.getRGB(r,g,b);
  y=rint(0.3*r+0.59*g+0.11*b);
  __pSum+=y;
  __pCnt+=1;
  if(__pMin>y){
    __pMin=y;
  }
  p.setRGB(y,y,y);
  return p;
}
mutex_t redMutex;
void __Ave(int __thN){
  mutex_lock(&redMutex);
  pSum+=__pSum-__initpSum;
  pCnt+=__pCnt-__initpCnt;
  if(pMin>__pMin){
    pMin=__pMin;
  }
  mutex_unlock(&redMutex);
}
int main(){
  RV_Image* input_img;
  input_img=new RV_img();
  //画像ファイルを読み出し
  // input_img に格納
  pCnt=0;
  pSum=0;
  pMin=256;
  __pCnt=pCnt;
  __pSum=pSum;
  __pMin=pMin;
  __initpCnt=pCnt;
  __initpSum=pSum;
  input_img->reduction(__Ave);
  input_img=
     input_img->procPix(Ave,4);
  pSum=rint(PSum/pCnt);
  return 0;
}
```

図 9: 変換前の逐次コードと変換後の並列コード

文中で呼び出されているすべての高階メソッドに対して 4.3 節で述べた条件の下で並列化可能であるか否かを判断し，並列化可能であれば変換処理を施す．図 9 のコードでは 46 行目の procPix が並列処理可能であると認識され，procPix の引数に並列数である 4 を加える．

プリプロセッサによって各処理ステップの高階メソッドを並列処理するよう変換したら，次に reduction 演算が必要か否かの判定にうつる．reduction 演算とは並列処理において，複数のサブタスクが 1 変数に対する読み出しおよび書き込みをする際の競合を回避するための処理である．各サブタスクが 1 変数の代替として定義した一時変数 (以下代替変数とする) に対して読み書きを行い，並列処理終了後に reduction 演算を用いてすべての代替変数を逐次的に統合する．従来の並列処理ライブラリでは reduction 演算の対象となる変数や統合の演算方法をプログラマがディレクティブ形式で指定していたが，reduction 演算自体を自動で検出するために高階メソッドの引数として呼ばれる画素毎の処理関数内を解析する．競合の発生するプログラムである条件として，4.2.2 項で示した大域変数に対する読み書きが挙げられるため，同じ大域変数に対して読み出しと書き込みの両方がなされている場合は，その大域変数を reduction 演算 の対象であると判断する．

```
int foo1=0,foo2=0,foo3=0;
RV_Array fooArray;

void bar5(RV_Pixel p,RV_Coord coord){
  if(foo > p.getR()){    //(4) 読み出しのみ
    foo1=p.getR();       //(1) 書き込みのみ
    foo2=foo2+p.getG();  //(2) 読み出しおよび書き込み
    foo3+=1;             //(3) 読み出しおよび書き込み
  }
  fooArray.incData(coord); //(5) 読み出しおよび書き込み
}
```

たとえば (左辺) = (右辺); といった一般的な式において，上の図中の (1) のように (左辺) だけが大域変数があれば書き込みと判断し，(2) のように (右辺) にも大域変数があれば読み出しと書き込みがあると判断する．なお (3) のように，+ = や − = といった複合代入演算子を用いた演算は，読み出しと書き込みの両方を行っている．また (4) のように if 文や for 文，while 文の条件式に大域変数が使われている場合も，その大域変数への読み出しと判断する．さらに，大域変数が RaVioli で定義されているクラスメソッドの引数として使われていれば，メソッドの仕様に沿って大域変数に対

する読み書きを判断する．(5) は配列クラス RV_Array のオブジェクト `fooArray` 内の `coord` で示す位置の配列要素に対してインクリメントをしている．これは演算としては + = と同義であるため，読み出しと書き込みであると判断できる．以上が大域変数への読み書きの判断方法であり，同じ大域変数に対して読み出しと書き込みの両方がなされている場合には，その大域変数が reduction 演算の対象となる．また大域変数に対する読み書きは，演算子だけではなく関数の中で行われることもある．使用されている関数が同じプログラム内で定義されている場合は，その関数内も解析する．しかし使用されている関数がプログラム内ではなくライブラリなど外部で定義されたものである場合は，関数の定義を探す事が困難である．よってその関数の実引数に大域変数が含まれていれば，解析の段階で並列化が不可能であるとしている．図 9 の例では，8∼19 行目で定義されている画素毎の処理関数 `Ave` 内が解析部分である．関数内の 12 行目の式 `pSum=pSum+y;` において，`pSum` に対する読み書きがされ，また 13 行目の式 `pCnt+=1;` では `pCnt` に対する読み書きがされている．また `pMin` は 14 行目の条件式内で読み出されており，15 行目で書き込まれている．よってこの 3 つの大域変数が reduction 演算の対象 (以下 reduction 大域変数) となる．

検出された reduction 大域変数は，並列処理時に各サブタスクが読み書きを同時に行うと競合が発生する変数である．このため reduction 大域変数には自身の他に，各サブタスク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．ここで本プリプロセッサは，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，変数の前に `__thread` と書くことにより，その変数のコピーを各スレッド毎に保持する機能である．スレッドが生成されるタイミングで作られ，スレッドが終了すると同時に消滅する．よって，代替変数として `__pSum`, `__pCnt`, `__pMin` を定義する際に，それぞれのスレッドにコピーが生成されるように `__thread` を指定しておく．ただし __thread 指定子は，テンプレートを用いるクラスオブジェクトをサポートしていない．例えば 3.2.2 項で示した RV_Array クラスは，テンプレートを用いて配列要素の型を指定することから，`__thread` を用いることはできない．このためテンプレートを用いるクラスオブジェクトは代替変数を並列数分，別途配列として定義する．スレッドは代替変数を自分のローカル変数として読み書きする代わりに，代替変数の配列の，スレッド ID と対応したインデックスの要素に対して読み書きをする．具体的には，8 行目の関数 `Ave` の引数にスレッド ID を持つ引数 `__thN` を宣言し，これを用いて大域変数である代替変数の配列要素に読み書きする．またこれらの代替変数を利用する前には，元の変数と同じ初期値を代入しておく．

次に reduction 演算の検出に移るが，その前に，検出しやすいようコードを単純化しておく必要がある．コード内のコメントアウトやタブ及び空行の削除，1 行に対して 1 演算のみにすることはプログラムの読み出しの直後に行うが，それとは別に，大域変数に対して読み書きをしているテンポラリ変数をなるべく省略できるようにする．

```
int foo=0;
void bar6(RV_Pixel p){
  int tmp;
  tmp=p.getR()+p.getG()+p.getB();
  foo=foo+tmp;
}
```

→

```
int foo=0;
void bar6(RV_Pixel p){


  foo=foo+p.getR()+p.getG()+p.getB();
}
```

ここに示す図の左側のコードでは，関数内で定義した変数 tmp に対して値を代入し，その変数を大域変数 foo に書き込んでいる．こういった記述に対しては，図の右側のコードのように大域変数に対して直接演算が適用されるようにする．

上記の変換を適用した後は reduction 演算の生成および適用に移る．reduction 演算の生成とは reduction 大域変数を含む演算式を抽出して，その演算式を変換することであり，検出された演算式は並列処理後の代替変数の統合に使われる．**図 10** は変換前のコード内の関数 Ave から reduction 演算を生成する過程を表している．図中①で抽出した reduction 大域変数を含む演算式の中で，四則演算を含む式については reduction 大域変数に対して複合代入演算子の形をとる式に変換する必要がある．pSum, pCnt, pMin が使われている行が該当するが，pSum=pSum+y; を pSum+=y; に書き換える．また if 文など制御文で用いる条件式において，関係演算子をはさんで右側に reduction 大域変数があった場合は，左側にくるよう書き換える．その後は図中②のように，reduction 大域変数を用いた演算式 (図中 detectCode) を__(アンダーバー 2 つ) で始まる代替変数に対して読み書きを行う部分 (図中 seqCodePar) と，代替変数から最後に統合をする部分 (図中 sepCodeSeq) とに分割する．この分割のメカニズムについては次項で詳しく述べる．結果として図 9 の変換後のコードのように関数 Ave 内に sepCodePar のコードが適用される．一方で図 10 の sepCodeSeq で示す reduction 演算部分は関数として別途定義し，高階メソッドが実行される前の段階で間数ポインタをライブラリ側に渡しておく．図 9 の変換例では，21 行目の__Ave() が reduction 演算として定義された関数にあたり，mutex を用いることにより逐次的な処理を実現している．ここで 20,22,28 行目

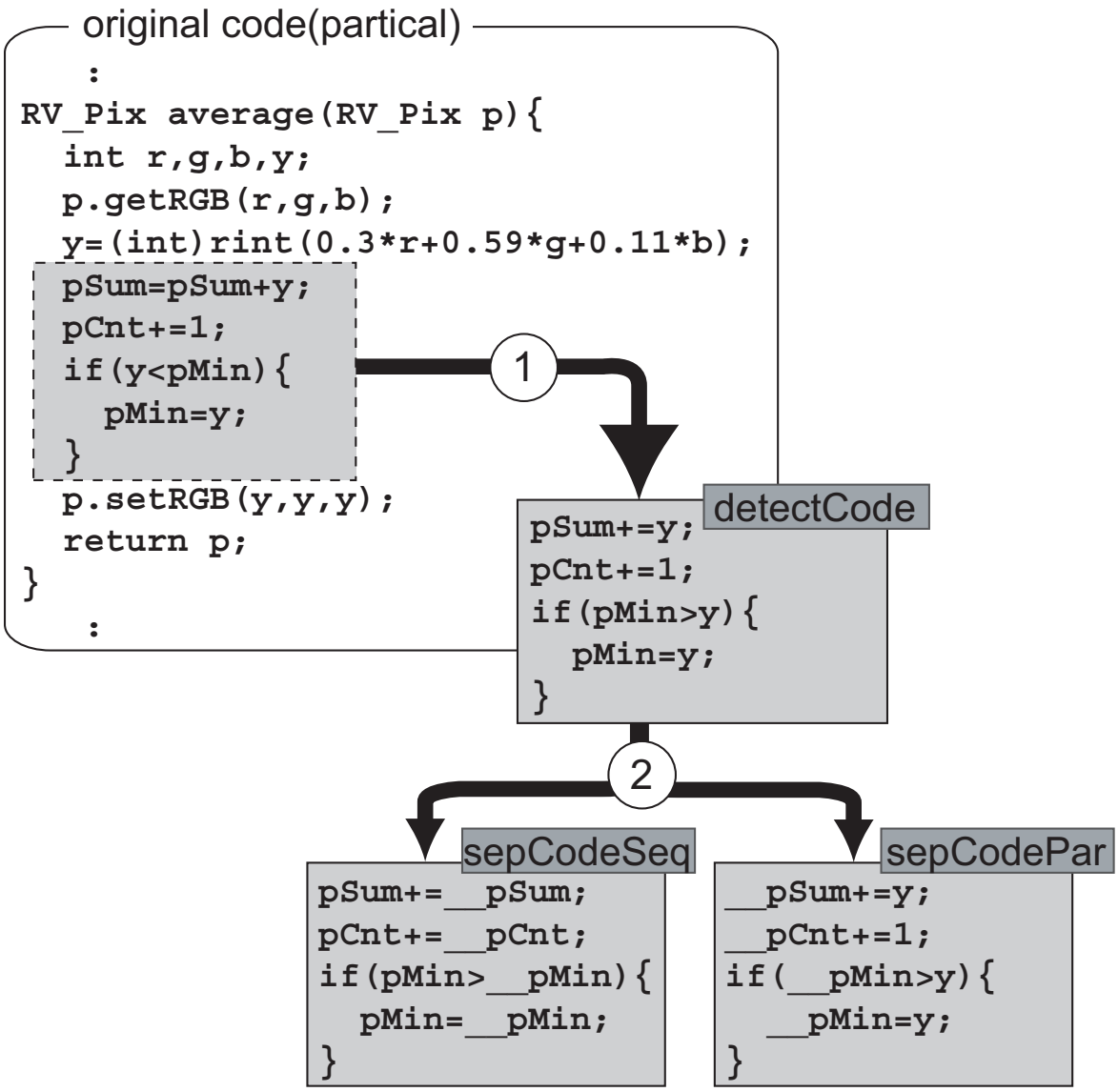


図 10: 処理の分割による reduction 演算の生成

は mutex の前に pthread_ と付け加える必要があるが，紙面の都合上省略する．__Ave() の関数ポインタは，45 行目で procPix が呼び出される前に，reduction という関数を通して RaVioli 側に渡される．こうすることにより，RaVioli は__Ave() をサブタスク毎に実行する．また並列処理の前にそれぞれの代替変数に初期値を代入していたことから，この初期値を並列数分除去する必要がある．初期値を除去する方法は reduction 大域変数に対する複合代入演算子により決定する．reduction 大域変数に対して加算または減算をしていたなら初期値を並列数分減算し，乗算または除算をしていたなら初期値を並列数分除算すればよい．図 9 では初期値が事前に代入されている__init で始まる変数を用いて，23,24 行目で pSum, pCnt 共に減算が適用されている．

以上のような reduction 演算の適用および変換方法を逐次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逐次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．

4.3.2 reduction 処理検出のメカニズム

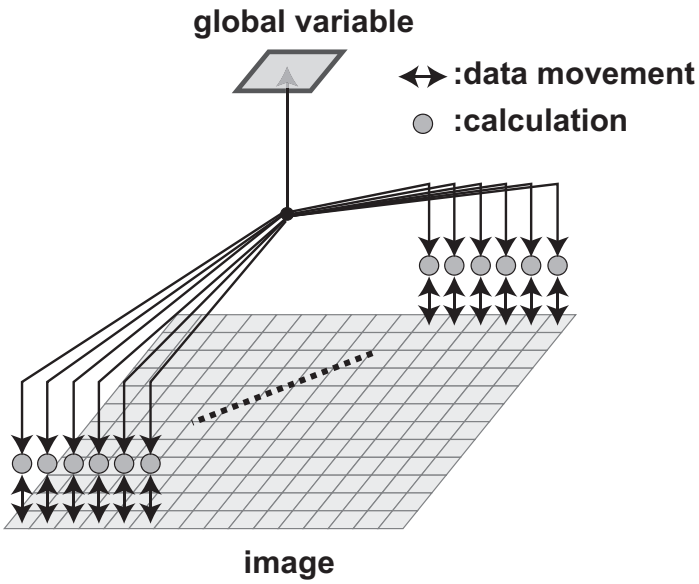
前項では並列処理する上で必要な reduction 演算を適用するために，大域変数に対して読み書きしている処理を図 10 のように分割した．この図に関して，sepCodePar は画素毎の関数内に適用するため並列に動作し，sepCodeSeq は画素の走査の終了後に逐次で処理するが，本稿ではこの分割方法が成り立つ理由とその条件について示す．

図 10 の detectCode を逐次で動作させることを考える．例として 6 画素からなる画像に対して関数 Ave を適用することを考える．引数である画素オブジェクト p に，始めに渡される画素値を p1，次に渡される画素値を p2 とし，p1 から求めた変数 y の値を y1，p2 から求めたものを y2 として，以降はこれに準じて p1〜p6 および y1〜y6 を定義する．ここで，pSum に対して最終的に書き込まれる結果を一行の式で表すと，

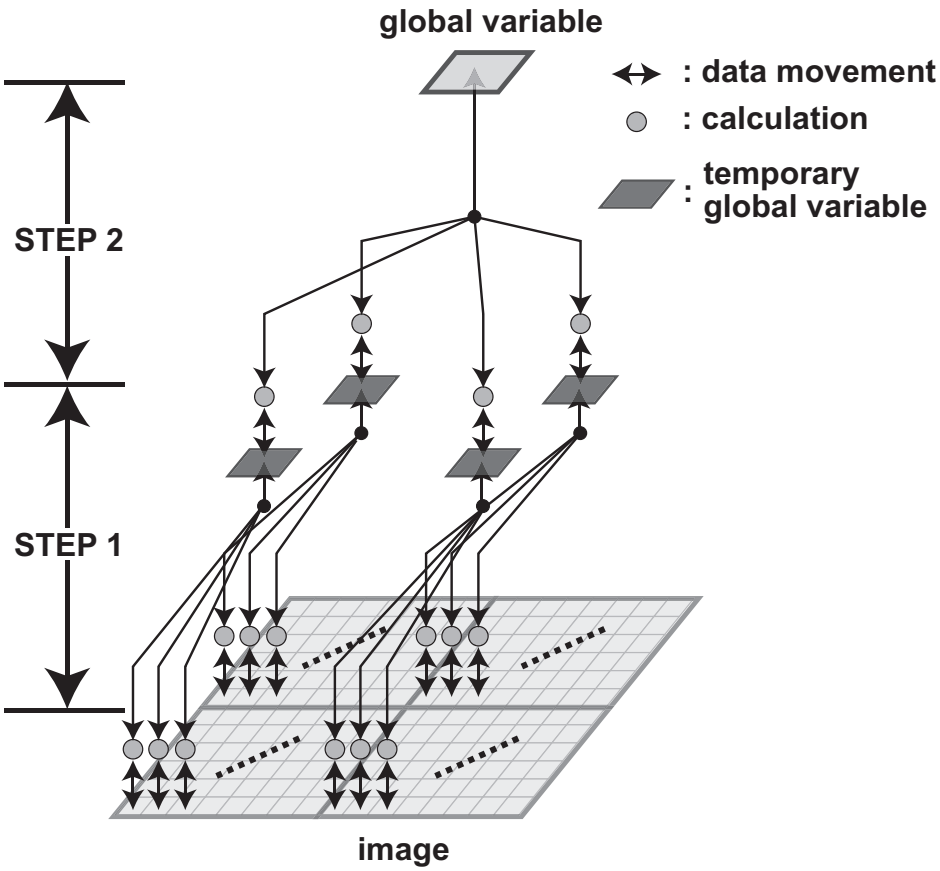
$$pSum = ((((((pSum + y1) + y2) + y3) + y4) + y5) + y6);$$

となる．この処理が処理順に依存していないことは処理順依存の検出をパスしたことから分かっているため，任意に選んだ 2 つの yn を交換しても pSum に出力される結果は変わらない．このことから解釈をかえれば，yn および $+=$ の演算は，データ並列性があるとも言える．この概念を一般化したものが**図 11**(a) である．この図は画像データと大域変数とのデータのやり取りおよび演算の並列性を示している．先の pSum への代入式と照らし合わせると，yn は image，$+=$ が calculation，そして pSum が global variable にそれぞれ対応する．

代替変数をはさんで分割した式を，図 11(a) に適用した場合が図 11(b) である．処理



(a) image processing with sequence.



(b) image processing with parallel.

図 11: 推移律から抽出される reduction 処理のイメージ

順は下から上であり，1 ステップ目で各画素値との演算を代替変数に対して行い，2 ステップ目で代替変数に対して同じ演算をし，結果を大域変数に書き込む．1 ステップ目のそれぞれの代替変数に対して読み書きする演算を並列して行い，2 ステップ目では逐次的に演算を行う．ここで，この処理の分割が成り立つのは，演算が推移律を満足するためである．一般的に，演算 R に対し，

$$\forall a, b, c \in X, aRb \vee bRc \rightarrow aRc$$

を満足する場合に推移律が成り立つ．ここで，このような推移律が成り立つということは，aRc の処理を aRb と bRc に分割できるともいえる．このため，上の式の b にあたる変数を代替変数として演算を分割し，aRb を並列処理した後に bRc を reduction 演算として逐次的に処理することができる．このように処理を分割することにより，逐次画像処理を reduction 演算を含めた並列画像処理に変換している．

5 評価

5.1 RaVioli の記述能力と動作の評価

動画像に対する処理のサンプルとして顔検出プログラムを用い，RaVioli の記述能力や解像度変更への対応の評価を行った．

5.1.1 評価プログラム

RaVioli の動画像処理能力を評価するにあたり，サンプルプログラムとしてハフ変換を使った複数人の顔検出プログラム [14] を実装した．

まず背景画像と各時刻での入力画像における色の差分値を求め，差分値が閾値以上なら黒，閾値未満なら白とした 2 値画像を得る．次に，画像中で人が存在している可能性の高い領域を検出するために，縦方向のスキャンラインに対し，それぞれに含まれる黒の画素数をカウントし，画像の幅を区間とした黒の画素数の分布を算出する．黒の画素数の和が閾値以上あるラインが一定数以上続けば，その範囲は人が存在する可能性があると判断し，処理対象範囲とする．また同じ 2 値画像から，エッジの部分だけを黒とし他を白としたエッジ抽出画像を得る．エッジ抽出画像中の処理対象として指定された範囲毎に，ハフ変換を用いた円検出を適用する．検出された円のうち，円を構成する画素数の多い上位 5 つを顔の候補とし，5 つの円のうち領域内で一番上の位置にある円を顔とする．以上の顔検出プログラムを RaVioli を用いて記述し，動作させることにより評価をおこなった．

5.1.2 RaVioli 不使用/使用の比較

RaVioli を使用した場合と使用しない場合それぞれの，プログラマが記述するプログラムを**図 12** に示す．なお顔検出処理プログラムの中で，背景差分のソースのみを抽出して掲載した．また同プログラムのハフ変換のソースは付録 A.1 に掲載する．

具体的には，背景フレーム `bgFlm` と当該フレーム `newFlm` との r, g, b それぞれの差分の絶対値 `difr`, `difg`, `difb` を算出し，それらの値のいずれかが閾値以上である場合はその画素を黒に，そうでなければ白にするという処理を一定フレーム数毎に行うプログラムである．なお，プログラム中に出てくる閾値は 77(抽象濃度では 300 に相当) とした．

RaVioli を使用しない場合のプログラムにおいて，3 行目の `for` ループは繰り返しフレームをキャプチャするための記述である．キャプチャを行う関数 `readFlame()` を使って読み込んだフレーム `newFlm` と，事前に取得したフレーム `bgFlm` を用いて，2 重の `for` ループで背景差分を行う (5,6 行目)．ここで，フレームレートや画素数を動的に変動させる際には，インクリメント幅である `ST` や `SI` を動的に変動させるプログラムを追加する必要がある．

これに対し RaVioli を使用する場合，RV_Streaming クラスに定義されている高階メソッド `proc1Frm()`，及び RV_Image クラスに定義されている高階メソッド `compImg()` を用いる．18 行目の `proc1Frm()` には 1 フレームを処理単位とした関数 `interframe()`(プログラムの 12 行目) を渡すだけでよく，また 14 行目の `compImg()` には背景フレームと当該フレームにおける同位置の 2 つの画素 `newP` と `bgP` を処理単位とした関数 `compare()`(プログラムの 1 行目) を渡すだけでよい．このように，プログラマがフレームレートや構成画素数の値を利用することなく，RaVioli を使用しない場合と全く同じ処理を記述することができる．また，本来であれば画素数やキャプチャフレームの位置などを強く意識して記述する必要のある部分であるループなどの繁雑な箇所が省かれる．これによりバグ混入の可能性を低下させるとともに，RaVioli 内で自由に解像度を変更することができ，演算量の調整が可能になる．

5.1.3 処理解像度が変動した場合の動作

5.1.1 項で示した顔検出プログラムを RaVioli を用いて記述し，処理画素数の解像度ストライド S_I を変えて動画像処理を行った結果を**図 13** に示す．(a) $S_I = 1$, (b) $S_I = 2$ (処理画素数 1/4) , (c) $S_I = 3$ (処理画素数 1/9) とした場合では画像中の顔の位置に円が描かれてことが分かる．この結果から，プログラムを一切変更することなく解像度の変更に対応できており，期待した結果が得られることを確認した．

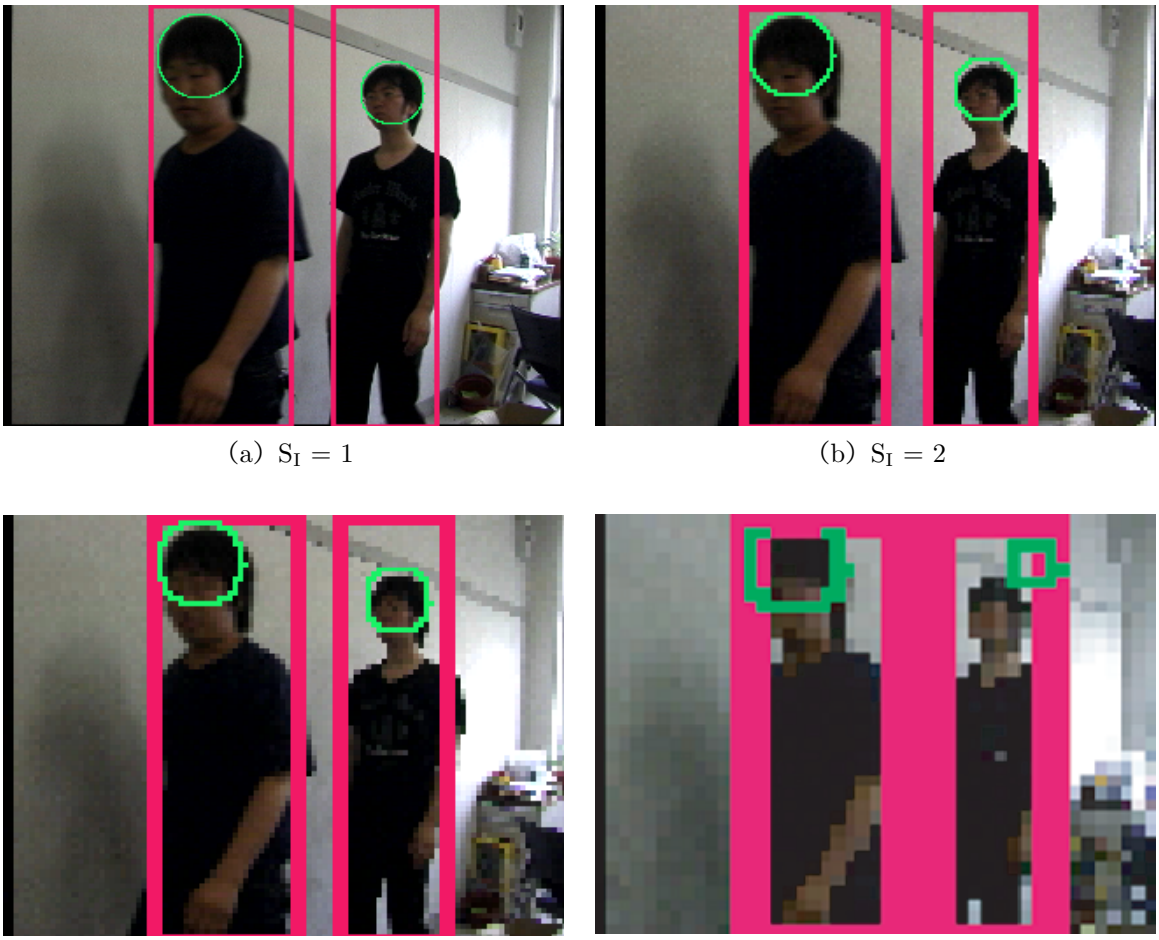
RaVioli を用いない記述

```
int main(){
  IMG bgFlm; //背景画像（事前に取得）
  for(;;CapFlm += ST){
    newFlm = readFrame(CapFlm);
    for(y = 0; y < img.H-SI; y += SI){
      for(x = 0; x < img.W-SI; x += SI){
        int difr, difg, difb;
        difr = abs(newFlm[x][y].R - bgFlm[x][y].R);
        difg = abs(newFlm[x][y].G - bgFlm[x][y].G);
        difb = abs(newFlm[x][y].B - bgFlm[x][y].B);
        if( difr > 77 || difg > 77 || difb > 77)
          newFlm[x][y].R=
          newFlm[x][y].G=
          newFlm[x][y].B=0;
        else newFlm[x][y].R=
            newFlm[x][y].G=
            newFlm[x][y].B=255;
}}}}
```

RaVioli を用いた記述

```
RV_Pixel compare(RV_Pixel newP, RV_Pixel bgP){
  int r, g, b, br, bb, bg, difr, difg, difb;
  newP->getRGB(r, g, b);
  bgP->getRGB(br, bg, bb);
  difr=abs(r-br); difg=abs(g-bg); difb=abs(b-bb);
  if( difr > 300 || difg > 300 || difb > 300)
    newP->RGBabs(0, 0, 0);
  else newP->RGBabs(1000, 1000, 1000);
  return(newP);
}
RV_Image bgFlm; //背景画像（事前に取得）
void interframe(RV_Image newFlm){
  RV_Image outFlm;
  outFlm=newFlm->compImg(compare, &bgFlm);
}
int main(){
  RV_Streaming video;
  video.Proc1Frm(interframe);
}
```

図 12: RaVioli 不使用/使用の場合のプログラム（部分）



(a) $S_I = 1$　(b) $S_I = 2$

(c) $S_I = 3$　(d) S_I=7(処理の失敗例)

図 13: 各解像度における出力画像

また S_I の値を上げていくと，S_I=7 の時点で画素数の減少により情報量が少なくなりすぎたために，抽出された円と顔の部分が一致せず認識に失敗した．この失敗例を図 13（d）に示す．顔検出のような画像から特徴を検出する処理は，一定の処理画素数を保持する必要があるため，処理画素数を優先させるように優先度パラメータを指定する必要がある．

5.1.4 RaVioli の適用範囲

プログラマは 3.2.2 節で述べたクラス及び高階メソッドを使用することで，処理画素数や処理フレームレートを意識することなく，さまざまな動画像処理プログラムを記述することができることを，サンプルプログラムを作成することにより確認した．画像処理においては，閾値判定による二値化やグレースケール化といった画素毎に独立した処理，メディアンフィルタによるノイズ除去やゾーベルフィルタによる輪郭抽出等の近傍処理，テンプレートマッチングによる顔検出などの処理が記述可能であった．また画像以外の配列を利用した画像処理として，ハフ変換による直線抽出などの処理も記述可能であった．さらに画素の色情報を使った処理に限らず座標の変換として，回転や拡大縮小や台形への変換など画像の形を自由に変更できることを確認した．また本論文中には示していないが，画素の処理順を決定する高階メソッドを利用することで，2 値化画像における輪郭追跡や領域のような逐次追跡型の処理が記述できることも確認している．

動画像処理の例として，1 フレームに対する処理を毎フレームに適用する処理や，背景差分やフレーム間差分といった 2 枚の画像を比較するような処理が記述可能であった．

一方で，二次元画像からの三次元モデルの再構成や，投影による二次元画像への変換については未だ検討段階である．今後ワイヤーフレームモデルやサーフェスモデルのような三次元モデル [15] を実現するインターフェースの実装，また二次元データとの変換が可能となるメソッドの生成により実現可能であると考えている．

また RaVioli では，動画像処理において単純に画素およびフレームをスキップして処理を適用する方法を取るため，高周波成分情報の損失など，出力結果の精度低下という悪影響を及ぼすおそれがある．例えばゾーベルフィルタを用いたエッジ検出において，スキップされた画素の輝度値が，隣り合った処理対象画素の輝度値と著しく異なっていても，エッジとして検出することができない．またオプティカルフローを用いた移動物体のトラッキング等においては，移動物体の進行方向が頻繁に変わったり動作が速くなると対応できなくなる．もちろん，画素をスキップする際にバイリニア法やハイパキュービック法など一般的な手法を用いて，スキップした画素やフレーム

の情報を補完し処理対象に反映させることにより，出力結果の精度を一定のレベルで保証することはできる．しかし RaVioli が行う解像度変更は，あくまで演算量低減のためのものであり，解像度変更自体のために演算量を増加させるべきではない．そもそも RaVioli の目的は，CPU リソース量に制限がある汎用 PC において動画像処理のリアルタイム性を維持する上で，プログラマに負担を与えず演算量を自動的に削減することであり，また多少精度が落ちても処理を成立させること，そしてなるべく動画像処理プログラムの意図に応じた処理結果を出力することである．このため RaVioli は「優先度」というインタフェースを提供している．プログラマはこの優先度により，動画像処理プログラムにとって本質的に重要である解像度（1 フレームの構成画素数またはフレームレート）を維持することができる．

5.2　動画像処理のリアルタイム性の評価

処理画素数と処理フレームレートの調整による疑似リアルタイム処理の評価を，フレーム間差分を用いて行った．用いたシミュレーション環境を**表 2** に示す．

サンプルとして用いたフレーム間差分プログラムは，時間軸上で隣り合う 2 枚のフレームのすべての画素において，RGB それぞれの差分の絶対値をとり，抽象濃度 100(RaVioli 不使用の場合は 25) より大きい場合はその画素を白，それ以外の画素を黒とすることで移動体を検出する処理である．

カメラから 30fps で転送される解像度 320×240 のフレームをキャプチャし，フレーム間差分プログラムを，空間解像度と時間解像度の優先度を変更して適用することで評価を行った．それぞれ優先パラメータを設定した場合の処理画像，出力結果，および出力ピクセル数と出力フレームレートの変化を**図 14** に示す．

（a）は出力フレームレートと出力画素数の優先度パラメータ (P_I, P_T) を (1, 0) とした場合, (b）は (0, 1) とした場合, (c）は (7, 3) とした場合の，処理開始から 6 秒後までの時間変化を表したグラフである．なお処理開始の 2 秒後から 2 秒間，別プロセスとして無限ループを行うプロセスを動作させ，使用可能な CPU リソースを減少させた．(a）では出力フレームレートは常に最大値が維持されている一方で，出力画素数は負荷を与えた時点で自動的に低下して行き，約 4.5 秒後には元の画素数に戻る様子が読みとれる．(b）においても同様に，出力画素数は最大値を維持しつつ，出力フレームレートは負荷を与えた時点で自動的に低減し，その後に元のフレームレートの上昇が読み取れる．また（c）も，負荷が与えられた時間において優先度に沿った解像度の削減が行われていると言える．

表 2: 動画像処理シミュレーション環境

CPU	AMD Opteron Dual-Core (2GHz)
Memory	2GB
Camera	SONY DCR-TRV900
Capture board	I·O DATA GV-VCP2M
Format	NTSC
Interface	S-video (S1)

以上の結果から，RaVioli がプログラマの指定する優先度に基づき正しく自動負荷調整を行えること，および処理画素数や処理フレームレートが変動した場合でもプログラムが正しく動作することを確認した．

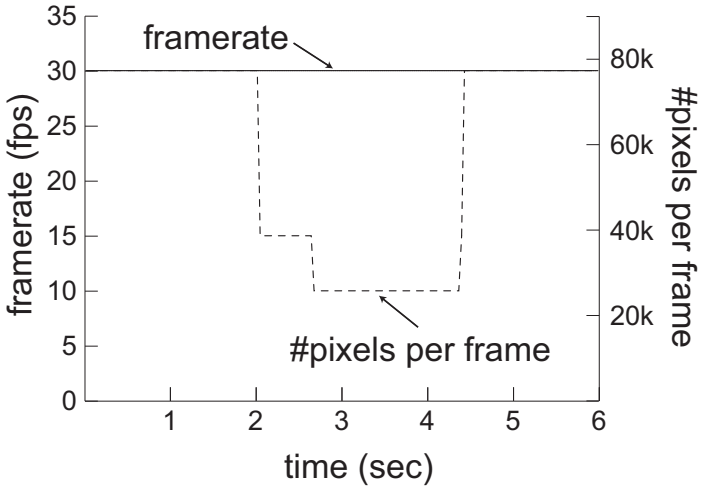
5.3 自動並列化機能の評価

RaVioli で書かれたさまざまな逐次プログラムに対してプリプロセッサを適用することで，自動並列化の適用範囲を明らかにした．またプリプロセッサにより並列化した様々なプログラムをマルチコア環境で動作させ，並列処理による速度向上の評価を行った．

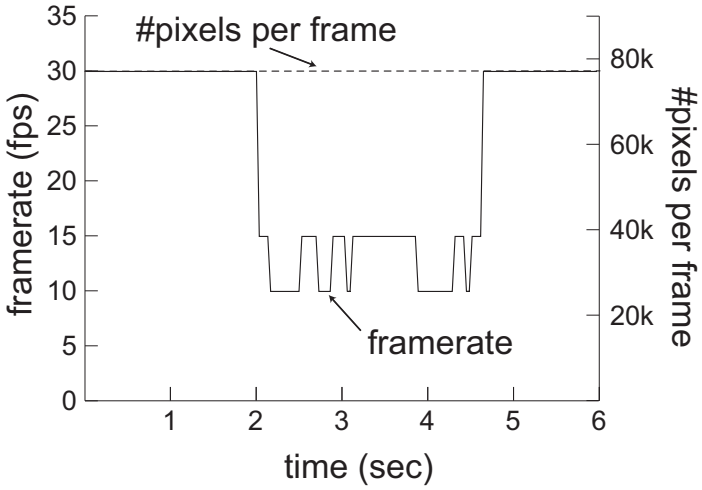
5.3.1 プリプロセッサの適用範囲

本プリプロセッサは perl で実装しており，RaVioli を用いて書かれた逐次画像処理プログラムを引数として受け取る．そして`#parallel` n (n は並列数) という記述が見つかった場合には，並列画像処理プログラムに変換し，出力する．評価として，`#parallel` n と書かれたさまざまな逐次プログラムをプリプロセッサの引数に渡し，出力されるプログラムが並列で動作するのかを確認した．

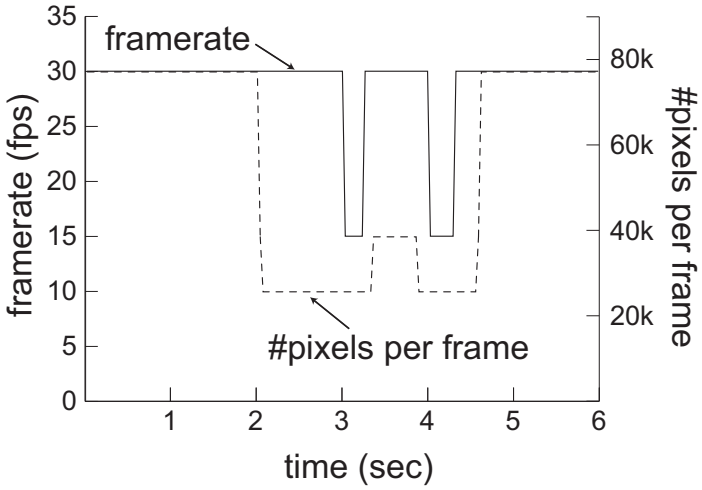
まず画像の二値化とグレースケール化に適用することで，procPix() を使う処理が正しく変換されている事を確認した．またラプラシアンフィルタによるエッジ抽出，ハイパスフィルタ，voronoi 図の生成に適用することで，画素毎に独立した処理には広く適用が可能であることを確認した．voronoi 図の生成の逐次プログラムを，プログラム例として付録 A.2 に掲載する．次に，競合の起りうる処理に対して，結果の整合性が取れるように reduction 演算ソースが生成され，適用されているかを検証した．画素値や画素数をカウントする処理や，ハフ変換による中間データ作成処理を適用した結果，処理の整合性が保証されるよう書き換えられていることを確認し，さらに処理結果や



(a) $(\mathrm{P_I}, \mathrm{P_T}) = (1, 0)$



(b) $(\mathrm{P_I}, \mathrm{P_T}) = (0, 1)$



(c) $(\mathrm{P_I}, \mathrm{P_T}) = (7, 3)$

図 14: 優先度を設定した場合の解像度の時間変化

表 3: 並列化シミュレーション環境

	Solaris/T1	Solaris/Core2Q	Fedora/Core2Q
OS	Solaris10	Solaris10	Fedora core 9
CPU	Sun UltraSPARC T1	Intel Core2 Quad (Q9650)	Intel Core2 Quad (Q9550)
クロック周波数	1.00GHz	3.00GHz	2.83GHz
core 数	8	4	4
並列 thread 数 (コアあたり)	4	1	1
memory	16GB	8GB	8GB
compiler	Sun Studio 12 (Sun C++ 5.9)	Sun Studio 12 (Sun C++ 5.9)	GNU g++ 4.3.0
compiler option	-fast -m64 -chip=ultraT1	-fast -m64 -xarch=sse3	-fno-rtti -O3 -fomit-frame-pointer -march=core2 -funroll-all-loops
thread library	pthread	pthread	pthread

中間データが逐次処理の場合のそれと同じ値であったことから，値の正当性が保証されていることを確認した．さらに処理に順序依存があるプログラムを作成し，プリプロセッサを適用した場合には，並列処理プログラムへの変換は行わずユーザに警告を提示していることを確認した．

なお本プリプロセッサは，現段階では動画像処理プログラムには対応していない．しかし RaVioli で書かれる動画像処理プログラムは，フレーム 1 枚もしくは時系列で隣り合うフレーム 2 枚を用いた処理をプログラマが記述するため，画像処理に対する変換手法と同じアルゴリズムを用いればよい．よって，動画像処理への適用は容易に実現可能である．

5.3.2 自動並列化による速度向上

マルチコア環境として Sun UltraSPARC T1 プロセッサおよび Intel Core2 Quad を用いて評価を行った．各評価環境の詳細および用いたコンパイラや thread 生成に用い

たライブラリ等の情報を表 3 にまとめる．そしてこれら 3 台の計算機を用いて，並列数を変化させた場合のさまざまなサンプルプログラムの処理にかかる時間を測定した．測定の対象としたサンプルプログラムは，以下の 4 つである．なお括弧内はグラフ上で用いる略称である．

- 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て領域毎に分けるボロノイ図の作成 (voronoi)
- 近傍画素を使ったフィルタリング処理であるラプラシアン (laplacian)
- 画素の平均値の算出処理 (pixAverage)
- ライン抽出プログラム内のハフ変換処理 (hough)

なお，画素値の合計処理およびハフ変換は競合の発生する可能性のある処理であり，reduction 演算が適用されている．今回の測定ではファイルの書き込みおよび読み出しの時間は計測しておらず，処理ステップにあたる部分，いわゆる高階メソッドの呼び出しからその終了までの時間を計測した．ただし reduction 演算および同処理に使う変数の初期化や，並列化する際のスレッド生成，終了待機および解放も計測の対象としている．またハフ変換を用いたライン抽出では，画像に対する前処理として閾値との比較による二値化やエッジ抽出などが含まれているが，それらは測定の対象とせず，$\rho - \theta$ 2 次元空間への投票部分のみの時間を計測した．

それぞれのサンプルプログラムを評価したグラフの見方を示す．グラフはそれぞれの逐次プログラムの処理時間を 1 として正規化した場合の高速化率を示している．また UltraSPARC T1 は，8core で 32thread が並列実行可能であるため，Solaris/T1 では並列数を 2,4,8,16,32 と変化させた場合のそれぞれの速度を測定した．一方 Solaris/Core2Q と Fedora/Core2Q は 4core 構成であるため，並列数を 2,4 として測定した．

voronoi を各計算機で動作させた結果を**図 15** に示す．グラフから，並列数 2 から 8 において台数に応じた結果が得られた．ここで UltraSPARC T1 は 1core で 4thread が動作可能であり，1core で逐次的に処理をするよりも高速に処理が可能であるという特性を持つ．これはパイプラインストールの際の待ち時間に他の thread の命令を実行することでオーバーヘッドを隠蔽し，全体のスループットを向上させる CMT(Chip Multithreading Technology)[16] によるものである．図 15 の Solaris/T1 の並列数 16 や 32 の結果から，voronoi でパイプラインストールが比較的多く発生しており，T1 がこれを効率よく隠蔽していることがわかり，結果として 8 コアでも逐次処理の 8 倍以上の並列処理性能を実現できている．

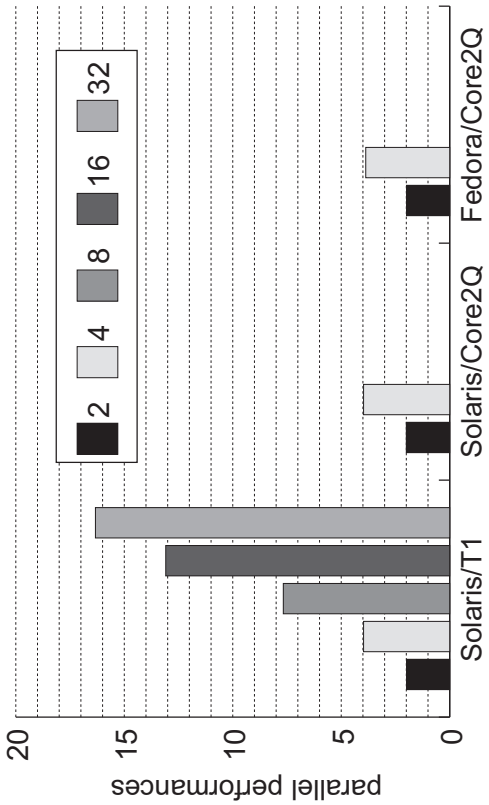


図 15: voronoi の高速化率

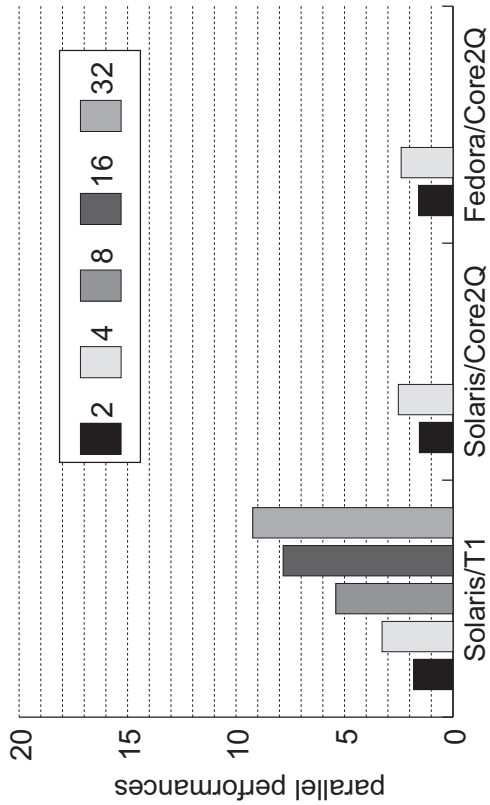


図 16: laplacian の高速化率

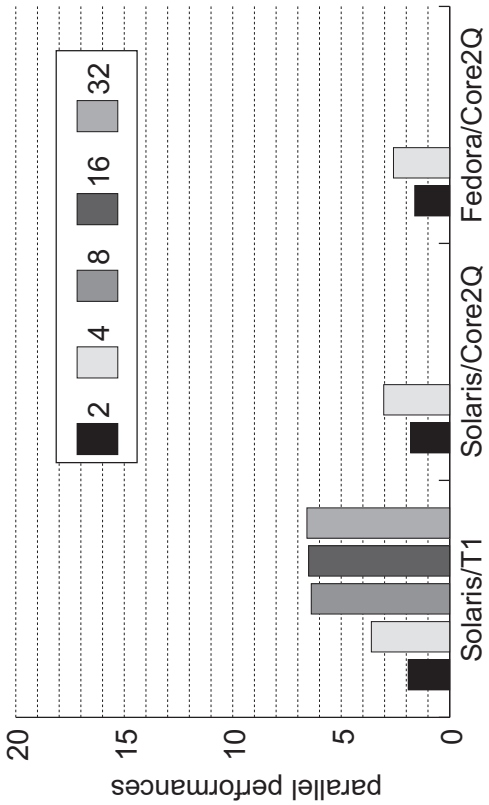


図 17: pixAverage の高速化率

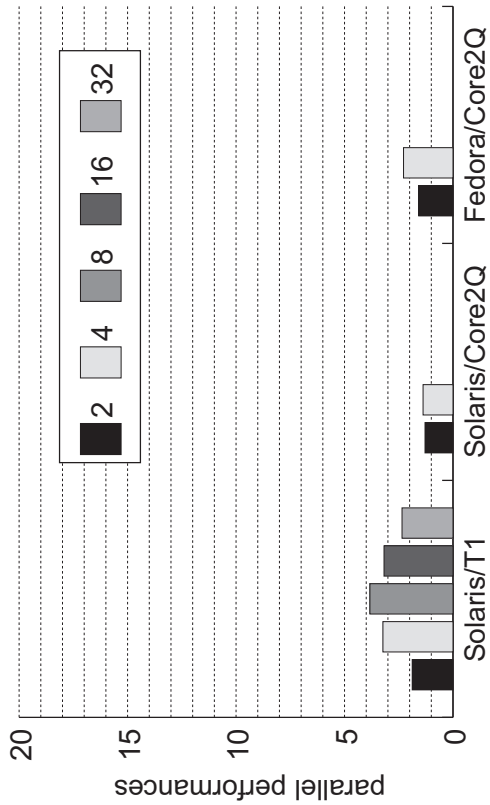


図 18: hough の高速化率

次に laplacian を各計算機で動作させた結果を**図 16** に，pixAverage の結果を**図 17** 示す．グラフから，Solaris/T1 では laplacian と pixAverage の両方において並列度に近い高速化率となった．また laplacian では 16 および 32 の並列数において，8 並列を上回る高速化を実現することができた．ここで Solaris/Core2Q や Fedora/Core2Q の 4 並列の結果では台数に応じた効果は得られなかった．この要因として，高速化に伴うオーバーヘッドの露呈が挙げられる．laplacian と pixAverage の両者とも 1 画素の処理量が少ないため，並列化の際には thread の統合や終了，pixAverage では reduction 演算といったオーバーヘッドが無視できない値となる．また並列数が上がるに従い thread 数や reduction 演算も増えることから，voronoi 程の効果が出なかったと考えられる．しかしこれらのオーバーヘッドは，逐次プログラムを手動で並列化したとしても同じような処理が行われると想定されるため，本プロセッサによる自動並列化は有効に働いていると言える．

最後に hough を各計算機で動作させた結果を**図 18** に示す．hough において，Solaris/T1 では 8 並列までは並列度に近い高速化率となったが，16,32 並列以降では期待した結果が得られず，逆に 8 並列時よりも遅くなってしまった．また Solaris/Core2Q や Fedora/Core2Q でも，4 並列での高速化率が期待した程得られなかった．この原因として，reduction 演算にかかるオーバヘッドの増大が考えられる．hough は reduction 演算の対象となるものが $\theta-\rho$ の 2 次元配列であり，int 型の要素を $360\times\sqrt{width^2\times height^2}$ 数分持つ (ただし $width$ と $height$ は画像の幅と高さとする)．よって並列数分の代替変数の初期化および reduction 演算にかかるオーバーヘッドが，本質的な処理であるハフ変換に対してかなり多い時間となってしまったと考えられる．

ここで hough を各並列数で処理した場合の，reduction 演算，reduction 演算のための初期化，実質的なハフ変換のそれぞれに要する時間の内訳を観測した．Solaris/T1 上で処理した結果を**図 19** に示す．グラフは逐次プログラムの処理時間を 1 に正規化した場合の，それぞれの並列数での処理時間を表している．なお reduction 演算にかかる時間は reduction，初期化の時間は init，ハフ変換にかかる時間は hough に相当する．

この結果からわかるように，ハフ変換の処理自体にかかる時間は並列数に応じて削減されている．しかし並列数が増えるにしたがって，reduction 演算と初期化のオーバヘッドが無視できない大きさになっている．これは reduction 演算をする上では回避できないオーバーヘッドであるが，今後 RaVioli およびプリプロセッサではこれを少しでも削減していくことを考える．例えばハフ変換の際の reduction 演算は $\theta-\rho$ の 2 次元配列の加算であるため，SIMD 演算を使ったベクトル化が有効である．よって

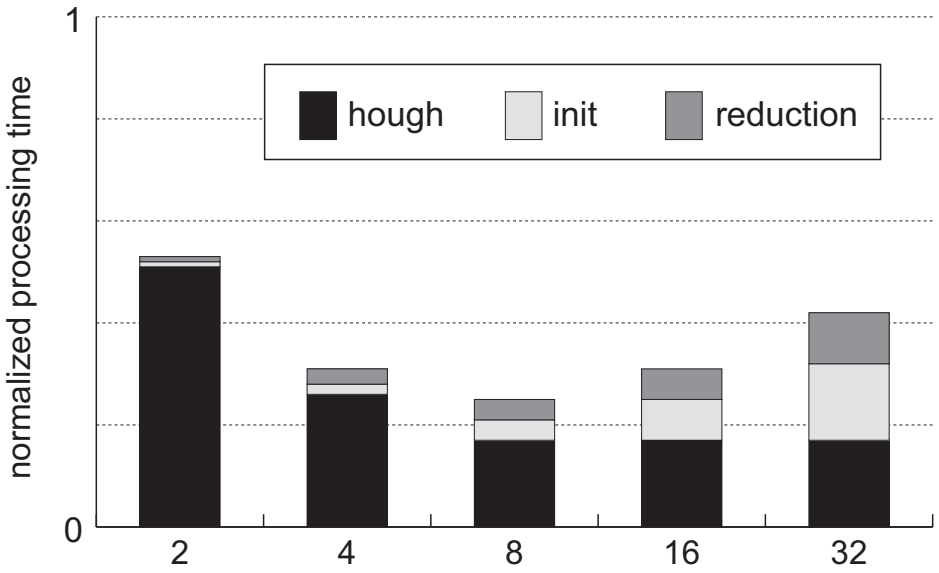


図 19: ハフ変換にかかる処理時間の内訳 (Solaris/T1)

reduction 演算を SIMD 命令を使用するよう変換する機能を付け加えることで，自動並列化プリプロセッサの更なる高性能化を目指す.

以上の結果から，提案したライブラリ RaVioli およびプリプロセッサを用いた場合には，プログラマが並列化を意識した記述を一切しなくても，逐次画像処理プログラムから並列画像処理プログラムを生成できることを確認した．またさまざまな画像処理プログラムで並列処理による高速化が確認できたことから，本プロセッサによる自動並列化は一般的な画像処理にも有用であることを示すことができた.

6 おわりに

本稿では，動的に使用可能な CPU リソースが変動するような環境において，解像度を自動で変動させることにより擬似的なリアルタイム動画像処理を保証する解像度非依存型動画像処理ライブラリ RaVioli を提案した．さらに，変動する解像度を考慮したプログラミングは複雑であることから，RaVioli は解像度に依存しないプログラミングパラダイムを提供することを示した．解像度を隠蔽するために動画像をカプセル化し，動画像処理に対する様々な高階メソッドを実装することでこのプログラミングパラダイムを実現した.

またマルチコアプロセッサが広く一般に普及したことから，余剰コアを利用した並列画像処理による高速化に焦点をあてた．画像処理のデータ並列化プログラムを実装する際には，

- 順序依存のある処理は並列化できないため，これを検出する必要がある
- 一つの大域変数に対して複数のサブタスクからアクセスする際には競合が発生するため，reduction 演算をする必要がある

という 2 つの大きな考慮点があり，プログラマがこれらを理解した上で並列化を適用しなければ，処理結果の正当性が無くなるという問題を示した．本稿では，プログラマが並列化に関する問題を一切考慮しなくても並列画像処理ができるよう，RaVioli を用いた逐次画像処理プログラムを並列画像処理プログラムに変換するプリプロセッサを提案し，従来には無かった画像処理の完全な自動並列化を実現した．

ハフ変換による顔検出プログラムと，フレーム間差分の検出プログラムを用いて RaVioli の記述容易性とリアルタイム性の保証に関する評価を行った．評価の結果，プログラムを一切変更することなく期待する動作が得られ，かつ擬似的にリアルタイム性を保証した処理ができることを確認した．

また自動並列化機能の評価として，まずプリプロセッサを様々な画像処理プログラムに適用することで，プリプロセッサの適用範囲を明らかにした．結果として，競合や順序依存のない画像処理はすべて並列処理に変換できた．また競合の起こる可能性のある処理は reduction 演算を適用しており，順序依存のある処理を見つけた場合には，警告を発するとともに逐次処理のままとしていることが確認できた．次に変換された並列画像処理プログラムの速度評価を行った．マルチコア環境として，UltraSPARC T1(8 コア) と Core2 Quad(4 コア) を搭載した計算機を用いて速度評価を行ったところ，reduction 演算のない処理は台数効果に近い速度向上が確認できた．

今後の課題として，処理結果に依存した動的な優先度切り替え機能の実装が挙げられる．例えば動画像中に現れる人物の検出を行う場合，歩行中の人物の有無を認識するためには高いフレームレートでサンプリングを行うことが重要であるのに対し，人物の顔などの詳細な認識を行う場合には 1 フレームの構成画素数が多いことが重要である．従って，歩行中の人物が現れるまではフレームレートを優先した処理を行い，人物が検出された時点で構成画素数を優先した処理を行う優先度の切り替えが必要であると考える．よってプログラマが，過去の処理結果によって明示的に優先度切り替えができ，さらに部分的 (例えば顔領域) に優先度を指定できるインターフェースの導入を検討する．

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電力化への要求が高まっていることから [17]，RaVioli でも高速化および省電力化手法を検討する．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** [18] を RaVioli に追加実装する．これにより，計算の省略による更なる高速化や省電力が見込める．

さらに RaVioli による画像処理の高速化を図るために，処理に対するさまざまな粒度の並列化の検討・開発を進めている．まず，RaVioli は画像内のすべての画素に対して同じ処理を適用することから，SIMD 命令を用いて複数の画素を一度に処理することを考える．SIMD 化の問題として RaVioli 記法では，プログラマが 1 画素に対する処理を記述するため，ライブラリ側でこれを複数画素に対する処理に書き換え SIMD 化することはできない．現状ではプログラマが記述したプログラムを SIMD 命令を用いたものに変換するトランスレータを実装することにより，この問題に対応している．また，ストリーミング画像処理はソフトウェアパイプライン的に演算処理を行うことが可能な処理が多い．例えば，ハフ変換によるライン抽出画像の背景差分をとってから，エッジ抽出，ハフ変換をするといったように，画像単位の処理が段階的に行われることが多く，それらをステージとして分割し，タイムスライス化することで高速化を図る．しかしステージ毎に処理量が違うことから，1 ステージに割り当てられる時間に対して処理時間が極端に少なかったり，逆にオーバーしてしまう可能性がある．現状ではこの問題に対し，処理量の少ない 2 つのステージを統合して 1 つのステージとして処理したり，逆に処理量の多いステージはフレーム単位で並列化することにより，なるべくステージ毎の処理を平均的に扱う機能を実装している．そしてさらに，CPU だけでなく GPU への処理の割り当てによる高速化も検討している．パイプライン化の場合と同じく画像処理は段階的に行われるという考えから，それらの処理のうち処理量の大きいものを GPU に割り当てることを考えている．現状では，CPU のキャッシュやメモリからのデータ転送時間がボトルネックとなることが，予備評価から分かっている．このことから，転送時間に対して処理時間の十分大きな処理を GPU に割り当てることによって，全体としてのスループットを向上させる．

以上のような提案を実装し，処理をマルチグレイン並列化することで高速化を図る．またこのマルチグレイン並列化に対して，プログラマがまったく並列化を意識しなくても記述できるような専用の言語の検討・開発も行っていきたい．

謝辞

本研究の為に多大な御尽力をいただき，日頃から熱心な御指導を賜った名古屋工業大学の津邑公暁准教授に深く感謝致します.

また，本研究に対しての熱心なご検討をいただきました，名古屋工業大学の松尾啓志教授に深く感謝致します．これに加えて，たびたびご検討・助言をしていただいた同大学の齋藤彰一准教授ならびに松井俊浩助教にも深く感謝致します.

最後に，本研究の際に多くの助言・協力をしていただいた松尾・津邑研究室ならびに齋藤研究室の皆様，特に RaVioli の開発に携わっていただいた桜井寛子さん，大野将臣君，白木夢斗君に対し心より感謝致します．本当にありがとうございました.

参考文献

[1] 細田寛人: 車載用画像認識プロセッサ IMAPCAR, NEC 技報, Vol. 59, No. 5, pp. 22–25 (2006).

[2] Sato, H. and Yakoh, T.: A real-time communication mechanism for RTLinux, *Industrial Electronics Society(IECON 2000)*, Vol. 4, pp. 2437–2442.

[3] Köthe, U.: *VIGRA - Vision with Generic Algorithms*, 1.6.0 edition (2008).

[4] 飯尾淳, 谷田部智之, 比屋根一雄, 米元聡, 谷口倫一郎: 動画像処理ライブラリ MAlib を利用した 3 次元ユーザインタフェースの実装, オブジェクト指向 2002 シンポジウム (OO2002), 情報処理学会, pp. 117–120 (2002).

[5] Bradski, G. and Kaehler, A.: *Learning OpenCV: Computer Vision With the Opencv Library*, Oreilly & Associates Inc (2008).

[6] 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).

[7] 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).

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

[9] 吉本廣雅, 有田大作, 谷口倫一郎: 実時間ビジョンシステムのための信頼度駆動メモリ, 情報処理学会論文誌, Vol. 44, No. 10, pp. 2428–2436 (2003).

[10] 奥村文洋, 松尾啓志: 疑似リアルタイム機能を備えた動画像処理系 Streaming VIOS の開発, 第 2 回動画像処理実利用化ワークショップ, 精密工学会, pp. 62–65 (2001).

[11] Campbell, D. K. G.: Towards the Classification of Algorithmic Skeletons, Technical report, Dept. of Computer Science, Univ. of York (1996).

[12] 田中義純, 田浦健次朗, 米澤明憲: OpenMP におけるネストした並列性の実装と評価, 情報処理学会論文誌プログラミング, Vol. 41, No. SIG 2(PRO 6), pp. 54–64 (2000).

[13] 松尾啓志, 川脇智英: 分散画像処理環境 VIOS-III, 電子情報通信学会論文誌 D-II, Vol. J84-D-II, No. 6, pp. 955–964 (2001).

[14] 馬場功淳, 江島俊明: HeadFinder:単眼視動画像を用いた複数人追跡, 画像センシングシンポジウム, pp. 363–368 (2001).

[15] Ganter, M. A.: From Wire-Frame to Solid-Geometric: Automated Conversion of Data Representations, *Computers in Mechanical Engineering*, Vol. 2, No. 2, pp. 40–45 (1983).

[16] Weaver, D.: *OpenSPARC Internals*, Lulu.com (2008).

[17] 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).

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

付録

A.1　ハフ変換のプログラム例

5.1 節で述べた顔検出プログラムの一部を**図 A.1** に示す．このプログラムはエッジが抽出された 2 値画像に対して，円のハフ変換を適用する部分の記述を抜き出したものである．この処理は一つの黒画素に対して，その黒画素が構成要素となりうる円をすべて検出するという行程を，画像中のすべての黒画素に対して行う．そして検出された円は，x 座標，y 座標，円の半径をインデックスとする 3 次元空間の該当部分に投票される．

ハフ変換のプログラムの処理内容について概説する．43 行目から始まる `main` 文では RV_Streaming インスタンス `video` を定義し，1 フレームに対する処理を記述した関数 `serialProc` を動画像中の該当フレームに適用させる．28 行目から始まる関数 `serialProc` では，32 行目に至るまでに，画像に対して 2 値化，人が存在する可能性のある領域の決定，エッジ抽出といった前処理を行う．なお 32 行目以降で使われている変数 `start` 及び `end` は，人が存在する可能性のある領域の左上と右下の座標とする．このため 32,33 行目では，その領域の大きさを `areaWidth` と `areaHeight` に格納している．34∼37 行目では抽出する円の半径の最大値 `radius_max` 及び最小値 `radius_min` を決定する．次に 38 行目で，円候補に投票する 3 次元配列の RV_Array オブジェクト `counter` を生成する．39 行目のメソッド `setSize` は配列に対して 1 次元及び 2 次元のサイズを確保するメソッドであり，次元数は引数の数に対応している．40 行目のメソッド `procVal` は高階メソッドであり，`setSize` メソッドにより生成された 2 次元配列の各要素に対して処理を適用する．そして 1 次元配列を生成する関数 `newCounter`(5 行目) を各要素に適用することにより，3 次元配列を実現する．次に 40 行目で `start` から `end` の範囲に関数 `hough` を適用する．21 行目で定義されている関数 `hough` では，黒画素 `p` の座標を `point` として一時的に大域変数に格納しておき，24 行目で `counter` の高階メソッド `procCoord` を呼び出す．`procCoord` の引数として呼ばれる 8 行目の `dataSet` は，`p` を構成要素とするすべての円候補を検出し，`counter` に投票する．

```
RV_Length radius_max, radius_min;
RV_Coord point, start, end;
RV_Array<RV_Array<int> > counter;

void newCounter(RV_Array<int>* factor){
  factor->setSize(radius_max);
}
void dataSet(RV_Array<int>* factor,
             RV_Coord coord){
  RV_Length radius;
  int tmpValue;
  radius=(point-coord).getLength();
  if(radius < radius_max){
    if(radius > radius_min){
      tmpValue=factor->getData(radius);
      tmpValue++;
      factor->setData(radius,tmpValue);
    }
  }
}
RV_Pixel hough(RV_Pixel p,RV_Coord coord){
  if(p->getR() == 0) {
    point=coord-start;
    counter.procCoord(dataSet);
  }
  return p;
}
void serialProc(RV_Image* Fnew){
  RV_Length areaWidth;
  RV_Length areaHeight;
     : //前処理
  areaWidth=(end-start).getXLength();
  areaHeight=(end-start).getYLength();
  if(areaWidth < areaHeight)
    radius_max=areaWidth/2;
  else radius_max=areaHeight/2;
  radius_min=radius_max/10;
  counter.setSize(areaWidth,areaHeight);
  counter.procVal(newCounter);
  Fnew=Fnew->procCoord(hough,start,end);
     : //ハフ逆変換, 出力
}
int main(){
  RV_Streaming video;
  video.StreamProc(serialProc);
}
```

図 A.1: **RaVioli** を用いたハフ変換による円抽出（部分）

A.2　voronoi 図作成のプログラム例

5.3.2 節でサンプルプログラムとして評価に用いた voronoi 図の作成プログラムを**図 A.2** に示す．voronoi 図は，画像中の任意の位置に配置された複数個の母点を持ち，各画素がどの母点に一番近いかによって領域分けされた図である．読み込まれる画像はすべて書き換えられるため，どのような画像でもよい．

voronoi 図のプログラムの内容について概説する．21 行目から始まる `main` 文では RV_Image インスタンスのポインタ `image` を定義後，`new` を用いて領域を確保している．24 行目で定義している RV_FileHandler クラスはファイルの入出力を行うクラスであり，次の行でコマンドライン引数として受け取った画像ファイル名を取り込み，ファイルの中の画像データを `image` に格納している．また 33 行目の `outBMP` では，逆に `image` のデータをビットマップファイルとして書き込んでいる．3 行目の大域変数 `zero`，`Wid`，`Hei` はいづれも RV_Length インスタンスであり，`zero` には 0 を `Wid`，`Hei` にはそれぞれ画像の幅と高さを格納しているが，中身を隠蔽しているためすべて抽象的に扱う．次に母点の座標群 `g` を random 関数を用いて決定する．`g` のクラスである RV_Coord もまた中身を隠蔽しており，`Wid`，`Hei` に対する割合で座標値を指定する (29~31 行目)．次の 32 行目では RV_Image クラスの高階メソッド procCoord() を用いて各画素を，一番近い母点が持つ規定色となるよう色分けをする．なお母点の規定色とは，母点毎に固有の色であり，今回は母点配列のインデックスとしている．

次に procCoord() の引数として呼ばれている関数 `voronoi_calc`(5 行目) の説明に移る．高階メソッドの引数であるこの関数は，ある 1 画素の値とその座標に対する処理を記述し，その結果を書き出す内容となる．具体的には，それぞれの母点座標と当該画素の座標との差を求め，一番小さい差分値であった母点の規定色を書き出している．なお，当該画素が母点であった場合は白を書き出している．プログラムの 8 行目で，まず変数 `min` を，画像内で取りうる最大長よりも大きな長さとする．そして 10 行目でそれぞれの母点座標 `g[i]` と当該画素の座標 `Co` との差 `dist` を求める．`dist` が `min` よりも小さければ，`min` に `dist` を代入するとともに，画素 `P` の色を母点のインデックス値 `i` にする (11~13 行目)．また `dist` が 0 であれば，その画素は母点であると判断し，`P` の色を白にする (14 行目)．なお `dist` は長さであり負の値は存在しないため考慮しなくてもよい．

```
#define GNUMBER 80
RV_Coord g[GNUMBER]; //母点の座標
RV_Length Hei,Wid,zero;

RV_Pixel voronoi_calc(RV_Pixel P, RV_Coord Co){
  int color;
  RV_Length min,dist;
  min = Wid + Hei;
  for(int i=0; i< GNUMBER; i++){
    dist = (g[i] - Co).getLength();
    if(dist < min){
      color = i;
      min=dist;
      if(dist == zero){color = 1000;}
    }
  }
  P.setRGB(color, color, color);
  return(P);
}

int main(int argc, char* argv[]){
  RV_Image* image;
  image=new RV_Image();
  RV_FileHandler file;
  file.readBMP(argv[1], *image);
  zero.Zero();
  Wid = image->Width;
  Hei = image->Height;
  for(int i=0; i<GNUMBER; i++){
    g[i].setCoord(Wid*rand()/RAND_MAX,Hei*rand()/RAND_MAX);
  }
  image=image->procCoord(voronoi_calc);
  file.outBMP(argv[2],*image);
}
```

図 A.2: **RaVioli** を用いた voronoi 図の作成