解像度調整機能を備える 並列動画像処理ライブラリ RaVioli の実装

大 野 将 臣[†1] 桜 井 寛 子[†1]
津 邑 公 暁[†1] 松 尾 啓 志[†1]

動画像処理は携帯端末から汎用 PC まで幅広いプラットフォームで動作させる必要がある．しかし，プラットフォーム毎に，処理精度やバッテリ持続時間など，重要視される要求性能ポイントが異なる．そのため，プログラマは各プラットフォームに対する深い知識が必要になり，大きな負担となる．

そこで，ユーザが予め指定した優先度パラメータに基づき，解像度を動的かつ透過的に変動させることを可能にする動画像処理ライブラリを提供する．これにより，リアルタイム性が要求される動画像処理において，自動的に解像度を低減させることで，高負荷時にも疑似リアルタイム処理を実現可能になる．さらに，入力動画像に変化が現れない場合などに自動的に解像度を低減させることで，処理による消費電力を削減する．

また，解像度を動的に変更するためにはプログラマから解像度情報を隠蔽する必要があるが，これはプログラマにとって障碍とはならず，むしろ本来人間の映像認識過程には存在しない「画素」「フレーム」などの概念から開放されることで，より直感的なプログラミングが可能となる．また，一般に動画像処理はフレームのエリア分割や，複数ステップに分割した上でのパイプライン化など，様々な粒度での並列化が可能である。本ライブラリでは，解像度を隠蔽することで処理中の繰り返し処理単位が明確になるため，これらの並列化をプログラマの手を煩わせることなく，比較的容易に自動化することも可能となった．

RaVioli: a Parallel Video Processing Library with Auto Resolution Adjustability

Masaomi Ohno,[†1] Hiroko Sakurai,[†1]
Tomoaki Tsumura[†1] and Hiroshi Matsuo [†1]

Video processing applications are in demand on a great variety of platforms such as mobile devices or high performance computers. On these platforms, principal points differ from each other. One requires high processing speed, another requires long battery time, and so on. Therefore, a deep knowledge about various platforms is required for programmers. This would make the burden heavy for them.

This paper proposes a parallel video processing library RaVioli. RaVioli hides two resolutions, framerate and pixel number, from users, and provides a dynamic and transparent resolution adjustability based on user-preferred priority parameter. This makes pseudo realtime video processing feasible by decreasing resolutions automatically and dynamically. Meanwhile, when the input video images are static, decreasing resolutions reduces power consumption.

Generally, video processing has some parallelism in its algorithm. For example, pixels in a frame have data parallelism, and many video processing algorithm can be divided into some processing steps which can be pipelined. Hiding resolutions makes implicit parallelism more obvious. Hence, RaVioli can parallelize programs semi-automatically.

1. はじめに

これまでの情報システムは，応用分野からの性能要求が原動力となりコンピュータシステムやプロセッサ，更には半導体技術を改善することが重要視されてきた．しかし今日，Nehalem や Niagara といったマルチコア・メニーコア環境が一般的になるに従い，これら多くのコアを有効活用できるようにアプリケーションやコンパイラやライブラリを改良することが重要になってきている．

特に，リアルタイム動画像処理アプリケーションには様々な最適化が必要である．今日，動画像処理アプリケーションは携帯電話などの携帯端末から高性能なサーバまで幅広いプラットフォームで実行されている．しかしそれぞれのプラットフォーム上で高い性能を実現するには，プログラマは対象とするプラットフォームのプロセッサやメモリアーキテクチャに対する深い知識が必要になる．さらに，プラットフォーム毎に，バッテリ持続時間，消費電力や処理速度など，重要視される要求性能ポイントが異なる．その結果，プログラマは動画像処理の本質とは異なる様々な点に注意を払わなければならない．

一方，動画像処理には一般にデータ並列性およびタスク並列性が存在するため，動画像処理アプリケーションは並列システムで処理するのに適している．動画像フレームを構成する

†1 名古屋工業大学
Nagoya Institute of Technology

画素間にはデータ並列性があり，SIMD 命令やブロック分割並列処理を行うことで様々な粒度で並列処理が可能である．また，動画像処理プログラムの多くはいくつかのステージに分割することができ，パイプライン的に各ステージでフレームを次々と処理することで並列化することができる．例えば人物検出の場合では，二値化，エッジ抽出，ハフ変換といったステージに分割しパイプライン化することができる．

しかしこのような並列プログラミングは容易ではなく，既存の逐次プログラムを書き直す必要がある．データ並列性を利用するには，データ依存やリダクション処理を意識しながらプログラムを記述しなければならない．また，効率的なパイプライン処理を実現するには，プログラマは処理量を意識しながら各ステージを構成し，それら処理量を平均化する必要がある．このような動画像処理の本質でない点に留意しながらのプログラミングはプログラマにとって負担となる．

そこで，これらの問題を解決するために 全く新しい動画像処理プログラミングパラダイムを提供する C++ ライブラリ **RaVioli** (Resolution Adaptable Video Operating Library) を提案する．RaVioli は動画像処理における時間解像度 (フレームレート) と空間解像度 (処理画素数) をプログラマから完全に隠蔽し，それらをライブラリ内で制御する．また，負荷が高くなりリアルタイム処理が困難になったとき，自動的に処理画素数やフレームレートを低減させることで，処理量を減少させることが可能である．処理画素数またはフレームレートどちらを低減させるかは，プラットフォームの性能要求に応じて，プログラマが設定した優先度に従って決定される．さらに，解像度を隠蔽することで処理中の繰り返し処理単位が明確になるため，データ依存やリダクション処理を自動的に検出・解決しデータ並列化を適用することが可能となる．また，パイプライン処理を容易に構成することができるインタフェースを提供する．このインタフェースを通じて定義されたパイプラインステージは自動的にスレッドに割り当てられ，それら複数スレッド間の負荷も自動的に均衡化される．

2. 関 連 研 究

2.1 動画像処理

リアルタイム動画像処理では，使用可能な CPU リソース量に応じて処理量を調節することが重要である．しかし，これまでは複数の予め定義されたルーチンを切り替えることが，これを実現する唯一の方法であった．例えば Imprecise Computation Model[1] は計算時間の長さに応じて処理精度を変化させるモデルである．またこのモデルに基づく，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，複数のルーチンから適切なルーチンを動的に選択する信頼度駆動アーキテクチャ[2] も提案されている．しかし，このモデルでは，処理を計算負荷の異なる複数のルーチンで実装する必要がある．

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

図 1 リダクション処理を含む OpenMP を用いた並列プログラム
Fig. 1 Parallel program with OpenMP containing reduction operation

一方で，良く知られる動画像処理ライブラリに VIGRA[3] や OpenCV[4] がある．これらのライブラリは動画像処理の抽象化を目的としたもので，VIGRA では C++の STL と同様にテンプレートを用い，プログラマに抽象的な処理を提供している．また，OpenCV では多くの動画像処理アルゴリズムを C 言語の関数や C++のメソッドとして提供している．しかし，これらのライブラリを用いて実装されたプログラムで処理量を動的に調節したりリアルタイム性を保証することは難しい．

RaVioli の手法はこれら既存の計算モデルや動画像処理ライブラリとは全く異なる．RaVioli ではプログラマから画素やフレームを隠蔽し，ライブラリ内で空間解像度および時間解像度を負荷に応じて動的に変動させることで，自動的に処理量を調節することが可能となる．

2.2 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という概念がある．例えば，C++ テンプレートライブラリ Intel Threading Blocks Blocks (TBB)[5] は並列スケルトンの形で書かれたプログラムを並列に動作させるライブラリであり，TBB はタスクのスケジューリングを管理するため，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なくなる．しかし，予め並列化可能な処理を並列スケルトンとして記述しなければならないため，プログラマはアルゴリズムを設計する段階から並列化を考慮して開発する必要がある．

そのほかの並列プログラミングモデルに OpenMP[6] がある．OpenMP ではプログラマは、‘pragma’ と呼ばれるコンパイラ指示子を使用しコードブロックをどう処理するかを指

定する．例えば，**図 1** に示すように，`parallel` プラグマを用いることで，`parallel` プラグマに続く for 文のイタレーションは並列処理される．しかし，この図 1 の例では変数 `sum` への代入は排他的に行われなければならない．したがって，複数スレッドを用いて並列化した場合，このプログラムが正しい結果を得るには次の手順で処理を進める必要がある．

(1) 各スレッドが担当する加算結果を格納しておくスレッドローカル変数 `sum` を作る．
(2) 各スレッドは担当する加算結果をスレッドローカル変数 `sum` に格納する．
(3) 最後に各スレッドの加算結果を全て足し合わせる．

OpenMP にこの手順を踏ませるためには，図 1 に示される，リダクション処理を行う `reduction` プラグマとリダクション変数を指定する必要がある．このように OpenMP ではプログラマが，どのブロックが並列処理可能であるかや，リダクション処理が必要な箇所を明示的に指定する必要がある．

一方，RaVioli は半自動的にプログラムを並列化する機能を有する．RaVioli を用いたプログラムでは，データの依存関係やリダクションが必要な箇所を容易に検出することができる．本研究では，RaVioli を用いて記述された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実装した．これはデータ依存関係やリダクション必要性を自動検出し処理するため，プログラマは並列化を行う際に必要となる問題を考慮する必要がなくなる．

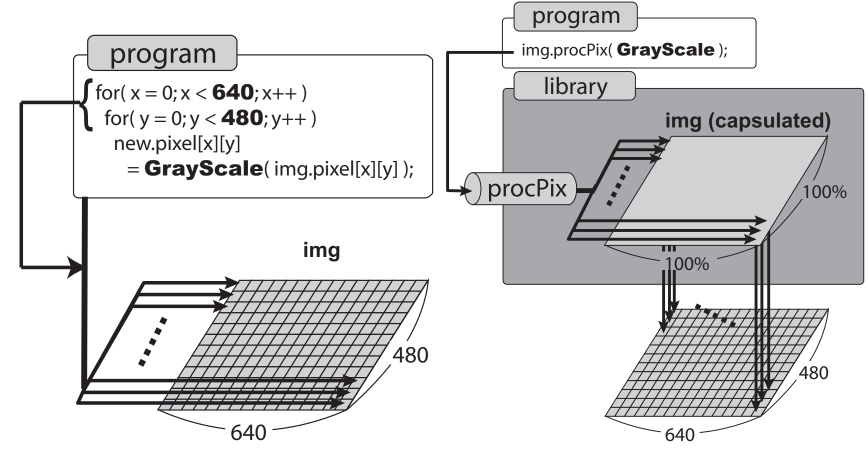
3. 動画像処理ライブラリ RaVioli

3.1 動画像処理の抽象化

RaVioli は動画像の空間解像度および時間解像度を隠蔽することで，プログラマが直感的にプログラムを記述可能となる，全く新しい動画像処理プログラミングパラダイムを提供する．

そもそも人間が物体を認識する際には解像度といった概念は存在しない．しかし，例えば動物体検出プログラムは，一般に隣接フレーム間の画素値の差などからフレーム間の類似度を求め動物体を検出する．このように量子的に情報を扱う必要がある計算機上では解像度の概念は不可欠であるため，プログラマは本来意識下に存在しない解像度をプログラム中で管理しなければならなくなり，直感的なプログラミングが困難となる．

このように量子化された動画像データを扱うため，一般に動画像処理にはループが良く使用される．例えば画像をグレースケールに変換するとき，**図 2(a)** に示すように各画素を変換する処理は最内ループに記述され，この処理が画像中のすべての画素に繰り返し適用される．一方 RaVioli を用いた画像処理プログラムでは，**図 2(b)** に示すように，画素に対



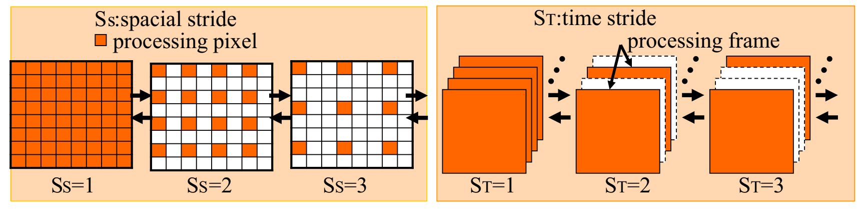
(a) Traditional image processing.　(b) Image processing with RaVioli.

図 2 画像処理
Fig. 2 Digital image processing.

する処理を記述した関数 GrayScale() を，画像を保持するクラスのインスタンス img の高階関数 procPix() に引数として渡すのみで良い．RaVioli では，このような画像を構成する画素，動画像を構成するフレームを処理単位とする関数を**構成要素関数**と呼ぶ．ここで高階関数 procPix() は，そのインスタンスが保持する画像の構成画素すべてに対し，渡された関数 GrayScale() を繰り返し適用する．さまざまな繰り返しパターンに対応する高階関数が用意されているが，それら詳細については文献 7) を参照されたい．図 2(b) の例では，GrayScale() はプログラマによって定義された構成要素関数で，この関数を画像インスタンスである img の高階関数 procPix() に引数として渡すことで，GrayScale に定義された処理がすべての画素に適用される．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の記述ができる．

3.2 処理量の自動調節

RaVioli は解像度をプログラマから隠蔽したことにより，ライブラリ内で負荷に応じて処理解像度を動的に変動させることが可能になった．解像度には**空間解像度**と**時間解像度**の 2 つがあり，空間解像度は 1 フレームを構成する画素数を意味し，時間解像度はフレームレートを意味する．RaVioli はこの 2 つの解像度を変動させることで処理量の調節を行いリア



(a) Spacial resolution.　(b) Time resolution.

図 3　解像度の変更
Fig. 3　Change of resolutions.

ルタイム性を保証する．ライブラリ内では**空間解像度ストライド** ($\mathrm{S_S}$) および**時間解像度ストライド** ($\mathrm{S_T}$) を増減させることで両解像度を変動させている．例えば，**図 3(a)** は空間解像度ストライド $\mathrm{S_S}$ の値を上げることで処理画素数が減少し処理量が低減している．同様に**図 3(b)** は時間解像度ストライド $\mathrm{S_T}$ の値を上げることでフレームをスキップし処理量が低減している．

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た解像度の維持を行う．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，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，RaVioli は空間解像度を優先的に低減させる．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，時間解像度が優先的に低減され，精細さの確保を行い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 $(\mathrm{P_S}, \mathrm{P_T})$ の組である**優先度セット**を指定する事で設定することができる．$\mathrm{P_S}$ は空間解像度に対する優先度を表し，$\mathrm{P_T}$ は時間解像度に対する優先度を表す．例えば，$(\mathrm{P_S}, \mathrm{P_T}) = (3, 7)$ と設定されたとき，RaVioli は空間解像度ストライドと時間解像度ストライドを $7:3$ の割合で維持しようとする．動画像処理アプリケーションを異なるプラットフォームに移植する際，プログラマはプログラムを書き直す必要がなく， 優先度セットの値をそのプラットフォームに適した値にするだけでよい．その結果，プログラマはプラットフォームに対する要求性能ポイントを満たし，実時間処理を実現する動画像処理アプリケーションを容易に実装することが可能となる．

4.　動画像処理の並列化

プログラマから解像度を隠蔽したことにより，繰り返し処理単位が明確になり，動画像処理の並列化も容易となる．本章では，RaVioli が提供する 2 種類のプログラム並列化手法について述べる．

4.1　空間分割並列化

一般に画像処理では 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ものが多くあるため，データ並列化に適している．例えばグレースケール化処理の場合では画像をブロック分割し，それぞれの分割画像を別スレッドで処理させることによりデータ並列化を実現することができる．しかし，他の画素の処理結果を用いて，別の画素に対する処理を行うといった順序依存のある処理は並列化することができない．また図 1 で見たように，イタレーション間で共通して読み書きアクセスする変数が存在するようなプログラムを並列化する場合は，リダクション処理を記述しなければならない．そのため，プログラマは並列化の際に，順序依存やデータ依存問題が起きる処理を予め把握し，これに対応しなければならない．

そこで，これらの問題に対し，RaVioli を用いた逐次プログラムを自動的に並列プログラムへ変換するプリプロセッサを実装した．RaVioli を用いたプログラムでは図 2(b) に示されるように，プログラマは始めに画素などの画像の構成要素を処理する関数である構成要素関数を定義する．次に，適切な高階関数を選択し，定義した構成要素関数をその高階関数に渡す．高階関数はライブラリ内ではループを用いて実装されており，渡された構成要素関数を画像を構成する全画素に適用する．ここでループイタレーションにはデータ並列性があるため，**図 4** に示されるように自動的に画像を分割し並列化することが可能である．

プリプロセッサは始めにプログラム中から高階関数を検索し，検出された高階関数が並列化可能であれば画像を分割しそれぞれを別のスレッドで処理することで並列化する．同時に，プリプロセッサは並列化時にリダクション処理が必要かどうか自動的に判定する．このプリプロセッサを用いることで，一度記述した動画像処理プログラムをマルチコア並列システム用に書き直す必要がなくなる．以下にこの詳細な手順を述べる．

まず RaVioli の高階関数は，構成要素に対する適用順を規定していない。つまり，**図 2** の例で procPix() に渡された GrayScale() は，画像インスタンスを構成する各画素にどのような順番で適用されるかはプログラマには分からない．しかし構成要素関数内でグローバル変数にアクセスするようなプログラムの中には，この適用順に依存するプログラムも記述

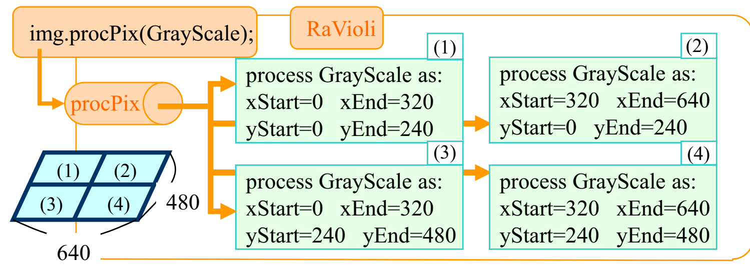


図 4 空間分割並列化
Fig. 4 Spacial Parallelization

し得る．これは従来の画像処理プログラミングに親しんだプログラマが犯しうる間違いであるため，プリプロセッサはまずこれを検出して警告する．具体的には，以下のような条件を見たす場合は処理順依存であると判断する．

- 大域変数に対し書き込みのみが存在する場合
  (例: `foo = pixel.getR();`)
- 読み書きのある大域変数に対し ‘+’ もしくは ‘-’ を使用し，かつ ‘*’ もしくは ‘/’ を使用している場合
  (例: `foo = foo/p.getR()-2;`)
- 条件式で使われている大域変数に対し，比較した値とは異なる値が代入されている場合
  (例: `if(foo > p.getR()) foo = p.getG();`)
- 四則演算を施した大域変数を画素へ書き込んでいる場合
  (例: `foo += pixel.getR(); pixel.setR(foo);`)
- 他のライブラリで定義された関数の引数に大域変数を使用している場合

次に，各構成要素に対する処理間で共有変数にアクセスしており，かつ上記条件にあてはまらない場合は，リダクション処理が必要であると判断する．RaVioli では各構成要素に対する繰り返し処理自体がライブラリの高階関数内に綴じているため，この変数共有は必ず構成要素関数内における帯域変数へのアクセスとして発言する．よって，リダクション処理が必要である個所の検出は，構成要素関数内における大域変数へのアクセスが存在するか否かを調べることで実現できる．

次に，プリプロセッサがリダクション処理を生成する方法を示す．構成要素関数内に大域変数へのアクセスを検出し，リダクション処理が必要だと判断したとき，リダクション変数

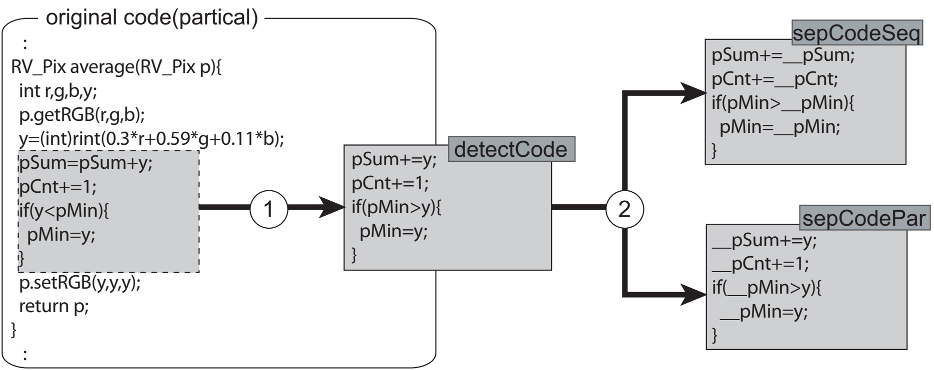


図 5 リダクション処理の変換
Fig. 5 transform of a reduction unit

を**スレッドローカルストレージ (TLS)** として定義する．Sun Studio C++ コンパイラ[8] や GNU C++ コンパイラを使用している場合，スレッドローカル変数は __thread 指示子で指定できる．リダクション変数はブロック分割した部分処理の結果を格納するのに使用され，すべての部分処理が終了したあとに，各リダクション変数の値を逐次的に統合する．

例えば**図 5** に示す関数 `average()` は画像中の全画素値の和，最小値，画素数を計算する構成要素関数であり，`pSum`，`pCnt` および `pMin` は大域変数である．始めに，リダクション処理を検出しやすいように，プリプロセッサはコメントやタブ及び空行を削除し，1 行に 1 演算のみが含まれるように整形し，大域変数に対し読み書きしているテンポラリ変数をなるべく使用しない形にコードを変換する．次に，図 5 のコードを上述した条件と照らし合わせ，処理順に依存しないことが確認される．これが確認されると，このコードは推移律を満たすこととなり，図 5 に示すように，各スレッド用コード sepCodePar と統合用コード sepCodeSeq の 2 つに分割することができる．`__pSum`，`__pCnt`，`__pMin` はスレッドローカルなリダクション変数で，それぞれ大域変数 `pSum`，`pCnt`，`pMin` に対応している．sepCodePar は各スレッドで並列に実行され，その後，sepCodeSeq は 1 つのスレッドで 1 度だけ実行され，部分計算の結果が統合される．

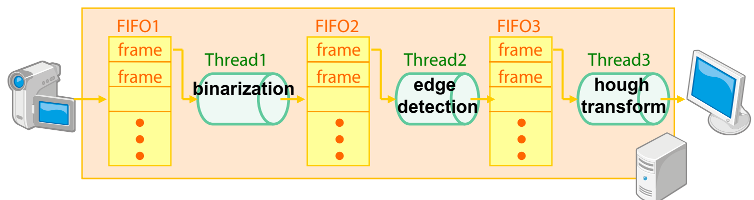
プリプロセッサを用いた並列化のもう一つの例を付録 A.1 に示す．このプログラムはカラー画像をグレースケールに変換し，その後，すべての画素値の平均値と最小値を計算する．

```
binarization(){
  while(video_stop!=1){
    while(load(frame)!=1){}
    process the frame
  }
}
```

```
edge_detection(){
  while(video_stop!=1){
    while(load(frame)!=1){}
    process the frame
  }
}
```

```
hough_transform(){
  while(video_stop!=1){
    while(load(frame)!=1){}
    process the frame
  }
}
```

(a) Pseudo codes of pipeline.



(b) Pipelined video processing.

図 6 パイプライン化
Fig. 6 Pipelining

4.2 パイプライン化

動画像処理には処理を複数のステージに分割し，パイプラン的に実行可能であるものが多く，マルチコアプロセッサを用いたパイプライン処理が有効である．そこで RaVioli は容易にパイプライン処理を実現可能なインターフェースを提供する．一般に動画像処理をパイプライン化する場合，まず全体の処理を複数のステージに分割する必要がある．例えば，顔検出の場合では，**図 6(a)** に示すように，2 値化，エッジ抽出，ハフ変換に分割することができる．次に，スレッドを作成し，そのスレッドにステージとして分割した処理を割り当て，ステージ間でのデータの授受を管理するための FIFO（パイプラインレジスタ）も作成する必要がある．

パイプライン化された動画像処理の例を**図 6(b)** に示す．カメラからキャプチャされたフレームは図 6(b) の FIFO1 に格納され，2 値化ステージは FIFO1 からフレームを取得し，フレームを処理し，その結果を FIFO2 に格納する．同様に，エッジ抽出ステージは FIFO2 からフレームを取得し，処理し，その結果を FIFO3 に格納する．各ステージを並列に動作させた場合，各ステージでのフレーム処理はオーバラップされ，動画像処理全体を高速化することができる．

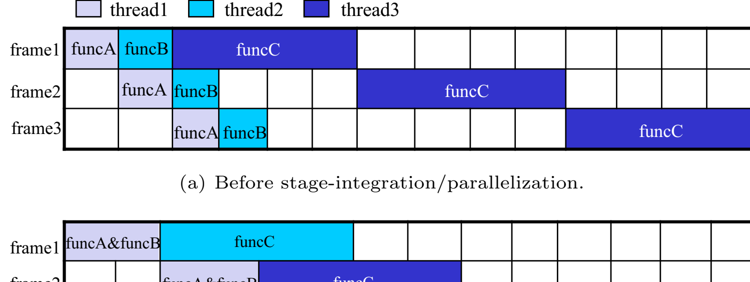
```
RV_Pipedata* GrayScale(RV_Pipedata* data){
  //Gray-scale processing for a frame
  return data;
}
RV_Pipedata* Laplacian(RV_Pipedata* data){
  //Laplacian filter processing for a frame
  return data;
}

int main(){
  RV_Pipeline pipe;      //Making an instance of pipeline class
  pipe.setParam(7,3);    //Setting priority of resolutions
  pipe.push(GrayScale); //Creating a Gray-scale stage
  pipe.push(Laplacian); //Creating a Laplacian stage
  pipe.run();            //Starting pipelining
  return 0;
}
```

図 7 RaVioli を用いたパイプラインプログラム
Fig. 7 An example of pipelining program with RaVioli

しかし，効率の良いパイプライン処理を実現することは容易ではない．まず各ステージ間で処理量に違いがあるため，プログラマはステージ間で処理量が均等になるように負荷分散を意識しなければならない．さらに，パイプライン処理を実現するために必要な FIFO などの機構の構築は動画像処理の本質ではなく，プログラマにとって煩雑である．

RaVioli ではこれらの問題に対しパイプライン処理を容易に実現するインターフェースを提供する．このインターフェースを用いてパイプラインステージを作成したとき，パイプライン化時に必要となる機構は自動的に定義され，さらにスレッドに割り当てるステージを負荷状況に応じ自動的に変動させることでパイプラインのスループットを向上させる．

RaVioli を用いたパイプライン処理の記述例を**図 7** に示す．プログラマは始めにパイプラインステージに割り当てるための，フレームを処理対象とした構成要素関数を定義する，次にその関数をパイプラインインスタンスの高階関数 push() に渡す．push() 関数はスレッドを作成し，渡された関数をパイプラインステージとしてスレッドに割り当てると同時に，必要な FIFO の作成も自動的に行う．このようにプログラマはステージで行う処理を記述



(a) Before stage-integration/parallelization.

(b) Integrated/parallelized the stages.

図 8 パイプライン処理
Fig. 8 Processing states of each frame

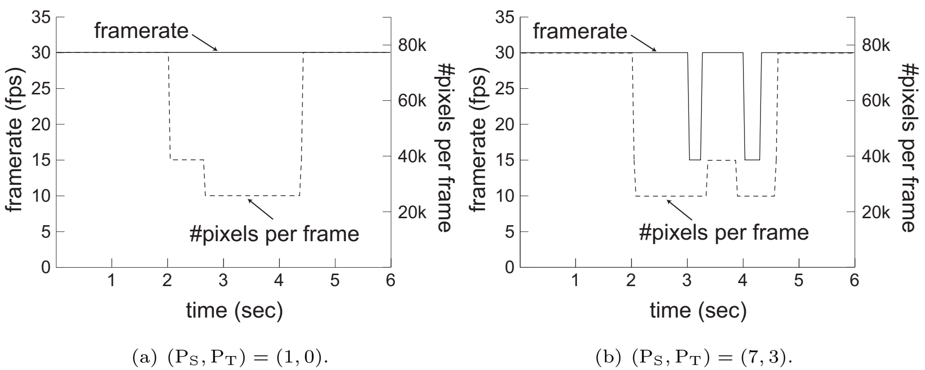
した関数を定義し，ステージの作成を行う push() 関数に渡すだけでパイプライン処理を容易に実現することが可能となる．

また，RaVioli はパイプラインステージ間で処理量の不均衡を検出したとき，スレッドに割り当てるステージを動的に変更することでステージの統合または並列化を行い，スレッド間で処理量を分散させる． 例えば**図 8(a)** に示すように，3 つの関数 `funcA`，`funcB`，`funcC` がパイプラインステージに割り当てられており，`funcC` の処理量が他のステージ `funcA` と `funcB` の約 4 倍である場合，`funcC` に割り当てられているスレッドはフレームごとにストールが発生する．そのため，`funcC` のステージがボトルネックとなりパイプライン全体のスループットが低下してしまう．RaVioli は FIFO 内のデータ数を監視することでステージ間で処理量の不均衡が発生していないかどうかを検出する．もしあるパイプラインステージがボトルネックとして検出された場合は，直ちにそのステージを並列化する．並列化は複数のスレッドを 1 つのステージに割り当てることで実現している．割り当てられた複数のスレッドはパイプラインレジスタから交互にフレームを取得し，処理する．また，隣接した処理量の少ないステージを 1 つのスレッドに割り当て，複数のステージを統合する．こうすることで**図 8(b)** に示されるようにストールの発生を抑えることが可能になりパイプラインのスループットを向上することができる．

プログラマはこれら 2 つの並列化機構を有効に使うことで，容易に並列処理を実現することが可能となり，動画像処理プログラムのスループットを半自動的に向上させることがで

表 1 動画像処理評価環境

OS	Fedora core 9
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)
Resolution	320 × 240
Framerate	30fps



(a) $(P_S, P_T) = (1, 0)$.　(b) $(P_S, P_T) = (7, 3)$.

図 9 優先度を設定した時の解像度の変化
Fig. 9 Resolution transitions with several priorities.

きる．

5. 評　　価

5.1 リアルタイム性の評価

RaVioli の自動解像度変動機構を，フレーム間差分プログラムを用いて評価した．評価環境を**表 1** に示す．

3.2 節で述べた優先度セットを $(P_S, P_T) = (1, 0)$ および $(P_S, P_T) = (7, 3)$ に設定した場合の評価結果を**図 9** に示す．これは処理開始から 6 秒後までの解像度の変化を表したグラフで，処理開始 2 秒後から 4 秒後まで，他のプロセスを動作させ負荷を与え，評価プログ

表 2　並列化評価環境

OS	Solaris 10
CPU	Sun UltraSPARC T1
Frequency(MHz)	1.0 GHz
Number of cores	4
Threads per core	8
Memory	16BG
Compiler	Sun Studio 12 (Sun C++ 5.9)
Compiler option	-fast -m64 -xchip=ultraT1
Thread library	pthread

ラムの使用可能な CPU リソースを減少させた．

図 9(a) では，時間解像度は維持され，空間解像度だけが 2 秒後から低下している．また，負荷を与えていたプロセスが終了する約 4 秒後から空間解像度は初期の定常状態に戻っていることが分かる．一方，**図 9(b)** では，両解像度ともに減少している．また，時間解像度より空間解像度の減少する割合が大きくなっていることが分かる．この結果から，RaVioli はシステムが高負荷時に，プログラマが設定した優先度に応じて解像度を適切な値に変動させることを確認した．

また，RaVioli を用いたプログラムの記述能力は，ラプラシアンフィルタ，テンプレートマッチング，ハフ変換，画像の回転などの様々なアプリケーションを実装することで確認した．RaVioli を用いた場合と用いなかった場合のフレーム間差分プログラムを付録 A.2 に示す．

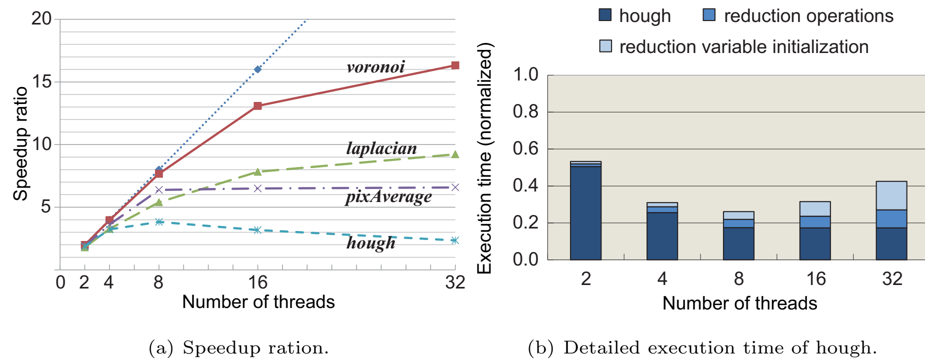
5.2　並列化機構の評価

次に**表 2** に示す環境で 2 種類のプログラム並列化手法の評価を行った． Sun UltraSPARC T1 は 8 つのコアを持ち，チップ・マルチスレッディング・テクノロジ (CMT)[9] により同時に 32 スレッドを動作させることが可能である．

5.2.1　空間分割並列化

4.1 で述べたプリプロセッサを用い，次に示す評価プログラムを自動並列化したときの，並列処理による速度向上の評価を行った．

- *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*: 近傍処理によるエッジ抽出処理
- *pixAverage*: 画素の平均値の算出処理



(a) Speedup ration.　(b) Detailed execution time of hough.

図 10　UltraSPARC T1 での評価プログラムの実行結果
Fig. 10　Sample programs executed on UltraSPARC T1 processor.

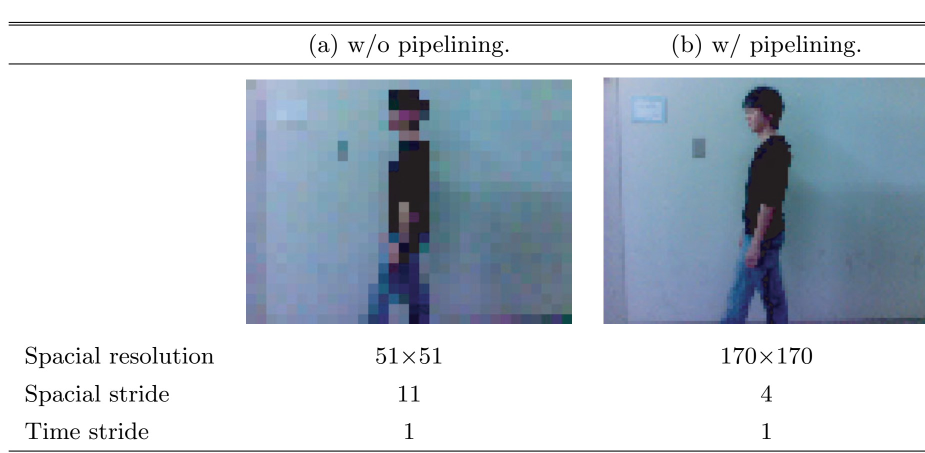
- *hough*: $\rho - \theta$ パラメータの投票による線検出

このうち *pixAverage* と *hough* の 2 つはリダクション処理を必要とするプログラムである．**図 10(a)** に逐次プログラムに対する速度向上比を示す．8 並列まではすべてのプログラムで並列度に近い高速化率となっていることから，プログラムの書き換え必要なしに，RaVioli のプリプロセッサによる並列化だけで UltraSPARC T1 プロセッサの 8 つのコアを有効活用できていることが確認できた．とくに *voronoi* と *laplacian* の 2 つのプログラムでは，並列数を大きくしたとき，各コアでマルチスレッド処理されることにより，コア数である 8 を越えて速度が向上している．

しかし，*hough* では 8 並列以降で速度が低下している．*hough* の詳細な処理時間の内訳を**図 10(b)** に示す．図 10(b) の各グラフは 1 スレッドで実行した場合の実行時間を 1 として正規化した場合の実行時間である． 図 10(b) に示されるように 8 並列以上では，リダクション処理とリダクション変数の初期化のオーバーヘッドが，処理時間に対し大きな割合を占めていることが分かる．これは，今後リダクション処理に SIMD 命令を用いるなどプリプロセッサおよび RaVioli の改良により解決していくべき問題である．

5.2.2　パイプライン化

次に，自動負荷均衡化機能を備えるパイプライン機構の評価を行った．評価に用いたプログラムは，3 つのステージがあり，それぞれを異なるスレッドに割り当てた．ここで，第 3 ステージの負荷は第 1，第 2 ステージの 4 倍とした．また，時間解像度が優先されるよう



	(a) w/o pipelining.	(b) w/ pipelining.
Spacial resolution	51×51	170×170
Spacial stride	11	4
Time stride	1	1

図 11 手動パイプライン化を行った場合と本パイプライン化機構を用いた場合の処理画像
Fig. 11 Output images w/o and w/ pipelining mechanism.

に，優先度セットを $(0, 1)$ として設定した．

パイプライン機構を用いず手動でパイプライン化した場合の結果を**図 11(a)** に示し，パイプライン機構を用いた場合の評価結果を**図 11(b)** に示す．パイプライン機構のステージ統合・並列化による自動負荷均衡化機能により空間解像度ストライドを 11 から 4 に抑えられたことを確認した．

6. おわりに

本稿では 2 種の並列化機能を備える動画像処理ライブラリ RaVioli を提案した．RaVioli はプログラマから解像度という概念を隠蔽し，新しいプログラミングパラダイムを提供する．RaVioli は使用可能な CPU リソース量に応じて自動的に空間解像度および時間解像度を変動させる．それゆえ，プログラマは RaVioli を用いることで様々なプラットフォームに対応した動画像処理アプリケーションを容易に開発することができる．また，半自動ブロック分割並列化機構と自動負荷分散パイプラインインターフェースを提供する．

評価プログラムとして，フレーム間差分やボロノイ図を作成するプログラムを実装した．疑似リアルタイム機構が適切に動作することと，並列化機構による解像度維持を確認した．また，実際に様々なプログラムを記述することで記述能力の高さを確認した．

現在，高速化に加えて省電力技術も重要となってきている．今後の課題として，RaVioli に自動省電力化機能を実装することが考えられる．具体的には，入力フレームに変化がない場合，または入力の変動が小さい場合に，自動的に解像度を低下させることで消費エネルギーの削減を図る．また，メモ化[10),11)] 手法の実装，Cell/B.E.[12)] 等の様々なプラットフォームへの対応，CUDA[13)] を用いた GPU の活用などによるさらなる高速化が課題としてあげられる．さらに，RaVioli に適した新しい動画像処理言語の開発も併せて検討していきたい．

参 考 文 献

1) 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).
2) 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).
3) Köthe, U.: *VIGRA - Vision with Generic Algorithms*, 1.6.0 edition (2008).
4) Intel Corp.: *Open Source Computer Vision Library* (2001).
5) Reinders, J.: *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, O'Reilly (2007).
6) Dagum, L. and Menon, R.: OpenMP: an Industry Standard API for Shared-Memory Programming, *IEEE Computational Science and Engineering*, Vol. 5 (1998).
7) 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).
8) Sun Microsystems, Inc.: *Sun Studio 12: C++ Users Guide* (2007).
9) Spracklen, L. and Abraham, S.G.: Chip Multithreading: Opportunities and Challenges, *Proc. of 11th Int'l Symp. on High-Performance Computer Architecture (HPCA-11)*, pp.248–252 (2005).
10) Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).
11) Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp.245–250 (2007).
12) Sony Computer Entertainment: *Cell Broadband Engine Architecture*, 1.01 edition (2006).
13) NVIDIA Corp.: *NVIDIA CUDA Programming Guide*, 2.0 edition (2008).

付　　録

A.1　プリプロセッサによって生成された並列プログラム

Input: a sequential program

```
int pSum;
int pCnt;
int pMin;

RV_Pixel Ave(RV_Pixel 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_Image* input_img;
  input_img=new RV_Image();
//load image data
//store the data into input_img
  pCnt=0; pSum=0; pMin=256;

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

Output: a parallel program

```
#include<pthread>
int pSum,__initpSum; __thread int __pSum;
int pCnt,__initpCnt; __thread int __pCnt;
int pMin,__initpMin; __thread int __pMin;

RV_Pixel Ave(RV_Pixel 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_Image();
//load image data.
//store the data into input_img
  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;
}
```

A.2　フレーム間差分プログラム

A program written with RaVioli

```
void CompFrame(RV_Pixel* Pnew, RV_Pixel* Pbfr){
  int r,g,b,br,bb,bg,ar,ag,ab;
  Pnew->getRGB(r, g, b); Pbfr->getRGB(br,bg,bb);
  ar=abs(r-br);
  ag=abs(g-bg);
  ab=abs(b-bb);
  if(ar > 77 || ag > 77 || ab > 77)
    Pnew->RGBabs(255,255,255);
  else
    Pnew->RGBabs(0,0,0);
}
void FrameDifference(RV_Image* newFlm, RV_Image* bfrFlm){
  newFlm->procImgComp(CompFrame, bfrFlm);
}
int main(){
  RV_Streaming video;
  video.setParam(3, 7);
  video.SetStreamProc(FrameDifference);
}
```

A program written with only C++

```
int main(){
  int ar,ag,ab;
  for(;;FrameNum+=ST){
    newFlm = readFrame(FrameNum);
    for(y=0; y<newFlm.H-SI; y+=SI){
      for(x=0; x<newFlm.W-SI; x+=SI){
        ar=abs(newFlm[x][y].R - bfrFlm[x][y].R);
        ar=abs(newFlm[x][y].R - bfrFlm[x][y].R);
        ar=abs(newFlm[x][y].R - bfrFlm[x][y].R);
        if(ar > 77 || ag > 77 || ab > 77)
           newFlm[x][y].R=newFlm[x][y].G=newFlm[x][y].B=255;
        else
           newFlm[x][y].R=newFlm[x][y].G=newFlm[x][y].B=0;
      }
    }
    bfrFlm = newFlm;
  }
}
```