# 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 ： 黒川 耕平
学 籍 番 号 ： 2210990085-4
指 導 教 員 ： 山崎 勝弘 教授

提 出 日 ： 2003 年 2 月 21 日


立命館大学 理工学部 情報学科

## 内容梗概

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、BM法、KMP法、ランレングス圧縮、Hough変換の6つのプログラムについて実行した。その結果、全ての問題に関して、速度向上が得ることができた。しかしスレッド数が1台のときと、16台のときを比較すると、速度向上が約16にならないと理想的な速度向上が得られたとはいえない。全ての問題について、理想的な速度向上は得られなかった。定義するデータ量が多ければ多いほど速度向上は大きかったが、データ量が大きすぎると、SCASHの制約にひっかかってしまう。いかにしてSCASHの制約にひっかからずに、問題を解決していくかが今後の課題だと思われる。

# 目次

１．はじめに ..... 1
２．並列処理と OpenMP ..... 3
2.1 並列処理とは ..... 3
2.2 OpenMP とは ..... 7
2.3 並列化アルゴリズム ..... 10
2.4 PC クラスタとは ..... 12
3. マンデルブロ ..... 15
3.1 問題定義 ..... 15
3.2 実行結果 ..... 16
3.3 考察 ..... 17
4. 文字列照合 ..... 18
4.1 KMP 法 ..... 18
4.2 BM 法 ..... 20
4.3 実行結果 ..... 22
4.4 考察 ..... 24
５．ランレングス圧縮 ..... 25
5.1 問題定義 ..... 25
5.2 実行結果 ..... 26
5.3 考察 ..... 26
６．Hough 変換 ..... 27
6.1 問題定義 ..... 27
6.2 実行結果 ..... 28
6.3 考察 ..... 29
７．おわりに ..... 30

謝辞 ..... 31

参考文献 ..... 32

付録１　マンデルブロ（ブロック分割）mandel_saitekika_org.c ..... 33
付録２　マンデルブロ（サイクリック分割）mandel_cyclic.c ..... 34
付録３　KMP 法　heiretu_kmp.c ..... 36
付録４　BM 法　bmh6.c ..... 40
付録５　ランレングス圧縮　parallel_pointer2.c ..... 43
付録６　Hough 変換　omp_hutuu.c ..... 47

## 図目次

図 1:逐次処理と並列処理……3
図 2:共有メモリモデル……5
図 3:分散共有メモリモデル……6
図 4:分散メモリモデル……6
図 5:fork-join モデル……7
図 6:並列化アルゴリズムのモデル……11
図 7:ブロック分割とサイクリック分割……12
図 8:本研究室の PC クラスタの構成図……13
図 9:マンデルブロの分割手法……15
図 10:マンデルブロの速度向上比 (解像度 280×349)……16
図 11:マンデルブロの速度向上比 (解像度 2800×3499)……17
図 12:KMP 法の例……19
図 13:BM 法の例……21
図 14:KMP 法の速度向上比……23
図 15:BM 法の速度向上比……23
図 16:ランレングス圧縮の分割手法……25
図 17:ランレングス圧縮の速度向上比……26
図 18:xy平面……27
図 19:θ ρ 平面……27
図 20:Hough 変換の速度向上比……29

## 表目次

表 1-1:Parallel 構文と For 構文……8
表 1-2:Section 構文と Single 構文……9
表 1-3:同期のための構文……9
表 2:ブロック分割の実行結果……16
表 3:サイクリック分割の実行結果……16
表 4:KMP 法の実行結果 (pattern は 30 個)……22
表 5:BM 法の実行結果 (pattern は 30 個)……22
表 6:ランレングス圧縮の実行結果……26
表 7:Hough 変換の実行結果……

# 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はなく、並列I/O の点で計算機クラスタに向いた性質がある。

並列の計算機には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/スイッチ経由で、主記憶に接続された共有メモリモデル（SMP）。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で、プロセッサはほかのプロセッサの主記憶の読み書きができる共有分散メモリ。プロセッサと主記憶からなるシステムが複数個互いに接続された形態で、プロセッサは他のプロセッサの主記憶の読み書きができない分散メモリがある。分散メモリに即した並列プログラミングライブラリとしては、PVM（Parallel Virtual Machine）、および、MPI(Message Passing Interface)が有名である。また、共有メモリに即した並列プログラミングモデルとしては、OpenMP が主流になってきている。最近の汎用マイクロプロセッサの高性能化は凄まじく、一昔前までは手の出なかった高性能スーパコンピュータに匹敵するマイクロプロセッサが、今やWS やPC に使用され、さらにネットワークにおいても近年目覚しい発展がある。ギガビットイーサ、Myrinet などのギガビット級のネットワークが妥当な価格で手に入るようになってきた。ハードウェアばかりでなく通信におけるソフトウェアオーバヘッドを低減化した、低遅延の通信方式も数々開発されてきている。結果としてPC、WS などの汎用計算機と汎用ネットワークを用いた計算機クラスタが、コストだけでなく性能の点でも大きな可能性を持つようになってきた。

本研究室ではこれまで、京都産業大学のSUN Enterprise 4CPU SMPマシンと本研究室の2CPU SMPマシン上でOpenMPプログラムを実行してきた。しかし昨年16台のPCから成るSCore型PCクラスタを構築したことにより、スレッド数16台までの並列処理が実行可能となった。さらにSCore型PCクラスタでは、ソフトウェア分散共有メモリシステム（SCASH）により、分散メモリのクラスタ上で共有メモリプログラミングを可能にする。これによって、Scoreクラスタシステム上でのOpenMP 並列プログラミングが可能になっている。

本研究では、OpenMP によるプログラミングを、PC16台のSCore 型クラスタ上で行った。2章では、並列処理とOpenMPについて、3章から6章に関しては、マンデルブロ、文字列照合、ランレングス圧縮、Hough変換という各問題についての問題定義と実行結果を示している。

マンデルブロとは、ある数式で定義される複素数の数列が有界であるような点を求める計算で、その点の集合をマンデルブロ集合という。この集合は複素数平面上で非常に複雑な形をしており、どんなに拡大しても入り組んだ境界が見られるというものである。

文字列照合にはKMP法とBM法を取り上げており、それぞれ代表的な文字列照合のアルゴリズムである。

ランレングス圧縮は圧縮技術の1つであり、2値の画像ファイルの圧縮などによく用いられる。本

論文では文字列に対して、ランレングス圧縮を実行している。

Hough変換とは2値の画像に対して、直線や円の抽出を行なうときに用いる手法である。本論文では2値画像ファイルと仮定した、0か1の2次元配列の中から1で描かれた直線を抽出する。θ ρ -Hough変換について述べている。それぞれ各章で、問題定義、アルゴリズム、実行結果、考察などを述べている。

## 2. 並列処理と OpenMP

### 2.1 並列処理とは

並列処理とは、いくつかのプロセッサを用いて同一の計算を分担して処理したり、もしくは関連性の無い問題を各プロセッサが同時に処理することである。並列処理の目的には次の3点があげられる。[1]

- 高速性を追求するため。
- プログラミングが楽になるため。
- 本質的に並列なものを扱うため。

最初の理由はいわゆる「1つの仕事をみんなですれば早くおわる。」ということである。つまり3個のプログラムがあったときに、それを3台のプロセッサで並列に処理すれば、処理時間は単純計算で1/3になる。これは、1人では1日かかる仕事を2人ですれば約半日で終わらせられるかもしれないと考えれば、わかりやすいと思われる。この並列処理を簡単に図示したものを図1に示す。この図1はプロセッサ数が1台（逐次処理）のときと、3台（並列処理）のときの流れを示している。

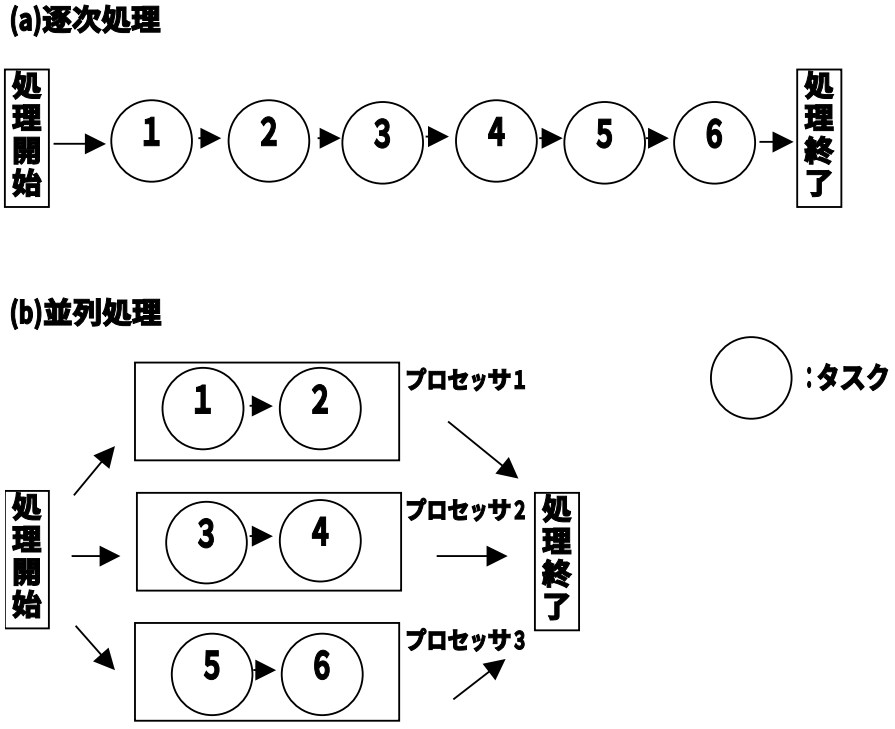



図 1：逐次処理と並列処理

第2の理由は意外だと思われるかもしれないが、マルチスレッドプログラミングはこの観点から行なわれることが多い。しかし楽になると同時に、難しくなるのは明らかである。以上が基本的な並列処理の目的である。

しかし実際に並列処理を行なっても、プロセッサ数に比例した速度向上が得られるとは限らない。その原因として次の3点があげられる。

⑴ 逐次処理部分のオーバーヘッドの問題

これは1つのプログラムの中の逐次処理部分が、どれだけの割合を占めているかということが深く関係している。例えば1台実行時にプログラムの逐次部分の割合が5%であれば、1000台のプロセッサを用いても得られる速度は高々20倍なのである。これは有名なアムダールの法則によって求められる。

この問題に対する対応策として、アルゴリズムやライブラリレベルで並列部分の割合が大きくなるように、プログラマが工夫する必要がある。

⑵ 負荷分散の問題

各プロセッサに与えられた仕事の負荷が大小様々であるとき、つまり、プロセッサ1には0.5秒かかる仕事を与え、プロセッサ2には3.0秒かかる仕事を与え、同時に実行するとき、プロセッサ1が処理を終えても、プロセッサ2はまだ処理途中であり、その処理が終わるまでプロセッサ1はプロセッサ2を待たなければならない。つまり負荷の最も大きいプロセッサに、足を引っ張られることになる。

この問題に関してはプログラマが各プロセッサに仕事を与えるとき、できるだけ各プロセッサにかかる負荷が等しくなるようにすべきである。

⑶ 通信/同期のオーバーヘッドによる問題

並列処理では、他のプロセッサと通信したり、全プロセッサ同期をとることが必要になる。もちろんこの時間は逐次処理では発生しないオーバーヘッドである。

本論文で取り上げているOpenMPは共有メモリ用の並列プログラミング言語であるため、パイプやメッセージパッシングなどの、プロセッサ間の通信のオーバーヘッドは存在しない。しかしながら、同期のオーバーヘッドは存在するので、この問題に関してはプログラミングする際に注意する必要がある。

この問題は唯一、ハードウェアアーキテクチュアで対応できる問題である。

並列処理を行なう上で、プログラマが意識すべきアーキテクチュア上の特徴で、プログラミングに影響を与える因子としてメモリモデルがあげられる。主なメモリモデルとしては、以下の3点があげられる。

(1) 共有メモリモデル

複数のプロセッサがメモリバス/スイッチ経由で、主記憶に接続される形態である(図2参照)。このアーキテクチュアを有するシステムのことをSMP (Symmetrical Multi Processor) と呼ぶ。この形態の利点はメモリモデルが最も汎用でプログラムが組みやすいことと、並列処理だけでなく、スループットを重視するサーバマシンとしても適しているということである。逆にキャッシュの一貫性をいかに実現するかという課題点も有している。

(2) 分散共有メモリモデル

プロセッサと主記憶から構成されるシステムが、互いに接続された形態である。ただしプロセッサは、他のプロセッサの主記憶の読み書きを行なうことができる(図3参照)。この形態の利点は自由度が高く、大規模なシステムの構築が可能だということである。逆に1個でもバリア同期の場所を間違えると、タイミング依存で非常にわかりにくいバグを作りこむことになるという欠点も持つ。

(3) 分散メモリモデル

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である。ただしプロセッサは、他のプロセッサの主記憶の読み書きを行なうことができない(図4参照)。この形態の利点は、大規模なシステム構築が可能であることと、デッドロックさえ気をつければ、タイミングに依存する嫌なバグは発生することが少ないことである。一方、通信をプログラミング時に全てスケジューリングすることが、プログラマにとって非常に多大な負担になるという欠点も持つ。

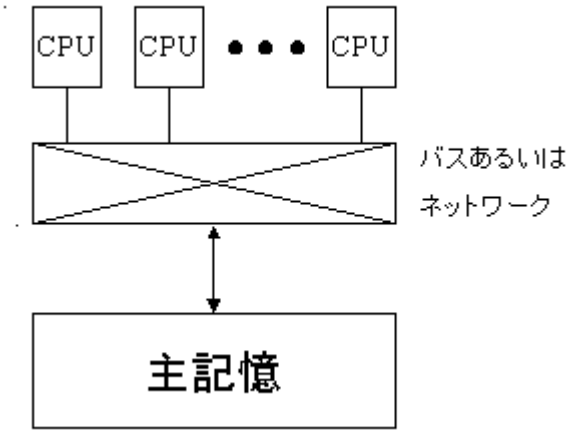



**図２：共有メモリモデル**

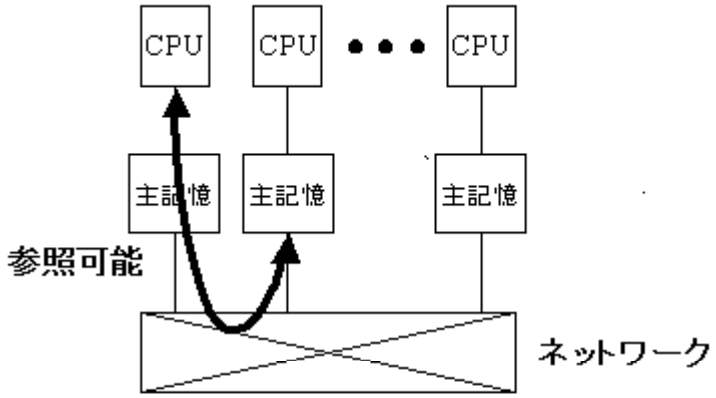



図 3：分散共有メモリモデル

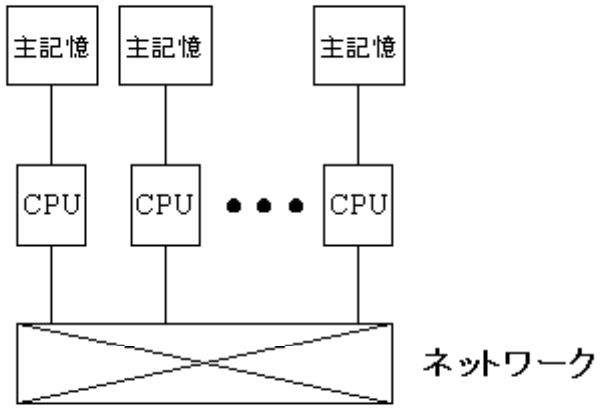



図 4：分散メモリモデル

## 2.2 OpenMP とは

OpenMPは共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデルであり、ベース言語（Fortran、C、C++）をコンパイラ指示文（directives/pragma）、ライブラリ、環境変数によって拡張したものである。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なう。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、逐次版と並列版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並列実行モデルを用いている。プログラムはまず、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指示文に遭遇すると、複数のスレッドから成るチームを作成し、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実行される部分を並列リージョンといい、マスタスレッドだけで実行される部分を逐次リージョンという。ワークシェアリング構文では、no wait指示文が指定されてなければ、暗黙のバリア同期が自動的に行なわれ、プログラマは同期についてあまりとらわれずにプログラミングをすることができる。図5はfork-joinモデルで書かれたプログラムのスレッドの動きを表したものである。

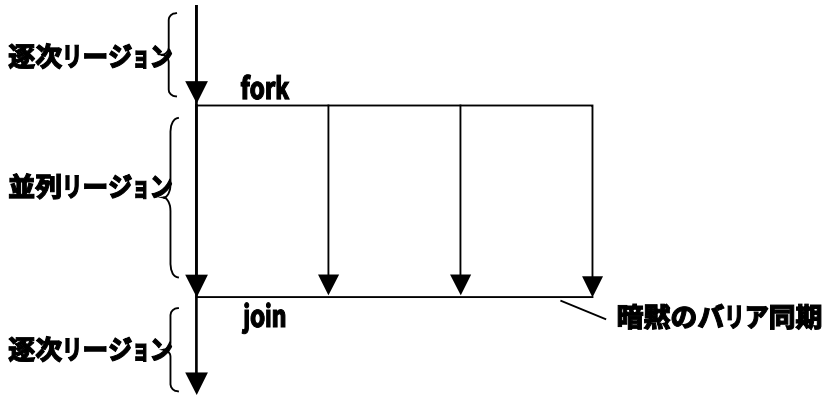



**図５：fork-join モデル**

OpenMPのC言語での指示文は、pragma行を用いて、

#pragma omp　OpenMP構文名　clause　・・・

と記述される。基本的な構文とそのclauseを表1-1、1-2に示す。またOpenMPには同期のための構文も存在する。同期のための構文を表1-3に示す。

表 １－１：Parallel 構文と For 構文

| | | |
|---|---|---|
| Parallel構文 | #pragma omp parallel ……<br>{block-statement}　　で書き出し。 | 並列リージョンを定義する。この指示文は並列実行を開始する基本的な構文である。 |
| private(list) | | 変数listが各スレッドで異なる値を持つ。 |
| shared(list) | | 変数listが各スレッド間で共有される。 |
| default(SHARED ¦ NONE) | | デフォルトで適用される属性を示す。 |
| firstprivate(list) | | はじめに元のオブジェクトからコピーされる以外はprivateと同じ。 |
| reduction({operator} : list) | | 各スレッドで求められたlistをoperatorで定義された演算子によって演算し、1つにまとめる。演算子には(+,+,-,^,&,&&,¦,¦¦) がある。 |
| if(scalar_logical_expression) | | parallel構文でのみ使用可能であり、条件が真のときのみ並列実行される。 |
| For構文 | #pragma omp for ……<br>for(…; …; …) {body}　　で書き出し。 | この構文は構文直後forループを並列実行する。ただしforループは正規型(canonical) でなくてはならない。 |
| private(list) | | parallel構文と同じ。 |
| firstprivate(list) | | parallel構文と同じ。 |
| lastprivate(list) | | 最後にオブジェクトに書き戻す。その規則はfor directiveに現れたときにはその中の条件などは関係なく、最後のinterationを実行したスレッドが元のオブジェクトを更新する。 |
| reduction({operator} : list) | | parallel構文と同じ。 |
| schedule(type[, chunk]) | | ループをどのようにスレッドに分割して実行するかを指定する。 |
| nowait | | Nowaitが指定されているときはimplicitに同期がとられる。 |

表 1 - 2：Section 構文と Single 構文

| | | |
|---|---|---|
| Section構文 | #pragma omp sections<br>{<br>#pragma omp section<br>{section1}<br>#pragma omp section<br>{section2}<br>}　で書き出し。 | チーム内のスレッドで分割して実行する構文の集合を指示する、非繰返しワークシェアリング構文である。各セクションはチーム内のスレッドによって1度だけ実行される。 |
| Single構文 | #pragma omp single<br>{block-statement}　で書き出し。 | 対応する構造ブロックがチーム内の1つのスレッドのみで実行されることを指示する構文である。 |

表 1 - 3：同期のための構文

| | |
|---|---|
| master構文 | マスタがスレッドのみで実行する構造ブロックを指示する構文である。マスタ以外のスレッドは指定されたステートメントを実行しない。マスタセクションの入り口と出口は暗黙のバリア同期を実行しない。 |
| critical構文 | 排他的に実行されるcritical sectionを指定する指示文である。大域的な名前をつけることができ、同じ名前のcritical sectionは排他的に実行される。名前のない場合は他の名前がないcritical sectionと排他的に実行する。 |
| barrier構文 | バリア同期を指示する構文である。チーム内のスレッドが同期点に達するまでまつ。並列リージョンのおわり、ワークシェアリング構文でnowait指示文が指定されない限り、暗黙のバリア同期が行なわれる。 |
| atomic構文 | 複数の同時書き込みを行なう可能性のあるスレッドに対して、指定されたメモリをアトミックに更新することを指示する構文である。 |

## 2.3 並列化アルゴリズム

並列化するにあたってまずプログラマが意識しなければならないものに、並列化アルゴリズムがあげられる。並列化アルゴリズムは一般的に大きく4つに分類される。

(1) 分割統治法

まず問題に対して何らかの計算を行なう。そしてその結果をある条件をもとに分割し、またその計算を行なう。ある条件が満たされるまで計算と分割を繰り返していき、その部分解を全て統合することによって結果を得る。

(2) プロセッサファーム

まずマスターによって処理を開始する。マスターは与えられた問題に対し複数の独立した計算に分割し、マスターと各スレーブがそれぞれ計算を行ない、その結果を再びマスターが回収し結果を得る。このアルゴリズムはマスターとスレッドで独立して計算が行なわれるので、並列効果が出やすいといわれる。

(3) プロセスネットワーク

ある問題に対して、計算ステージを複数に分割する。データはあるステージで計算され、それが終わったら次のステージに移り計算される。というふうに各ステージを複数のデータが流れていく。パイプライン処理とも呼ばれる。

(4) 繰り返し変換

ある問題を複数のオブジェクトに分割し、各オブジェクトは複数の計算の繰り返しによって値が変更される。オブジェクトの値がある条件を満たすまで繰り返し実行することで解を求める。

本論文ではこれら4つの並列化アルゴリズムのうち、高い並列効果が期待できるプロセッサファームを用いて並列プログラムを作成し実験を行っている。その実験方法や並列化手法は、各問題の章で詳しく述べている。(1) から (4) の並列化アルゴリズムのモデルを図6に示す。

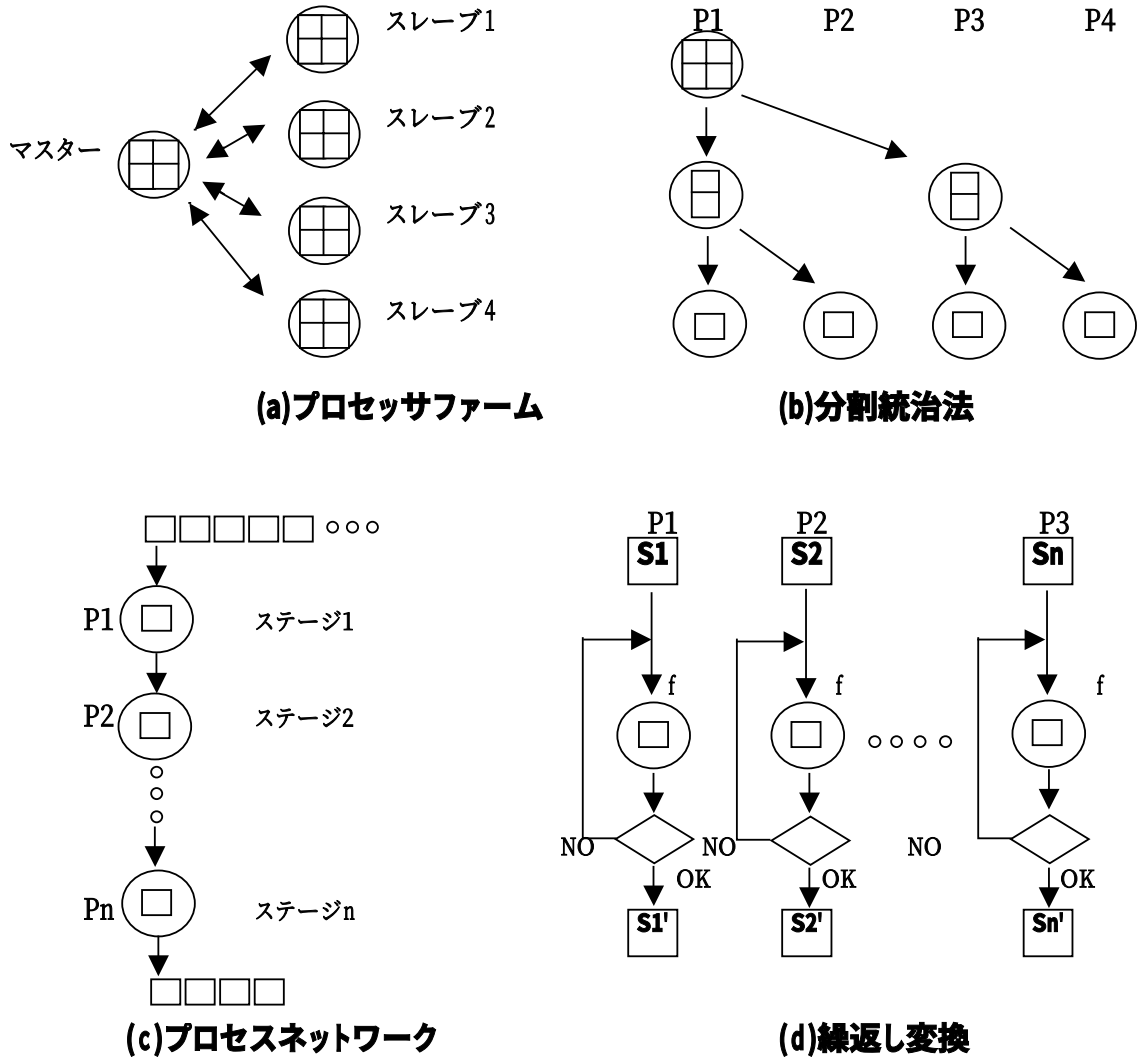



図６：並列化アルゴリズムのモデル

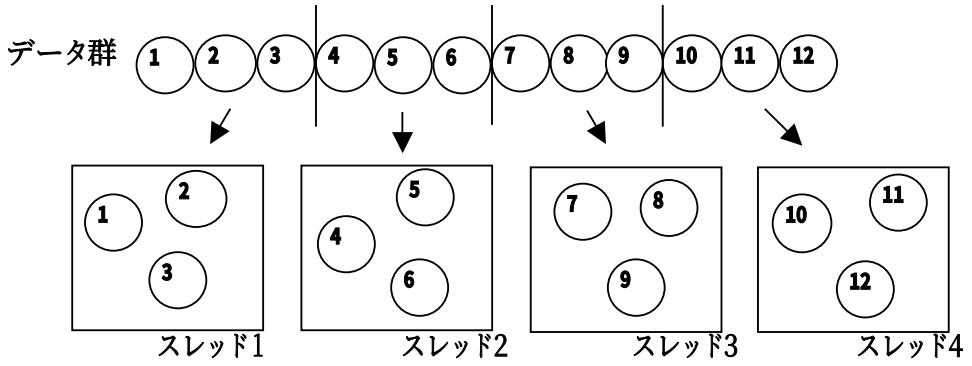
次に並列化するときの重要な概念として、分割手法があげられる。分割という概念には各スレッドに同一の処理をさせるとき、その処理のためのデータ群を分割するものと、互いに相反しないタスクを各スレッドが独立に処理していくという、タスクを分割するものが存在する。本論文では前者のデータ群を分割し、各スレッドに割り振るという概念を全てのプログラムに対して用いている。そのデータの分割手法として以下のブロック分割とサイクリック分割を取り上げていて、問題によってどの分割手法が適しているかを考えなければならない。ブロック分割とサイクリック分割のモデルを図7に示す。

(1) ブロック分割

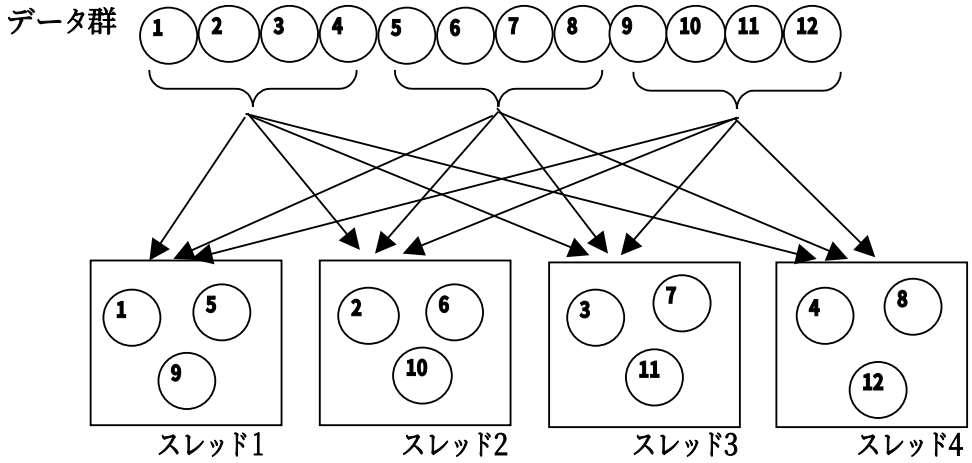
各スレッドに計算範囲を割り振るとき、全体の計算範囲を起動するスレッド数で割り、その範囲を１ブロックとして、各スレッドに割り振る。

(2) サイクリック分割

各スレッドに計算範囲を割り振るとき、その計算範囲を細かく分割し、スレッド1から順に割り振っていく手法である。細かく範囲を区切っていくので、負荷均衡をとりやすいというのが、この分割手法の利点である。




**(a)ブロック分割**




**(b)サイクリック分割**

**図７：ブロック分割とサイクリック分割**

### 2.4 PC クラスタとは

コンピュータをネットワークでつなげた処理を行うシステムをクラスタと呼ぶ。とくに PC を使って並列処理を行うシステムは PC クラスタと呼ばれている。本研究室の PC クラスタは、計算ノードとして PC１６台と、それらを管理するサーバを 1 台設置し，研究室内の 100BASE-T Ethernet に接続してある。また、Myrinet2000 を用いて PC16 台をつないでいる。

Myrinet2000 では 4Gbps(片方向 2Gbps，双方向通信)の高速通信が可能である。次にその上に RWCP で作成された Score というソフトウェアを搭載している。

また SCore には分散メモリのクラスタ上で共有メモリプログラミングを可能にする SCASH というシステムが存在し、2.1 で述べた分散共有メモリアーキテクチャを実現している。つまり、これにより OpenMP の実行が可能となっている。本研究室の PC クラスタの構成図を図 8 に示す。

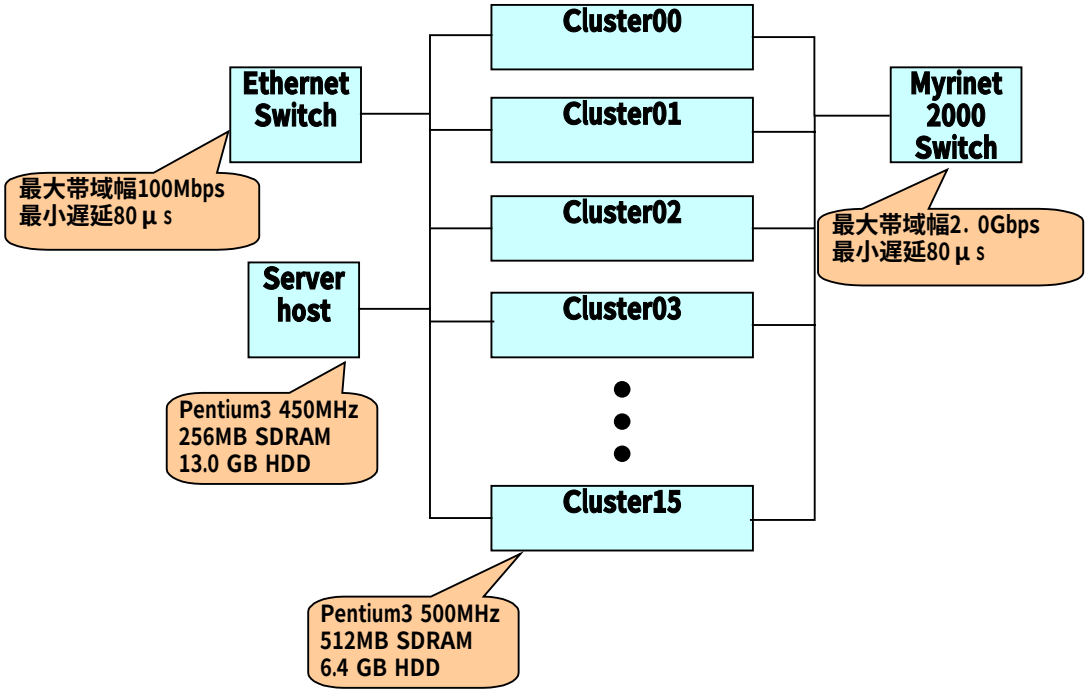



**図 8：本研究室 PC クラスタの構成図**

SCore クラスタ上で OpenMP プログラミングをするときの制約や注意点について、以下のことがわかっている。

(1) ユーザが使用可能なメモリ空間

Score では分散共有メモリを SCASH で実現しているため、ユーザが使用可能なアドレス空間には制限がある。もしその制限を越えた場合、OpenMP で書かれたプログラムは実行時に、VM Fault out of DSM area PC=[080bb0d6]:ADDR=[00000024] のようなエラーが出る。この他にもメモリに関するエラーは存在する。

(2) 共有引数のスタックオーバーフロー

引数メモリサイズは, デフォルトでは 4KB になっている. 関数内で宣言する変数のサイズの合計がこれを超える場合には, 環境変数(OMNI_SCASH_ARGS_SIZE)で引数メモリサイズを指定しなければならない。この設定を行わず実行してしまうと、実行時に　_ompsm_scash_fatal: shared arg stack overflow　というエラーが出る。

SCASH の制約エラーはこの他にも多数存在していると思われる。上の2つはその中でも最も頻繁に現れたエラーであり、なぜこのようなエラーが起こるのかということも、プログラミングを行っていくうちにわかってきたことである。今後まだ解明されていないエラーについては調査する必要があると思われる。

## 3. マンデルブロ

### 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、$Z_k(k\rightarrow\infty)$の値が収束か、発散かを求める。

(－2.0≦実部<0.6)、(-2.0≦虚部<1.5)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なう。その結果 $|Z_i|>4.0$ で k<=30 のとき発散、$|Z_i|<=4.0$ で k>30 のとき収束するとして、後者の結果となる座標点が、マンデルブロ集合の要素となる。各座標点に対して、マンデルブロ集合に属する座標点の合計数を count に格納する。

マンデルブロの計算を並列に計算するためには、各スレッドに計算領域を縦に分割して割り振らなければならない。そのときの計算領域の分割手法に、ブロック分割とサイクリック分割の２つの手法を試みた。サイクリック分割とブロック分割の分割手法を以下の図 9 に示す。

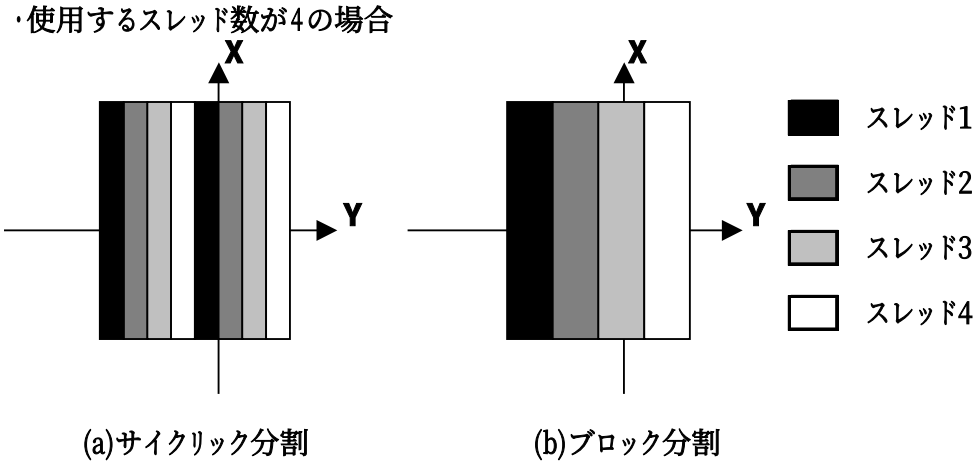



図９：マンデルブロの分割手法

### 3.2 実行結果

実行時の解像度は 280×349 のと、2800×3499 に設定した。それぞれブロック分割のときとサイクリック分割のときの実行結果を表 2、表 3 に示し、速度向上比を図 10、図 11 に示す。

表 2：ブロック分割の実行時間（単位は秒）

| スレッド数 | 1 | 2 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|
| 280*349 | 0.202 | 0.068 | 0.039 | 0.025 | 0.020 | 0.017 |
| 2800*3499 | 20.20 | 6.72 | 3.69 | 2.14 | 1.51 | 1.19 |

表 3：サイクリック分割の実行時間（単位は秒）

| スレッド数 | 1 | 2 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|
| 280*349 | 0.213 | 0.108 | 0.055 | 0.029 | 0.021 | 0.017 |
| 2800*3499 | 21.23 | 10.60 | 5.30 | 2.65 | 1.77 | 1.33 |

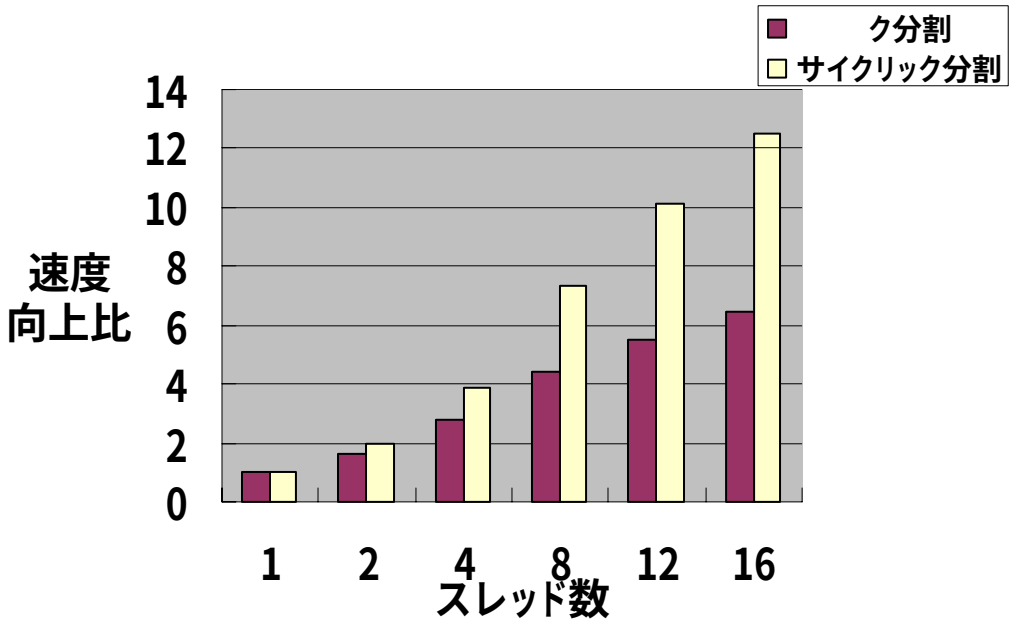



図 10：マンデルブロの速度向上比（解像度 280×349）

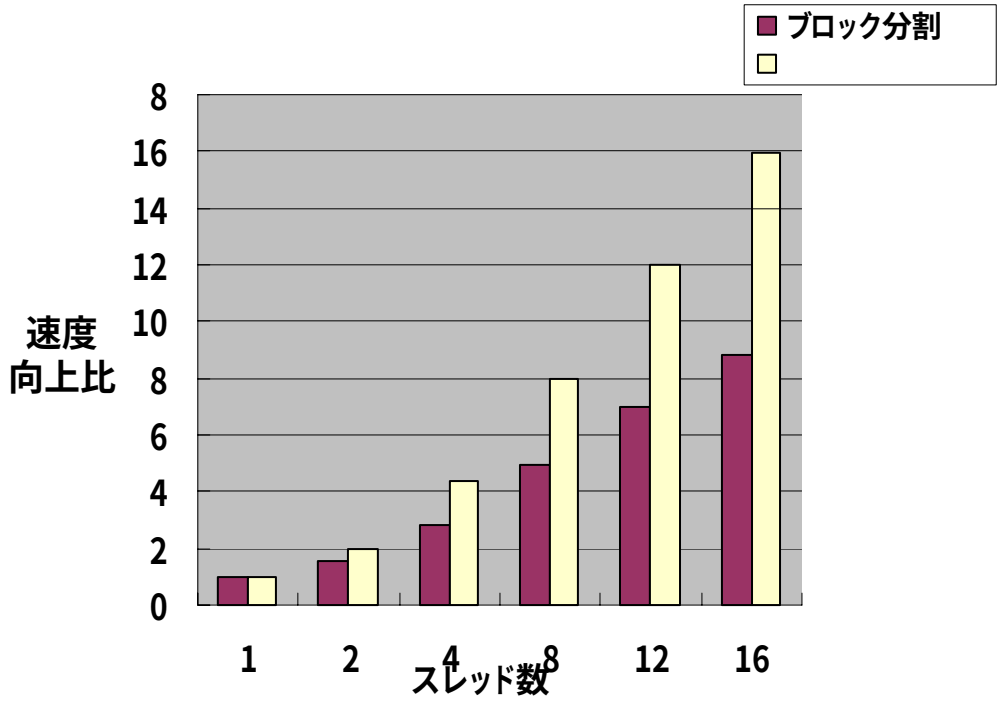



**図 11：マンデルブロの速度向上比（解像度 2800×3499）**

### 3.3 考察

マンデルブロでは、ブロック分割とサイクリック分割の２種類の分割法により、実行した。両者の速度向上比を比較してみると、サイクリック分割のほうがブロック分割よりも理想的かつ大きな速度向上が得られた。サイクリック分割とブロック分割のちがいは x の範囲によって発散せずに収束するまで計算される複素数の点の数がちがうことである。そのため、x の範囲を大きく分割したブロック分割では、各スレッドでの計算量も大きく異なる。それにより、プログラム全体の処理時間も、最も遅いスレッドに引っ張られることになる。このことは、２章で述べた負荷分散のばらつきの実例といえる。つまりマンデルブロの場合、サイクリック分割とブロック分割では、サイクリック分割のほうが負荷均衡がよくとられており、速度向上比も大きいということが結論である。

## 4. 文字列照合

### 4.1 KMP 法

KMP 法とはある文字列 text から pattern と呼ばれる文字列を探索する、いわゆる文字列照合アルゴリズムの１つである。KMP 法が単純な文字列照合アルゴリズムに比べて、計算量が少ない理由は、text を遡って読むことがないことである。

単純なアルゴリズムでも KMP 法でも text 文字列 ABCABCABA から pattern である ABCABA を探すとき、まず左端をあわせて左から１文字ずつ比較していく。

ＡＢＣＡＢＣＡＢＡ
ＡＢＣＡＢＡ

すると赤丸で囲んだ左から６文字目の文字で合わなくなる。単純なアルゴリズムならば、pattern を１文字右にずらしてもう１度 pattern のはじめの文字から比較していくのだが、このとき、pattern の１、２文字目と３、４文字目が両方ともＡＢであることを知っていれば、text の４，５文字目のＡＢと pattern の１、２文字目のＡＢを合わせて、pattern の３文字目から照合をはじめればよい。

ＡＢＣＡＢＣＡＢＡ
　　ＡＢＣＡＢＡ

このことを能率よく行なうためには、「pattern の６文字目（pattern[5]）で合わなくなったときは、pattern の３文字目（pattern[2]）から比較を開始すればよい」という決まりを作っておけばよい。この決まりを next[5]=2 と定義することにする。つまり、この決まり next[j]は pattern の先頭ｊ文字について、その頭と尾が何文字一致するかを調べたものであるといえる。つまり、この pattern「ABCABA」に関しては、

next[1]=next[2]=next[3]=0　　next[4]=next[6]=1　　next[5]=2

となる。これは２文字目、３文字目、４文字目で合わなかったら pattern の 1 番目の文字（pattern[0]）から照合を開始し、５文字目で合わなくなるか、pattern が見つかったときは pattern の２文字目（pattern[1]）から照合を開始し、pattern の６文字目で合わなくなったときは pattern の３文字目（pattern[2]）から照合を開始するという決まりを表している。この next[j]を求めるには、次のように pattern を１文字ずらして並べ、next[1]=0 をはじめとして KMP 法を用いれば求めることができる。

ＡＢＣＡＢＡ
　ＡＢＣＡＢＡ‥‥‥　と KMP 法を実行していく。

KMP 法の例を以下の図 12 に示す。

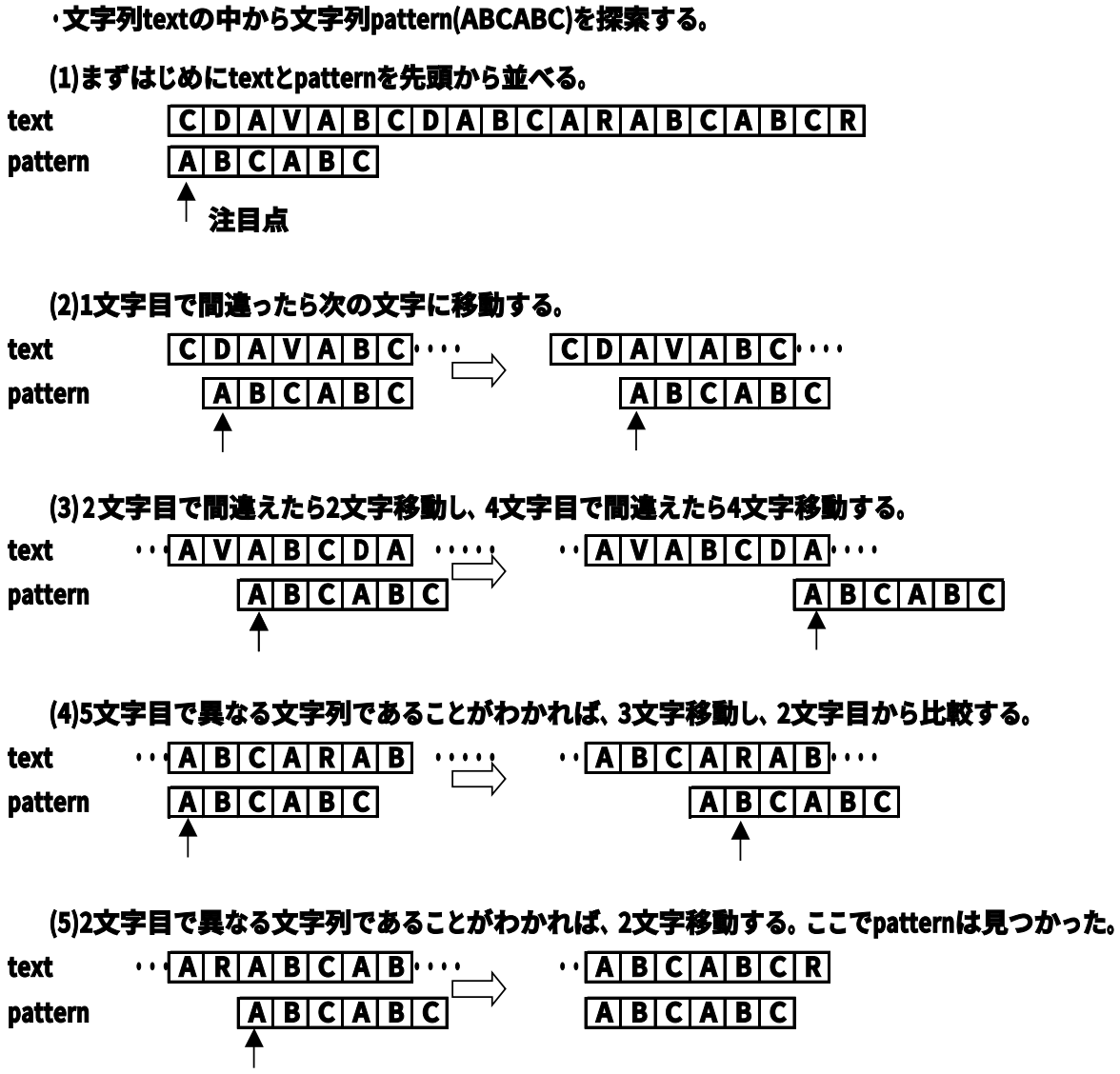



図 12：KMP 法の例

文字列探索の対象である配列 text はあらかじめファイルに格納しておき、用意した pattern に対して１つずつ照合を行っていき、text の末端部まで検索し終わったら、次の pattern を設定して、また検索を始める。pattern が text の何文字目に存在したかを、配列 result に格納する。

text は使用するスレッド数と同数のブロックに分割され、各スレッドは自分の文字列範囲の中で照合を行なっていく。隣接した２つのブロックの境界をはさんで pattern が存在す

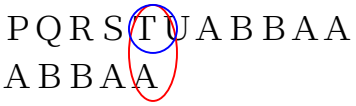
る場合があるので、境界付近の一定領域は重複して２つのスレッドに割り振られる。pattern の情報は全スレッドに割り振られ、それぞれがその pattern の情報を解析して、text と照合していく。

### 4.2 BM 法

BM 法も KMP 法と同じ、文字列照合アルゴリズムである。BM 法と KMP 法の違いは、pattern の照合方法にある。KMP 法では、pattern の先頭から照合するが、BM 法では逆に、pattern の末尾から文字列照合を行なっていく。末尾からの照合があわなかった時点で、それより前の照合は行なわずに、pattern を pattern の長さだけずらして、次の検索を再開する。BM 法は文字列照合の中では、最も高速なアルゴリズムである。

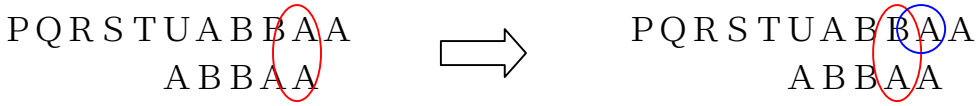
text 文字列 PQRSTUABBAA から pattern である ABBAA を検索するとき、まずはじめに text と pattern を先頭から並べる。このとき注目する文字はⅠ段階の赤丸で囲まれた pattern の末尾の文字（A）とその上の text の文字（T）である。

**Ⅰ段階**




T と A は一致しないことがわかる。そして、青丸で囲まれた T という文字は pattern には 1 つもないので、照合文字列を一気に 5 文字分（pattern の長さ分）ずらす。そして注目する文字はⅡ段階の赤丸で囲まれた pattern の末尾の文字（A）とその上の text の文字（A）である。

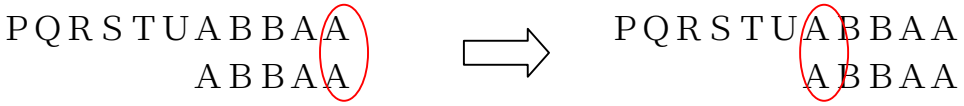
**Ⅱ段階**




これらは互いに一致する。一致すればその左の文字を 1 文字ずつ比較していく。しかし次の文字（B と A）ですぐに合わなくなる。この場合はⅠ段階のように pattern を 5 文字移動するわけにはいかない。このときは青丸で囲まれた文字（A）に注目する。青丸で囲まれた文字が pattern に含まれる文字の場合、pattern のその文字と青丸で囲まれた文字を合わせる（この場合 1 文字ずらす）。A という文字は pattern の中に３文字存在するが、その中で青丸の A と重ねるのは４文字目の A である。５文字目の A はすでに pattern と重なっているので当然選ぶことはない。１文字目の A を選んでしまうと４文字ずらすことになり、pattern を見逃す場合があるので選ばない。つまりこの場合では４文字目の A を選ぶことになる。つまり選ぶ文字は、末尾以外の１番右の文字であるといえる。そして再び末尾から

比較していく。注目文字はⅢ段階の A と A である。

**Ⅲ段階**




A と A は同じ文字でありそれから順に text と pattern を１文字ずつ比較していくと、pattern が text の中から発見されたことがわかる。これらの動作を効率的に行なうためには、「照合があわなかったときの pattern の末尾と対応する text の文字が、pattern に含まれる文字ならば、その pattern の末尾の文字と対応する text の文字を合わせて、再び末尾から比較をしていく。」という決まりをあらかじめ pattern に対して設定しておけばよい。つまりこの pattern（ABBAA）の場合、A と B の２つの文字で構成されている。「もし照合があわなかったときの、pattern の末尾と対応する text の文字が A のときは、pattern４文字目の A と text の A を合わせること（1 文字分ずらすこと)」「照合があわなかったときの、pattern の末尾と対応する text の文字が B のときは、pattern３文字目の B と text の B を合わせること（２文字分ずらすこと)」「照合が合わなかったときの text の文字が A でも B でもなかったら、５文字分（pattern の長さ分）だけ pattern をずらすこと」という決まりを設定しておけばよい。この決まりを skip[A]=1、skip[B]=2、skip[A,B 以外の文字]=5 と定義する。プログラムの中では文字コードを用いるので、0～255 で表される半角の全ての文字に対して、配列 skip[…]を設定する。BM 法の例を図 13 に示す。

文字列探索の対象である配列 text はあらかじめファイルに格納しておき、用意した pattern に対して、１つずつ照合を行っていき、text の末端部まで探索し終わったら、次の pattern を設定して、また探索を始める。pattern が text の何文字目に存在したかを、配列 result に格納する。

BM 法も KMP 法と同じく、text は使用するスレッド数と同数のブロックに分割され、各スレッドは自分の文字列範囲の中で照合を行なっていく。隣接した２つのブロックの境界をはさんで pattern が存在する場合があるので、境界付近の一定領域は重複して２つのスレッドに割り振られる。pattern の情報は全スレッドに割り振られ、それぞれがその pattern の情報を解析して、text と照合していく。

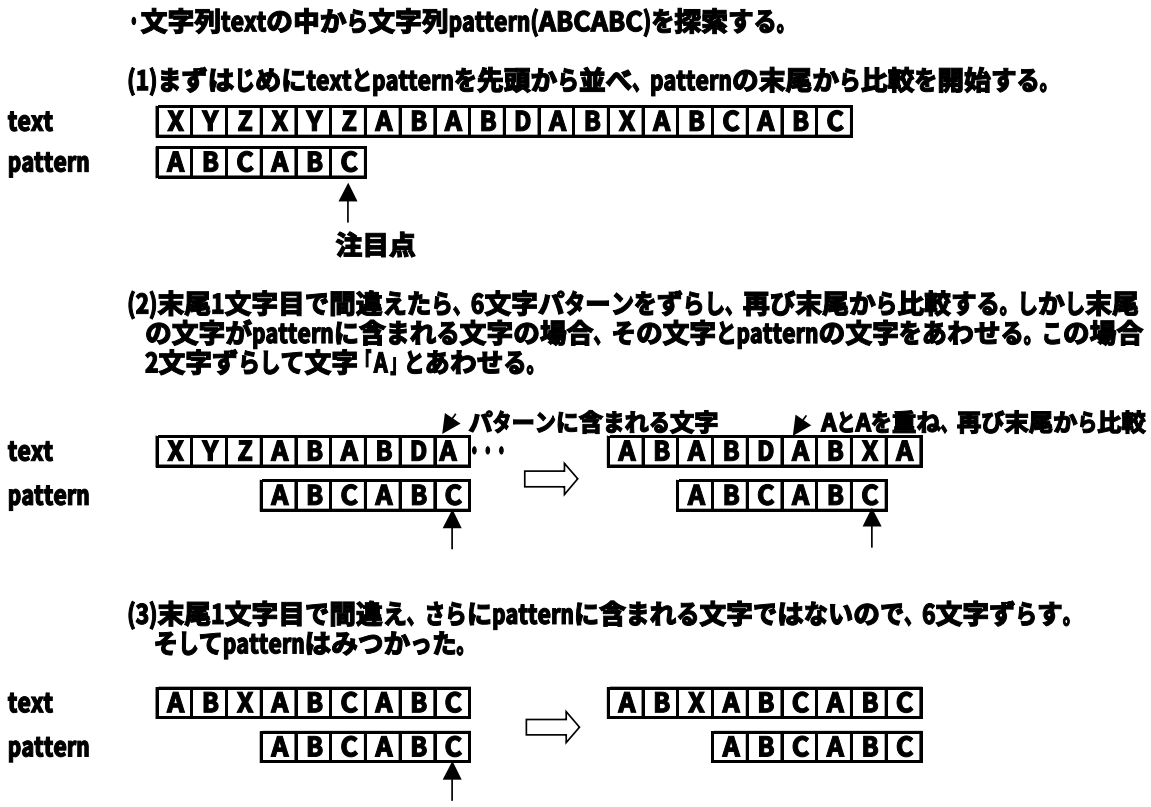



図 13：BM 法の例

### 4.3 実行結果

プログラムでは KMP 法、BM 法ともに text の文字数を 100 万文字、200 万文字、300 万文字、pattern を 30 個に設定して、実行した。KMP 法と BM 法の実行時間をそれぞれ表 4、表 5 に示し、それぞれの速度向上比を図 14、図 15 に示す。

表 4：KMP 法の実行時間 (単位は秒)

| スレッド数 | 1 | 2 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|
| 100 万文字 | 2.99 | 1.58 | 0.85 | 0.50 | 0.39 | 0.33 |
| 200 万文字 | 5.98 | 3.16 | 1.69 | 0.97 | 0.74 | 0.62 |
| 300 万文字 | 8.97 | 4.72 | 2.56 | 1.45 | 1.09 | 0.91 |

表 ５：BM 法の実行時間 (単位は秒)

| スレッド数 | 1 | 2 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|
| 100 万文字 | 1.07 | 0.58 | 0.37 | 0.26 | 0.23 | 0.21 |
| 200 万文字 | 2.15 | 1.24 | 0.74 | 0.49 | 0.43 | 0.38 |
| 300 万文字 | 3.23 | 1.85 | 1.13 | 0.73 | 0.62 | 0.56 |

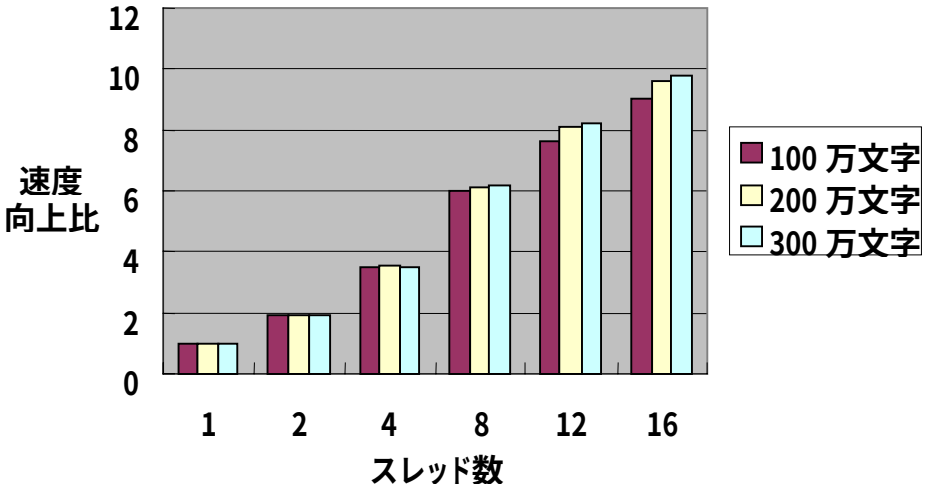



図 14：KMP 法の速度向上比

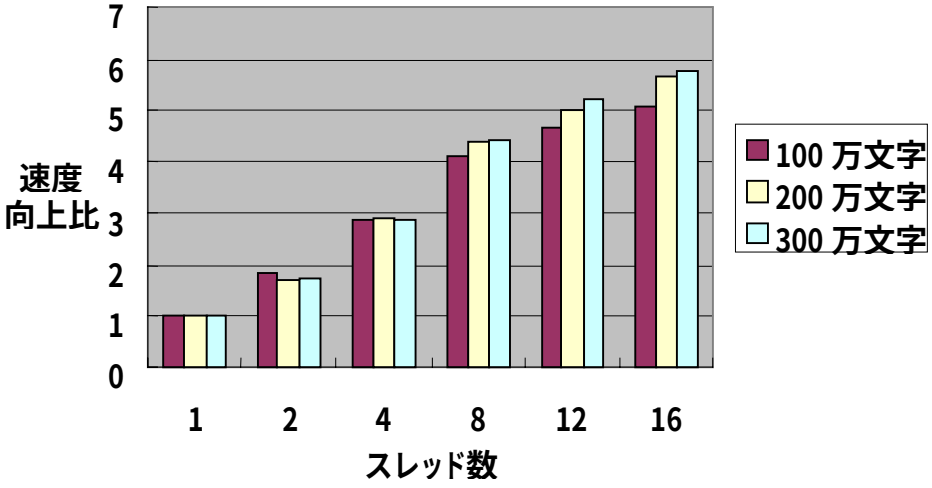



**図 15：BM 法の速度向上比**

### 4.4 考察

KMP 法、BM 法も共に理想的とまではいえないが、並列効果は十分に現れていると思われる。KMP 法の実行時間は BM 法と比べて長かった。この原因として考えられるのはアルゴリズムの違いと考えられる。KMP 法の方が１文字１文字に対する計算量が大きいので、その点から見ても、BM 法より時間がかかってしまうのは理解することができる。しかし速度向上比では KMP 法の方が大きかった。１文字１文字の計算時間が大きい KMP 法の方が並列処理の効果が大きいということがわかった。もしこれ以上大きいデータ量の text で実行することができれば、KMP 法の方が BM 法の計算速度を上回るのかもしれないが、データ量が大きすぎると、SCASH の制約にひっかかってしまうため、その実験をすることはできなかった。KMP 法、BM 法、共に pattern の数や text 文字数が大きければ大きいほど、速度向上比は増加していった。これはよりデータ量の大きい text や多くの pattern を用意すれば、より大きな並列効果が現れることを示していると考えられる。

# 5. ランレングス圧縮

## 5.1 問題定義

圧縮とはファイルのサイズを小さくし、記憶領域などを節約する技術であり、その技術によるプログラムとしては、UNIX の gzip や Windows の lha などがある。ランレングス圧縮とは例えば、AAAAAAFFFFFDDDDDD という文字列があったときに、この文字列を A6F5D6 というように、文字とその文字の連続する数をとの組み合わせに置き換える方法である。しかしこの方法では、例えば、AFD という文字列は A1F1D1 となって、冗長になるので、１文字のときは処理しないことにする。

並列化するにあたって、まず圧縮したい文字列 text を使用するスレッド数で割り、それぞれを各スレッドにブロックとして割り振っていくの。しかしブロックとブロックの境界に連続した文字が存在していた場合、前のスレッドの持つブロックに連続する文字を全てくっつける。各スレッドへの処理範囲の割り振り方を図 16 に示す。

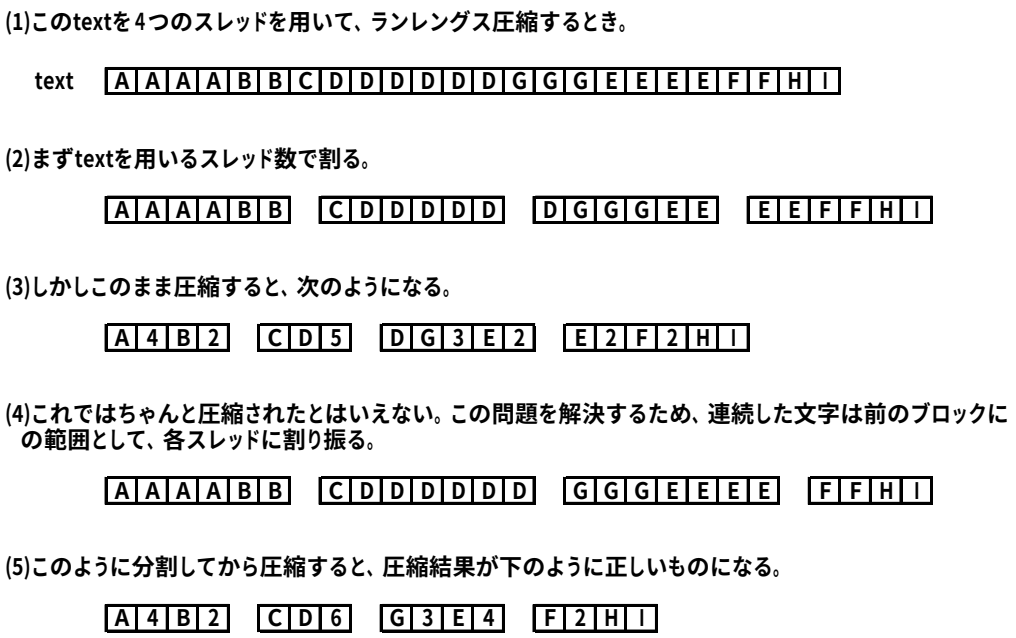



**図 16：ランレングス圧縮の分割手法**

text はあらかじめファイルに格納しておき、各スレッドに private に宣言された配列 parallel_text に割り振られていく。各スレッドは独立に処理し、最後に text に処理後の配列を再び格納する。

### 5.2 実行結果

ランレングス圧縮の実行結果を表 6 に示し、速度向上比を図 17 に示す。実行するときの text の文字数は 50 万文字、100 万文字、150 万文字に設定し、実行した。

**表 6：ランレングス圧縮の実行時間**　（単位は秒）

| スレッド数 | 1 | 2 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|
| 50 万文字 | 0.28 | 0.18 | 0.12 | 0.091 | 0.083 | 0.080 |
| 100 万文字 | 0.39 | 0.25 | 0.17 | 0.13 | 0.114 | 0.110 |
| 150 万文字 | 0.85 | 0.53 | 0.36 | 0.27 | 0.24 | 0.23 |

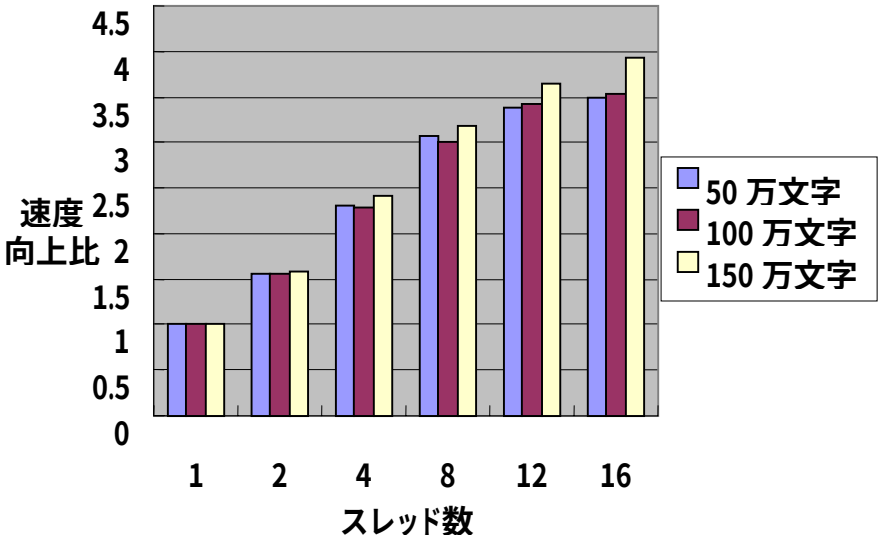



**図 17：ランレングス圧縮の速度向上比**

### 5.3 考察

ランレングス圧縮は思ったほどの速度向上は得られなかったが、文字列照合の KMP 法と BM 法と同様に、text の文字数を増やせば増やすほど、より大きな速度向上が得られた。しかしあまり text の文字数を増やしすぎると、SCASH の制約に引っかかってしまい、実行することができなかった。この問題を解決することができれば、より大きなデータ量の text においても実行することができ、より大きな速度向上が得られたのではないかと考えられる。

# 6. Hough 変換

## 6.1 問題定義

Hough 変換とは２値画像に対して、直線や円の抽出を行う有効な手法である。今回作成したプログラムは、Hough 変換の中でも最も基本的で、利用度が高い直線の抽出を行う、θρ-Hough 変換のプログラムである。

θρ-Hough 変換は原画像を走査して、図 18 に示すｘｙ平面上で図形画素 P1($x_1, y_1$)を検出したとき、その座標($x_1, y_1$)を次式に代入する。P2、P3 に対しても同様の操作を繰り返す。

$$\rho = x*\cos\theta + y*\sin\theta \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (A)$$

この式はθρ平面上では、図 19 のような曲線で表される。P1、P2、P3 の座標を代入して得られた曲線を、それぞれ C1、C2、C3 とする。

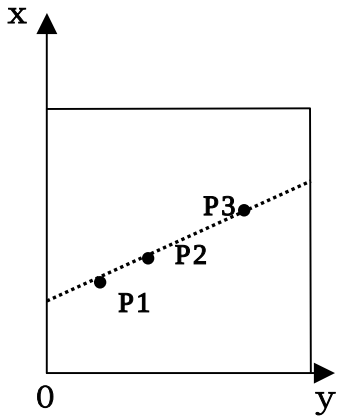



図 18：ｘｙ平面

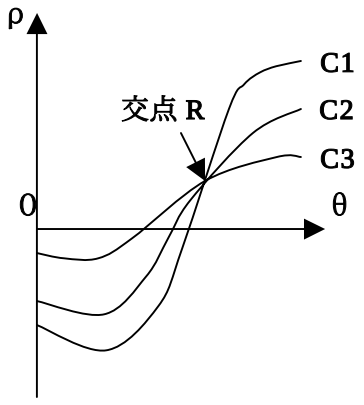



図 19：θρ平面

C1、C2、C3 がθρ平面上で 1 点($\theta_r, \rho_r$)で交わったとき、P1、P2、P3 は直線

$$\theta_r = x*\cos\theta_r + y*\sin\theta_r$$

上に存在するということができる。

つまり、原画像中の全図形画素についてθρ平面上に曲線を描いたあと、θρ平面で曲線が何回も重なっている点を見つけ、それらの点を原画素上の直線に戻すことによって、原画素中の直線を抽出しようというのが、θρ-Hough 変換である。

今回作成したプログラムでは、ｘｙ平面上の 0<=ｘ<10、0<=y<10 の範囲を 1000×1000 のメッシュに区切り、全てのメッシュにおいて０か１の乱数を発生させる。プログラムではこのｘｙ平面を区切ったメッシュを２次元配列 mesh1[x][y]としている。そして mesh1[x][y]の全てのメッシュについて以下の処理を行なう。

- mesh1[x][y]の値が０か１か判断。
- もし 0 ならば何も処理せず次のメッシュを調べる。
- １の場合はそのメッシュの座標（ｘ，ｙ）を（A）式に代入し、求められた曲線が通る座標（θ，ρ）をθ=0～180 をそれぞれθに代入して求める。
- その曲線が通るメッシュである２次元配列 mesh2[θ][ρ]を求め、その値を１増やす（mesh2[θ][ρ]の初期値は 0）。
- mesh1[x][y]の全てのメッシュについて以上の処理を繰り返す。
- 最後に mesh2[θ][ρ]の値が 500 以上のθとρを求め、その値を再び（A）式に代入することによって、xy 平面での直線の式を求める。

これにより、直線の抽出が成されることになる。

並列化手法はｘｙ平面に乱数を発生させるとき、各メッシュ mesh1[x][y]について mesh2[θ][ρ]を求めるとき、mesh2[θ][ρ]>500 のときのθとρを求め（A）式に代入させるときのそれぞれの for ループを自動並列化した。

### 6.2 実行結果

Hough 変換の実行結果を表 7 に示す。実行するときの解像度は 1000×1000 に設定して実行した。実行時間を表 7 に示し、速度向上比を図 20 に示す。

**表 7：Hough 変換の実行時間**　（単位は秒）

| スレッド数 | 1 | 2 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|---|
| 1000×1000 | 8.69 | 4.71 | 2.63 | 1.65 | 1.32 | 1.15 |

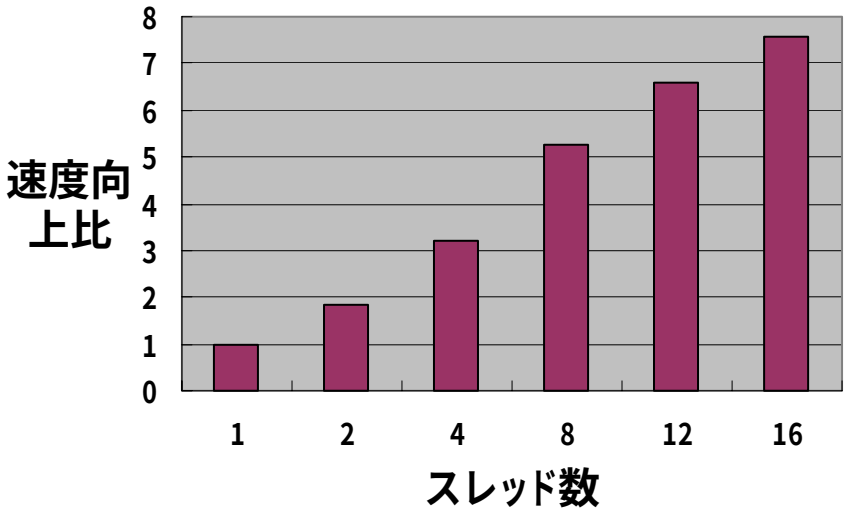



**図 20：Hough 変換の速度向上比**

### 6.3 考察

Hough 変換はスレッド数 16 台のとき 7.5 倍で最高であった。理想的な速度向上とはいえないが、速度向上は十分にあらわれている。この問題を応用して、２値画像に対しての Hough 変換も実行できるようになると思われる。解像度をより大きくして、計算量を増やして実行すれば、より大きな速度向上比が得られるのではないかと考えられる。

## 7. おわりに

本研究室では、京都産業大学の SUN Enterprise 4CPU SMP マシンに続き、本研究室で構築した SCore 型 PC クラスタ上での OpenMP 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、BM 法、KMP 法、ランレングス圧縮、Hough 変換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くことは確実であることがわかった。しかし逆に OpenMP プログラミングをしていくうちに、ソフトウェア分散共有メモリ SCASH に制約があることがわかった。SCASH の制約エラーは現在メモリに関連して起きたり、各スレッドで共有される変数がオーバーフローを起こすことによって起きることがわかっている。そしてそれに対する対応策もだんだんとわかってきた。今後、本研究室でも、より複雑でより大規模な並列プログラミングが行われていくと思われるが、そのときこの SCASH エラーがより深く解明されていたら、スムーズに研究が進んでいくのではないかと思われる。

今後の課題としては、先ほど述べた SCASH の制約エラーの解明、大規模な問題に対する OpenMP 並列プログラミングなどがあげられる。また SCore のバージョンアップと共に OpenMP のコンパイラや SCASH の性能もよくなっていけば、PC クラスタ上での OpenMP 並列プログラミングは今後実用化されていくことと思われる。

# 謝辞

本研究の機会を与えてくださり、様々なアドバイスを頂きました山崎勝弘教授、小柳滋教授に心より感謝いたします。また本研究にあたり、励ましの言葉や貴重な意見を頂きました本研究室の皆様、先輩方に心より感謝いたします。

# 参考文献


[1] 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（bit 別冊）、共立出版、1998.

[2] R.Chandra, L.Dagum, D.Kohr, D.Maydan, J.McDonald and R.Menon：Parallel Programming in OpenMP、Morgan Kaufman Publishers、2000.

[3] RWPC Omni OpenMP コンパイラプロジェクト
http://phase.etl.go.jp/omni/home.ja.html

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

[5] OpenMP C／C++ API 日本語版、RWPC、2000.

[6] 山崎勝弘：事例ベース並列プログラミングの研究平成 9 年度～平成10年度科学研究費補助金基盤研究（C）（2）研究成果報告書、1999.

[7] 林晴比古：改訂新 C 言語入門（ビギナー編、シニア編）、ソフトバンク、1998

[8] 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.

[9]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

[10]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

[11]古川智之、松田浩一、安藤彰一：並列プログラム事例集、1996

[12]米田健治、徳山美香、青地剛宇：並列プログラム事例集 2、1999

[13]柿下裕彰：PC クラスタ上での OpenMP 並列プログラミング[Ⅰ]、2002

## 付録1　マンデルブロ（ブロック分割）mandel_saitekika_org.c

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

main(){
        unsigned long long int count;

        int I,J,K,n,m;
        double d=0.001;                                 /*点と点の距離*/
        double xmin=-2.2, xmax=0.6;
        double ymin=-2.0, ymax=1.5;
        double lx, ly;                                  /*実部と虚部の長さ*/
        double at,bt;

        count=0;

        lx= xmax - xmin;
        n= lx / d;                                      /*実軸方向の点の数*/
        ly= ymax- ymin;
        m= ly / d;                                      /*虚軸方向の点の数*/
        printf("n*m=%d*%d¥n",n,m);
        bt = seconds();

#pragma omp parallel shared(d,n,m)reduction(+:count)
        {
        int I,J,K;
        double x,X,y,Y,A,B;

#pragma omp for                                         /*for ループを自動並列化*/
        for(I=0;I<=n;I++)                               /*マンデルブロの単位計算*/
            {
                A = I*d + xmin;
            for(J=0;J<=m;J++)
```

```
                {
                  B = J*d + ymin;
                  x = 0.0;
                  y = 0.0;
               for(K=0;K<=30;K++)
                   {
                      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++;                              /*収束する点の数を数える*/
                   }

               }
            }
}
        at=seconds();
        printf("syuusyokutennnokazu=%d¥n",count);
        printf("time=%lf¥n",at-bt);
        }
```

## 付録 2　マンデルブロ（サイクリック分割）mandel_cyclic.c

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

main(){
        unsigned long long int count=0;
```

```
        long int N,m,n,i;
        double d=0.01;                                  /*点と点の距離*/
        double xmin=-2.2, xmax=0.6;
        double ymin=-2.0, ymax=1.5;
        double lx, ly;                                  /*実部と虚部の長さ*/
        double at, bt;

        lx= xmax - xmin;
        n= lx / d;                                      /*実軸方向の点の数*/
        ly= ymax - ymin;
        m= ly / d;                                      /*虚軸方向の点の数*/
        printf("n*m = %d*%d¥n",n,m);

        bt=seconds();

#pragma omp parallel shared(n,d,xmin,xmax,ymin,ymax)reduction(+:count)
{
        int I,J,K,L,j,r;
        int NUM, ID;
        double A,B,X,x,Y,y;
        double Start[3000];
        NUM = omp_get_num_threads();
        ID = omp_get_thread_num();

        r=0;
        for(j=ID; j<=n; j+=NUM)                         /*サイクリック分割*/
            {
                Start[r] =j*d;
                r++;
            }

for(J=0;J<=r;J++){                                      /*与えられた計算範囲の*/
          A = xmin + Start[J];                          /*中で各スレッドは計算*/
              for(B=ymin;B<=ymax;B+=d)
                  {
                        x=0.0;
```

```
                    y=0.0;
                for(K=0;K<=30;K++)
                    {
                     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++;}

                }
        }
}
        at=seconds();
        printf("syuusyokutennnokazu=%d¥n",count);
        printf("time=%lf¥n",at-bt);
}
```

## 付録 3　KMP 法　　heiretu_kmp.c

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

#define N 1000000                                   /*text 文字数*/

main(){
        unsigned char text[N+1];
        unsigned char pattern[30][30] =             /*pattern を定義*/
            {
                "ABAAABBB",
```

```
        "ASSABABA",
        "ABASSAABB",
        "ABSSBAB",
        "ABABAASSA",
        "ABAAASAABB",
        "ABAASAAASSSBS",
        "ABBBASAAASAA",
        "AAABBBSASA",
        "ABAAAASSB",
        "ABAAABSBSA",
        "ABBBAAHSHSA",
        "BAAASSSBVB",
        "BAAAASASDS",
        "BAAAWESRW",
        "BAABABASAB",
        "ABBABABAS",
        "ABSSSSBA",
        "ABAAAAASSSBS",
        "ABBBAAAASAA",
        "AAABBBSASA",
        "ABAAAASSB",
        "ABAASSABBA",
        "ABBBAAHHSSSSA",
        "BAAASSSSBVB",
        "BAAASAASDS",
        "BASSAAWERW",
        "BAABSSABAAB",
        "ABBABASSBA",
        "ABSSAASSBA",
    };
static int next[31];
int ac, cc, bc=0;
int pat_len, I, i, j;
int ANS[16][30][200];                    /*pattern の位置を格納*/
long text_len, n;
long count[16][30];
```

```
        double at, bt;
        FILE *KMP;

        if((KMP = fopen("TEXT","r"))==NULL)          /*ファイルから text を読*/
                                                     /*み込む*/
            {
                printf("ERROR file");
                exit(1);
            }
        fgets((char *)text,N+1,KMP);
        fclose(KMP);
        text_len = strlen((char *)text);
        printf("text = %ld ¥n", text_len);

        bt = seconds();
#pragma omp parallel private(i,j)shared(ANS,count,text,pattern,text_len,pat_len,
next)
{
        unsigned char parallel_text[3000001];
        int a, b, c, y, w, z;
        int ID, NUM;
        long ptex_len, x;

        NUM = omp_get_num_threads();
        ID = omp_get_thread_num();

        x = text_len / NUM;
        y = text_len % NUM;
        b = 0;

        if(ID!=NUM-1) w = 20;
        else          w = y;
        for(a=ID*x;a<=(ID+1)*x+w;a++)                /*各スレッドに範囲を*/
            {                                        /*割り振る*/
                parallel_text[b] = text[a];
                b++;
```

```
            }
        ptex_len = strlen((char *)parallel_text);

for(I=0;I<30;I++)
    {
        pat_len = strlen((char *)pattern[I]);          /*pattern の構造を解析*/
         i=1;                                          /*そして pattern の表を*/
         j=0;                                          /*作成する*/
         next[1]=0;
        while(i <= pat_len)
            {
                if(pattern[I][i] == pattern[I][j])
                    {
                        i++;
                        j++;
                        next[i] = j;
                    }
                else if(j == 0)
                    {
                        i++;
                        next[i] = j;
                    }
                 else    j = next[j];
            }

        n = 0;
        i = 0;
        j = 0;
        while(i <= ptex_len)                           /*照合開始!!*/
            {
                if(parallel_text[i] == pattern[I][j])
                    {
                        i++;
                        j++;
                        if(j == pat_len)
                            {
```

```
                                ANS[ID][I][n] = ID*x + (i - j + 1);   /*pattern がみつ*/
                                j = next[j];                    /*かったら ANS にその位*/
                                n++;                            /*置を格納*/
                            }
                        }
                    if(parallel_text[i] != pattern[I][j])
                        {
                            if(j==0) i++;
                            else     j = next[j];
                        }

                }
                    count[ID][I] = n;                           /*pattern の数をカウント*/
            }
}
                    at = seconds();
                    printf("time= %lf¥n",at-bt);
}
```

## 付録4　BM 法　bmh6.c

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

#define N 1000000                                   /*text 文字数*/
double at, bt;

main(){
        unsigned char text[N+1];
        unsigned char pattern[30][30] =             /*pattern を設定*/
        {
                "ABAAABBB",
                "ASSABABA",
```

```
        "ABASSAABB",
        "ABSSBAB",
        "ABABAASSA",
        "ABAAASAABB",
        "ABAASAAASSSBS",
        "ABBBASAAASAA",
        "AAABBBSASA",
        "ABAAAASSB",
        "ABAAABSBSA",
        "ABBBAAHSHSA",
        "BAAASSSBVB",
        "BAAAASASDS",
        "BAAAWESRW",
        "BAABABASAB",
        "ABBABABAS",
        "ABSSSSBA",
        "ABAAAAASSSBS",
        "ABBBAAAASAA",
        "AAABBBSASA",
        "ABAAAASSB",
        "ABAASSABBA",
        "ABBBAAHHSSSSA",
        "BAAASSSSBVB",
        "BAAASAASDS",
        "BASSAAWERW",
        "BAABSSABAAB",
        "ABBABASSBA",
        "ABSSAASSBA",
};
int skip[256];
int pat_len;
int I, K, l, y, z, ac, bc=0;
int ANS[16][20];
long result[16][30][200];
long text_len, A, M, x;
FILE *BM;
```

```
        if((BM = fopen("text","r"))==NULL)          /*ファイルから text を読み込む*/
            {
                printf("ERROR file");
                exit(1);
            }
        fgets((char *)text,N+1,BM);
        fclose(BM);

        text_len = strlen((char *)text);
        printf("text = %ld¥n", text_len);

        bt = seconds();
#pragma omp parallel private(skip)shared(x, y, z, text, text_len, pattern, pat_l
en, I, M, ANS, ac, bc)
{
        char c, tail;
        int i, j, w, r=0, a;
        int NUM, ID;
        int k, J;

        NUM = omp_get_num_threads();
        ID = omp_get_thread_num();
        x = text_len / NUM;
        y = text_len % NUM;

        for(I=0;I<30;I++)
            {
                J=0;
                pat_len = strlen((char *)pattern[I]);          /*pattern を解析*/
                tail = pattern[I][pat_len-1];

                for(i=0;i<=255;i++)                            /*そして表を作成*/
                    skip[i]=pat_len;
                for(i=0;i<pat_len-1;i++)
                    skip[pattern[I][i]] = pat_len-1-i;
```

```
                if(ID != NUM-1) w = pat_len - 2;
                else            w = y;

                for(i=ID*x+pat_len-1;i<(ID+1)*x+w;i+=skip[c])      /*探索開始*/
                 {
                    c = text[i];
                    if(c==tail)
                     {
                        k = i;
                        j = pat_len - 1;
                        while(pattern[I][--j] == text[--k])
                         {
                            if(j==0)
                             {
                                result[ID][I][J] = k + 1 ;   /*pattern の位置を格納*/
                                J++;
                             }
                         }
                     }
                 }
                ANS[ID][I]=J;                                /*pattern の数を格納*/
        }
}
                printf("%d", result[0][0][1]);
                at = seconds();
                printf("time=%lf¥n", at-bt);
}
```

## 付録5　ランレングス圧縮　parallel_pointer2.c

```
#include<stdio.h>
#include<string.h>
#include<time.h>
#include<stdlib.h>
#include<omp.h>
```

```
#include"second.c"

#define N 1000000                                   /*text 文字数*/

main(){
        char text[N+1];
        char RESULT[N+1];
        int Q[16];
        int a=0,p,q;
        int ac,bc,R,M,J,K;
        int ANS;
        long text_len, ptex_len, parallel_len;
        FILE *RUN, *OUTPUT;
        double at, bt;

        if((RUN = fopen("TEXT","r")) == NULL)     /*text をファイルから読み込む*/
            {
                printf("ERROR!!!¥n");
                exit(1);
            }
        if((OUTPUT = fopen("OUTPUT","w")) == NULL)     /*出力用ファイル*/
            {
                printf("ERROR!!!¥n");
                exit(1);
            }
        fgets(text,N+1,RUN);
        text_len = strlen(text);
        printf("text = %d moji¥n", text_len);

bt = seconds();
#pragma omp parallel shared(RESULT,text,text_len,Q,p,q,a)reduction(+:ptex_len)re
duction(+:parallel_len)
{
        int i, cc, j, k, n, m, r;
        int I, y, w;
        int NUM, ID;
```

```
        int Start, End;
        int save_number[750001];
        char save_word[750001];
        char Z[5];
        char parallel_text[1000001];
        char *tx, *p1, *p2, *p3;
        long Z_len;
        long x;

        NUM = omp_get_num_threads();
        ID = omp_get_thread_num();

        M = NUM;
        x = text_len / NUM;
        y = text_len % NUM;
        r = 0;

        for(p=1;p<NUM;p++)                          /*ブロック境界に文字が連続しな*/
{                                                   /*いように調整*/
                q=0;
                while(text[x*p-1] == text[x*p+q])
                        q++;
                Q[a] = q;
                a++;
            }
//for(k=0;k<NUM;k++)
//printf("%d ",Q[k]);
        if(ID != NUM-1)                             /*計算範囲を割り振る*/
            {
                Start = ID*x + Q[ID-1];
                End = (ID+1)*x + Q[ID];
            }
        else if(ID != 0)
            {
                Start = ID*x + Q[ID-1];
                End = (ID+1)*x + y;
```

```
    }
if(ID == 0)
    {
        Start = 0;
        End = (ID+1)*x + Q[ID];
    }

for(J=Start;J<End;J++)
    {
        parallel_text[r] = text[J];
        r++;
    }

parallel_len += strlen(parallel_text);
printf("ID = %d,ptex = %d    ",ID, parallel_len);
tx = parallel_text;
p1 = tx;

i=0;j=0;k=1;cc=0;                                   /*圧縮開始*/
while(i < parallel_len)
    {
        if(*p1 == *(p1+1))
            {
                p1++;k++;
            }
        else
            {
                save_word[cc] = *p1;
                save_number[cc] = k;
                p1++;cc++;k=1;
            }
        i++;
    }
n=J=0;p1=tx;
while(save_word[n] != '¥0')
    {
```

```
                *p1 = save_word[n];
                p1++;
                if(save_number[n]==1)
                    ;
                else
                    {
                        sprintf(Z,"%d",save_number[n]);
                        Z_len = strlen(Z);
                        *p1 = Z[0];p1++;
                        if(Z_len == 2)
                        {*p1 = Z[1];p1++;}
                    }
                n++;
            }
        *p1 = '¥0';
        ptex_len += strlen(tx);
        printf(" assyukugo ha %d ¥n",ptex_len);
}
        printf("---------------------------------hajime ha %d¥n",parallel_len);
        printf("---------------------------------assyukugo ha %d¥n", ptex_len);
        at = seconds();
        printf("---------------------------------time = %lf¥n", at-bt);
}
```

## 付録6 Hough 変換 omp_hutuu.c

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

#define rad(I) ((I)*3.14159/180.0)
```

```
main(){
        int x, X, y, Y, RHO, theta;
        int mesh1[1001][1001], mesh2[180][2001];
        double rho;
        double x1, y1, a, b;
        double xmax=10.0, xmin=0.0;
        double ymax=10.0, ymin=0.0;
        double at, bt;
        double D = 0.01;

        X = (xmax - xmin) / D;
        Y = (ymax - ymin) / D;



#pragma omp parallel for                        /*xy 平面の初期化*/
        for(x=0;x<X;x++){
            for(y=0;y<Y;y++){
                mesh1[x][y]=0;
            }
        }
#pragma omp parallel for private(x,y)           /*乱数発生*/
        for(x=0;x<X;x+=10){
            for(y=0;y<Y;y++){
                mesh1[x][y] = rand()%2;
            }
        }

#pragma omp parallel for                        /* θ ρ平面の初期化*/
        for(x=0;x<180;x++){
            for(y=0;y<2000;y++){
                mesh2[x][y] = 0;
            }
        }

bt = seconds();                                 /*Hough 変換開始*/
```

```
#pragma omp parallel for private(x,y,x1,y1,theta,rho,RHO)shared(mesh1,mesh2,D)
        for(x=0;x<X;x++){
            for(y=0;y<Y;y++){
                if(mesh1[x][y] == 1){
                    x1 = x*D;
                    y1 = y*D;
                for(theta=0;theta<=180;theta++){
                    rho = x1*cos(rad(theta)) + y1*sin(rad(theta));
                    RHO = rho/D;

                    mesh2[theta][RHO]++;
                }
                }
            }
        }

at = seconds();
#pragma omp parallel for private(x,y,rho,a,b)        /*500 以上の点を求め直線の式を*/
        for(x=0;x<180;x++){                          /*を抽出*/
            for(y=0;y<2000;y++){
                if(mesh2[x][y] >500){
                rho = y*D;
                a = -(cos(rad(x)) / sin(rad(x)));
                b = -( rho / sin(rad(x)));
                    printf("y =%2.2lfX +%2.2lf¥n",a,b);

                }
            }
        }

        printf("TIME = %lf¥n",at-bt);
}
```