卒業研究論文

解像度非依存型動画像処理ライブラリRaVioliのTBBを用いた自動並列化

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 20 年度入学　20115033 番

大平 真司

平成 24 年 2 月 8 日

解像度非依存型動画像処理ライブラリ RaVioli の TBB を用いた自動並列化

大平 真司

内容梗概

侵入者検知システムや自動車の衝突回避システムなどリアルタイム性の重要なシステムの開発が盛んに行われている．また，汎用計算機の高性能化と低価格化により，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，汎用システム上でリアルタイム動画像処理を行うことが今後多くなると予想される．しかし，汎用システムでは並行実行プロセスなどの外乱により，リアルタイム動画像処理に必要な CPU リソース量を常に確保することは困難である．

そこで，擬似的にリアルタイム性を保証する動画像処理ライブラリ RaVioli が提案されている．RaVioli では確保可能な CPU リソースの減少によりリアルタイム動画像処理が困難になった場合，解像度を変動させることで処理量を調整し，擬似的なリアルタイム性を保証している．また，RaVioli では解像度を動的に変動させるために，プログラマから解像度を隠蔽している．これにより，人間の映像認識過程には存在しない解像度という概念を排除することができ，より直感的な動画像処理プログラミングを可能にしている．しかし，RaVioli は抽象化のオーバヘッドによって処理速度が低下してしまうという問題点がある．そこで，GPU や Cell/B.E. などの特殊なプロセッサを利用した，RaVioli の高速化が取り組まれている．一方，マルチコアプロセッサの普及が進んでおり，汎用 PC にもそのようなプロセッサが搭載されつつある．そのため今後は，GPU や Cell/B.E. のような特殊なハードウェア環境だけでなく，一般的なマルチコア環境における高速化も必要になると考えられる．

一方で，マルチコア環境のための並列化フレームワークとして TBB が開発されている．TBB は，一般的に用いられる並列処理パターンを，テンプレートライブラリとして抽象化して提供している．これを用いることで，ループの並列処理やパイプライン処理などが比較的容易に実装可能である．そのため，一般的にデータ並列性やタスク並列性を持つ動画像処理の並列化に TBB を用いることで，処理速度の向上が期待できる．そこで本研究では，RaVioli に TBB を組み込んだ RaVioli/TBB を提案し，RaVioli の処理速度向上を目指す．また，従来の RaVioli で記述されたプログラムを，RaVioli/TBB で記述されたプログラムへと自動的に変換するプリプロセッサも提案する．これにより，従来の RaVioli と同様の記述方式で，TBB を用いた処理の高速化が

可能となる．

提案手法の有効性を確認するため，5つの画像処理プログラムを用いて，従来手法との実行速度を比較した．4並列で実行した結果，従来のRaVioliに対して，テンプレートマッチングプログラムにおいて最大3.8倍の速度向上が達成できることを確認した．

解像度非依存型動画像処理ライブラリ RaVioli の TBB を用いた自動並列化

目次

1 はじめに

侵入者検知システムや自動車の衝突回避システムなどリアルタイム性の重要なシステムの開発が盛んに行われている．また，汎用計算機の高性能化と低価格化により，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 PC および汎用 OS 上でリアルタイム動画像処理を行うことが多くなると予想される．

しかし，汎用 OS 上で，1/30 または 1/60 秒毎に処理を行うリアルタイム動画像処理の実現は困難である．その主な理由として，1 フレームあたりの処理量の変動や，複数プロセスの並行実行による利用可能な CPU リソース量の変動が挙げられる．

そこで，汎用システム上で擬似的にリアルタイム性を保証する動画像処理ライブラリ **RaVioli (Resolution-Adaptable Video and Image Operating Library)** 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) を変動させることで，擬似的にリアルタイム性を保証する．

このように，動的に解像度を変動させる場合，一般にプログラマが 1 フレームあたりの画素数やフレームレートの変動に対応した記述をしなければならない．しかし，これら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は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，RaVioli では，画像や画素のカプセル化のオーバヘッドにより処理速度が低下するという問題点がある．そこで，GPU や Cell/B.E. などの特殊なプロセッサを利用した，RaVioli の高速化が取り組まれている．一方，マルチコアプロセッサの普及が進んでおり，汎用 PC にもそのようなプロセッサが搭載されつつある．そのため今後は，GPU や Cell/B.E. のような特殊なハードウェア環境だけでなく，一般的なマルチコア環境における高速化も必要になると考えられる．

一方で，マルチコア環境のための並列化フレームワークとして **Intel TBB (Intel Threading Building blocks)**，(以下 TBB) が開発されている．TBB は，一般的に用いられる並列処理パターンを，テンプレートライブラリとして抽象化して提供している．これを用いることで，ループの並列処理やパイプライン処理などが比較的容易に実装可能である．そのため，一般的にデータ並列性やタスク並列性を持つ動画像処

理の並列化に TBB を用いることで，処理速度の向上が期待できる．そこで本研究では，TBB を用いることで RaVioli の処理速度の向上を目指す．そのために，RaVioli に TBB を組み込んだ RaVioli/TBB を提案する．また，従来の RaVioli で記述されたプログラムを，RaVioli/TBB で記述されたプログラムへと自動的に変換するプリプロセッサも提案する．これにより，従来の RaVioli と同様の記述方式で，TBB を用いた処理の高速化が可能となる．

以下，2 章では関連研究，動画像処理ライブラリ RaVioli，並列フレームワーク TBB について述べる． 3 章では TBB を組み込んだ RaVioli を提案し， 4 章では，従来の RaVioli を用いて記述されたプログラムを，RaVioli/TBB を用いて記述されたプログラムに自動変換するプリプロセッサについて述べる．次に 5 章で提案の評価とそれに対する考察を述べる．最後に 6 章で本論文全体をまとめる．

2 背景

本章では従来の動画像処理に存在する問題点を述べ，それらに対する既存研究について説明する．また，本研究の対象となる RaVioli と TBB の概要について説明する．

2.1 関連研究

前章で述べたように，汎用 OS 上では 1 フレームあたりの処理量の変動や，複数プロセスの並行実行による利用可能な CPU リソース量の変動などにより，リアルタイム動画像処理の実現は困難である．この問題の解決策の 1 つとして，利用可能な CPU リソース量に応じて動画像の処理量を調整するという方法がある．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，このモデルに基づき，処理精度および処理時間に関する知識を利用し，適切なアルゴリズムを動的に選択するアーキテクチャも提案されている [2]．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，プログラマの負担が大きくなる．

また，計算機を使って画像を処理する際には解像度という概念が不可欠である．従来の画像処理では，画像の幅や高さを用いて画像に対する処理内容を記述する．しかし，画像処理プログラムを記述する人間にとっては，解像度という概念は不自然である．そもそも人間が物体を認識する過程には解像度という概念は存在しないため，解像度を意識した画像処理プログラミングは直感的でない．

そこで，擬似的にリアルタイム性を保証する解像度非依存型動画像処理ライブラリ

RaVioli[3, 4] が提案されている．RaVioli では利用可能な CPU リソース量の不足によりリアルタイム処理が困難になった場合，解像度を自動調節することで処理量を調整し，リアルタイム性を保証する．また，動的に解像度を変動させるためにプログラマから解像度を隠蔽することにより，プログラマは解像度を意識することなく処理を記述できる．

一方で，動画像処理プログラミングを抽象化するためのライブラリはこれまでにも提案されている．その中でもよく知られた動画像処理ライブラリに VIGRA[5] や OpenCV[6, 7] がある．VIGRA 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 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，これらは単純に処理内容を抽象化してライブラリの形で提供しているものであり，プログラマは解像度を意識することなく画像に対する処理内容を記述することはできない．そのため，これらのライブラリが行う画像処理の抽象化は，RaVioli とは全く異なっている．

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 [8, 9]．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 に似ているが，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．

2.2 RaVioli

本節では，RaVioli の擬似的なリアルタイム性の保証について述べ，次に RaVioli を用いた画像処理について説明する．そして，RaVioli が持つ問題点についても述べる．

2.2.1 リアルタイム性の保証

2.1 節で述べたように，一般的に汎用 PC および汎用 OS 上では，リアルタイムに動画像を処理することは困難である．そこで RaVioli では，利用可能な CPU リソース量に応じて動画像の解像度を変動させ，処理量を調整することでこれを解決する．

動画像における解像度には空間解像度と時間解像度の 2 種類がある．空間解像度とは 1 フレームを構成する画素数のことである．一方，時間解像度とはフレームレートのことである．RaVioli は空間解像度および時間解像度を制御するための，**空間解像度ストライド (Ss)** と**時間解像度ストライド (St)** を持っており，利用可能な CPU リソース量に応じて，これらのストライドを増減させることで処理量の調整を実現している．

ここで，空間解像度を変動させるときの処理方法を**図 1** に示す．空間解像度ストラ

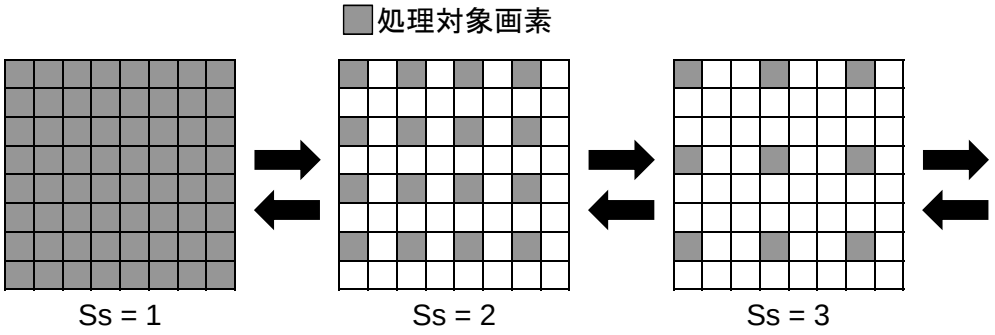

図 1: 空間解像度ストライド (Ss) の変更

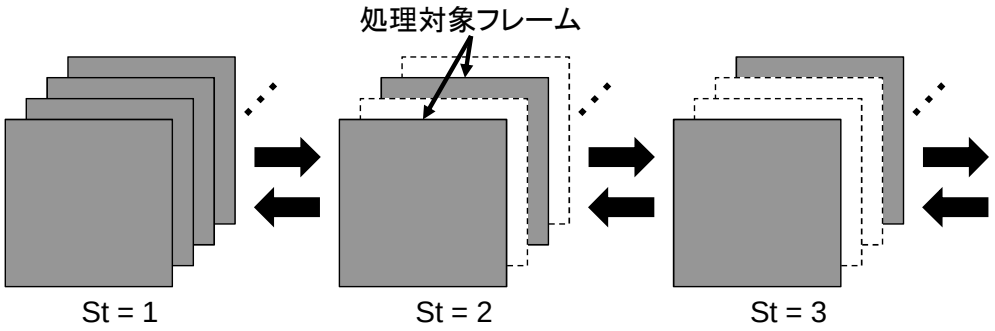

図 2: 時間解像度ストライド (St) の変更

イド Ss= 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 Ss= 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 Ss= 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 Ss= 3 とすると，処理画素数は 1/9 となる．

一方，時間解像度を変動させるときの処理方法を**図 2** に示す．時間解像度ストライド St= 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 St= 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 St= 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 St= 3 とすると，処理フレーム数は 1/3 となる．

なお，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，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

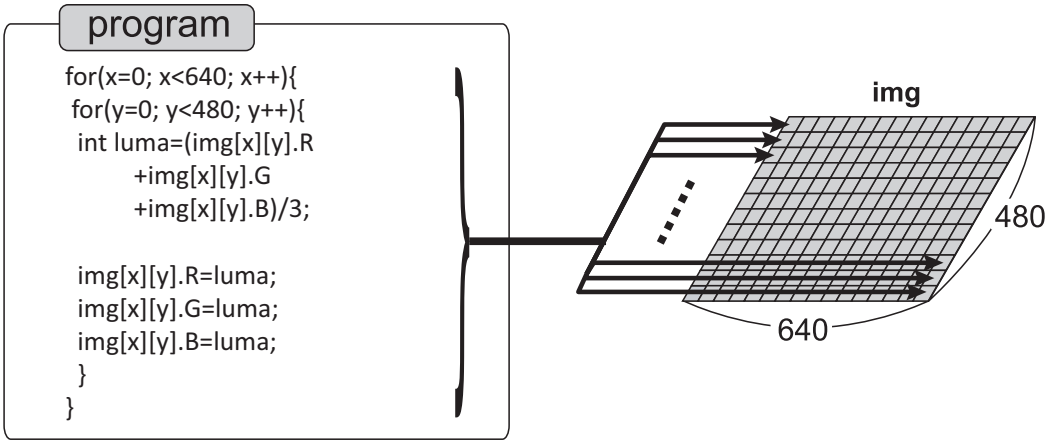

図 3: 一般的な動画像処理プログラム記述例

に設定することで，時間解像度が優先的に低減され，精細さを確保しつつリアルタイム性を実現することができる．

2.2.2 RaVioli を用いた画像処理

前項で述べたように，RaVioli は擬似的にリアルタイム性を保証するために，空間解像度および時間解像度を制御している．そのため，RaVioli では，プログラマからこの 2 つの解像度を隠蔽する．これにより，プログラマは動画像のフレーム数や画像の幅や高さを意識した記述を省略できる．

RaVioli を使用しない一般的な画像処理では，**図 3** のように，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，量子化された動画像データを扱うためにループがよく用いられるが，ループを用いる画像処理では，プログラマは画像の幅と高さを意識した記述をしなければならない．

一方，RaVioli では，画像の幅，高さや画素配列を RV_Image クラスに，画像を構成する画素の色情報などを RV_Pixel クラスにそれぞれカプセル化している．そして，画像の構成要素である画素や部分画像，また動画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，動画像中の全ての構成要素に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**と呼び，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 を用いて画像をグレースケールに変換する処理の様子を**図 4** に示す．1 画素をグレースケール化する処理を記述した関数 GrayScale() を定義し，この関数を RV_Image インスタンスの高階メソッド

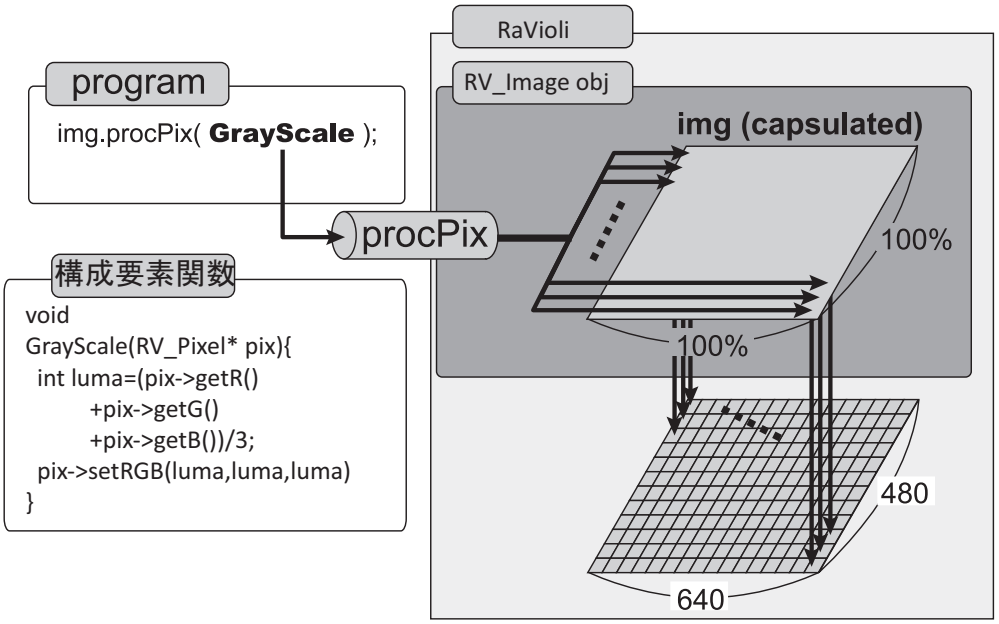

図 4: RaVioli 使用時プログラム記述例

procPix() に渡す．高階メソッドの procPix() が，RV_Image インスタンスが持つ画像の全ての画素に GrayScale() を繰り返し適用することで，図 3 と同様の処理が実現できる．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を記述できる．

2.2.3 RaVioli が持つ問題点

前項で述べたように，RaVioli はプログラマから解像度を隠蔽し，より直感的なプログラミングを可能にしている．しかし，解像度隠蔽のためのオーバヘッドによって処理速度が低下してしまうという問題点がある．

ここで，**表 1** に RaVioli 不使用時と使用時の処理速度の比較と処理時間の増加率を示す．表中の C++は RaVioli を使用せず C++で記述したプログラム，RaVioli は RaVioli を使用して記述したプログラムを表す．また処理時間の増加率は，RaVioli 使用時の処理時間が RaVioli 不使用時から何倍に増加したかを表す．ここで処理時間は，画像入出力の時間は含めない，画像に対する処理のみの実行時間である． 表 1 に示すように，RaVioli を使用した場合，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でそれぞれ約 5.1 倍，1.6 倍，1.9 倍の速度低下がみられた．

このような RaVioli の処理速度の低下を解決するために，GPU や Cell/B.E. を利用した高速化が行われている．GPU はグラフィックボード上に搭載されている画像処理に特化したプロセッサである．このプロセッサは簡素化された命令発行方式を採用す

表 1: 画像処理の速度比較 (ms)

プログラム名	C++	RaVioli	処理時間の増加率 (倍)
グレースケール化	0.899	4.621	5.140
エンボスフィルタ	5.234	8.474	1.619
テンプレートマッチング	3657.100	6794.989	1.858

CPU:Core2Quad(2.83GHz)，memory:3GB

ることで，チップ上における命令制御関連のユニットの占める面積を抑え，チップ上の面積の多くを演算処理ユニットに割いている．そのため，汎用プロセッサと比べ高い演算性能を持つ．また，Cell/B.E. は PS3 上に搭載されているプロセッサであり，1 個の汎用的なプロセッサコアと 8 個の演算プロセッサコアを組み合わせたヘテロジニアスマルチコアである．このように，これらのプロセッサはアーキテクチャ構造が特殊であるため，これらを利用した高速化手法は，それぞれ独自のものとなってしまう．そのため，これらの高速化手法をそのまま汎用プロセッサに適用することは困難である．一方で，汎用プロセッサのメニーコア化が進んでおり，汎用 PC にもマルチコアプロセッサが搭載されるようになってきた．そのため今後は，特殊なプロセッサだけでなく，より一般的なマルチコア環境における RaVioli の高速化も必要になると考えられる．そこで本研究では，マルチコア環境のための並列化フレームワークである TBB[10] に着目する．

2.3 TBB

マルチコアプロセッサの普及に伴い，汎用 PC にもそのようなプロセッサが搭載されつつある．ここでマルチコアプロセッサを有効に活用するためには，並列プログラミングによって，タスクを適切にコアへ割り当てる必要がある．よく知られる並列プログラミング手法としては，ネイティブスレッドのインターフェースである Pthread を用いた手法がある．しかし Pthread を用いる場合，タスクの分割や分割したタスクのスレッドへの割り当てといった処理をプログラマ自身が記述しなければならないため，プロセッサ上の複数のコアを十分に活用したプログラムの作成は困難である．

そこで，マルチコアプロセッサを有効に活用するプログラムを作成するために，並列プログラミングをサポートするフレームワークが多く開発されている．それらの 1 つに TBB がある．TBB は Intel により開発された C++テンプレートライブラリであ

表 2: TBB のテンプレート関数およびテンプレートクラス

テンプレート	実装できる並列処理
parallel_for 関数	ループ間で依存性がないループの並列処理
parallel_reduce 関数	リダクション処理を含むループの並列処理
parallel_scan 関数	並列プリフィックスの計算処理
parallel_while クラス	構造化されていないストリームの並列処理
pipeline クラス	パイプライン処理

り，マルチコア環境における CPU リソースの効率的な利用を目的としている．TBB を用いる場合，プログラマは任意のクラスを定義し，そのクラス内に並列化したい処理内容をタスクとして逐次プログラムと同様に記述する．そして，そのクラスを TBB が提供しているテンプレート関数あるいはテンプレートクラスに引数として渡す．こうすることで，TBB の内部でタスクのスケジューリングが行われ，処理を並列実行することが可能になる．

ここで，TBB が提供しているテンプレート関数およびテンプレートクラスを **表 2** に示す．parallel_for 関数，parllel_reduce 関数および parallel_scan 関数は，ループ処理を並列実装するためのテンプレート関数である．一方，parallel_while クラスと pipeline クラスは，ストリーム処理を並列実装するためのテンプレートクラスである．このように，TBB はループの並列処理やパイプライン処理などを並列実装するための枠組みを提供している．

一方で，動画像処理は並列性を持っている場合が多い．例えば，動画像中の 1 フレームにあたる画像は静止画像であり，その画像中の画素配列は一般的にデータ並列性を持っていることが多いため，画素データに対する処理を並列に実行することが可能である．また，複数フレームの処理間に依存関係がない場合は，複数フレームをパイプライン処理することも可能である．これらの並列処理は，前述したように TBB を用いることで実装可能である．このため，動画像処理に TBB を適用することで，動画像処理を高速化できる可能性が高いと考えられる．

また TBB は，タスクのスケジューリングをランタイム時に行っており，タスクをより小さなタスクに再帰的に分割し，スレッドに割り当てている．そのため，プロセッサのコア数を考慮してプログラムを書き分ける必要がなく，今後より多くのコアを搭載するプロセッサが登場したとしても，それに対応することができる．このように，プ

ログラマがプラットフォームを意識する必要なくプログラムが記述できるという特徴も TBB は持っており，これは様々なプラットフォームでの動作が必要とされる動画像処理において非常に有用である．

3 RaVioli/TBB

2.2.3 項で述べたように，RaVioli は処理速度が低下してしまうという問題を持っており，一般的なマルチコア環境における高速化が必要になると考えられる．一方，TBB を用いると，マルチコア環境において CPU リソースを効率良く利用した並列処理が実装可能である．そこで，一般的なマルチコア環境で処理を高速化するために，RaVioli に TBB を組み込んだ RaVioli/TBB を提案する．ここで TBB を組み込むために，RaVioli の高階メソッドを拡張する．また，並列化に伴って共有変数の競合が発生する場合にも処理の整合性が保てるようにする．本章では，高階メソッドの拡張と共有変数の競合が発生する場合への対応について述べる．

3.1 高階メソッドの拡張

2.2.2 項で述べたように RaVioli を用いた画像処理では，プログラマは 1 画素や近傍画素集合などに対する処理を構成要素関数として定義するのみである．そして，ライブラリ内の高階メソッドが構成要素関数を受け取ることにより，画像内の各画素に処理が適用される．そのため，画像の構成要素に対する処理単位が明確であり，データ並列性が抽出し易い．

一方で， 2.3 節で述べたように TBB は処理を並列実装するためのテンプレート関数やテンプレートクラスを提供している．本研究では RaVioli の高速化にあたって，ループの並列化のためのテンプレート関数の中で，最も基本となる parallel_for を利用する．

この parallel_for の使用方法をプログラム例を示しながら説明する．ここで， **図 5** は要素数が N である int 型の配列 array に 0 から N-1 を順に代入していく単純な逐次プログラムである．また， **図 6** は， 図 5 のプログラムを parallel_for を利用して並列化したプログラムである．

parallel_for を利用する際には，まず専用のクラスを定義しなければならない． 図 6 の例では Parallel という名前の専用クラスを定義している (1〜11 行目)．このクラスではメンバ変数として N 要素の int 型配列 array を保持し (3 行目)，配列 array_ptr を引数にとるコンストラクタを定義する (5 行目)．これにより，外部から受け取った配列を扱えるようする．

```
1  int main(){
2    int array[N];
3    for(int i=0;i<N;i++)
4      array[i]=i;
5  }
```

図 5: parallel_for を用いない逐次プログラム

```
1  class Parallel{
2  private:
3    int array[N];
4  public:
5    Parallel(int array_ptr[]):array(array_ptr){}
6
7    void operator()(const blocked_range<int>& range)const{
8      for(int i=range.begin();i<range.end();i++){
9        array[i]=i;
10     }}
11 };
12
13 int main(){
14   int array[N];
15   Parallel obj(array);
16   parallel_for(blocked_range<int>(0,N),obj,auto_partitioner());
17 }
```

図 6: parallel_for を用いた並列プログラム

また，専用クラス内でコピーコンストラクタおよびデストラクタを定義する必要がある．しかし，コピーコンストラクタおよびデストラクタはプログラマが記述しなかった場合，コンパイル時にコンパイラが自動的に定義するため，それらの記述は省くことできる．そのため，例ではコピーコンストラクタおよびデストラクタの記述は省いている．

さらに，専用クラス内では並列化したいループ処理を operator() にオーバーロードする必要がある (7〜10 行目)．このとき，仮引数には TBB の blocked_range<T>クラスを用いる．このクラスはテンプレート型 T の一次元の反復区間であることを示しており，ループ範囲などの情報を管理している．また TBB は二次元の反復区間を示す blocked_range2d<T,T>クラスも提供している．ここで，図 5 の 3，4 行目から，並列化したいループが一次元であり，ループのイタレータ変数として int 型の変数が使用されていることがわかる．そのため，図 6 では blocked_range<int>クラスを使用している (7 行目)．また，for ループにおけるイタレータ変数の初期化および終了判定には，blocked_range<int>クラスのインスタンス range が持つメンバ関数 begin() および end() を用いる (8 行目)．そして，ループの本体部分は，逐次の場合と同様に記述する (9 行目)．

専用クラスを定義した後は，専用クラス Parallel をインスタンス化する (15 行目)．インスタンス化の際に，14 行目に宣言してある配列 array を引数として与えることで，処理するデータをインスタンスに渡している．次に，parallel_for を呼び出す (16 行目)．parallel_for の呼び出し時には，ループ範囲，専用クラスのインスタンス，ループの反復区間の分割方法の 3 つを引数として与える．ループ範囲は，blocked_range<T>クラスの第 1 引数にループ開始の値，第 2 引数にループ終了の値をそれぞれ与えることで指定できる．図 6 では，第 1 引数に 0 を与え，第 2 引数に N を与えることで，0 から N までの範囲を指定している (16 行目)．また，ループの反復区間の分割方法には，auto_partitioner()，simple_partitioner()，affinity_partitioner() の 3 種類がある．まず，auto_partitioner() は，TBB 側で自動的に粒度を決め，反復区間を分割する．次に，simple_partitioner() は，プログラマから指定された粒度を元に反復区間を分割する．最後に，affinity_partitioner() は，キャッシュを有効に活用できるように反復区間を分割する． 図 6 において，ループの反復区間の分割方法には，auto_partitioner() を指定している (16 行目)．

parallel_for では，先述した分割方法の中から，引数として与えられた分割方法を元に反復区間を分割する．ここで，分割後の各反復区間のことをチャンクと呼ぶ．実行

```
1  //専用クラス
2  class _proc_tbb{
3  private:
4    RV_Image* Img;
5    void (* UserProgram)(RV_Pixel*);
6  public:
7    _proc_tbb(RV_Image* _Img,void (* _UserProgram)(RV_Pixel*))
8      :Img(_Img),UserProgram(_UserProgram){}
9
10   void operator()(const blocked_range2d<int,int>& range)const{
11     blocked_range2d<int,int>::row_range_type w=range.rows();
12     blocked_range2d<int,int>::col_range_type h=range.cols();
13     for(int y=h.begin();y<h.end();y++){
14       for(int x=w.begin();x<w.end();x++){
15         UserProgram();
16       }}}
17 };
18
19 //高階メソッド
20 void proc_tbb(void (* UserProgram)(RV_Pixel*)){
21   _proc_tbb obj(this,UserProgram);
22   parallel_for(blocked_range2d<int,int>(0,width,0,height),
23                obj,auto_partitioner());
24 }
```

図 7: parallel_for を組み込んだ高階メソッドの実装方法

時には，このチャンクを個別のスレッドに割り当て実行させることで並列化している．以上のように parallel_for を利用し，ループ処理を並列化することができる．

この parallel_for を使用し，RaVioli の並列化を実現する．そのために，RaVioli の高階メソッドを拡張する．従来の高階メソッド proc に対し，parallel_for を組み込んだ高階メソッド proc_tbb の具体的な実装方法を **図 7** に示す．

まず専用クラスのメンバ変数として，RV_Image クラスのインスタンスへのポインタ Img(4 行目) と構成要素関数 UserProgram へのポインタ (5 行目) を保持する．次に専用クラス内で，画像中の全ての画素に対し構成要素関数 UserProgram を適用する処理 (13〜16 行目) を operator() にオーバーロードする．この処理は二次元のループに対する処理であるため，blocked_range2d<int,int>クラスを用いる (10 行目)．こ

のクラスは，二次元のループ範囲を扱いやすくするために，内側のループに対して row_range_type，外側のループに対して col_range_type という型を用意している．これらの型は，blocked_range<T>クラスの別名として定義されている．また，これらの型のインスタンス生成時に，blocked_range2d<T,T>のメンバ関数 rows() および cols() を呼び出し，これらを用いて初期化することで，内側のループおよび外側のループの範囲の情報をインスタンスに渡すことができる．図 7 の例では，row_range_type 型のインスタンス w を range のメンバ関数 rows() を用いて初期化している (11 行目)．同様に，col_range_type 型のインスタンス h を range のメンバ関数 cols() を用いて初期化している (12 行目)．このようにして，インスタンス w および h に内側のループおよび外側のループにおけるループ範囲の情報をそれぞれ保持させる．そして，これらのインスタンスを for ループにおけるイタレータ変数の初期化および終了判定に用いる (13, 14 行目)．

一方，高階メソッド内では，専用クラス_proc_tbb のインスタンス化 (21 行目) および parallel_for の呼び出しを行う (22，23 行目)．ここで，専用クラスのコンストラクタでは，引数として RV_Image クラスのインスタンスと構成要素関数のそれぞれのポインタを受け取り，専用クラス内のメンバ変数を初期化するようにしている (7，8 行目)．そのため，専用クラスのインスタンス化の際には，第 1 引数として this ポインタを与えることで，高階メソッドの呼び出し元である RV_Image クラスのインスタンスへのポインタを渡し，第 2 引数には高階メソッドの引数として受け取った構成要素関数のポインタを渡す (21 行目)．続いて，parallel_for の呼び出しの際には，図 6 の例と同様に 3 つの引数を与えるが，ここで，ループ範囲の指定には blocked_range2d<int,int> クラスを用いる (22 行目)．このクラスの第 1 引数に 0 を，第 2 引数に画像の幅を表す RV_Image クラスのメンバ変数 width を与えることで，内側のループの範囲を画像の幅のサイズに指定する．また，第 3 引数に 0 を，第 4 引数に画像の高さを表す RV_Image クラスのメンバ変数 height を与えることで，外側のループの範囲を画像の高さのサイズに指定する．

以上のように，parallel_for を内部で使用するよう高階メソッドを拡張した結果，プログラマは拡張した高階メソッドを呼び出すだけで，parallel_for を用いた処理の並列化が可能となる．ここで，従来の高階メソッドと拡張した高階メソッドのそれぞれの呼び出しの例を **図 8** に示す．4 行目は従来の高階メソッド呼び出しである．ここでは，構成要素関数 GrayScale を引数に渡して高階メソッドを呼び出している．一方，拡張後の高階メソッド呼び出しの場合，5 行目のように従来の高階メソッドの末尾に_tbb を付

```
1 int main(){
2   RV_Image img;
3   readBMP(img);
4   img.proc(GrayScale);      // RaVioli
5   img.proc_tbb(GrayScale); // RaVioli/TBB
6 }
```

図 8: 高階メソッド呼び出しの例

け加えるだけで，拡張した高階メソッドを呼び出すことができる．この高階メソッドの引数には，従来の RaVioli の場合と同様に構成要素関数を渡すだけでよい．このようにプログラマは従来のプログラムに対してほとんど変更を加えることなく，parallel_for を組み込んだ高階メソッドを呼び出すことができる．

3.2 共有変数の競合への対応

一般的に画像処理は，画素毎の処理に順序依存が無い場合が多い．そのような場合は，画素毎の処理を並列に実行することができる．しかし，順序依存が無い場合でも，並列化することにより複数のスレッドが 1 つの共有変数に対して読み出しおよび書き込みをして競合が生じる場合がある．このような競合に対する一般的な解決手法として，共有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．しかし排他制御する場合，1 つのスレッドが共有変数の読み書きをしている間，他のスレッドはその変数に対してアクセスができず，処理が進められないため，並列性が低下してしまう．一方，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し，最後に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と，最後に行う統合処理を除き各スレッドは完全に独立して動作するため，排他制御する場合よりも並列度を高く保つことができる．

そこで，本研究では共有変数の競合が発生する場合，リダクション処理を利用することでこの問題に対応する．そのために，前節で述べた高階メソッドをさらに拡張した，リダクション処理が適用可能な高階メソッドを新たに実装する．この高階メソッドの具体的な実装方法を **図 9** に示す．

まず，専用クラスのメンバ変数としてリダクション処理用の関数 Reduction のポインタを追加し (6 行目)，コンストラクタの引数にも同様の関数ポインタを追加する (9 行目)．これにより，専用クラスが外部からリダクション処理用の関数を受け取れるよ

```
1  //専用クラス
2  class _proc_tbb_reduction{
3  private:
4    RV_Image* Img;
5    void (* UserProgram)(RV_Pixel*);
6    void (* Reduction)();
7  public:
8    _proc_tbb
9    (RV_Image* _Img,void (*_UserProgram)(RV_Pixel*),void (*_Reduction)())
10     :Img(_Img),UserProgram(_UserProgram),Reduction(_Reduction){}
11
12   void operator()(const blocked_range2d<int,int>& range)const{
13     blocked_range2d<int,int>::row_range_type w=range.rows();
14     blocked_range2d<int,int>::col_range_type h=range.cols();
15     for(int y=h.begin();y<h.end();y++){
16       for(int x=w.begin();x<w.end();x++){
17         UserProgram();
18       }}
19     Reduction();
20   }
21 };
22
23 //高階メソッド
24 void proc_tbb(void (* UserProgram)(RV_Pixel*),void (* Reduction)()){
25   _proc_tbb_reduction obj(this,UserProgram,Reduction);
26   parallel_for(blocked_range2d<int,int>(0,width,0,height),
27                obj,auto_partitioner());
28 }
```

図 9: リダクション処理が適用可能な高階メソッドの実装方法

```
1  int count=0;
2  __thread int local_count=0;
3  spin_mutex mutex;
4
5  void Increment(RV_Pixel* p){
6    // count+=1;
7    local_count+=1;
8  }}
9
10 void Reduction(){
11   spin_mutex::scoped_lock lock(mutex);
12   count+=local_count;
13   local_count=0;
14 }
15
16 int main(){
17   RV_Image* Image;
18   Image->proc_tbb(Increment,Reduction);
19 }
```

図 10: リダクション処理が適用可能な高階メソッドの使用例

うにする．そして operator() のオーバーロードでは，構成要素関数の適用処理後にこのリダクション処理用の関数を呼び出す (19 行目)．

一方，高階メソッドの引数にリダクション処理用の関数を追加し (24 行目)，この関数を専用クラスのインスタンス化時に引数として渡す (25 行目)．このようにして，高階メソッドの引数としてリダクション処理用の関数を受け取り，高階メソッドの呼び出し時にその関数を呼び出すことで，リダクション処理を適用することができる．

ここで，この高階メソッドの使用例を **図 10** に示す．構成要素関数 Increment は，画像中の画素数をカウントしている．ここで，このカウント処理を並列化すると，画素数のカウントに使用している大域変数 count に対して競合が発生する．そこで，その変数 count に対して，スレッド毎の一時格納領域としてスレッドローカル変数 local_count を定義する (2 行目)．スレッドローカル変数は宣言時に __thread 指定子をつけることで指定でき，その変数のコピーがスレッド毎に保持される．そして，構成要素関数 Increment では local_count に 1 加え画素数をカウントする (5〜7 行目)．

また，リダクション処理用の関数 Reduction を定義する (10 行目)．この関数内では

スレッド毎に保持される画素値の合計 local_count を大域変数 count に足し合わせる (12 行目)．ここで， 3.1 節で述べたように，parallel_for を使用すると，実行時に反復区間がチャンクという小さい単位に分割され，このチャンクが，それぞれのスレッドで実行される．このとき，チャンクの数はスレッド数よりも多くなるよう分割され，各スレッドは複数のチャンクを逐次的に処理する．このため，スレッドの実行終了時にスレッドローカル変数を初期化することで (13 行目)，次のチャンクの実行で，結果に不整合が生じないようにする必要がある．また，統合処理はスレッド毎に排他的に実行されなければならないため，リダクション処理用の関数 Reduction を排他的に実行する必要がある．この排他制御には，TBB が提供している同期プリミティブの中から，命令数が少ない場合に処理速度が速いという特徴を持つ spin_mutex クラスを使用する．これは，リダクション処理用の関数には統合処理およびスレッドローカル変数の初期化処理のみが記述されるため，命令数があまり多くならないと考えられるためである．spin_mutex クラスを利用するには，大域変数として spin_mutex クラスをインスタンス化する (3 行目)．次に，排他制御したいスコープ内において，ロックをかけるための記述をする (11 行目)．

最後に，高階メソッド呼び出しでは，拡張した高階メソッド proc_tbb を指定し，第 1 引数として構成要素関数 Increment，第 2 引数としてリダクション処理用の関数 Reduction をそれぞれ渡す (18 行目)．このようにして，プログラマが記述したリダクション処理を適用することができ，これにより処理結果の正当性を保証することができる．

4　自動変換プリプロセッサ

本章では，従来の RaVioli を用いた逐次プログラムを RaVioli/TBB を用いた並列プログラムに自動的に変換するプリプロセッサについて述べる．

4.1　プリプロセッサの動作フロー

3.2 節で述べたように，リダクション処理を適用する場合は，プログラマはスレッドローカル変数の定義およびその変数を使った処理の記述やリダクション処理用の関数の追加など，並列化を意識した記述が必要となる．また，リダクション処理が必要であるか否かを判断し，プログラムを書き分けなければならないため，これもプログラマに並列化を意識させ，負担になってしまうと考えられる．そこで，従来の RaVioli を用いた逐次プログラムから RaVioli/TBB を用いた並列プログラムに自動的に変換するプリプロセッサを実装する．

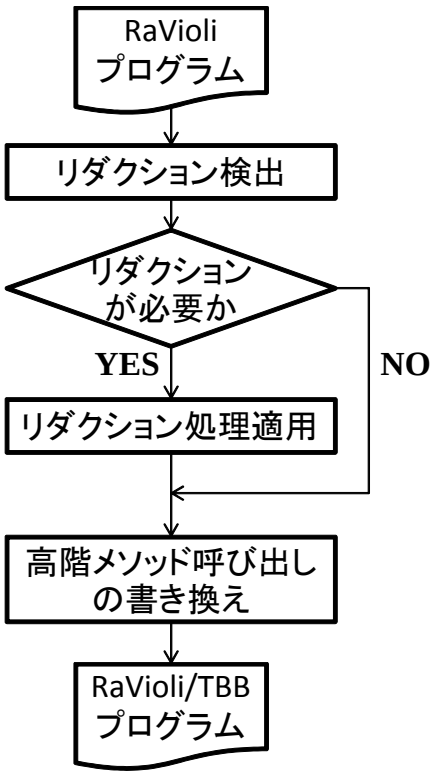

図 11: プリプロセッサの動作フロー

プリプロセッサの動作フローを **図11** に示す．プリプロセッサは，入力として RaVioli を用いた逐次プログラムを受け取る．そして，そのプログラムを解析し，構成要素関数内にリダクション処理が必要な変数が存在するか否かを判定する．リダクション処理が必要な変数が存在する場合，リダクション処理を適用する．そして，高階メソッド呼び出し部分の書き換えを行う．全ての変換が終わると，プリプロセッサは RaVioli/TBB プログラムをファイルに出力する．以下，4.2 節でリダクション処理の検出方法について述べ，4.3 節でリダクション処理の適用方法について述べる．

4.2 リダクション処理の検出

3.2 節で述べたように，複数スレッドによる共有変数へのアクセスが競合を発生させてしまう場合，本提案ではこの共有変数に対してリダクション処理を適用することで競合を回避している．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メソッド内に存在するため，RaVioli プログラムにおいて競合が発生しうる変数は構成要素関数内の大域変数に該当する．そこで，RaVioli プログラムを解析した結果，同じスコープ内で大域変数に対する読み出しおよび書き込みが両方行われている

```
1  int global0,global1,global2,global3,global4;
2  void func(RV_Pixel* p){
3    global0=100;                  // 読み出しのみ
4    global1=global1+p.getR(); // 読み出しおよび書き込み
5    global2+=1;                   // 読み出しおよび書き込み
6
7    if(global3 > p.getG()){   // 読み出し
8      global3=p.getG();         // 書き込み
9      global4=p.getB();         // 書き込みのみ
10   }
11 }
```

図 12: リダクション処理が必要な変数の例

事が検出された場合，その大域変数はリダクション処理が必要であると判断する．ここで，リダクション処理が必要な変数について **図 12** に示すサンプルプログラムを用いて説明する．

まず (左辺)=(右辺); のような一般的な式について考える (3〜5 行目)．このような式で，3 行目のように (左辺) にのみ大域変数が使用されている場合は書き込みと判断する．また，4 行目のように (右辺) にも大域変数が使用されている場合は，読み出しと書き込みが行われていると判断する．これに加え，5 行目のように+=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と判断する．このようにプログラムを解析することで，4 行目の global1 と 5 行目の global2 はリダクション処理の対象となる変数であるとわかる．

また，7 行目のように if 文の条件式で参照されている場合は読み出しであると判断する．8 行目のように，if 文の条件式で参照されている大域変数に対して，その if 文のボディで書き込みがされている場合，大域変数 global3 に対し，読み出しと書き込みの両方が行われていると判断する．なお，このように if 文の条件式で参照されている大域変数に対してリダクション処理が必要な場合，その if 文のボディで書き込みが行われている他の大域変数にも影響を与える可能性がある．そこで，このような場合は if 文のボディで書き込みのみ行われている大域変数に対しても，例外としてリダクション処理が必要であると判断する．9 行目では大域変数 global4 に対して書き込みしか行われていないが，この例外に該当するためリダクション処理が必要であると判断する．以上がリダクション処理の検出方法であり，リダクション処理が必要な変数が構成要

素関数内に存在する場合，リダクション処理を適用する．

4.3 リダクション処理の適用

リダクション処理の適用方法について，テンプレートマッチングのプログラム変換例を用いて説明する． **図 13** に従来の RaVioli を用いた逐次プログラムを， **図 14** にプリプロセッサによる変換後の RaVioli/TBB を用いた並列プログラムをそれぞれ示す．なおテンプレートマッチングとは，処理対象画像とテンプレート画像の差分が最小となる箇所を探索する処理である．

従来の 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 13 に示すように 2 つの構成要素関数を定義する (図 13 の 10〜12，14〜24 行目)．構成要素関数 Comp(図 13 の 14〜24 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 (図 13 の 40 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(図 13 の 10〜12 行目) を定義して，RV_DoppelImage インスタンスの高階メソッドに渡す必要がある (図 13 の 18 行目)．この RV_DoppelImage 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．RV_DoppelImage の高階メソッドを利用した場合，引数として渡された構成要素関数が部分画像中の全画素に対して適用される．なお，このとき実際に処理される画素配列は RV_Image インスタンスが保持しているものであり，RV_DoppelImage では RV_Image インスタンスの持つ画素配列へのポインタを用いて処理を適用する．このように，procBox() を使用する場合，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．

図 13 のプログラムにおいて，構成要素関数 Comp 内の 19〜22 行目の処理における大域変数 min および coord は，前節で述べたリダクション処理が必要な変数に該当する．そのため，これらの変数はリダクション処理を追加すべき変数であると判定される．リダクション処理を追加する場合は，まず，これらの変数の一時格納領域としてスレッドローカルな変数を新たに定義する必要がある．ここで本プリプロセッサは，スレッドローカル変数の宣言に__thread 指定子を使用し，大域変数 min に対応するスレッドローカルな変数__min を定義する (図 14 の 3 行目)．しかし，座標を保持する RV_Coord などは RaVioli 独自のクラスであるため，__thread 指定子が対応していない．そこでこのような場合は，TBB が提供しているテンプレートクラスである

```
1  RV_Image* tpImg;
2  int min=INT_MAX
3
4  int sum=0;
5
6  RV_Coord coord;
7
8
9
10 void Count(RV_Pixel p1,RV_Pixel p2){
11   sum+=abs(p1-p2);
12 }
13
14 void Comp(RV_DoppelImage* dImg,
15                RV_Coord Cstart){
16
17
18 dImg->procImgComp(Count,tpImg);
19   if(min > sum){
20     min=sum;
21     coord=Cstart;
22   }
23   sum=0;
24 }
25
26
27
28
29
30
31
32
33
34
35
36
37 int main(){
38   RV_Image* Img;
39   //画像ファイル読み出し
40   Img->procBox(Comp,tpImg->size);
41 }
```

図 13: 変換前

```
1  RV_Image* tpImg;
2  int min=INT_MAX
3  __thread int __min=INT_MAX;
4  int sum=0;
5  __thread int __sum=0;
6  RV_Coord coord;
7  enumerable_thread_specific<RV_Coord> __rv_coord;
8  spin_mutex mutex;
9
10 void Count(RV_Pixel p1,RV_Pixel p2){
11   __sum+=abs(p1-p2);
12 }
13
14 void Comp(RV_DoppelImage* dImg,
15                RV_Coord Cstart){
16   enumerable_thread_specific<RV_Coord>::
17       reference __coord=__rv_coord.local();
18   dImg->procImgComp(Count,tpImg);
19   if(__min > __sum){
20     __min=__sum;
21     __coord=Cstart;
22   }
23   __sum=0;
24 }
25
26 void __Comp(){
27   spin_mutex::scoped_lock lock(mutex);
28   enumerable_thread_specific<RV_Coord>::
29       reference __coord=__rv_coordS.local();
30   if(min > __min){
31     min=__min;
32     coord=__coord;
33   }
34   __min=INT_MAX;
35 }
36
37 int main(){
38   RV_Image* Img;
39   //画像ファイル読み出し
40   Img->procBox_tbb(Comp,tpImg->size,__Comp);
41 }
```

図 14: 変換後

enumerable_thread_specific<T>クラスを用いる．このテンプレートクラスは，T 型の各スレッドの変数をコンテナの構造で管理し，スレッドローカルな変数として使用可能にする．このため，RV_Coord クラスにはこのテンプレートクラスを適用し，__rv_coord という名前でインスタンス化する (図 14 の 7 行目)．そして，スレッドローカルなインスタンスとして参照するには，enumerable_thread_specific<T>クラスのメンバ関数 local() を利用する (図 14 の 17，29 行目)．また，enumerable_thread_specific<T>クラスでは，reference という名前でテンプレート実引数として与えられた型名を保持しており，スコープ演算子 (::) を利用することでこの型名を参照することができる．これを用いることで，RV_Coord クラスをインスタンス化する (図 14 の 16，17，28，29 行目)．このようにして RaVioli 独自のクラスのインスタンスに対応する，スレッドローカルなインスタンスを用意することができる．また，大域変数 sum は 2 つの構成要素関数 Count と Comp のスコープ内で使用されている．このように，複数の構成要素関数に使用されている大域変数に対しても，スレッドローカルな変数を定義しておく．大域変数 min と同様に，スレッドローカル変数__sum を定義する (図 14 の 5 行目)．

次に，リダクション処理の生成および適用について述べる．リダクション処理が必要な変数が使用されている，変換前の 19〜22 行目の処理を，スレッド毎で実行する処理と各スレッドの処理結果を統合する処理の二つに分割する．スレッド毎に実行される処理は変換前と同様に構成要素関数内に記述する (図 14 の 19〜22 行目)．ここで，スレッド毎の処理結果は，スレッドローカルな変数に格納する必要があるため，__min および__coord を使用する．一方，各スレッドの処理結果を統合する処理は，リダクション処理用の関数を新たに定義し，その関数内に記述する．ここで，リダクション処理用の関数__Comp を定義し (図 14 の 14 行目)，各スレッドの処理結果をそれぞれの大域変数 min，coord に統合する (図 14 の 30〜33 行目)．また，この関数の最後では 3.2 節で述べたように，スレッドローカル変数の初期化を行う (図 14 の 34 行目)．そして，この関数を排他的に実行させるために，TBB の spin_mutex クラスのインスタンス化 (8 行目)，およびロックをかけるための記述を追加する (27 行目)．

最後に，高階メソッド呼び出し部分を変換する．変換前のプログラムでは，高階メソッドの procBox が使われている (図 13 の 40 行目)．そこで，この高階メソッドの末尾に_tbb を追加することで，TBB を組み込んだ高階メソッド procBox_tbb を使用するように変換する (図 14 の 40 行目)．また，リダクション処理用の関数__Compare を引数として追加する (図 14 の 40 行目)．このようにプログラムを変換することで，共有変数に対する競合が発生する場合でも，処理結果の正当性を保証することができる．

表 3: 評価環境

OS	Fedora 15
CPU	Core2Quad
クロック周波数	2.83GHz
コア数	4
並列スレッド数 (コアあたり)	1
メモリ	3GB
TBB version	4.0
コンパイラ	GNU g++ 4.6.1
最適化オプション	-O3

5 評価

前章までに述べた RaVioli/TBB およびプリプロセッサを評価した.

5.1 評価環境

評価環境を **表 3** に示す. CPU は 4 コア構成である Core2Quad を使用し, 並列数は 4 として評価した. 評価には, サンプルプログラムとしてグレースケール化, エンボスフィルタ, 画素の平均値算出, テンプレートマッチング, 直線検出の計 5 つのプログラムを使用した. グレースケール化, エンボスフィルタ, 画素の平均値算出プログラムでは 512×512 ピクセルの画像を, 直線検出プログラムでは 450×540 ピクセルの画像を使用した. またテンプレートマッチングプログラムでは, 395×372 ピクセルの処理対象画像と 70×72 ピクセルのテンプレート画像を使用した. サンプルプログラムを用いて, RaVioli/TBB を使用する場合の処理時間を, RaVioli を使用しない場合, RaVioli を使用せず TBB のみ使用する場合, 従来の RaVioli を使用する場合のそれぞれの処理時間と比較し, 評価した.

5.2 実行時間の比較

評価結果を **表 4** に示す. “C++” は C++で記述した場合, “C++/TBB” は TBB を使用した C++の場合, “RaVioli” は RaVioli を使用した場合, “RaVioli/TBB” は RaVioli/TBB を使用した場合をそれぞれ表している.

結果から, 従来の “RaVioli” に対し, “RaVioli/TBB” ではグレースケール化プログ

表 4: 実行時間の比較 (ms)

プログラム名	C++	C++/TBB	RaVioli	RaVioli/TBB
グレースケール化	0.899	1.258	4.621	1.321
エンボスフィルタ	5.234	2.299	8.474	3.189
画素の平均値算出	0.185	0.968	2.631	0.716
テンプレートマッチング	3657.100	893.277	6794.989	1780.218
直線検出	35.401	45.810	45.790	76.371

ラムでは約 3.5 倍，エンボスフィルタプログラムでは約 2.7 倍，画素の平均算出では約 3.7 倍，テンプレートマッチングプログラムでは約 3.8 倍の高速化が達成できることを確認した．しかし，直線検出では速度低下してしまっている．

直線検出は，2 値化，エッジ検出，ハフ変換，逆ハフ変換の処理を順に画像に適用するプログラムである．このプログラムについて，“RaVioli” および “RaVioli/TBB” のそれぞれの処理時間の内訳を **図 15** に示す．“RaVioli/TBB” において，前述した 4 つの処理のうち，逆ハフ変換では従来の高階メソッドを使用しているため，処理時間がほとんど変化していない．これは逆ハフ変換に用いる高階メソッドが RV_Image クラスの高階メソッドではなく，配列を扱う RV_Array クラスの高階メソッドであり，本提案手法ではこの RV_Array クラスの高階メソッドは拡張していないためである．一方，残りの 3 つの処理は拡張した高階メソッドを使用している．2 値化とエッジ検出については，それぞれ “RaVioli” に比べ処理時間の削減が確認できたが，ハフ変換では処理時間が増加してしまっている．これは，リダクション処理によって生じたオーバヘッドが原因である．ハフ変換には，大域変数として定義されている配列にランダムアクセスする処理がある．この処理は並列化するとその配列に対し競合が発生してしまうため，リダクション処理を適用することで並列化している．しかし，その配列のサイズが大きいため，スレッドローカルな配列の生成に要する時間が増加し，また排他制御が必要となる統合処理によるオーバヘッドも大きくなる．その結果，処理時間が増加したと考えられる．

一方，“RaVioli/TBB” を “C++” と比べると，エンボスフィルタ，テンプレートマッチングプログラムで高速化が確認できた．しかし，グレースケール化と画素の平均値算出では速度向上が確認できなかった．これは，グレースケール化と画素の平均値算出はどちらも非常に軽い処理であるため，RaVioli を用いたことにより生じるオーバ

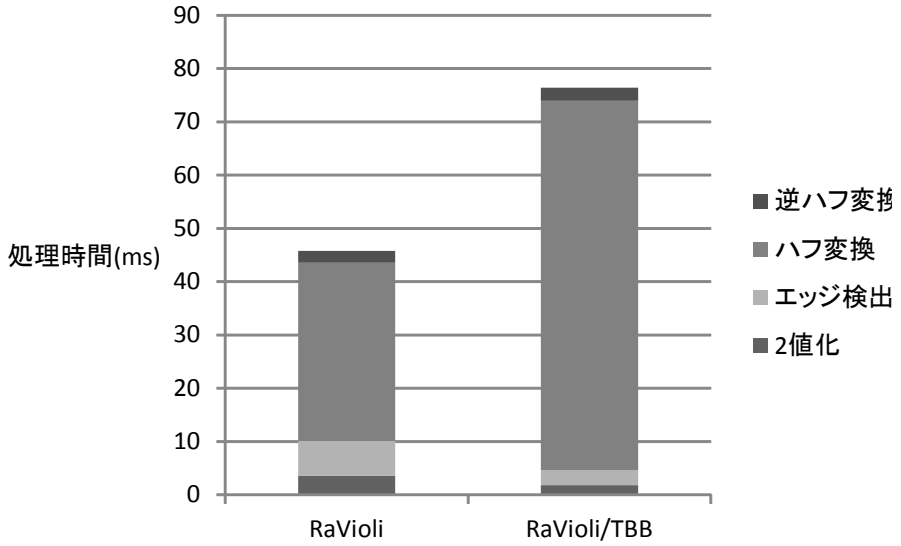

図 15: 直線検出の処理時間の内訳

ヘッドを，並列化して得られた速度向上で隠蔽しきれなかったためだと考えられる．

また，“C++/TBB” と比べると，ほとんどのプログラムで速度低下している．これは，“C++/TBB” と “RaVioli/TBB” ではどちらも TBB を用いているため，RaVioli を用いたことにより生じるオーバヘッドが速度差に表れていると考えられる．しかし，画素の平均値算出プログラムでのみ速度向上が確認できた．これは，“C++/TBB” と “RaVioli/TBB” のそれぞれで使用している TBB のテンプレート関数が違うためであると考えられる．“C++/TBB” では，テンプレート関数 parallel_reduce を用いている．この parallel_reduce を使用した場合，実行時には各スレッドはタスクの終了時に他のスレッドと待ち合わせ，処理結果を統合する．一方，“RaVioli/TBB” では，3 章で述べたように，parallel_for を用いており，リダクション処理の適用もこれを用いて行っている．このリダクション処理の適用において，各スレッドの処理結果の統合処理には排他制御を用いているが，parallel_reduce を用いる場合のようにスレッド同士で待ち合わせることはなく，各スレッドが独立して実行される．そのため，スレッド間で同期を取る parallel_reduce に比べて統合処理でのオーバヘッドが小さくなり，その結果 “C++/TBB” よりも処理速度が速くなったと考えられる．

5.3 プリプロセッサの適用可能範囲

本プリプロセッサは，RaVioli を用いて記述された逐次プログラムを入力として受け取り，RaVioli/TBB を用いたプログラムへと変換する．この実装には flex と bison を使用した．本プリプロセッサを評価するために，さまざまな逐次プログラムをプリプロセッサの入力として与え，出力されるプログラムが正しく動作するかを確認した．

まず変換後のグレースケール化プログラムを実行した場合，処理対象画像がグレースケール化され，正しく動作していることを確認した．これにより，処理単位が単一画素である画像処理プログラムを適切に変換できることが確認できた．また，エンボスフィルタプログラムの変換から，処理対象が近傍画素集合である画像処理プログラムの変換も可能であることを確認した．以上のことから，画素毎の処理に依存関係のない処理には広く適用可能であることを確認した．

次に，並列化した場合に共有変数の競合が発生するプログラムに対して，リダクション処理が適用され，処理結果の整合性が保たれているかを検証した．これは，画素の平均値を求める処理やテンプレートマッチングプログラム，また直線検出プログラム内のハフ変換処理において，リダクション処理が必要な変数の存在がプリプロセッサにより正しく検出され，その変数に対してリダクション処理が適用されていることから確認できた．さらに逐次処理と同じ出力が得られたことから，処理結果の正当性が保証され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 で記述される動画像処理プログラムは，フレーム 1 枚あるいは時系列で隣り合う 2 枚のフレーム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適用可能である．よって，動画像処理への適用も容易に実現可能であるため，今後実装を進めていく予定である．

6 おわりに

本論文では，動画像処理ライブラリ RaVioli に TBB を組み込むことにより高速化する手法を提案した．これを実現するために，RaVioli 内の高階メソッドを拡張した．拡張後の RaVioli を用いたプログラミングでは，画像の構成要素に対する処理を記述した関数を定義し，その関数を拡張した高階メソッドに引数として渡すだけで，画像全体への処理の適用を TBB を用いて並列化可能となった．また，並列化に伴って共有変数の競合が発生する場合があるが，そのような場合には，リダクション処理用の関数を新たに定義し，高階メソッドの引数に加えることで，処理結果の正当性を保証した

並列化を可能にした．しかし，リダクション処理の有無の判断やリダクション処理の記述は，プログラマに並列化を意識させることになってしまう．そこで，従来の RaVioli プログラムから拡張後の RaVioli プログラムに自動的に変換するプリプロセッサを提供することで，プログラマに並列化を意識させることなく，TBB を利用した画像処理が可能となった．

提案手法の有効性を示すため，5 つの画像処理プログラムを用いて，従来手法との実行速度を比較した．4 並列で実行した結果，従来の RaVioli に対して，テンプレートマッチングプログラムにおいて最大 3.8 倍の速度向上が達成できることを確認した．

今後の課題として，TBB の pipeline クラスを用いた動画像処理のパイプライン化が挙げられる．パイプライン化は，タスク並列性を利用した並列化であるため，本論文で提案したデータ並列とは違う軸での並列化である．そこで，本論文で提案した手法とパイプライン処理とを統合することによって，RaVioli のさらなる高速化が考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩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・津邑研究室および齋藤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近藤勝彦氏，稲葉崇文氏，小野亜実氏に感謝致します．

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] 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).

[4] 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).

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

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

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

[8] 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).

[9] 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).

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