卒業論文

OpenMP を用いた画像処理プログラムの並列化

氏　　名：大山 佳宣
学籍番号：2260050092-2
指導教員：山崎 勝弘 教授
提 出 日：2010 年 2 月 19 日

立命館大学　理工学部　電子情報デザイン学科

内容梗概

並列処理は大量のデータを高速に処理できる技術である。これまで並列処理は高価なスーパーコンピュータのための技術であったが、高性能な PC が安価で入手できるようになったことからクラスタリング技術を使った PC クラスタが広まってきた。更に近年ではマルチコアプロセッサが普及してきた。単一のプロセッサの性能向上だけに頼る高速化は限界に近づきつつあり、マルチコアプロセッサなどによる並列化を採用する方向へ転換しつつある。そのため、並列プログラミングは分散メモリ環境から共有メモリ（分散共有）環境へと移行しつつある。この共有メモリ環境用の並列プログラミングモデルとして OpenMP が注目されている。本論文では、OpenMP を用いた画像処理プログラムの並列化について述べる。並列化の対象として、「アルファブレンド」、「ラプラシアンフィルタ」、「エンボス処理」の 3 つの画像処理プログラムの並列化を行った。アルファブレンドでは 8 プロセッサで 5.2 程度、ラプラシアンフィルタでは 8 プロセッサで 7.5 倍、エンボス処理では 8 プロセッサで 7.1 倍の速度向上を得ることができた。

目次

図目次

表目次

１．はじめに

並列計算とは大量のデータを高速に処理するための手法の一つである。並列計算では複数の計算機を同時に利用し、データ処理の高速化を行う。特に気象予測、環境問題など大量のデータを扱う問題に対して有効な方法である。現在、高速処理が必要とされるコンピュータのほとんどは並列コンピュータである。有名な地球シミュレータやチェスの対戦で有名なディープ・ブルーなども並列コンピュータである。

並列コンピュータはメモリの使用方法により３つに分類することができる。複数のプロセッサがメモリバス/スイッチ経由で共通のメモリを使用する共有メモリ型と論理的に分散したシステムがネットワークで繋がり、メッセージなどで通信しながら、ひとつの問題を処理する分散メモリ型、各プロセッサが持つメモリをシステム全体で共有して、1 つのメモリ空間にまとめ、プロセッサ内のプロセッサ がメモリ空間全体にアクセスできるようにする分散共有メモリ型がある。

近年では、地球シミュレータやディープ・ブルーのような大規模な並列コンピュータではなく、PC などの汎用コンピュータを高速ネットワークでつないで並列処理をさせる PC クラスタとマルチコアプロセッサを搭載した並列コンピュータが普及している。

クラスタリングとは複数のコンピュータをつなぎあわせ、全体としてひとつのシステムとして利用する技術のことである。この技術を用いて構築されたシステムである PC クラスタはプロセッサの数に比例して処理能力が向上し、1 台のコンピュータでは得られない性能を得ることができる。大規模な並列コンピュータと比べコストパフォーマンスに優れている。

また、近年では複数のプロセッサコアを搭載した汎用コンピュータが普及している。近年のプロセッサは周波数のアップによる高速化から、プロセッサコアを増やす高速化に方向転換している。プロセッサの性能向上だけに頼る高速化は限界に近づきつつあり、マルチコアプロセッサなどによる並列化を採用する方向へ転換しつつある。つまり、ハードウェアとソフトウェアが協調し、処理を並列化して高速化を図るのである。

本研究では、画像処理プログラムの並列化による高速化の有効性を実験で検証することを目的とし、OpenMP によるビットマップファイルの「アルファブレンド」、「ラプラシアンフィルタ」、「エンボス処理」の並列化を行う。

並列化の対象として画像処理を選んだ理由は、近年の技術進歩によりデジタルカメラやディスプレイなどの機器の性能が大幅に向上し、数千万画素という大量のデータを扱うようになったことと、画像処理はデータに依存性がないため並列効果が見込まれるためである。

本論文では、２章で「アルファブレンド」、「ラプラシアンフィルタ」、「エンボス処理」の３つの画像処理プログラムのアルゴリズムについて説明し、３章では並列コンピュータと並列プログラミングについて述べ、4 章では OpenMP による並列化手法について

述べ、５章では実験条件、実験結果、考察を述べる。

２．画像処理アルゴリズム

２．１　アルファブレンド

アルファブレンドによる画像合成とは、2 枚の画像の混合度を変えて合成する方法のことである。入力画像として画像 a と画像 b という２枚の画像があり、画像 a 上の画素値を A、画像 b 上の画素値を B、出力する画像 c にセットする画素値を C とする。このときアルファブレンドでは次のような演算を行う。

$$C = \alpha A + (1 - \alpha)B$$

ここで、α は 0 から 1 までの値をとり、２枚の画像のうち、どちらの画像をより強く反映するか決定をする。α が 1 に近づくと A が、0 に近づくと B が強く強調される。α＝0.5 の場合は図 1 のように処理が行われる。

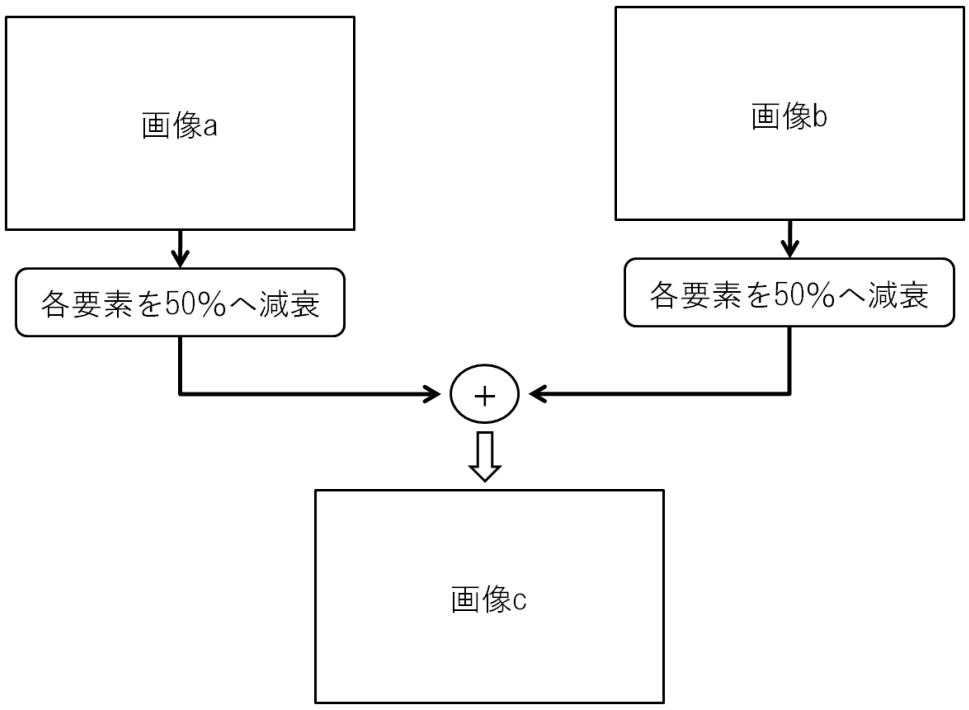

図 1：アルファブレンドによる画像合成

２．２　ラプラシアンフィルタ

ラプラシアンフィルタは画像中に含まれる物体の輪郭部分（エッジ）を抽出する。フィルタである。ラプラシアン∇^2は画像のような２次元のデータに２階微分を行う。次のような演算を行う。

$$\nabla^2 f(x,y) = f_{xx}(x,y) + f_{yy}(x,y)$$

これを差分形式で表わすと、次のようになる。

$$\begin{aligned}\nabla^2 f(x,y) &= f(x-1,y) + 2f(x,y) + f(x+1,y) + f(x,y-1) - 2f(x,y) + f(x,y+1)\\ &= f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y)\end{aligned}$$

これらを係数で表現すると、

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

となる。４５℃方向を含めると、

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

となる。これらの係数をマスクパターンとして注目画素の近傍に以下の 3*3 のオペレータを使って微分を行う。処理の流れを図 2 に示す。

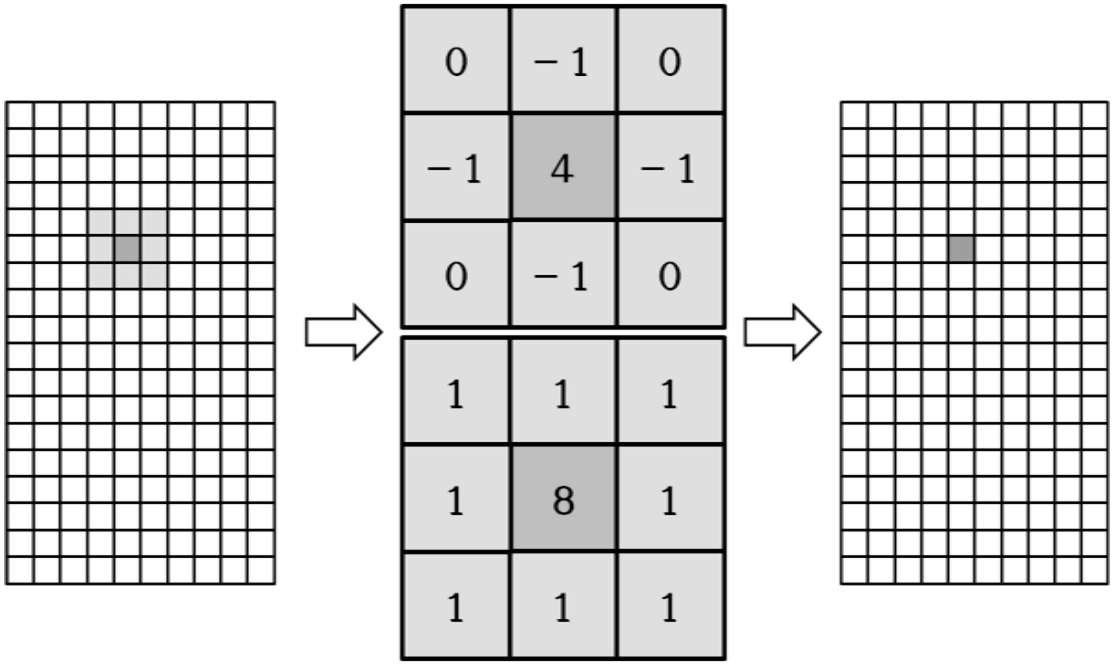

図 2：ラプラシアンフィルタ

２．３　エンボス処理

エンボス処理は彫刻のような効果を出す処理を行う。エンボス処理は斜め方向のエッジ抽出を行い、得られた値に除算・加算を行い抽出したエッジに下駄を履かせたような処理を行う。入力画像に係数を乗算し、その総数を４で割る。得られた結果に、画素の取りうる値の中央値（0～255 なら 128）を足す。以下のようにマスクパターンとして注目画素の近傍に以下の 3*3 のオペレータを使って微分を行う。処理の流れを図 3 に示す。

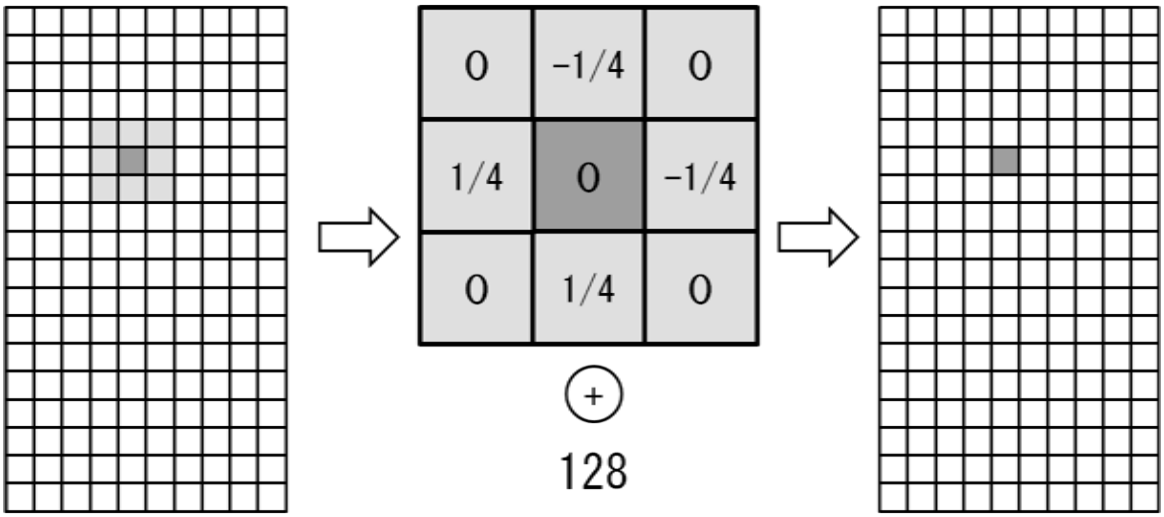

図 3：エンボス処理

３．並列コンピュータと並列プログラミング

３．１　並列コンピュータ

並列処理を行うコンピュータはメモリの使用方法によって分類できる。１つは図４に示すような共有メモリ型である。共有メモリ型は、各プロセッサが共通のメモリを使用し、メモリ全体にアクセスできる。各プロセッサの接続は密になり、高速に通信することが可能である。また、同じ空間で動作するためデータ交換や、同期、メッセージの交換は高速になる。これに対して、プロセッサとメモリを１つのプロセッサとし、複数のプロセッサが相互に通信をしながら並列計算をするのが図 5 で示す分散メモリ型である。プロセッサ内のメモリをそのプロセッサのみで利用する分散メモリ型と、各プロセッサが持つメモリをシステム全体で共有して、１つのメモリとして扱い、プロセッサ内のプロセッサがメモリ全体にアクセスできるようにする分散共有メモリ型がある。

分散メモリ型のクラスタでは、プロセッサ内では共有メモリ型と同様に高速な通信が行えるが、プロセッサ間のネットワークがボトルネックとなる。ソフトウェアを用いた分散共有メモリ型では、プロセッサ全体のメモリをあたかも１つのメモリ空間として共同使用するため、各プロセッサのプロセッサ は大きなメモリ空間を利用できるが、異なるプロセッサのプロセッサ とメモリ間の通信には遅延が生じるので、それを考慮する必要がある。分散共有メモリ型はプロセッサ間通信の能力がシステムで重要となる。

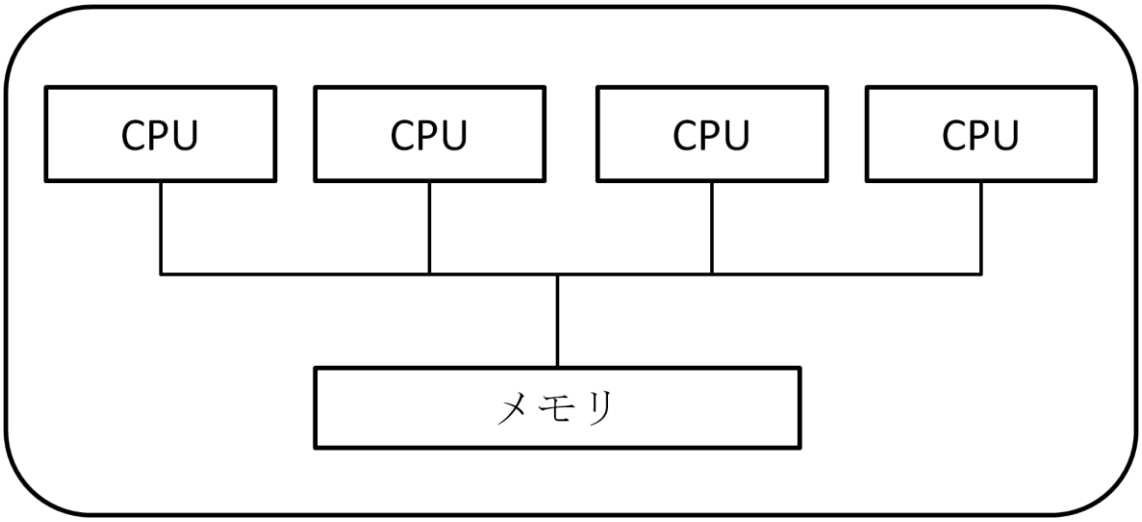

図 4：共有メモリ型

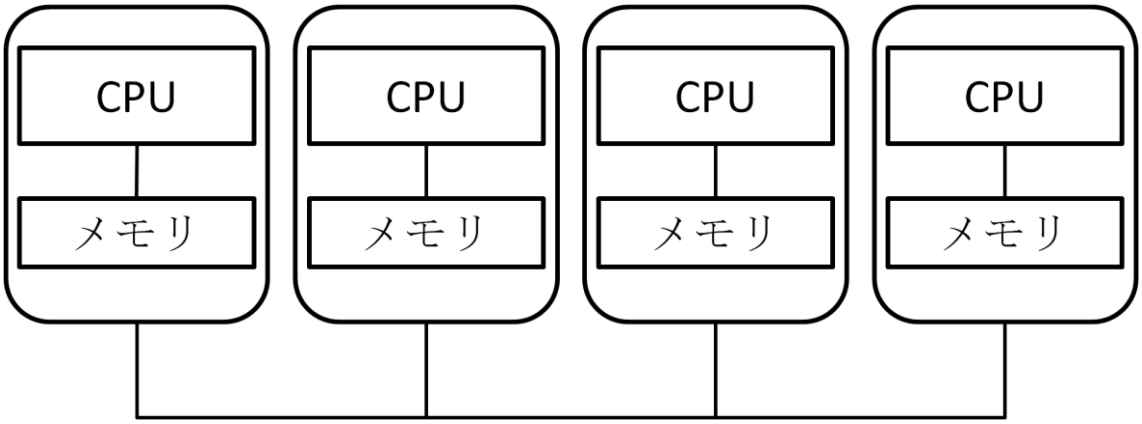

図 5：分散メモリ型

本研究では、nycto を使って実験を行っている。nycto の構成を図 6 に示す。nycto00,nycto01 の 2 台とそれらを管理する Server Host 1 台で構成されている。それぞれの Host は Gigabit Ethernet により接続されている。Server Host のプロセッサは Pentium4 3GHz、メモリは 2GB の SDRAM であり、nycto はそれぞれ、プロセッサは Quad Xeon 3GHz を 2 つ、メモリは 8GB の SDRAM である。Quad Xeon は 4 つのコアをもつマルチコアプロセッサである。nycto00、nycto01 はそれぞれ 8 つのコアをもち、合計 16 コアとなる。

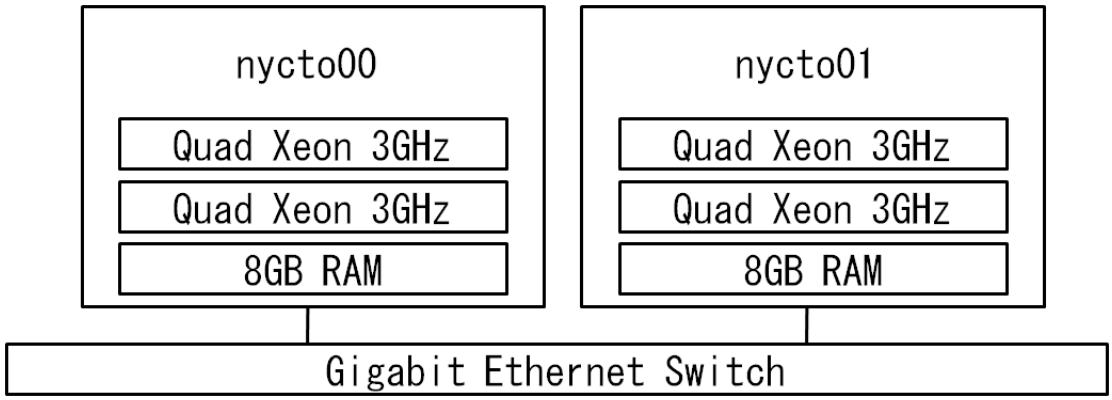

図 6：nycto の構成

３．２　並列プログラミング

並列プログラミングは OpenMP のような共有メモリ型の並列システムと、MPI や PVM を代表とする分散メモリ型の並列システムがある。

OpenMPは主に共有メモリ型の並列コンピュータで用いられる。1997年に米国のOpenMP Architecture Review Boardが、FortranをベースとしてAPIの仕様で開発したものである。以後、C/C++などにおいてもAPIの仕様で開発された。OpenMPの実装上の特徴は、従来のプログラミング手法で開発されたプログラムにいくつかの指示文を追加するだけで、簡単に並列化できるという点がある。分散型の並列化を行う場合、プログラム自体を書き換える必要がある。しかし、OpenMPでは、ループ部分に指示文を付加することで、並列化を実現できる。またコンパイルオプションからOpenMPを除くだけで、逐次型プログラムとソースを共有できる。

MPIはメッセージ通信のAPI仕様である。メッセージ制御は分散メモリ型の並列コンピュータで広く用いられているパラダイムである。MPIはPVMなど既存のメッセージ通信システムが持つ機能の数々を採り入れるようにして標準化が進められたため、事実上の標準としての力を持つこととなった。この方針により、数多くの有用な機能を持ち、多くの並列コンピュータやLAN環境で高い性能を実現できる優れた仕様となった。

PVM（Parallel Virtual Machine）は米国のオークリッジ国立研究所を中心に開発された、メッセージ通信による並列計算を行うためのソフトウェアであり、動作するマシンの種類が多いこと、比較的容易に入手できることもあって、広く利用されている。PVM は、ネットワークに接続された複数台のUNIXコンピュータを、単一の並列コンピュータとして利用することを可能にするソフトウェアシステムである。これによって、複数台のコンピュータの処理能力を、一つの大規模計算問題に結集して処理を行うことができる。

４．OpenMP による並列化

複数のプロセッサを実装したコンピュータであっても、並列プログラミングしなければ、単一のプロセッサがプログラムの最初から最後までを逐次的に処理してしまう。処理を担当しないプロセッサは空き状態となり、高速な処理を行うことができない。OpenMP ではループ部分に指示文を付加することで、並列化を実現できる。逐次処理で記述したプログラムを OpenMP で並列化した場合の概念を図 7 で示す。

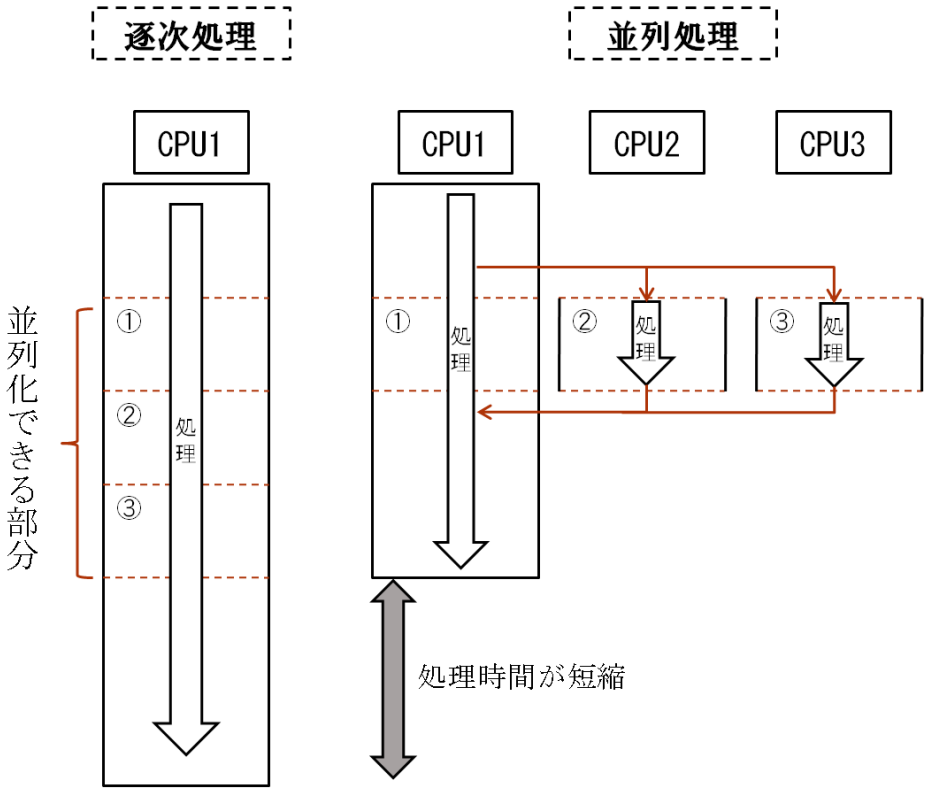

図 7：逐次処理と並列処理

並列化される部分を並列リージョンと呼ぶ。並列化された実行単位をスレッドといい、並列化される以前から存在するスレッドをマスタースレッドと呼ぶ。図 7 においてプロセッサ 1 が対応するスレッドがマスタースレッドである。本研究の実験には for 構文を用いてブロック分割をしている。

for 構文は関連付けられた for ループの繰り返しがスレッドに並列処理されることを指定する。繰り返し処理は parallel 構文で指定した並列リージョンに存在するスレッドで分割処理される。

```
#pragma omp for [指示句[[,]指示句…]
for ループ
```

処理対象であるビットマップファイルは図 8 のように 1 画素が 24 ビットで構成されている。

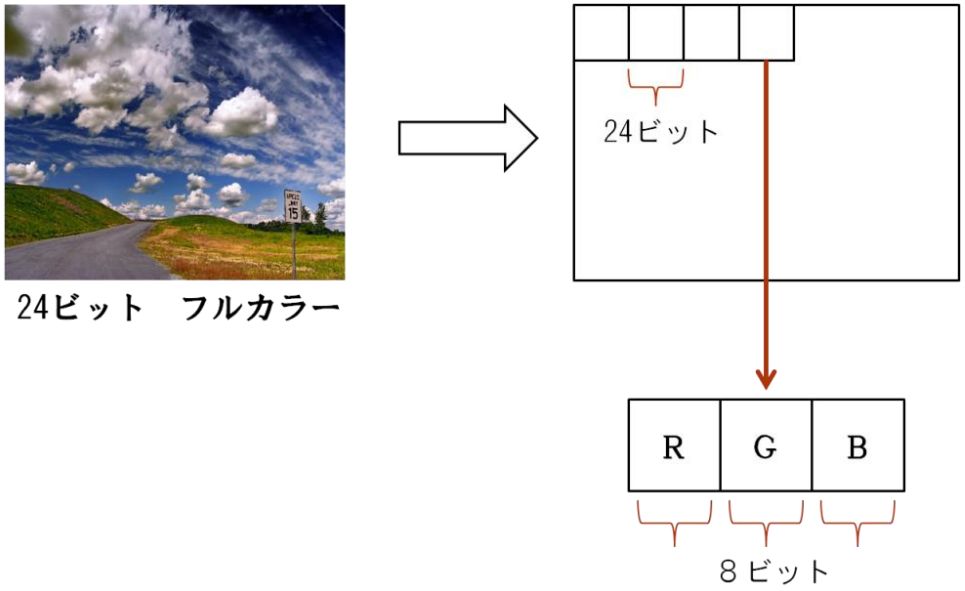

図 8：ビットマップファイルの構成

「アルファブレンド」、「ラプラシアンフィルタ」、「エンボス処理」において 20*20 画素の画像を 4 プロセッサで並列処理をした場合のイメージを図 9 に示す。for 構文ではループの処理回数を均等に各プロセッサに割り当てるため、各プロセッサは 5*20 画素の領域を処理する。そのため理論上は逐次処理の 4 倍の処理速度が期待される。

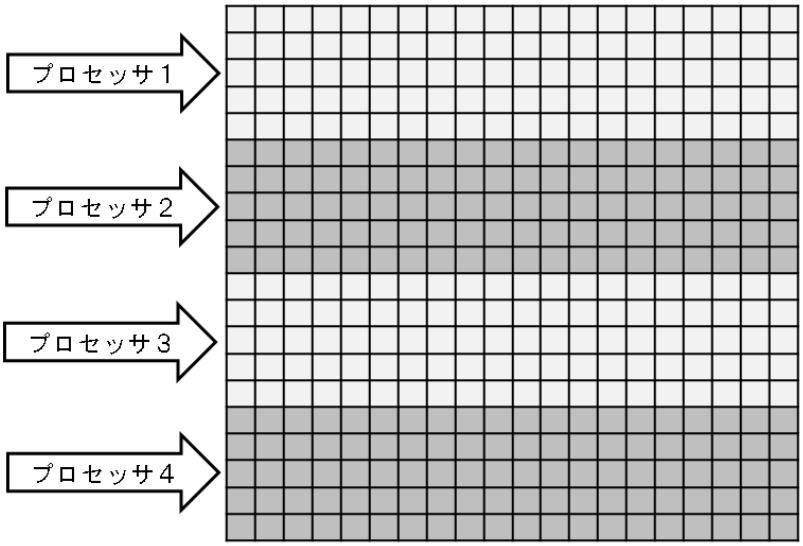

図 9：4 プロセッサでの並列処理

５．実験

５．１　実験条件

実験は nycto00 を使って行った。nycto00 は Quad Xeon 3GHz を 2 つ、メモリは 8GB の SDRAM である。Quad Xeon は 4 つのコアをもつマルチコアプロセッサである。画像処理の対象は 1000*1000 画素、3000*3000 画素、6000*6000 画素のフルカラービットマップファイルを使用した。実行時間の計測は並列化した部分のみで行い、画像の読み込みや書き込みは計測していない。またラプラシアンフィルタ、エンボス処理はモノクローム変換も実行時間に含まれる。

５．２　実行結果

５．２．１　アルファブレンド

α＝0.5 でのアルファブレンドの入力画像と出力画像を図 10 に示す。1000*1000 画素、3000*3000 画素、6000*6000 画素での実行時間、速度向上比それぞれを表 1 、表 2 に示す。速度向上比のグラフを図 11 に示す。1000*1000 画素では 1～7 プロセッサで実行した場合に処理速度の減少が確認できる。また、速度向上比は最大で 2.1 倍であった。2000*2000 画素では 1000*1000 画素と同様に 1～7 プロセッサで実行した場合に処理速度の減少が確認できる。速度向上比は最大で 4.2 倍である。3000*3000 画素ではプロセッサ数の増加とともに実行時間の減少が確認できた。速度向上比は最大で 5.2 倍であった。

図 10：アルファブレンド 実行結果

表１：アルファブレンド 実行時間　　単位（秒）

プロセッサ数	1	2	3	4	5	6	7	8
1000*1000	0.0146	0.0105	0.0079	0.0084	0.0078	0.0072	0.0068	0.0099
3000*3000	0.131	0.083	0.055	0.042	0.035	0.030	0.035	0.041
6000*6000	0.527	0.319	0.213	0.161	0.129	0.111	0.107	0.101

表２：アルファブレンド 速度向上比

プロセッサ数	1	2	3	4	5	6	7	8
理想値	1	2	3	4	5	6	7	8
1000*1000	1	1.38	1.84	1.72	1.87	2.01	2.12	1.48
3000*3000	1	1.58	2.37	3.12	3.66	4.24	3.75	3.21
6000*6000	1	1.65	2.47	3.27	4.08	4.72	4.91	5.20

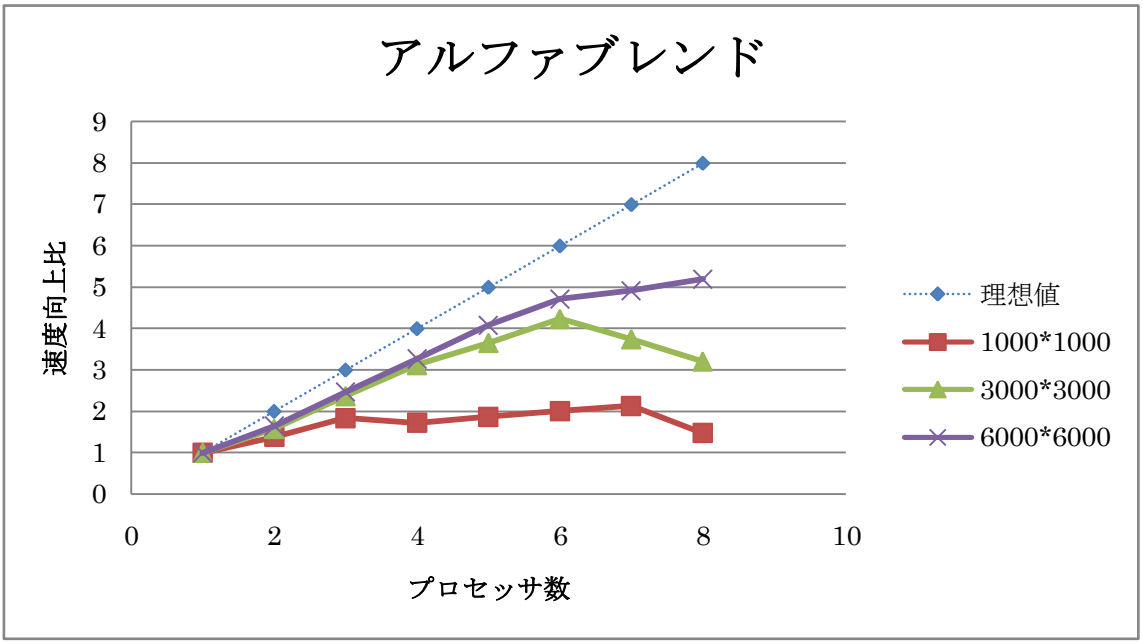

図 11：アルファブレンド 速度向上比

５．２．２　ラプラシアンフィルタ

ラプラシアンフィルタの入力画像と出力画像を図 12 に示す。1000*1000 画素、3000*3000 画素、6000*6000 画素での実行時間、速度向上比それぞれを表 3、表 4 に示す。速度向上比のグラフを図 13 に示す。1000*1000 画素ではプロセッサ数の増加とともに実行時間の減少が確認できるが、7 プロセッサ以上で実行した場合は実行時間が

増加している。また、速度向上比は最大で 3.5 倍であった。2000*2000 画素ではプロセッサ数の増加とともに比較的理想的な実行時間の減少が確認できた。速度向上比は最大で 6.3 倍である。3000*3000 画素ではプロセッサ数の増加とともに理想的な実行時間の減少が確認できた。速度向上比は最大で 7.5 倍であった。

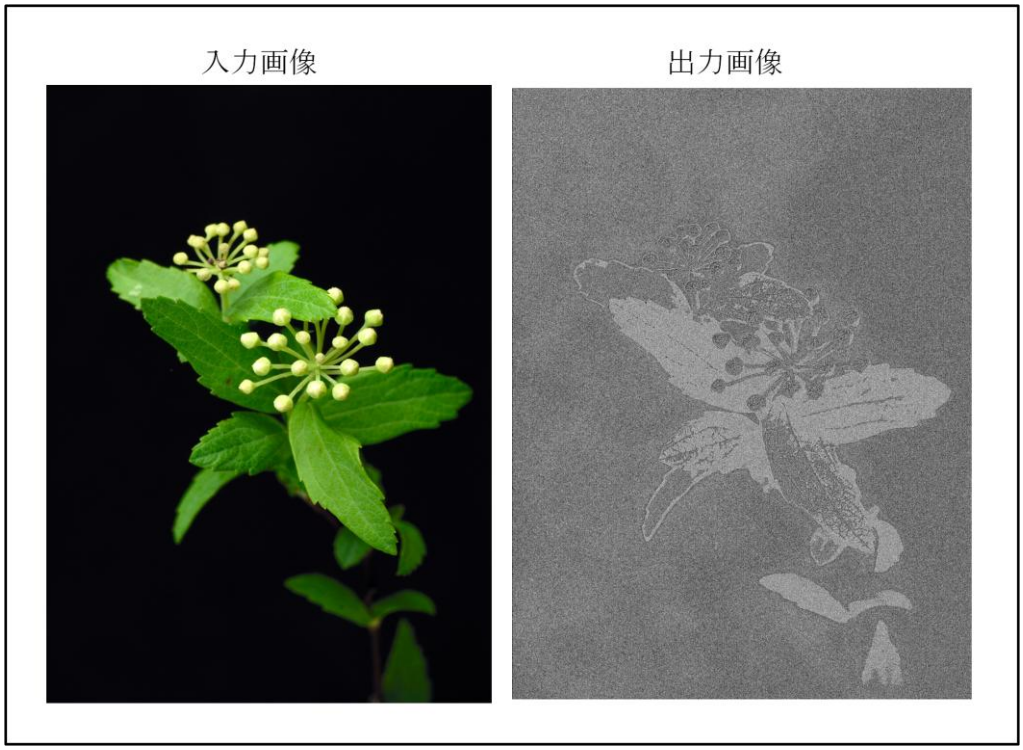

図 12：ラプラシアンフィルタ 実行結果

表 3：ラプラシアンフィルタ 実行時間 単位（秒）

プロセッサ数	1	2	3	4	5	6	7	8
1000*1000	0.094	0.050	0.036	0.031	0.027	0.026	0.030	0.031
3000*3000	0.843	0.436	0.293	0.222	0.178	0.154	0.140	0.133
6000*6000	3.38	1.74	1.24	0.87	0.70	0.585	0.51	0.45

表 4：ラプラシアンフィルタ 速度向上比

プロセッサ数	1	2	3	4	5	6	7	8
1000*1000	1	1.86	2.61	3.00	3.52	3.53	3.15	2.98
3000*3000	1	1.93	2.88	3.81	4.72	5.47	6.02	6.35
6000*6000	1	1.94	2.71	3.88	4.83	5.77	6.68	7.52

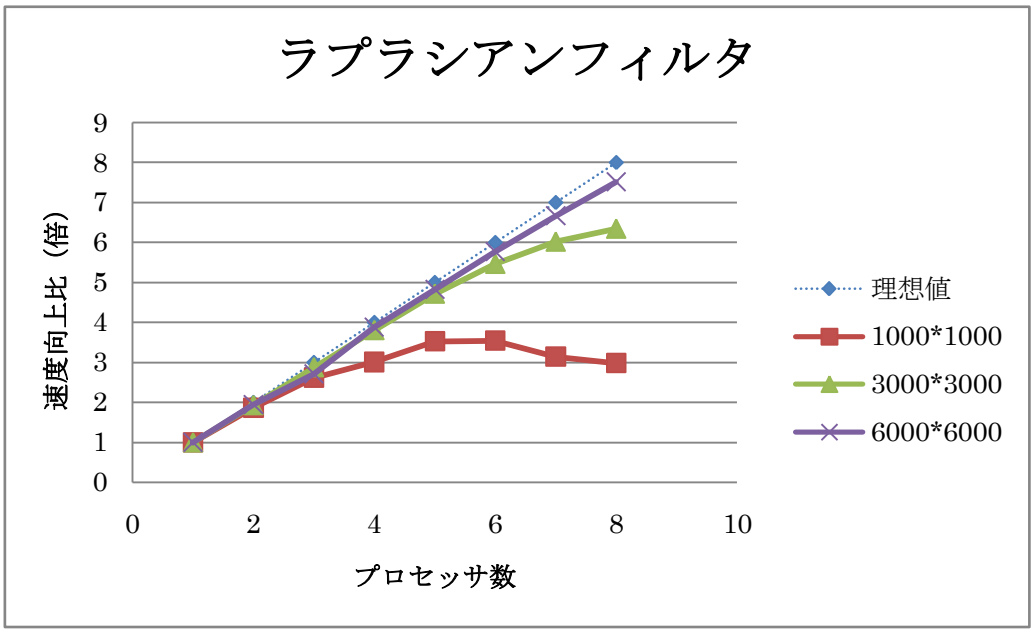

図 13：ラプラシアンフィルタ 速度向上比

５．２．３　エンボス処理

エンボス処理の入力画像と出力画像を図 14 に示す。1000*1000 画素、3000*3000 画素、6000*6000 画素での実行時間、速度向上比それぞれを表 5、表 6 に示す。速度向上比のグラフを図 15 に示す。1000*1000 画素ではプロセッサ数の増加とともに実行時間の減少が確認できるが、4 プロセッサ以上で実行した場合、実行時間はほぼ一定である。また、速度向上比は最大で 2.9 倍であった。2000*2000 画素ではプロセッサ数の増加とともに比較的理想的な実行時間の減少が確認できた。速度向上比は最大で 6.5 倍である。3000*3000 画素ではプロセッサ数の増加とともに理想的な実行時間の減少が確認できた。速度向上比は最大で 7.1 倍であった。

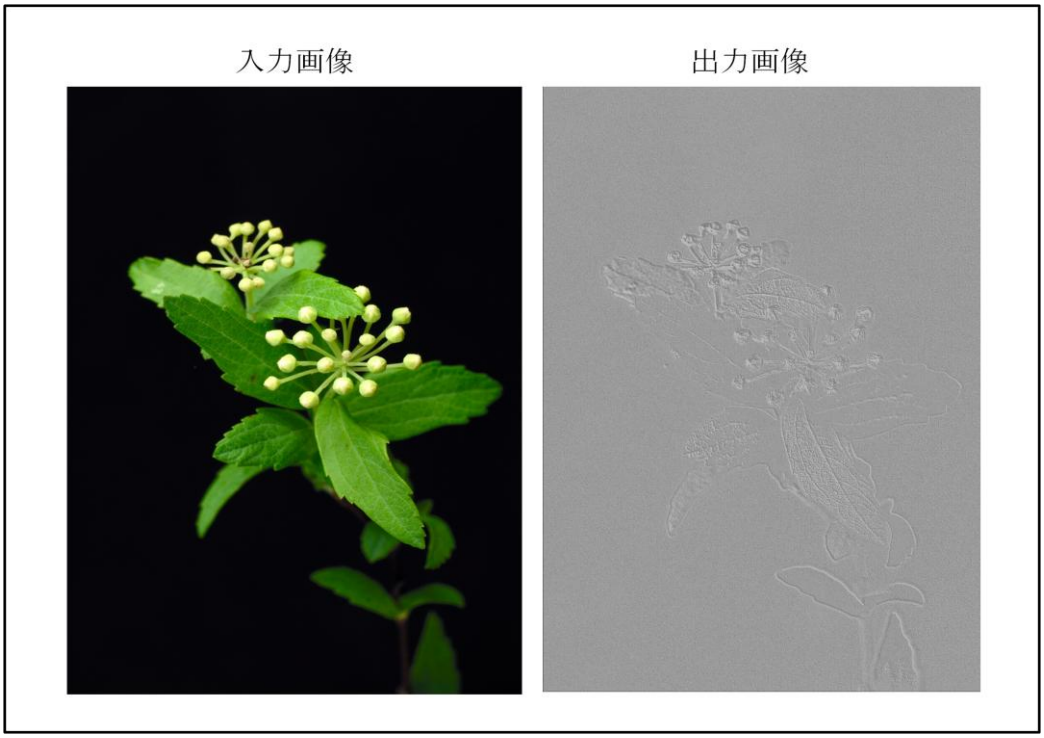

図 14：エンボス処理 実行結果

表 5：エンボス処理　実行時間　　単位（秒）

プロセッサ数	1	2	3	4	5	6	7	8
1000*1000	0.090	0.052	0.037	0.033	0.034	0.034	0.033	0.031
3000*3000	0.815	0.444	0.298	0.226	0.204	0.172	0.137	0.126
6000*6000	3.25	1.77	1.18	0.73	0.71	0.611	0.51	0.46

表 6：エンボス 処理速度向上比

プロセッサ数	1	2	3	4	5	6	7	8
理想値	1	2	3	4	5	6	7	8
1000*1000	1	1.72	2.4	2.7	2.7	2.7	2.7	2.9
3000*3000	1	1.83	2.73	3.60	3.99	4.74	5.93	6.49
6000*6000	1	1.84	2.75	3.83	4.56	5.38	6.33	7.08

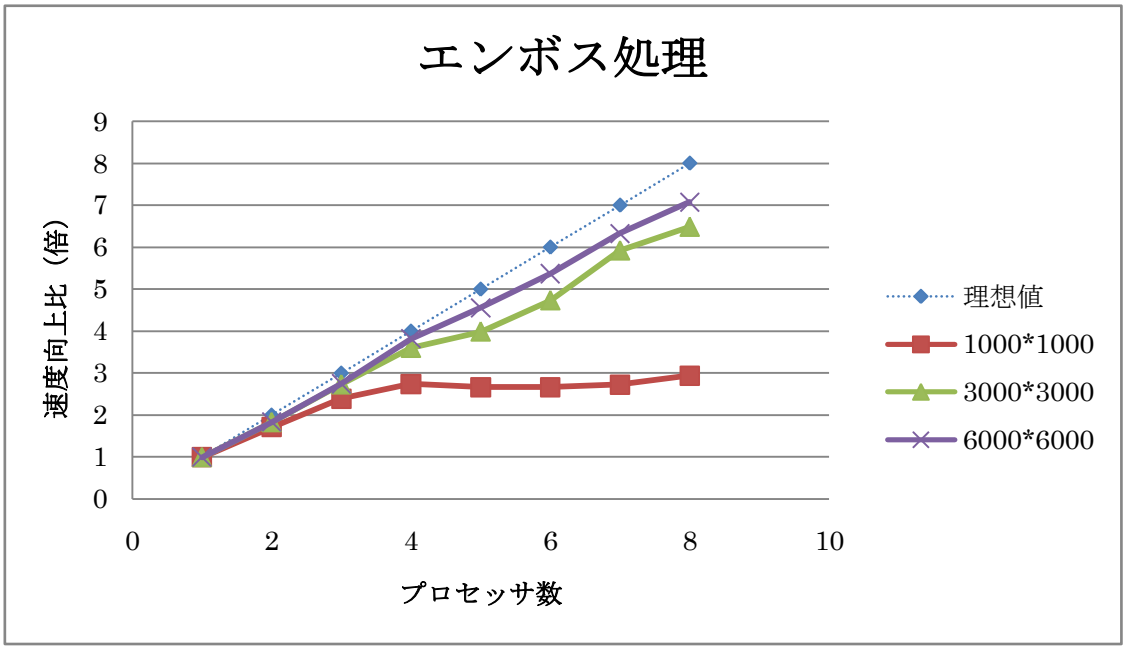

図 15：エンボス処理 速度向上比

５．３　考察

アルファブレンドの実験では、1000*1000 画素・3000 *3000 画素の場合にはプロセッサ数に比例した速度向上は確認できなかった。これは処理するデータ量が少ないことが原因だと考えられる。6000*6000 画素の場合はプロセッサ数の増加と比例して速度向上が確認できた。アルファブレンドではループ計算は１つだけであるため、計算回数が少なく相対的に速度向上比が低い結果となったと考えられる。ラプラシアンフィルタ・エンボス処理では前処理としてモノクローム変換を行っている。実行時間は入力画像のモノクローム変換からマスクパターンを用いたエッジ検出までを測定しているため、アルファブレンドに比べて同じ画素のデータでも計算量が多い。そのため、ラプラシアンフィルタ・エンボス処理では比較的理想的な速度向上を確認することができた。またそれぞれの実験を画素数別に比較すると、データ量や計算回数の増加に比例して速度向上比が高くなることが分かる。これは並列処理が大量のデータを処理することに有効であるということである。

６．おわりに

本研究では、画像処理プログラムの並列化による高速化の有効性を実験で検証することを目的とし、OpenMP によるビットマップファイルの「アルファブレンド」、「ラプラシアンフィルタ」、「エンボス処理」の並列化を行った。アルファブレンドでは 8 プロセッサで 5 倍、ラプラシアンフィルタでは 8 プロセッサで 7.5 倍、エンボス処理では 8 プロセッサで 7.1 倍の速度向上を得ることができた。また速度向上比のグラフから処理データの増加に比例して速度向上比が高くなることが確認できた。大量のデータを持つ画像に対して OpenMP による画像処理プログラムの並列化が非常に有効であると予想される。

謝辞

本研究の機会を与えてくださり、ご指導を頂きました山崎勝弘教授に深く感謝いたします。また、本研究に関して様々な相談に乗って頂き、貴重な助言を頂いた亀井雄介氏をはじめ、様々な面で貴重な助言や励ましを下さった研究室の皆様に深く感謝いたします。

参考文献

[1] 石川裕　他：Linux で並列処理をしよう,共立出版,2007.
[2] 北山洋幸：OpenMP 入門,秀和システム,2009.
[3] 昌達慶仁：詳細　画像処理プログラミング,ソフトバンククリエイティブ 2008.
[4] 田村秀行：コンピュータ画像処理,オーム社,2002
[5] 宮城 雅人：PC クラスタ上での OpenMP による JPEG2000 エンコーダの並列化,立命館大学理工学部情報学科卒業論文,2003
[6] 粂山直生：PC クラスタを用いた画像処理プログラムの並列化,立命館大学理工学部情報学科卒業論文,2005
[7] 多田修司：PC クラスタ上での OpenMP によるマンデルブロ集合の並列化, 立命館大学理工学部情報学科卒業論文,2005.
[8] 松岡毅：OpenMP による数独パズルの並列化, 立命館大学理工学部情報学科卒業論文,2007.
[9] Intel 社：インテル®コンパイラーOpenMP*入門,http://www.intel.co.jp/
[10] PC Cluster Consortium：PC クラスタコンソーシアム, http://www.pccluster.org/

付録A　BMP 形式ファイルの入出力プログラム

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#include"lap.h"
#define F_PI 2*M_PI

double second()
{
   struct   timeval   tv;
   gettimeofday(&tv, NULL);
   return tv.tv_sec + tv.tv_usec / 1000000.0;
}


int main(void)
{
      BITMAPINFOHEADER bitmapinfoheader;
      FILE *fp,*fp2,*fp3;
      unsigned long w1,h1,w2,h2,w3,h3;
      unsigned char *p1,*p2,*p3;
      unsigned long data;
      int n = 8;
            double       start, time;

      if ((fp = fopen("a.bmp", "rb")) == NULL) {
            printf("file open error!!¥n");
      }
      if ((fp2 = fopen("a_bibun.bmp", "wb")) == NULL) {
            printf("file open error!!¥n");
      }
            if ((fp3 = fopen("b.bmp", "rb")) == NULL) {
            printf("file open error!!¥n");
      }
      readbmpheader(fp,&bitmapinfoheader);

      w1=bitmapinfoheader.biWidth;
      h1=bitmapinfoheader.biHeight;
      data=bitmapinfoheader.bfSize-54;
      p1 = (unsigned char*) malloc (sizeof(unsigned char)*data);
      fread(p1,1,data,fp);
      p2 = (unsigned char*) malloc (sizeof(unsigned char)*data);
            p3 = (unsigned char*) malloc (sizeof(unsigned char)*data);
      fread(p3,1,data,fp3);

            start = second();

            //mono(data,p1,p2);

            //lap(w1,h1,data,p1,p2);

            //emboss(w1,h1,data,p1,p2);
```

```
        //gousei(data,p1,p2,p3);


        time = second() - start;

        printf("time = %f seconds¥n",time);

    writebmpheader(fp,fp2,&bitmapinfoheader);

    fwrite(p2,1,data,fp2);
    free(p1);
    free(p2);
        free(p3);
    fclose(fp);
    fclose(fp2);
        fclose(fp3);

    return 0;
}
```

付録Ｂ　画像処理ヘッダーファイルの OpenMP 記述

```
#define F_PI 2*M_PI
#include<omp.h>


struct BITMAPFILEHEADER {
   unsigned short bfType[2];
   unsigned long   bfSize;
   unsigned short bfReserved1;
   unsigned short bfReserved2;

   unsigned long   bfOffBits;
   unsigned long   biSize;
   long                biWidth;
   long                biHeight;
   unsigned short biPlanes;
   unsigned short biBitCount;
   unsigned long   biCompression;
   unsigned long   biSizeImage;
   long                biXPixPerMeter;
   long                biYPixPerMeter;
   unsigned long   biClrUsed;
   unsigned long   biClrImporant;
};
typedef struct BITMAPFILEHEADER BITMAPINFOHEADER;


unsigned long ch_to_int(unsigned char *s,int len)
{
      int i;
      unsigned int v=0;
      for(i=len-1;i>=0;i--){
            v=256*v;
            v=v+s[i];
            }
            return v;
}

unsigned long int_to_ch(unsigned char *s, int len,unsigned int v)
{
      int i;
      for(i=0;i<len;i++){
                  s[i] = v%256;
                  v = v/256;
            }
      return 0;
}


////////////////////////////// モノクローム変換//////////////////////////
int mono(unsigned long data,unsigned char *p1,unsigned char *p2)
{
      unsigned char B,G,R,Y;
      int i=0;
```

```
	#pragma omp parallel for private(i, R, G, B, Y) num_threads()

		for(i=0; i<data; i=i+3){

		B = p1[i   ];
	G = p1[i+1];
	R = p1[i+2];

		Y = (0.114478*B + 0.586611*G + 0.298912*R);

		p2[i   ]=Y;
		p2[i+1]=Y;
		p2[i+2]=Y;
		}
}

////////////////////////////ラプラシアンフィルタ////////////////////////////
int lap(unsigned long w1,unsigned long h1, unsigned long data,unsigned char *p1,unsigned
char *p2)
{
	unsigned char Y;
	int x,y, fx,fy;
	int i,ii;
	int filter[3][3] = { /*  ラプラシアンフィルタ  */
		{ 0,-1, 0},
		{-1, 4,-1},
		{ 0,-1, 0}

	};

	//p1 はモノクロ化された画像とする
	#pragma omp parallel for private(Y, x,y, fx,fy, i,ii) num_threads()
	{

	for(y=1; y<(h1-1); y++){
		for(x=1; x<(w1-1); x++){

			i=((w1 * y) + x) * 3;		// 2 次元 x と y から 1 次元 i に変換

			//画素に対してフィルターを計算する
			Y=0;
			for(fy=-1; fy<=1; fy++){
				for(fx=-1; fx<=1; fx++){
				ii=((w1 * (y+fy)) + (x+fx)) * 3;// 2 次元  -->  1 次元

				Y += p1[ii] * filter[fy+1][fx+1];
				}
			}
			p2[i   ]=Y;
			p2[i+1]=Y;
			p2[i+2]=Y;
			}
		}
```

```
		}
}


//////////////////////////エンボス処理//////////////////////////////
int emboss(unsigned long w1,unsigned long h1, unsigned long data,unsigned char
*p1,unsigned char *p2)
{
		unsigned char Y;
		int x,y, fx,fy;
		int i,ii;
		int filter[3][3] = { /*エンボス  */
				{ 0,-1, 0},
				{ 1, 0,-1},
				{ 0, 1, 0}

		};

		//p1 はモノクロ化された画像とする
		#pragma omp parallel for private(Y, x,y, fx,fy, i,ii) num_threads(2)
		for(y=1; y<(h1-1); y++){
				for(x=1; x<(w1-1); x++){

						i=((w1 * y) + x) * 3;		// 2 次元 x と y から 1 次元 i に変換

						//画素に対してフィルタを計算する
						Y=0;
						for(fy=-1; fy<=1; fy++){
								for(fx=-1; fx<=1; fx++){
										ii=((w1 * (y+fy)) + (x+fx)) * 3   / 2 次--> 1 次元

										Y += p1[ii] * filter[fy+1][fx+1];
								}
						}
						Y=(Y/4)+128; //中央値を加算
						p2[i   ]=Y;
						p2[i+1]=Y;
						p2[i+2]=Y;
				}
		}
}

//////////////////////////画像合成//////////////////////////////
int gousei(unsigned long data,unsigned char *p1,unsigned char *p2,unsigned char *p3)
{
	unsigned char B,G,R,Y;
	int i=0;

		#pragma omp parallel for private(i, R, G, B, Y) num_threads()
		{
	for(i=0;i<data;i=i+3){
		B = (p1[i] + p3[i])/2;
		G = (p1[i+1] + p3[i+1])/2;
		R = (p1[i+2] + p3[i+2])/2;
```

```
            p2[i]=B;
                        p2[i+1]=G;
                        p2[i+2]=R;
        }
    }
}


//////////////////////ヘッダーファイル//////////////////////////
int readbmpheader(FILE *fp, BITMAPINFOHEADER *bitmapinfoheader)
{
      int i = 0;
      unsigned char s[54];
      fread(s,1,54,fp);
      bitmapinfoheader->bfType[0] = s[0];i++;
      bitmapinfoheader->bfType[1] = s[1];i++;
      bitmapinfoheader->bfSize = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->bfReserved1 = ch_to_int(s+i,2);i=i+2;
      bitmapinfoheader->bfReserved2 = ch_to_int(s+i,2);i=i+2;
      bitmapinfoheader->bfOffBits = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biSize = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biWidth = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biHeight = ch_to_int(s+i,2);i=i+4;
      bitmapinfoheader->biPlanes = ch_to_int(s+i,2);i=i+2;
      bitmapinfoheader->biBitCount = ch_to_int(s+i,2);i=i+2;
      bitmapinfoheader->biCompression = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biSizeImage = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biXPixPerMeter = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biYPixPerMeter = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biClrUsed = ch_to_int(s+i,4);i=i+4;
      bitmapinfoheader->biClrImporant = ch_to_int(s+i,4);i=i+4;
}


int writebmpheader(FILE *fp,FILE *fp2,BITMAPINFOHEADER *bitmapinfoheader)

{
      int i = 0;
      unsigned char s[54];

      s[0]=bitmapinfoheader->bfType[0];i++;
      s[1]=bitmapinfoheader->bfType[1];i++;
      int_to_ch(s+i,4,bitmapinfoheader->bfSize);i=i+4;
      int_to_ch(s+i,2,bitmapinfoheader->bfReserved1);i=i+2;
      int_to_ch(s+i,2,bitmapinfoheader->bfReserved2);i=i+2;
      int_to_ch(s+i,4,bitmapinfoheader->bfOffBits);i=i+4;
      int_to_ch(s+i,4,bitmapinfoheader->biSize);i=i+4;
      int_to_ch(s+i,4,bitmapinfoheader->biWidth);i=i+4;
      int_to_ch(s+i,4,bitmapinfoheader->biHeight);i=i+4;
      int_to_ch(s+i,2,bitmapinfoheader->biPlanes);i=i+2;
      int_to_ch(s+i,2,bitmapinfoheader->biBitCount);i=i+2;
      int_to_ch(s+i,4,bitmapinfoheader->biCompression);i=i+4;
      int_to_ch(s+i,4,bitmapinfoheader->biSizeImage);i=i+4;
```

```
    int_to_ch(s+i,4,bitmapinfoheader->biXPixPerMeter);i=i+4;
    int_to_ch(s+i,4,bitmapinfoheader->biYPixPerMeter);i=i+4;
    int_to_ch(s+i,4,bitmapinfoheader->biClrUsed);i=i+4;
    int_to_ch(s+i,4,bitmapinfoheader->biClrImporant);i=i+4;

    fwrite(s,1,54,fp2);
}
```