卒業論文

PC クラスタ上での

OpenMP 並列プログラミング(Ⅱ)

氏名　：　前田　大輔

学籍番号　：　2210990198-2

指導教員　：　山崎勝弘　教授

提出日　：　2004 年 2 月 19 日

立命館大学　理工学部　情報学科

内容概要

並列処理は大規模な問題でも計算時間が大幅に短縮できるというメリットがあり、欠かせない技術の一つといえる。近年では、共有メモリ計算機の普及に伴い、並列プログラミングも分散メモリ環境から、共有メモリ環境へと移行しつつある。その共有メモリ用のプログラミングモデルとして現在注目を集めているのが OpenMP である。OpenMP は移植性が高く、プログラミングも比較的簡単なので、今後並列プログラミングの主流になると期待されている。また、高性能な PC が安価で手に入るようになり、Myrinet などの高速なネットワーク環境が普及してきたことから高性能な PC クラスタの構築が可能になった。本論文では、本研究室で構築した Score 型 PC クラスタ上での、OpenMP による並列プログラミングについて述べる。マンデルブロー集合、モンテカルロ法、組織的ディザ法、画像の３次元グラフ表示の６つのプログラムについて実行した。マンデルブロー集合では解像度を変えてそれぞれブロック分割で実行し、並列効果を得ることができた。モンテカルロ法についても、データ数を増やすにつれより速度向上を得ることができた。組織的ディザ法については４重ループにもなる for 文に多大な時間を費やしているので、その for 文に対して自動分割を行った。その結果、画素数を増やしていく事に速度向上を得ることができた。画像の３次元グラフ表示については描画ソフトウェアを用いて作成した画像を原画像とし、画素数を変えながら実行してみた結果、よい速度向上を得ることはできたが、およそ 1024×1024 画素を超える画像に対しては、あまり速度向上が伸びていくことはなかった。

目次

図目次

表目次

１．はじめに

並列処理の研究の応用分野として気象予測、環境問題、流体計算、ディジタル画像処理、遺伝子の解明、データベース処理などがあげられる。特に今後、画像処理などのマルチメディアなどとともに、最も並列処理の活用が広がると考えられているのが、データベース処理である。データベース処理は処理並列性があるばかりでなく、並列 I/O の点で計算機クラスタに向いた性質がある。

並列計算機には３つのメモリモデルがある。複数のプロセッサがメモリバス/スィッチ経由で、主記憶に接続された形態をもつ共有メモリモデル(SMP)とプロセッサと主記憶から構成されるシステムが複数個互いに接続された形態をもち、プロセッサは他のプロセッサの主記憶の読み書きを行うことができる共有分散メモリ、またプロセッサと主記憶から構成されるシステムが複数個互いに接続された形態をもちプロセッサは他のプロセッサの主記憶の読み書きを行うことができない分散メモリがある。

近年、汎用の PC を高速のネットワークで結合した PC クラスタが急速に広まっている PC クラスタには、米国 NASA のゴダード研究所で開発された Beowulf 型 PC クラスタと、日本の新情報処理開発機構の中のリアルワールドコンピューティング(RWCP)プロジェクトで開発された Score 型クラスタがある。PC クラスタの長所として、プロセッサ数に比例して実行速度が向上する、コストパフォーマンス高い、大規模な計算が可能などの点が挙げられる。

本研究室の PC クラスタは Score 型クラスタであり、また SCASH 等のソフトウェアにより分散共有メモリを実現している。また Myrinet や Ethernet を用いた高速通信により通信のオーバーヘッドをより小さくした環境である。

本研究では､OpenMP による並列プログラミングを PC16 台の Score 型クラスタ上で行った。並列処理は、一つの計算を複数のデータに対して処理を行う場合、大きな期待をすることができる。画像処理については一つの処理を適応画像の画素一つ一つに適用するので、並列処理を適用することで大きな効果を期待することができる。2 章では並列処理と OpenMP について、３章４章に関してはマンデルブロー集合、モンテカルロ法、５章６章では画像処理に関する問題である組織的ディザ法、画像の３次元グラフ表示という各問題について、様々な画像を使い、解像度を変えながらブロック分割とサイクリック分割で実行し、実行時間や速度向上比などを比較しながら並列効果にどのぐらい影響してくるのかを追求した。そして各々の問題については、問題定義、並列化手法、実行結果と考察を示した。

２．PC クラスタ上での並列プログラミング

2.1 PC クラスタ

PC クラスタとは、安価な PC にフリーの OS を載せ、それを高速のネットワークで複数接続し、分散して処理を行うシステムである。PC クラスタは、1990 年前後に、数千から数万台の CPU を搭載する超並列計算機の開発が進む一方で、TCP/IP ベースのネットワークで接続された複数台の計算機を仮想的に一台のマシンとしてとらえて並列プログラミングを走らせるような PVM(Parallel Virtual Machine)が開発された。PVM はそれぞれの計算機でメッセージ交換を行うメッセージ通信ライブラリであり、公開されたソフトウェアをインストールするだけで仮想的な並列処理環境が構築できた。これが PC クラスタの幕開けといえる。1995 年前後になると、イーサーネットスイッチや 100Mbit Ethernet などの技術も普及し比較的安価に高速ネットワークの構築が可能となった。さらに、Myricom 社の Myrinet などのクラスタを指向した専用の高速ネットワークが登場した。これにより、専用の並列計算機並みのスループットを持つネットワークの構築が可能となった。一方で、PC 向けのプロセッサの価格低下と急速な性能向上で、コストパフォーマンスに優れた PC クラスタの実行が可能となった。この時期に MPI フォーラムがメッセージ通信のプログラムを記述するために広く使われる「基準」を目指して作られたメッセージ通信の API である MPI(Massage Passing Interface)が広く普及した。現在では複数のプロセッサを搭載した SMP 型の WS や PC が比較的容易に入手可能となっており、これらをベースにした SMP クラスタが登場している。本研究室の PC クラスタは, Pentium3　500MH z を搭載した PC16 台を 100Mbit/sec の通信速度を持つ Ethernet と 1,280Mbit/sec の通信速度を持つ Myrinet との 2 重のネットワークで接続されており，SCore と呼ばれるクラスタシステムソフトウェアを用いて通信を行っている．本研究室の PC クラスタの構成を図１に示す

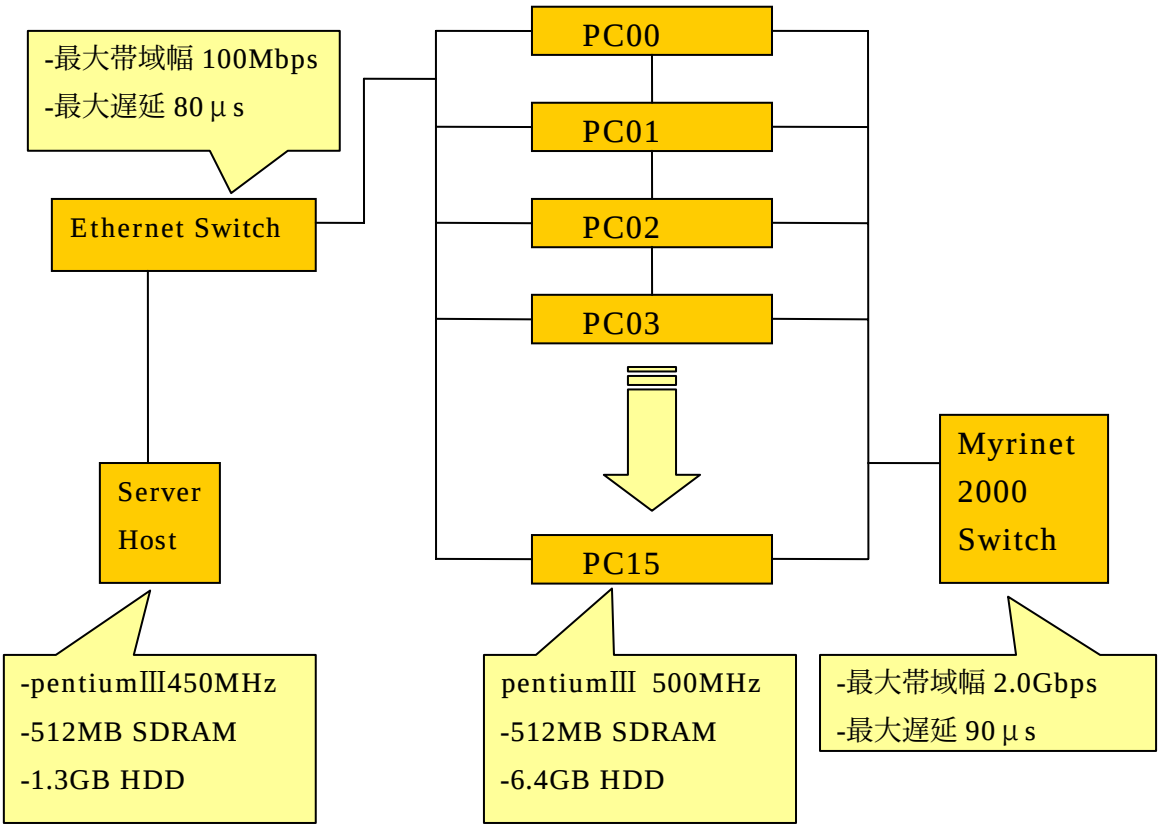


図１：PC クラスタの構成

2.2 並列プログラミング

並列プログラミングは大きく２つに大別できる。１つは分散メモリプログラミング、もう１つは共有メモリプログラミングである。分散メモリプログラミングとして挙げられるのは PVM、MPI のメッセージパッシング型のプログラミングである。共有メモリプログラミングとして挙げられるのは OpenMP の従来のプログラミング言語にコンパイラ指示文として並列処理の指示を挿入するプログラミングである。

(1) PVM(Parallel Virtual Machine)

PVM はアメリカのオークリッジ国立研究所(Oak Ridge National Laboratory)とエモリー大学(Emory University)の研究者らによる異機種分散計算の研究プロジェクトから生まれた。もともとは、ワークステーションクラスタのための TCP/IP ベースの通信ライブラリで、どこにでもある LAN 環境で手軽に並列処理プログラムを実行できることから、多くのユーザーがついた。その後、多くの並列計算機に移植されたため、PVM で書いた並列プログラムは、様々な並列計算機で実行できるようになった。

PVM は、ネットワークで接続された計算機群を仮想的な並列計算機として利用するためのソフトウェアシステムである。対応する計算機には、パーソナルコンピュータから並列計算機/スーパーコンピュータまで、多くの種類のものがある。

PVM では異なったアーキテクチャのホスト間で整数や浮動小数のデータを交換するため、送信されるデータは XDR(eXternal Data Representation)Standard によりエンコードされる。送信側はデータをバッファにパックした後、送信を行い、受信側はバッファにデータを受信した後、アンパックを行いデータを取り出す。基本的に送受信は非同期で、送信と受信が一致しない場合でも送信がブロックされることはない。なお、このデータのパック/アンパックには、データをまとめて送受信することによりオーバーヘッドを減少させる目的もある。

(2) MPI(Message Passing Interface)

MPI は、MPI フォーラムという、任意参加の会議で議論された。PVM とは異なり、MPI はメッセージ通信の API 仕様であって、MPI という特定のソフトウェアがあるわけではない。MPI は、２つの理由により事実上の標準としての力を持つことになった。１つは、MPI フォーラムには PVM を始めとする、主要メッセージ通信ライブラリの開発を行った研究者と、主要な並列計算機ベンダのほとんどが参加したこと、もう一つは、できあがった仕様が、数多くの有用な機能を持ち、多くの並列計算機や LAN 環境で高い性能を実現できる、優れたものに仕上がったことである。

MPI は， 並列処理用のメッセージパッシングの規格である． メッセージパッシングとは，メッセージと呼ばれる特定のデータ形式の受け渡しと，これらの一元的な管理に基づく通信手段の一つである．並列計算機の各 CPU 間では数値計算の過程で多くのデータの交換が行われているが，その交換手順を定めたものが MPI である．多くの並列計算機に標準実装され，また，フリーウェアとして mpich，LAM 等のパッケージが入手可能である．

2.3 OpenMP

OpenMP は共有メモリ方式マルチプロセッサの並列プログラミングのプログラミングモデルのことであり、ベースとなる言語(C言語や Fortran）に並列化の指示文を加えることによりコンパイラが並列化を行うように拡張されている。並列実行や同期をプログラマが明示することによって並列化を行う。また指示文を無視することによって逐次実行が可能なので、逐次版と並列を同じソースで管理でき、段階的な並列化が可能である。

OpenMP の実行モデルには、Fork-join モデルの並列実行モデルを用いている。OpenMP で記述されたプログラムは、マスタスレッドと呼ばれる単一のスレッドで実行を開始する。マスタスレッドは並列構文が現れるまで逐次リージョンを実行する。マスタスレッドは並列指示文に遭遇すると複数のスレッドからなるチームを作成し、そのチームのマスタとなる。ワークシェアリング構文に対応するブロックは、全スレッドによって実行されなければならない。全てのスレッドにより並列に実行される部分を並列リージョンという。また、ワークシェアリング構文に no wait 指示節が指定されていなければ、ワークシェアリング構文の最後で暗黙のバリア同期をチーム内の全スレッドに対して実行し、その後の処理はマスタスレッドのみが実行を続ける。一つのプログラムにおいて、いくつでも並列構文を使うことができる。したがってプログラムは実行中に fork と join を繰り返すことになる。図２に fork-join モデルの実行の流れを示す。

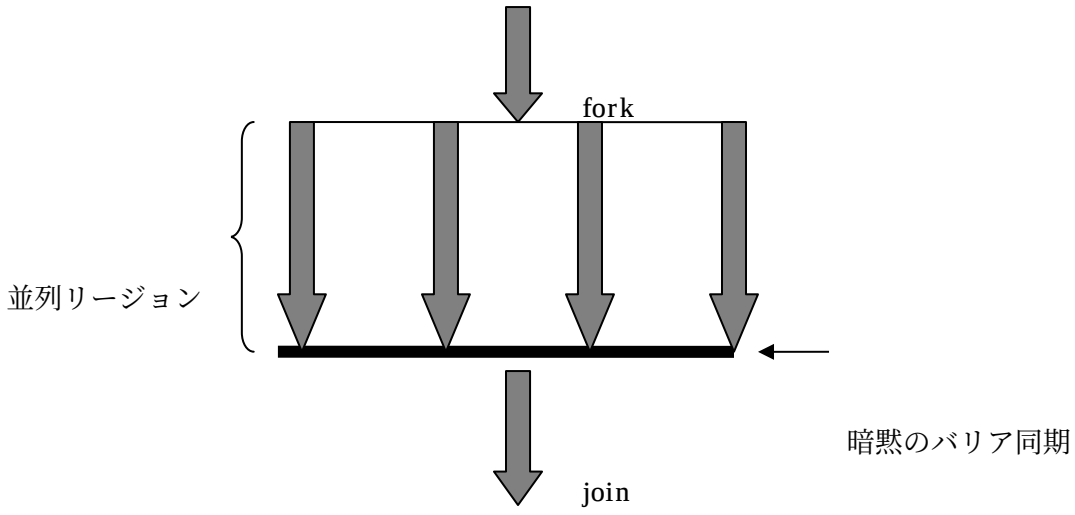


図 2：fork-join モデル

３．マンデルブロー集合

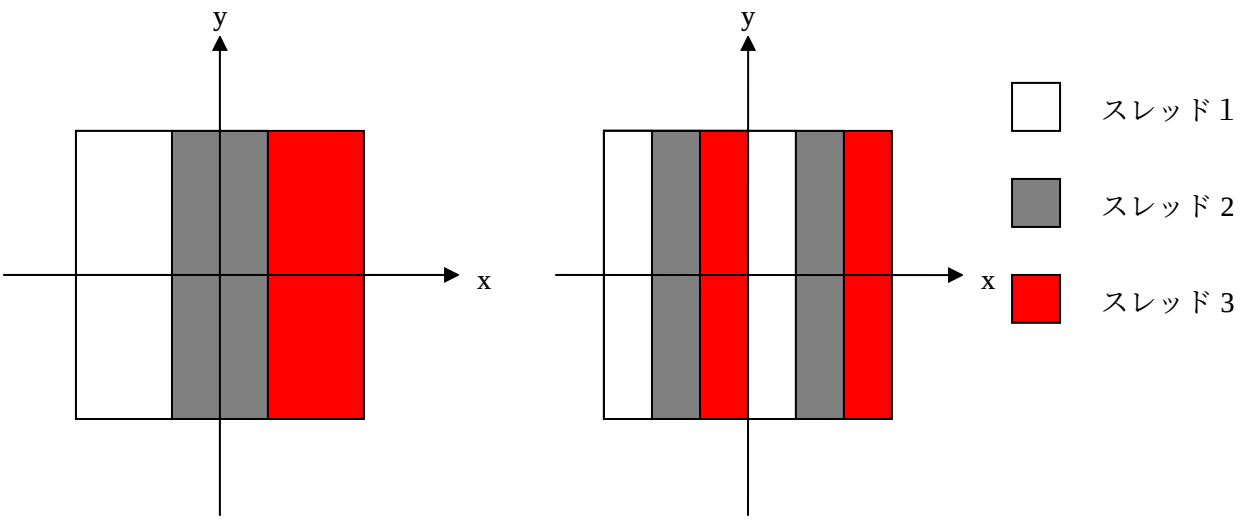
3.1 問題定義

複素関数(f(Z):$Z_{i+1}=Z^2+C$)に対して、初期値を$Z_0=(0,0)$と置き、$Z_1=f(Z_0)$、$Z_2=f(Z_1)$、・・・、$Z_k=f(Z_{k-1})$、のように反復計算を繰り返す。複素平面上の座標値 C の値を変化させ、Z_k（k→∞）の値が収束か発散かを求める。

（−2.5 実部<0.5)、(−2.0<1.0）の範囲の複素平面を X×Y のメッシュに区切り、X×Y 個の点について上の反復計算を行う。その結果$|Z_i|>2.0$ のとき発散、i＞200 のとき収束するとし、後者の結果となる座標点がマンデルブロー集合の要素となる、各座標点に対し、マンデルブロー集合に属する座標点の合計数を count に格納する。

3.2 並列化手法

並列化は複素平面上の x 標の範囲をブロック分割とサイクリック分割で実行する。各プロセッサ自分の受け持つ x 座標の範囲を受け持ち計算していく。以下にブロック分割とサイクリック分割の２つの手法を図 3 に示す。



(a)ブロック分割　　(b)サイクリック分割

図 3：マンデルブロー集合の分割手法

3.3 実行結果と考察

マンデルブロー集合をブロック分割とサイクリック分割で実行した結果を表１、表２に示す。またブロック分割とサイクリック分割で実行したときの速度向上比の違いを解像度別に図４、図５のグラフに表した。

表１：マンデルブロー集合のブロック分割での実行時間(s)

スレッド数	1	2	4	8	16
解像度 300×300	0.535	0.38	0.21	0.14	0.08
解像度 3000×3000	53.52	38.89	21.91	14.55	8.01

表２：マンデルブロー集合のサイクリック分割での実行時間(s)

スレッド数	1	2	4	8	16
解像度 300×300	0.47	0.36	0.20	0.10	0.05
解像度 3000×3000	47.74	36.05	20.38	10.13	5.13

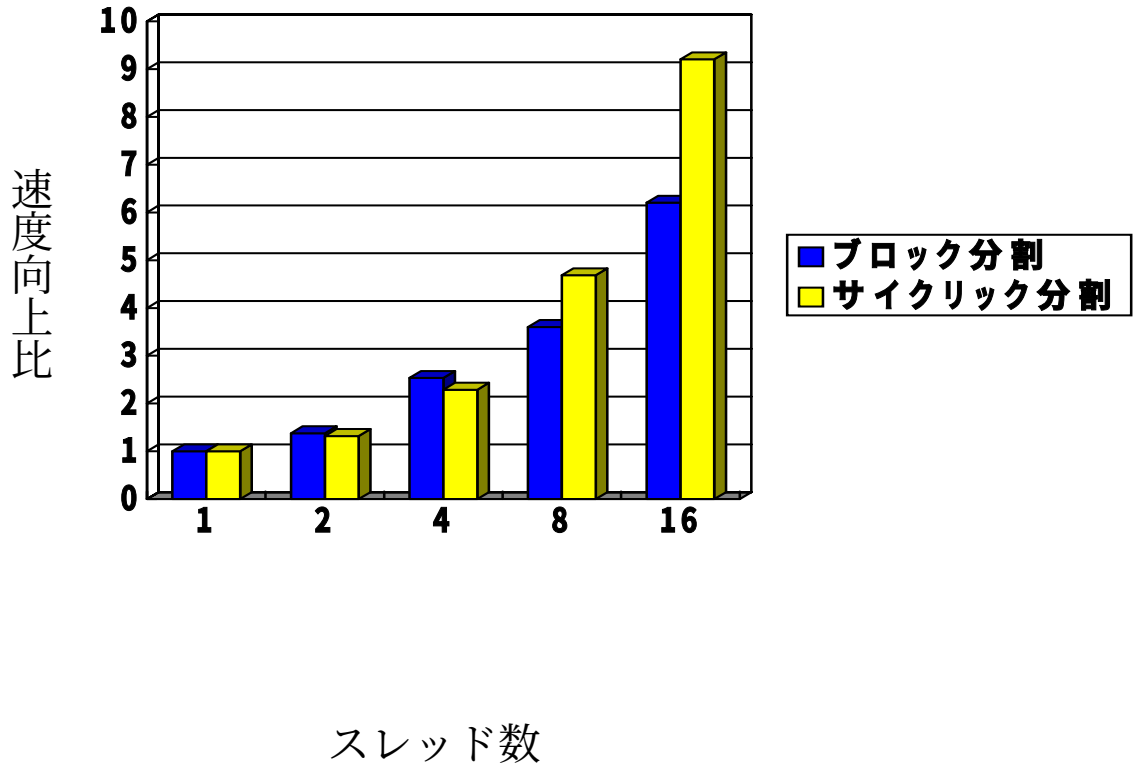


図 4：マンデルブロー集合の速度向上比(解像度 300×300)

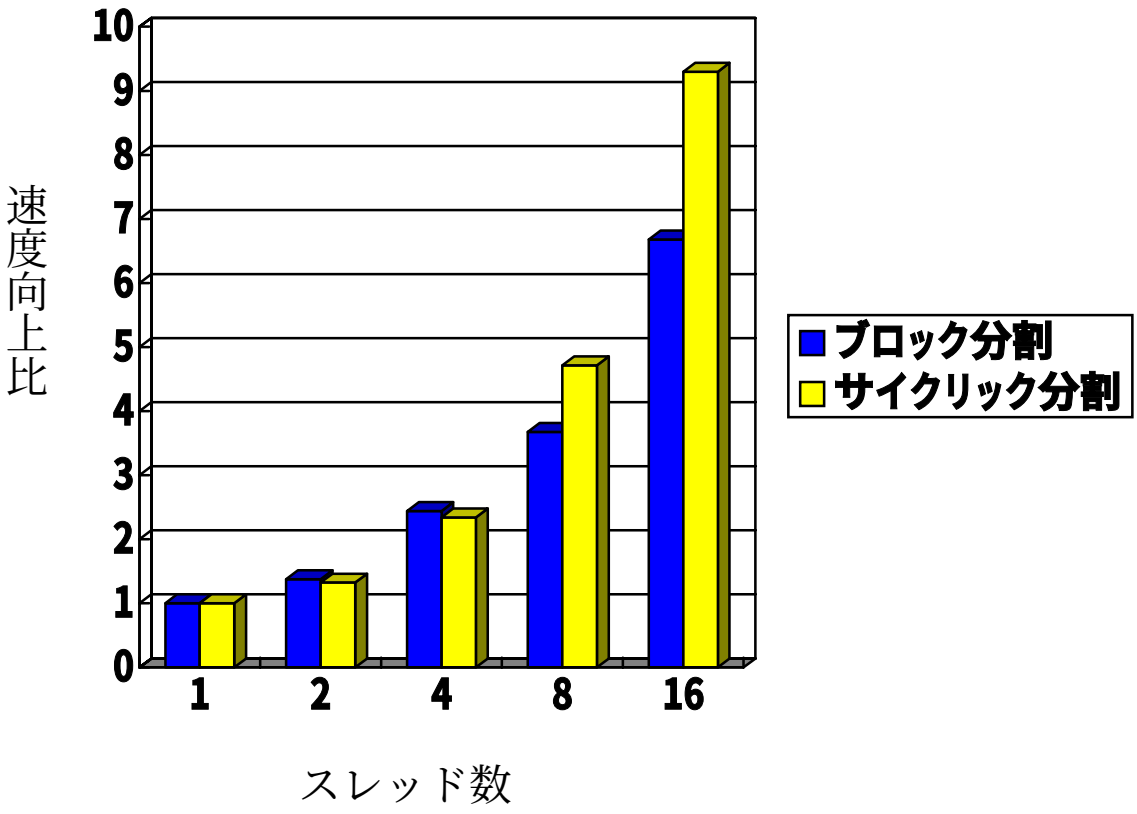


図 5：マンデルブロー集合の速度向上比(解像度 3000×3000)

考察

実行結果を見てみると、解像度 300×300、解像度 3000×3000 のときの両方ともブロック分割よりサイクリック分割の時の方が理想的な速度向上が得られているのがわかる。これはマンデルブロー集合をｘ座標で分割するときに、大きな範囲で区切って分割すると、発散せずに収束するまで計算されるZ_nが多く存在する範囲とそうでない範囲が存在することになり、ある範囲では計算量が多かったり、ある範囲では少なかったりと、一番遅いスレッドの計算に時間が引っ張られてしまうからである。サイクリック分割でｘ座標を細かく分割して実行したときは負荷均衡がとれてブロック分割のときよりも、実行時間が短くなっている。

４．モンテカルロ法

4.1 問題定義

モンテカルロ法による円周率πは、2 つの乱数を発生させ、下図のような正方形内に不規則に点を落とし、円の内部に発生した点の割合から円の面積を求めることができる。すなわち半径 1 の円の 4 分の 1（扇形）を、一辺の長さが 1 の正方形内に置く。すると、円の面積はπ／4 となる。そして、その正方形内にたくさんの不規則な点を落とし、（円のなかにある点の数 a）÷（全部の点の数 b）を円の面積と考える。円の中にある点かどうかは、円の内部の領域をあらわす不等式（$X^2+Y^2<1$）を満たすかどうかで判断する。

したがって、1/4×π：１×１＝ａ：ａ＋ｂより

$\pi = \frac{4a}{a+b}$ で求められる。

落とすｎ個の点を多くすればするほど、より正確な値に近くなると予想できる。

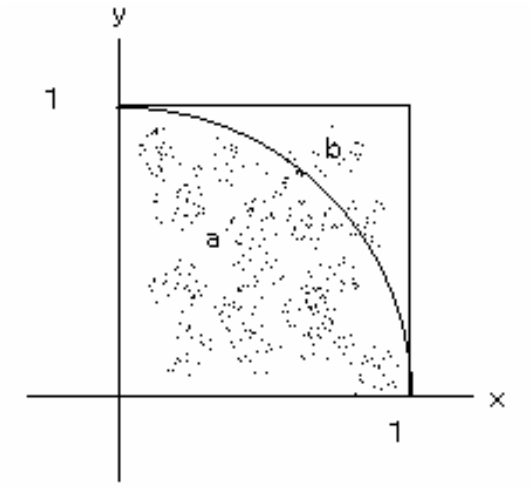


図 6：モンテカルロ法

4.2 実行結果

データ数を 300 万、3000 万、3 億、スレッド数を 1 から 16 まで増やして実行した結果を表３、表４、表５に示す。

表 3：モンテカルロ法のデータ数(300 万)

スレッド数	1	2	4	8	16
π	3.14106	3.141965	3.141435	3.139691	3.139861
処理時間	1.57	0.66	0.33	0.17	0.08
速度向上比	1	2.35	4.67	9.25	17.83

表 4：モンテカルロ法のデータ数(3000 万)

スレッド数	1	2	4	8	16
π	3.141683	3.140747	3.141667	3.142043	3.141828
処理時間	15.75	6.68	3.34	1.67	0.84
速度向上比	1	2.35	4.71	9.41	18.75

表 5：モンテカルロ法のデータ数(3 億)

スレッド数	1	2	4	8	16
π	3.14157	3.14166	3.14157	3.141325	3.141323
処理時間	157.57	66.90	33.45	16.72	8.35
速度向上比	1	2.35	4.71	9.42	18.85

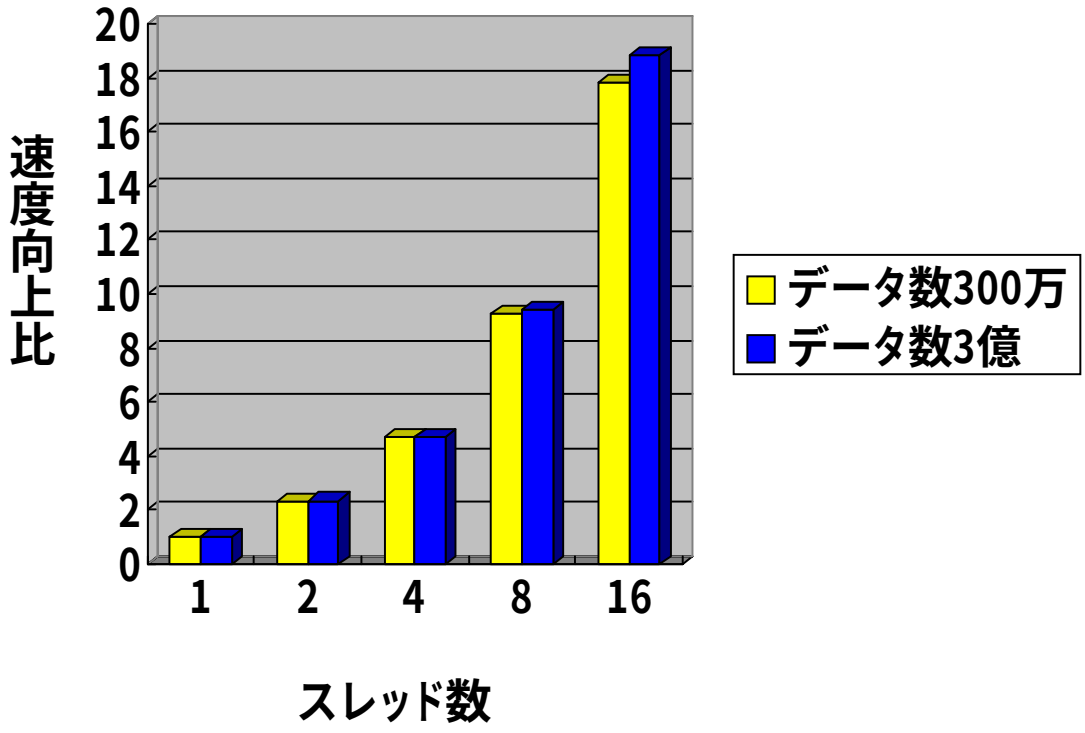


図 7：モンテカルロ法の速度向上比

4.3 考察

モンテカルロ法は比較的どのデータ数の時でも非常に高い速度向上が得られた。πの値については、理論的には点を打つ数を増やせば増やすほど円周率に近づくはずであるが、実際、実行結果を見て、データ数が多くなるにつれてπの値は円周率に近づいているといえる。

５．組織的ディザ法

5.1 問題定義

組織的ディザ法では、入力用に使う原画像を、図８のように、ディザ行列（４画素×４画素）の粗いメッシュに 分割していき、分割された各ブロックごとに２値化作業を行っていく。

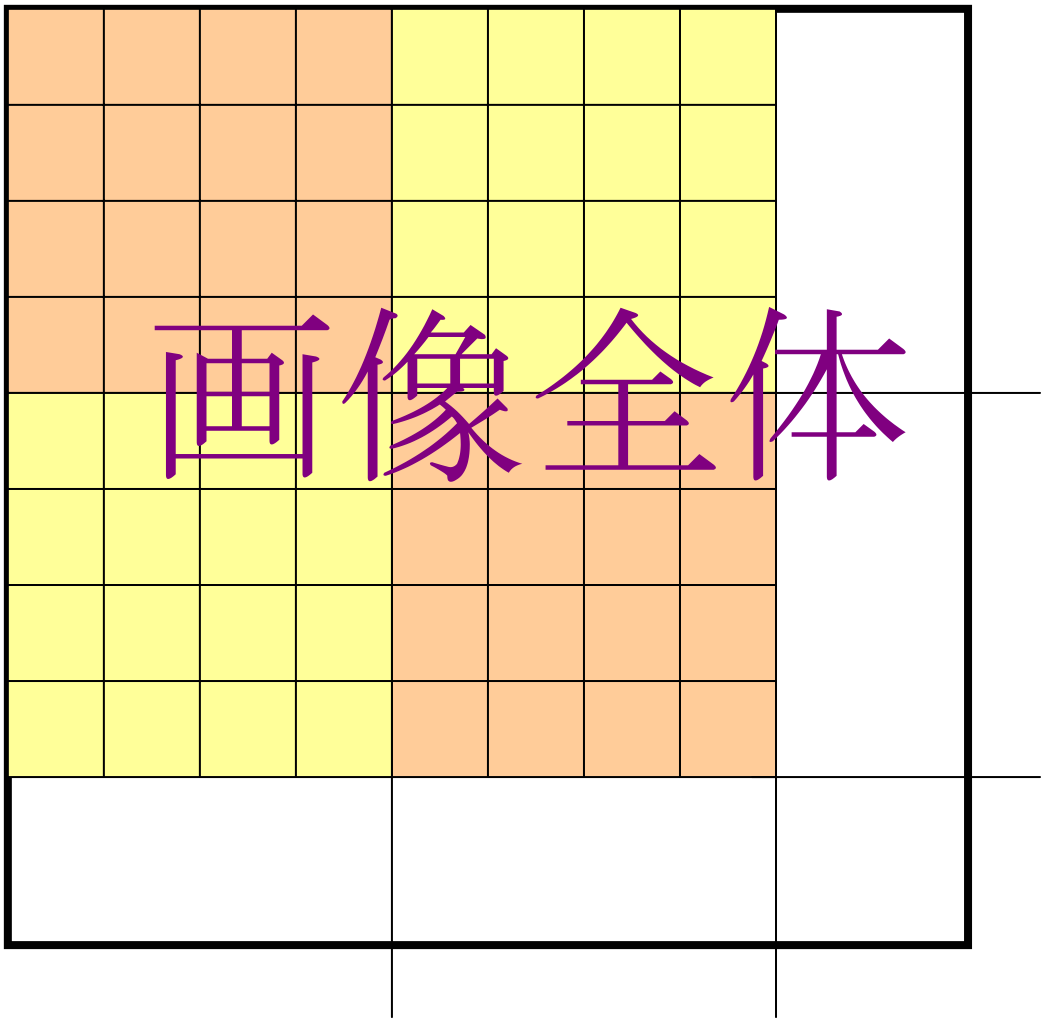


図８：4×４画素の正方短形領域のブロックにわけられた原画像

各ブロックは図９の(a)のような４×４の 16 個の画素からなる正方矩形領域なのであるが、この矩形領域における各画素の輝度を、図９の(b)のように、あらかじめ用意した４×4 の Bayer 型ディザ行列と比較して、表の対応する部分に書かれている数字が自分の輝度より小さければ白(輝度 255)に、大きければ黒(輝度 0)に置き換える。 その置き換えられた様子を図９の(c)に示す。

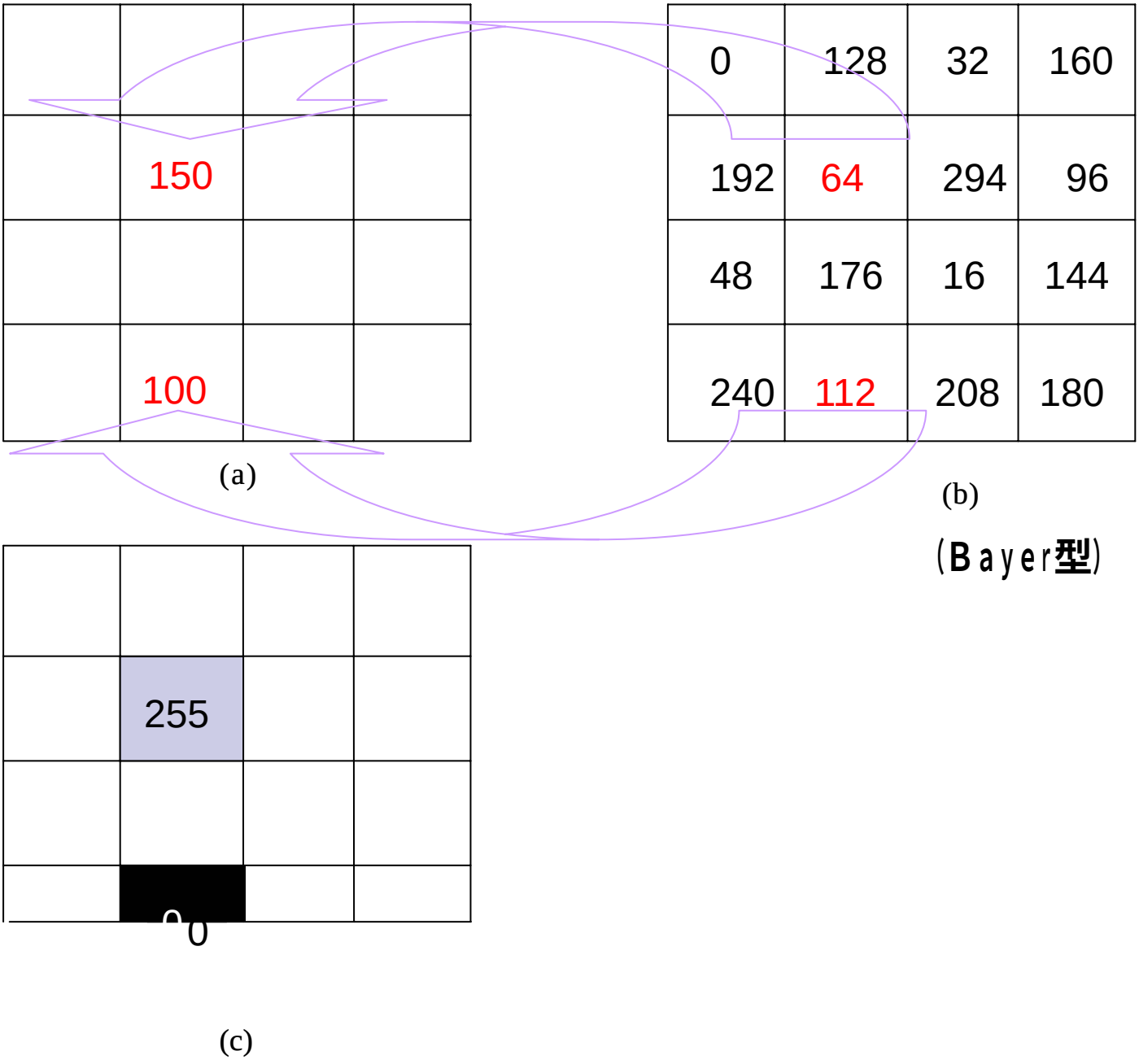


図 9：組織的ディザ法(Bayer 型)

5.2 並列化手法

並列化はブロック分割とサイクリック分割で行った。ブロック分割ではディザ画像を作るときに使われる for ループ内の変数が private となるように指定し、ｘ画素方向に向かって自動分割になるように、16 台のスレッドまで実行した。サイクリック分割では、chunk_size を 50 に設定し、そのイタレーションを各スレッドに静的に割り当ててｘ画素方向に向かって実行させた。

5.3 実行結果と考察

図 11 のような原画像の画素数を 500×500 画素、1000 画素×1000 画素、1500×1500 画素、2000×2000 画素と増やしていき、１６台のクラスタでブロック分割とサイクリック分割で実行したときの実行結果を表 6 と表 7 に示し、2000×2000 画素の原画像を使ってブロック分割とサイクリック分割で実行したとき速度向上比の差を図 10 に示した。またその出力結果は図 12 のような画像になった。

表 6：組織的ディザ法のブロック分割での実行時間(s)

スレッド数	1	2	4	8	16
解像度 500×500	0.12	0.19	0.11	0.08	0.085
解像度 1000×1000	0.48	0.51	0.27	0.20	0.15
解像度 1500×1500	1.06	0.93	0.53	0.36	0.26
解像度 2000×2000	1.88	1.46	0.84	0.54	0.37

表 7：組織的ディザ法のサイクリック分割での実行時間(s)

スレッド数	1	2	4	8	16
解像度 500×500	0.12	0.22	0.12	0.09	0.10
解像度 1000×1000	0.48	0.60	0.32	0.24	0.22
解像度 1500×1500	1.06	1.11	0.61	0.41	0.36
解像度 2000×2000	1.89	1.71	1.00	0.63	0.51

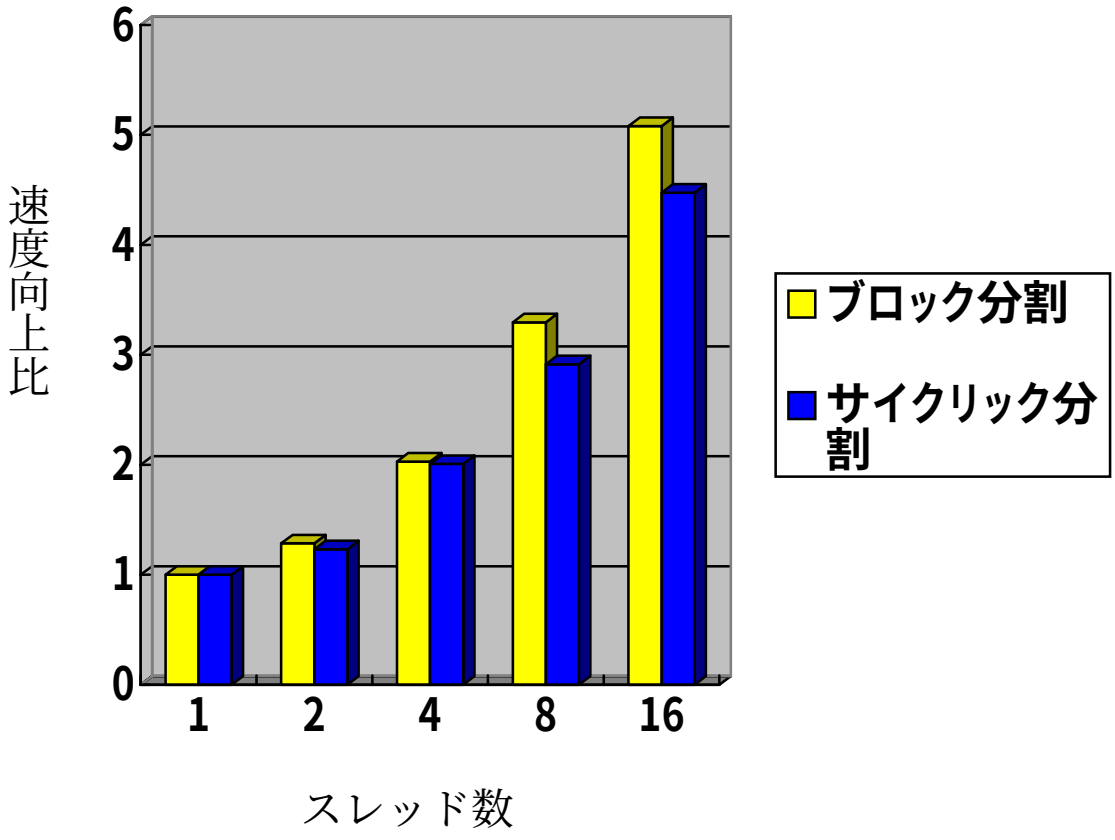


図 10：組織的ディザ法の速度向上比（画素数 2000×2000）

図 11：組織的ディザ法の入力画像

図 12：組織的ディザ法の出力画像

考察

組織的ディザ法ではブロック分割とサイクリック分割で実行したが、実行結果で見る限りではブロック分割では画素数が増やすほど速度向上は上がっているといえる。またサイクリック分割では chuk_size を 1～300 の間でいろいろ変えて実行したが 50 ずつ区切って静的に実行するのが一番時間を短くできることがわかった。動的に実行もしてみたが、やはり静的に実行する方がいい結果となった。しかし、図 10 から判断するとブロック分割と比較してサイクリック分割の方が速度向上が悪いといえる。これは画像処理の並列プログラムは画像の画素一つ一つの処理をスレッドに適用させるので、かなり部分によっては偏りのある処理があるといえる。並列処理として負荷均衡がうまくとれておらず、スレッド間のオーバーヘッドが大きくかかっているといえる。

６．画像の３次元グラフ表示

6.1 問題定義

対象とする画像は pgm 画像とし各画素は 0～255 の階調値をもつものとする。このとき図 13 のように、画像の水平方向の階調変化の様子を、画像の上の走査線から一定間隔ごとに描くことによって立体表現し、結果の画像を作成するものとする。ここで、各走査線上の階調変化曲線を図 13 のように右斜め下方向に 45 度の角度でずらしながら描画して、後ろの曲線を隠すようにしながら注目走査線上の階調変化曲線を描くことによって隠面消去する。このようにして作成される画像の大きさは、原画像が横 WX、縦 WY 画素であるとき、図 13 からわかるように、横 WX+WY、縦 WY+256 ×K 画素となる。ただし K は縦方向の表示倍率で今回はそのＫの値を 1.0 に設定した。

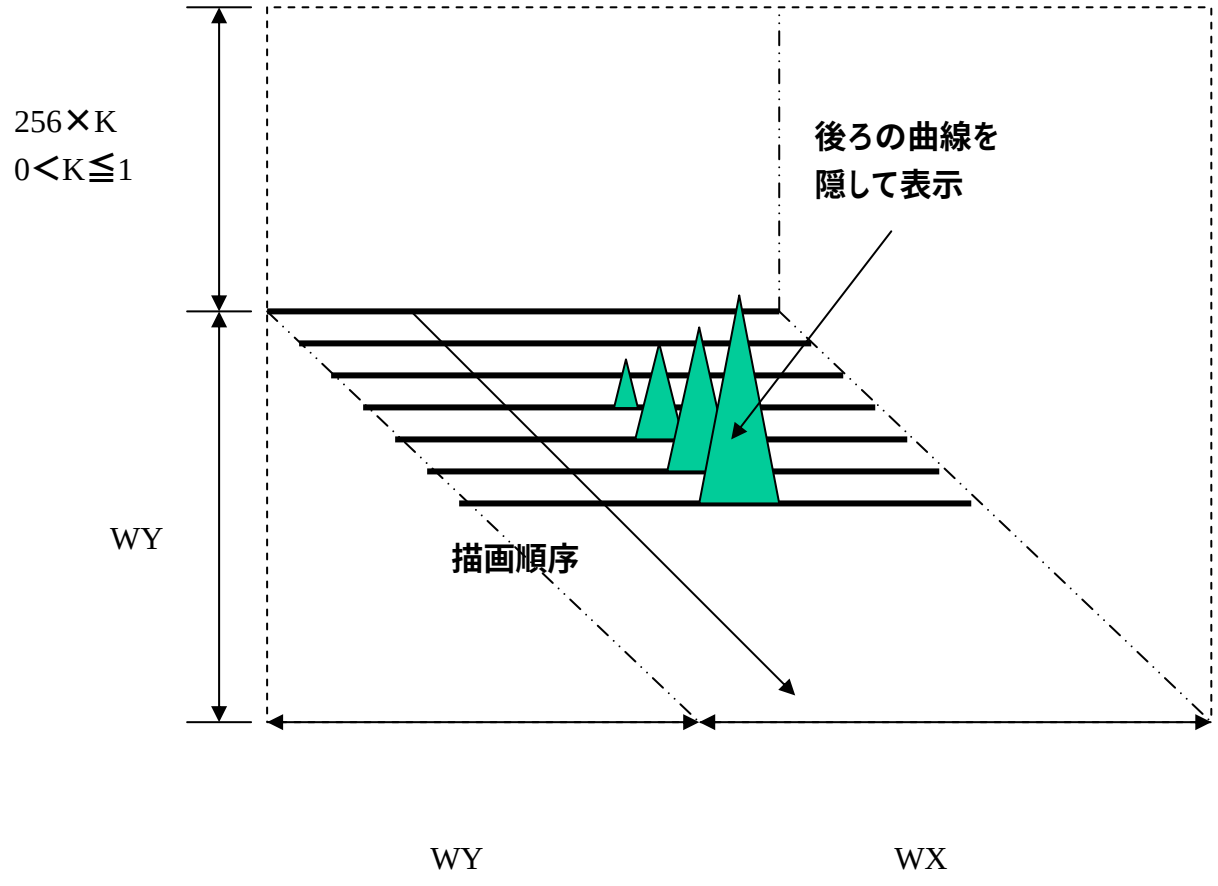


図 13：出力される画像の大きさを示す図

6.2 並列化手法

並列化手法はブロック分割とサイクリック分割で行ったが、図 14 のようにブロック分割では曲線を描いている for ループを x 画素方向に各スレッドがタスクを受け持つように自動分割で実行させた。またサイクリック分割では chunk_size を 50 にし、各スレッドに 50 画素ずつ値を割り当て、その範囲を図 15 に示すように静的にラウンドロビンで実行するものとする。

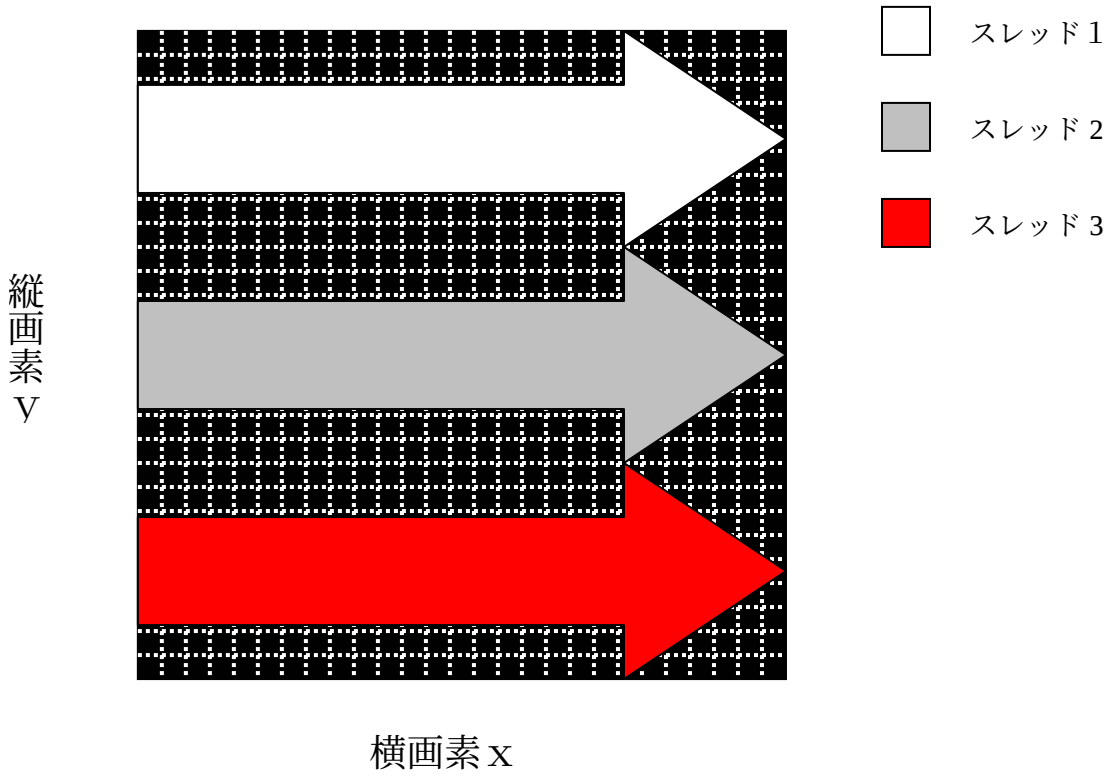


図 14：ブロック分割による並列化手法

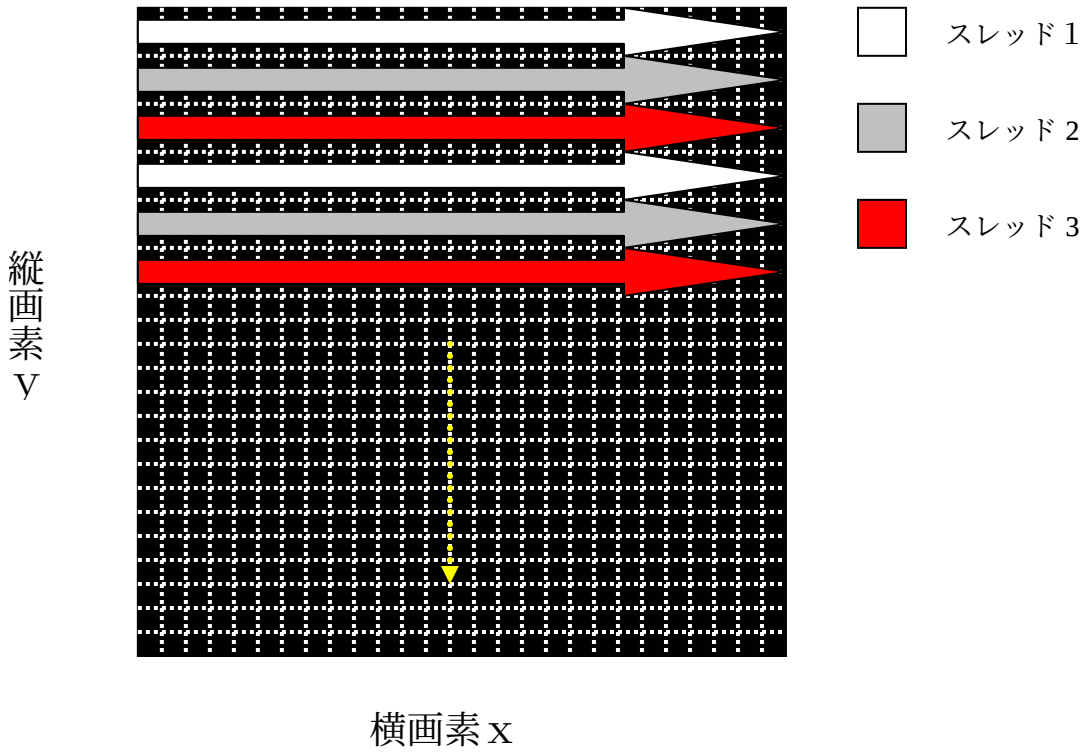


図 15：サイクリック分割での並列化手法

6.3 実行結果と考察

文字である「輔」「前田大輔」の画像と図 20 のような全画素が 1 以上の階調値をもつ画像を入力画像として 3 次元グラフ表示をブロック分割とサイクリック分割で実行した場合の実行結果を画素数別に、それぞれ表 8～表 10、表 11～表 13 に示す。また図 18 と図 20 についてはそれぞれ実行し得られた速度向上比をブロック分割とサイクリック分割で比較するために図 16、図 17 のようにグラフを作成した。

ブロック分割での実行結果

表 8：256×256 画素の時の実行結果（ブロック分割）

スレッド数	**1**	**2**	**4**	**8**	**16**
「輔」の実行時間	**1.04**	**0.67**	**0.61**	**0.40**	**0.30**
「前田大輔」の実行時間	**1.33**	**0.82**	**0.55**	**0.36**	**0.27**
全画素が 1 以上の階調値をもつ画像**の実行時間**	**3.45**	**2.06**	**1.21**	**0.70**	**0.45**

表 9：1024×1024 画素の時の実行結果（ブロック分割）

スレッド数	**1**	**2**	**4**	**8**	**16**
「輔」の実行時間	**15.55**	**8.69**	**7.83**	**4.16**	**2.52**
「前田大輔」の実行時間	**19.16**	**10.63**	**6.82**	**3.86**	**2.19**
全画素が 1 以上の階調値をもつ画像**の実行時間**	**54.85**	**30.61**	**17.71**	**9.13**	**4.67**

表 10：1600×1600 画素の時の実行結果（ブロック分割）

スレッド数	**1**	**2**	**4**	**8**	**16**
「輔」の実行時間	**37.72**	**20.92**	**18.80**	**9.87**	**5.92**
「前田大輔」の実行時間	**46.15**	**25.49**	**16.29**	**9.05**	**5.06**
全画素が 1 以上の階調値をもつ画像**の実行時間**	**134.28**	**74.58**	**43.10**	**22.12**	**11.20**

サイクリック分割での実行結果

表 11：256×256 画素の時の実行結果（サイクリック分割）

スレッド数	1	2	4	8	16
「輔」の実行時間	1.03	0.81	0.50	0.51	0.49
「前田大輔」の実行時間	1.33	0.83	0.44	0.45	0.43
全画素が 1 以上の階調値をもつ画像の実行時間	3.46	1.97	1.16	1.00	0.98

表 12：1024×1024 画素の時の実行結果（サイクリック分割）

スレッド数	1	2	4	8	16
「輔」の実行時間	15.53	9.72	5.43	3.71	2.13
「前田大輔」の実行時間	19.16	11.04	5.79	3.15	2.02
全画素が 1 以上の階調値をもつ画像の実行時間	54.87	30.38	15.36	8.63	5.11

表 13：1600×1600 画素の時の実行結果（サイクリック分割）

スレッド数	1	2	4	8	16
「輔」の実行時間	37.46	21.29	11.71	6.70	4.84
「前田大輔」の実行時間	46.15	26.37	13.96	7.47	5.02
全画素が 1 以上の階調値をもつ画像の実行時間	134.27	74.85	37.64	19.14	10.22

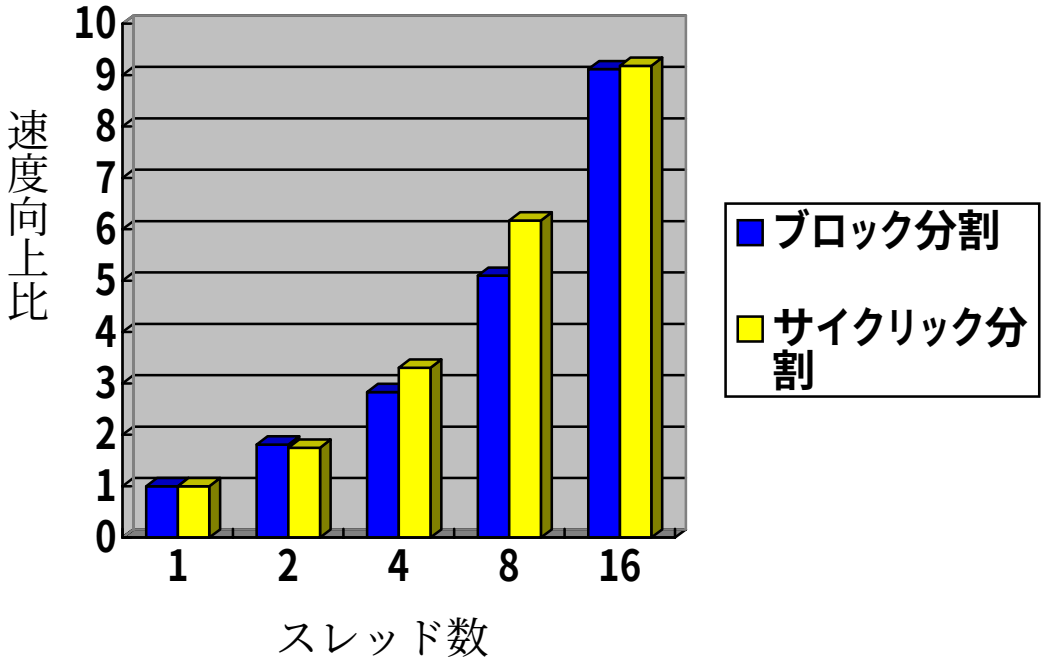


図 16：ブロック分割とサイクリック分割の速度向上比（「前田大輔」解像度 1600×1600）

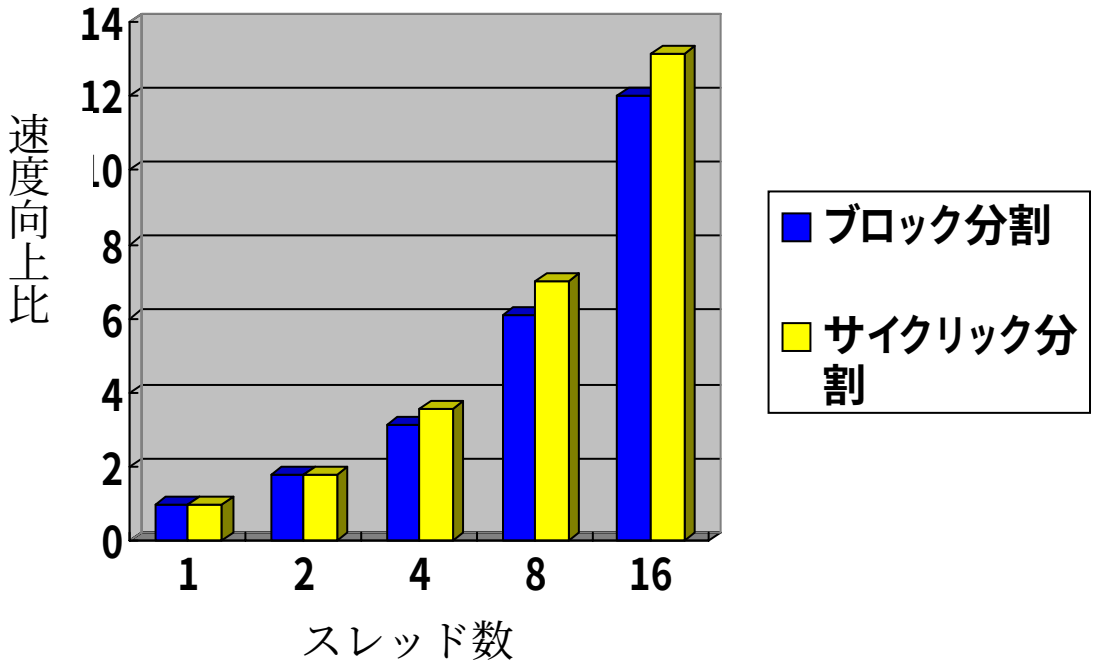


図 17：ブロック分割とサイクリック分割の速度向上比（「全画素が 1 以上の階調値をもつ画像**の実行時間**」解像度 1600×1600）

図 18：原画像「前田大輔」

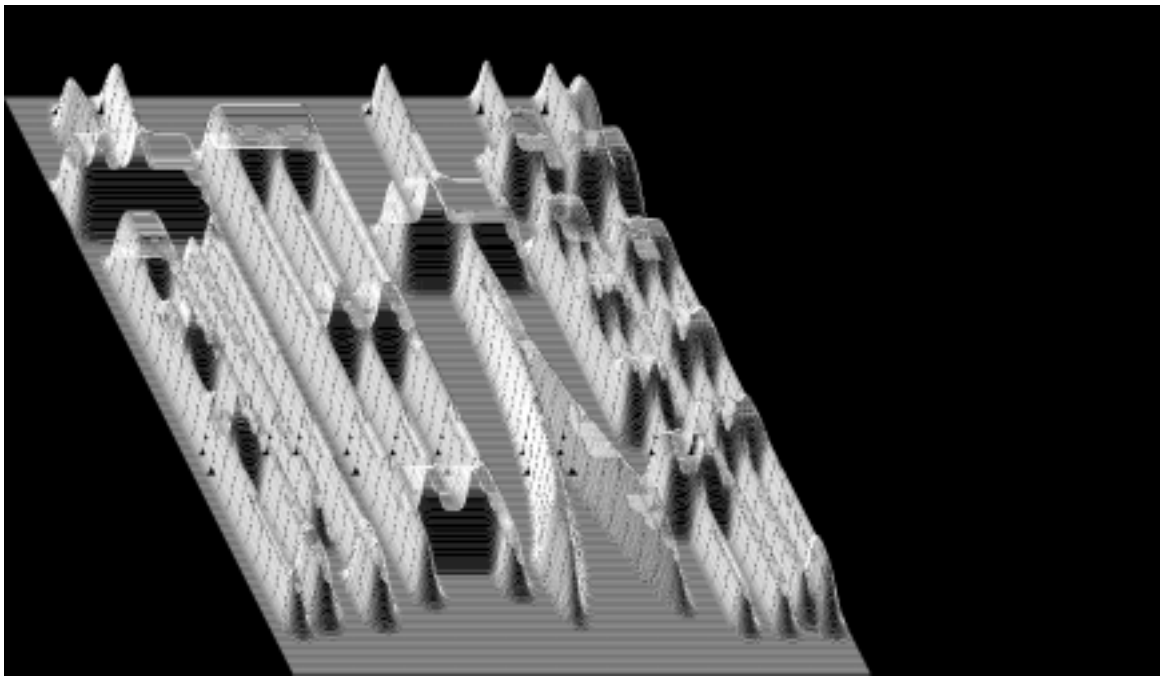


図 19：出力画像「前田大輔」

図 20：画素のすべてが 1 以上の階調値をもつ原画像

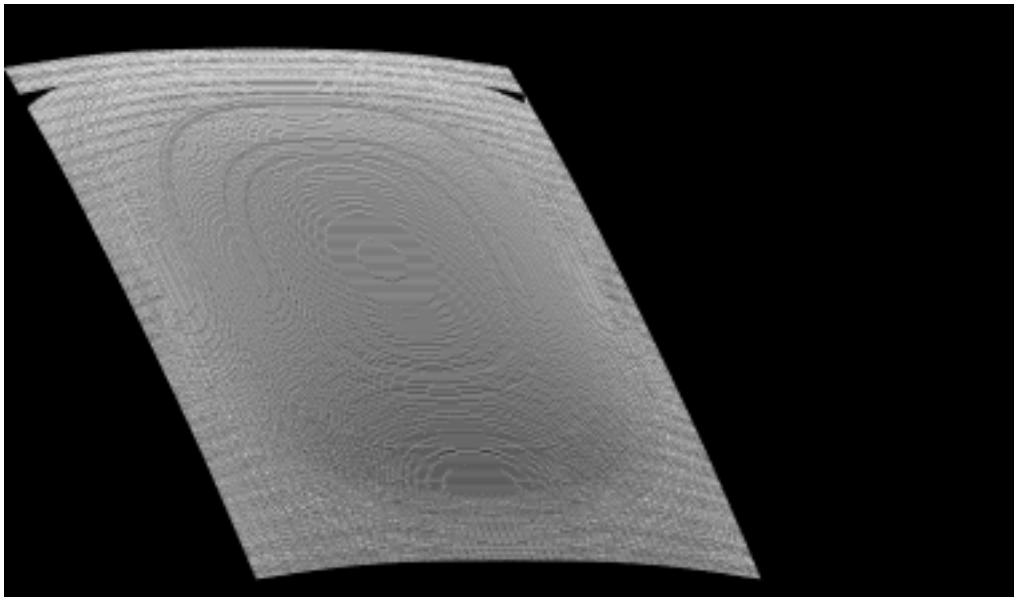


図 21：画素のすべてが 1 以上の階調値をもつ原画像の出力結果

考察

実行結果から得られた結果は図 20 のような原画像中に含まれる文字が占める割合、すなわち階調値が 1 以上でいくらかの値を持っている画素が多いほど実行時間は長くかかっているといえる。それは、画像を出力するときに、階調値の大きい画素がたくさんあるほど、その分だけ多く描写する必要があり、関数によって計算されるデータ量も増加するからである。また文字数が多い方が速度向上比もスレッド数を増やすにつれ良くなっているのがわかる。入力画像では「輔」という文字と「前田大輔」という文字を使って実行してみたが、やはり「前田大輔」という文字を使った時の方が実行時間が長く速度向上比も上がっていた。そこで図 20 のように原画像の画素のすべてが 1 以上の階調値をもっている場合で実行してみたところ、実行時間もかなり増え 16 台で、約 12 倍の速度向上比を得ることができた。

またサイクリック分割でも実行したが、今回では chunk_size を 50 に設定し、そのイタレーションを静的にラウンドロビンでスレッドに割り当てて実行するのが一番速度のでる分割であることがわかった。chunk_size を変え、1、100 のように極端に大きかったり少なかったりするとブロック分割の時よりも並列効果が落ち、うまく負荷均衡がとれていないという結果になった。

７．おわりに

本研究では、Score 型 PC クラスタ上での OpenMP による並列プログラミングを行ってきた。本研究で行ってきたマンデルブロー集合、モンテカルロ法、組織的ディザ法、画像の３次元グラフ表示の並列プログラムを実行することにより、Score 型 PC クラスタが並列効果を出していること、OpenMP が分散共有メモリ上でも正しく動いていることがわかった。しかし画像処理を並列化する時、画像をスレッドの台数分に分割してそのタスクを振り分けて、ブロック分割やサイクリック分割で実行している。数ある画像処理プログラムの内、一つの画素に処理を行う際にその画素の周りの画素の影響を受けるようなプログラムでは、画像を分割した際にその境目の部分では、処理を行う画素の周りの画素が他のスレッドに占有されているために参照することができない。その対策が最も難しく、研究する部分であると思う。また、画像ファイルは容量が大きく、ファイルをクラスタ間や、スレッド間の通信でかなりのオーバーヘッドがかかってしまう。その対策も考えるべきである。

謝辞

本研究の機会を与えてくださり、貴重な助言、ご指導いただきました山崎勝弘教授、小柳滋教授に深く感謝いたします。また本研究にあたり、いろいろな面で貴重なご意見や助言、励ましの言葉を頂きました本研究の皆様に深く心から感謝いたします。

参考文献

[1]　佐藤三久：JSPP'99 OpenMP チュートリアル資料、RWPC、2000

[2]　安居院猛、長尾智晴：C 言語による画像処理入門、昭晃堂、2001

[3]　OpenMP Home Page：http://www.openmp.org.

[4]　内田大介： OpenMP による並列プログラミング１、立命館大学理工学部情報学科卒業論文、2000

[5]　大村浩文： PC クラスタの動作テストと OpenMP 並列プログラミング、立命館大学理工学部情報学科卒業論文、2002.

[6]　柿下裕彰： PC クラスタ上での OpenMP 並列プログラミング（I)、立命館大学理工学部情報学科卒業論文、2003.

[7]　黒川耕平： PC クラスタ上での OpenMP 並列プログラミング（II)、立命館大学理工学部情報学科卒業論文、2003.

[8]　近藤嘉雪： C プログラマのためのアルゴリズムとデータ構造、ソフトバンク、1999.

[9]　古川智之、松田浩一、安藤彰一：並列プログラム事例集、立命館大学理工学部情報学科山崎研究室　/common/jirei/all/caseset、1996.

[10] 米田健治、徳山美香、青地剛宇：並列プログラム事例集２、立命館大学理工学部情報学科山崎研究室　/common/jirei/caseset2、1999.

[11] R.Chandra、L.Dagum、D.Maydan、J.McDonald、R.Menon：Parallel Programming in OpenMP、MORGAN KAUFMANN PUBLISHERS、2000.

[12]　三木光範他：PC クラスタ超入門 2000、PC クラスタ型並列計算機の構築と利用、超並列計算研究会、2000.

[13]湯浅太一、安村通晃、中田登志之：はじめての並列プログラミング、共立出版、1999.

付録1．マンデルブロー集合（ブロック分割）　mandel.c

```
#include<stdio.h>
#include<omp.h>
#include"second.c"
#include<time.h>
double second();
double s1,s2;


main()
{
      int l,N,n;
      int count;
      float D,T;
      float Xmin,Xmax;
      float Ymin,Ymax;
      s1=seconds();


        D=0.001;
        n=0;
        Xmin=-2.5     ,Xmax=0.5;
        Ymin=-2.0     ,Ymax=1.0;
        T=Xmax-Xmin;
        n=T/D;


#pragma omp parallel shared(D,n)reduction(+:count)
{
    int i,j;
    double A,B,x,X,y,Y;

#pragma omp for
    for(i=0;i<=n;i++)
            {
              for(B=Ymin; B<=Ymax; B+=D)
```

```
                {
                   x=0.0;
                   y=0.0;
                  for(j=1;j<=200;j++)
                   {
                         A=Xmin+i*D;
                         X=x*x-y*y+A;
                         Y=2*x*y+B;
                        if((X*X+Y*Y)>4.0)    break;
                         x=X;
                         y=Y;
                   }
                  if((X*X+Y*Y)<=4.0)
                   {
                         count++;
                   }
                 }
               }
             }
/*---------OpenMP-------*/

   printf("count=%d\n",count);

   s2=seconds();

   printf("TIME=%lf\n",s2-s1);
}
```

付録2．モンテカルロ法　monte.c

```
#include<stdio.h>
#include<time.h>
#include<math.h>
#include<stdlib.h>
#include<omp.h>
#include"second.c"

      double second();
      double s1,s2;

main()
{
      long i;
      long loop = 3000000;
      long count = 0;

      double x = 0.0, y = 0.0;
      double Z = 0.0;
      double pi = 0.0;

      s1 = seconds();


#pragma omp parallel for shared(x,y,Z)reduction(+:count)

      for(i = 0; i<loop; i++)

{
    x=(double)rand()/RAND_MAX;

    y=(double)rand()/RAND_MAX;


    Z=x*x+y*y;
```

```
            if(Z < 1.0){
            count++;
            }
}

pi =(double)4*count/loop;

printf("pi = %lf\n",pi);

s2 = seconds();

printf("time = %lf\n",s2-s1);
}
```

付録3． 組織的ディザ法（サイクリック） dither_cyclic.c

```
#include<stdio.h>
#include<stdlib.h>
#include"mypgm2.h"
#include<omp.h>
#include<time.h>
#include"second.c"
#define BLOCK_SIZE 4
#define NEW_LEVEL 16

double second();
double s1, s2;

void make_dither_image()

{
	double width;
	int new_gray;
	int x_block, y_block;
	int x,y,i,j,m,n;
	int dither_matrix[4][4] = {
		{0, 8, 2, 10},
		{12, 4, 14, 6},
		{3, 11, 1, 9},
		{15, 7, 13, 5}
	};

	if(x_size1 % BLOCK_SIZE != 0 || y_size1 % BLOCK_SIZE != 0){
		printf("原画像の縦・横の画素数が不適切です\n");
		exit(1);
	}

	width = MAX_BRIGHTNESS / (double)NEW_LEVEL;
	x_size2 = x_size1;
	y_size2 = y_size1;
```

```
s1 = seconds();

#pragma omp parallel for private(x,y) schedule(dynamic,50)
for(y = 0; y < y_size1; y++){
        for(x = 0; x < x_size1; x++){
            new_gray = (int)(image1[y][x] / width );
            if(new_gray > NEW_LEVEL - 1)
                new_gray = NEW_LEVEL - 1;
            image2[y][x] = (unsigned char)new_gray;
        }
    }

    printf("ディザ画像を作ります。\n");
    x_block = x_size1 / BLOCK_SIZE;
    y_block = y_size1 / BLOCK_SIZE;

#pragma omp parallel for private(i,j,m,n) schedule(dynamic,50)

    for(i = 0; i < y_block; i++){
        for(j = 0; j < x_block; j++){
            x = BLOCK_SIZE * j;
            y = BLOCK_SIZE * i;
            for(m = 0; m < BLOCK_SIZE; m++){
                for(n = 0; n < BLOCK_SIZE; n++){
                    if(image2[y+m][x+n] <= dither_matrix[m][n])
                        image2[y+m][x+n] = 0;
                    else image2[y+m][x+n] = MAX_BRIGHTNESS;
                }
            }
        }
    }

s2 = seconds();

printf("time = %lf\n",s2-s1);
```

```
}

main(int argc, char *argv[] )
{
        load_image_data(argv[1]);
        make_dither_image();
        save_image_data(argv[2]);
        return 0;
}
```

付録4． 画像の３次元グラフ表示（サイクリック）draw_cyclic.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include"mypgm.h"
#include<time.h>
#include<omp.h>
#include"second.c"

double second();
double s1,s2;


void draw_a_straight_line(int x1, int y1, int x2, int y2, int brightness)

{
	double distance, step, t;
	int x, y;

	distance = sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2-y1));
		step = 1.0 / (1.5 * distance);
	t = 0.0;
	while(t<1.0){
		x = (int)(t * x2 + (1.0 - t) * x1);
		y = (int)(t * y2 + (1.0 - t) * y1);
		if(0 <= x && x < x_size2 && 0 <= y && y < y_size2)
			image2[y][x] = (unsigned char)brightness;
		t = t + step;
	}
}

void draw_3D_graph()
{
	int y,x;
	int plot_height;
```

```
        x_size2 = x_size1+y_size1;
        y_size2 = y_size1+256*1.0;
        if(x_size2 > MAX_IMAGESIZE || y_size2 > MAX_IMAGESIZE){
            printf("画面が想定最大サイズを超えています。\n");
            exit(1);
        }else{


        plot_height = (int)(256.0 * 1.0);
        y_size2 = y_size1 + plot_height;

s1 = seconds();
#pragma omp parallel for private(x,y) schedule(static,50)
        for(y = 0; y < y_size2; y++)
            for(x = 0; x < x_size2; x++)
                image2[y][x] = 0;

        printf("画像生成中です。しばらくお待ちください。\n");

#pragma omp parallel for private(x,y) schedule(static,50)

        for(y = 0; y < y_size1; y++){
            if(y % 2 == 0){
                for(x = 0; x < x_size1; x++){
                    draw_a_straight_line(x + y / 2, plot_height + y,
                        x + y / 2,
                    plot_height + y - (int)(image1[y][x] * 1.0),0);
                }

                for(x=0; x < x_size1 - 1; x++){
                    draw_a_straight_line(x + y/2,
                    plot_height + y - (int)(image1[y][x] * 1.0),
                    x + y/2+1,
                    plot_height + y - (int)(image1[y][x+1] * 1.0),
                    MAX_BRIGHTNESS);
```

```
                }
            }
        }
        }
    s2 = seconds();

        printf("time = %lf\n",s2-s1);
}

main()
{
        load_image_data();
        draw_3D_graph();
        save_image_data();
        return 0;
}
```