卒業研究報告書

題目

# MPI による並列画像処理

指 導 教 員 石水　隆　助手

報告者

03-1-47-503

赤松啓介

近畿大学理工学部情報学科

平成 19 年 2 月 5 日提出

# 概要

情報処理の高度化かつ多様化に従い様々な分野で高速化が追求されている。高速化を行うためには複数の CPU に仕事を分散させて並列処理を行う並列計算機が必要とされている。しかし、一般的に、並列計算機は非常に高価であり、必ずしも必要な性能の並列計算機を使用できるとは限らない。そのため、複数の計算機をネットワーク接続することにより、仮想的に並列計算機を構築するクラスタ(Cluster)処理が注目されている。クラスタ処理は費用、機能拡張といった点で優れており、低コストで大規模な問題を解くことができるため、現在並列処理の主流となっており、ネットワークに接続された複数の各計算機はネットワークに通じて互いに通信しながら並列に処理を行うことがある。また、高速化が求められている分野の 1 つに画像処理がある。画像はデータが膨大であるため、画像の並列処理には高い関心が集められている。

クラスタ処理を行うソフトウェアの代表的なものに MPI(Message Passing Interface)[1] と、PVM(parallel Virtual Machine)[1]がある。本研究では、その内の一つである MPI を用いて、画素を対象とした並列画像処理であるエッジ抽出の高速化を目的とする.

目次

# 1 序論

## 1.1 本研究の背景

### 1.1.1 並列計算機

膨大な量や、複雑な計算には並列計算(Parallel Computing)が用いられている。並列計算とは一つの計算をいくつかに分割し、複数の CPU を用いて並列に処理を行うことを言い、計算処理の高速化を目的とするものである。複数のプロセッサを持ち並列計算を行うことができる計算機を並列計算機(Parallel Computer)と呼ぶ。複雑かつ膨大な計算を高速に処理することは計算機の誕生以来強く求められており、並列計算機をその要求に応えるものである。しかし並列計算機は高価であり、また計算機自体の性能拡張も容易でないため、利用しやすいものではない。そこで現在クラスタ(Cluster)処理が利用されることが多い。

### 1.1.2 クラスタ処理

クラスタ処理とはネットワークで接続された複数の計算機の集合を、一台の仮想の並列計算機として用いるものである。クラスタは複数の計算機とそれを結ぶネットワークからなり、低コストで大規模な問題を解くことができるため、現在における並列処理の主流となっている。

ネットワークで接続された各計算機は、多くの場合メッセージ・パッシグ(Message Passing)方式によってお互いに通信しながら並列に処理を行う。メッセージ・パッシング方式とはプロセス間の通信をお互いのデータ送受信にて行う方式である。

並列計算機(Shared Memory Parallel Computer)は、全ての CPU が共通のメモリに対して読み書きを行い、メモリを通じて通信を行う共有メモリ型計算機と、それぞれの CPU がメモリを持ち、メッセージ・パッシング方式によりネットワークを通して通信を行う分散メモリ型並列計算機(Distributed Memory Parallel Computer)に分類され、クラスタは後者に属する。メモリを計算機が排他的に管理するため、他のクラスタ上のデータを参照するためには明示的にデータのやり取りをしなければならず、クラスタ処理を使用する際には、計算に必要となるデータがどの計算機によって保持されるか、どのタイミングでデータが必要となるかを考慮し、送受信を効率良く行えるようにアルゴリズムを設計しなければならない。しかし、クラスタ処理ではデータの送受信において、計算機間でクラスタ処理を行う同期を取る必要が無く、タイミングに依存するバグの発生は少ない。

クラスタ処理を行うためのソフトとして MPI(Message Passing Interface)[1]と PVM(Parallel Virtual Machine)[1]などがある。現在 MPI、PVM により重要な科学・産業・医学上の問題が解かれており、クラスタ処理における事実上の標準となっている。

### 1.1.3 MPI (Message Passing Interface)

MPI(Message Passing Interface)[1]は、並列計算におけるメッセージ（データ）の高速な伝送と、PVM と同等の使いやすさを目的として作成さえたシステムで、MPI フォーラムという団体により 1994 年に考案されたものである。

MPI とはソフトウェアでなく MPI フォーラムが作った MPI 標準というインターフェイスの規格である。この MPI 標準に準じて実装されたライブラリを、MPI ライブラリという。特定の機能を持ったプログラムで、他のプログラムの一部として動く MPI ライブラリの種類はいくつもあり、無料のものもある。

計算機間で通信を行う場合、計算機のアーキテクチャ（構造）によってプロトコル（ネットワーク上の約束）が異なり、通信方法が違うため、計算機ごとにプログラムを書き分けなければならない。しかし、MPI ライブラリにはさまざまな通信関数が実装されており、MPI 規約に従ったプログラムを用いることで通信を代行してくれるため、並列処理を行うユーザーは並列アルゴリズムに集中することができる。

また、MPI はライブラリと、MPI を使用したプログラムのコンパイラと、起動スクリプトから成る。

MPI において並列処理を行う場合、UNIX や Windows などほとんどの OS・機種に対応している。しか

し、高速化に重点をおいているため機種、OS を統一する必要がある。また実行時の計算機の台数は基本的に固定されており、耐故障性(Fault Tolerant)は低くなっている。

#### 1.1.4 PVM (parallel Virtual Machine)

MPI(Message Passing Interface)[1]はインターフェイスの規格であるのに対して、PVM(parallel Virtual Machine)[1]は実装パッケージそのものである。PVM は、TCP/IP ネットワークで接続された何台もの計算機を仮想的に 1 台の並列計算機ととらえて、並列プログラムを走らせることのできるメッセージ・パッシングの環境とライブラリを提供する。その歴史は、1991 年にオークリッジ国立研究所とテネシー大学で開発が始まり、PVM プロジェクトとして発展してきた。PVM プロジェクトはその実験的性質から、副産物として科学者のコミュニティあるいはその他の分野の研究者に役立つようなソフトウェアをこれまで作り出している。元々は、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あったが、現在では多くの並列計算機にも移植されている。

しかしながら、PVM では各並列計算機ベンダが独立にチューニングを施した独自の PVM を開発しており、移植性に乏しいという欠点がある。これは、MPI フォーラムのような第三者的なしっかりとした決定機関を保有していないことを起因している。

#### 1.1.5 MPI と PVM の比較

PVM は、ワークステーションをクラスタすることで再利用するという世界で育ってきたものである。そのため不均質ないくつもの計算機やオペレーティング・システムを直接的に管理する。そして動的に仮想計算機を形成することができる。PVM のプロセス管理の機能では、アプリケーションの中から動的にプロセスを生成したり、停止したりすることができる。この機能は MPI-1 にはなく、MPI-2 に採り入れられている。さらに利用可能なノードのグループ管理する機能では、ノード数を動的に増減したり、調べたりすることができる。これは、MPI ではまだ採り入れられていない。PVM の利用における最大の問題点は、先ほどにも述べた移植性である。

それに対して MPI はその実装が MPP(Massively Parallel Processors)[1]やほとんど同一な特性のワークステーション・クラスタを対象としている。MPI は PVM の後に設計されたために、明らかに PVM における問題点を学んでいる。つまり MPI の方が PVM に比べて、高レベルのバッファ操作が可能であり、高速にメッセージを受け渡す事ができる。そして、オープンなフォーラムによって達成された「標準」であるために、MPI で作成されたプログラムは非常に移植性が高い。ただし、MPI の実装は数多くあるが、MPI-2 に完全にサポートしたものはまだない。また、Web や書籍などでの情報も MPI の方が入手しやすく便利であるといえる。

#### 1.1.6 MPICH

MPICH[1]は、アメリカのゴードン国立研究所(Argonne National Laboratory)が模範実装として開発し、無償でソースコードを配布したライブラリである。移植しやすさを重視したつくりになっているため盛んに移植が行われ、世界中のほとんどのベンダの並列マシン上で利用することができる。特に、MPICH では UNIX 計に限らず Windows 計へのサポートも充実している。さらに、SMP、Myrinet などのハード面にも対応している上に、DQS、Globus といった様々なツールを使用できることも大きな特徴の一つである。また、MPICH1.2.0 では、MPI-1.2 における全ての機能をカバーしており、MPI-2 に関しても幾つかの機能についてサポートしている。

### 1.2 本研究の目的

本研究では、クラスタ処理を行うソフトウェアの 1 つである MPI を用いて、画像処理の高速化を図る。また、画像処理の一分野であるエッジ処理を対象とし、MPI を用いてエッジ処理を並列化することにより効率

良く高速化を行う。

### 1.2.1 エッジ抽出

エッジ(Edge)とは、物体の外縁を表す線、または画像を特徴づける線要素である。画像処理においてエッジを抽出することを「エッジ抽出」と呼ぶ。

画像の中の物体と物体、あるいは物体の背景の境目はエッジであるので、「画像の濃度や色に急激な変化があるところ」がエッジとなる。つまり、エッジ抽出には濃度にだけ注目すればよい。

エッジは濃度値が急激に変化する部分にあるので、関数の変化分を取り出す微分演算がエッジ抽出に利用できる。微分には 1 次微分（グラディエント）と 2 次微分（ラプラシアン）があり、ともにエッジ抽出に利用される。

ディジタル画像ではデータが一定間隔にとびとび並んでいるため、本当の意味での微分演算はできない。このため、隣接画像同士の差をとる差分と呼ばれる演算で微分を近似する。微分演算を行うための隣接画像同士の演算を表現する係数の組を「微分オペレータ」と呼び、いろいろな種類がある。

### 1.2.2PGM

PGM とはグレースケールの画像データを扱うためのフォーマットである。PG にはアスキーとバイナリの 2 種類のフォーマットがある、ヘッダのマジックナンバーが P2 の時はアスキー、P5 のときはバイナリになる。マジックナンバーに続いて 10 進数で幅、高さ、グレースケールの最大値が記述される。

## 1.3 本報告書の構成

本報告書の構成を以下に述べる。2 節に本研究での環境と、並列化の方法を述べる。3 節に計測結果を記し、問題点等を考察した。4 節は本研究の結論と今後の課題を述べる。

## 2 研究内容

2.1 計算機環境

本研究では、MPI として MPICH2[1]を用いる。表 1 に、本研究で使用する計算機のスペックを示す。本研究を始めるにあたり、MPI のインストールに関しては、「MPI による並列プログラミングの基礎」[1]を参考とした。

表 1：本研究で使用する計算機のスペック

| | PC1 | PC2 | PC3 | PC4 |
|---|---|---|---|---|
| OS | Microsoft Windows 2000 | Microsoft Windows XP | Microsoft Windows 2000 | Microsoft Windows 2000 |
| CPU | Intel Celeron 768MHZ | Intel Pentium 1.9GHZ | Intel Pentium 1.9GHZ | Intel Celeron 768MHZ |
| Memory | 512MB | 670MB | 670MB | 512MB |

2.2 エッジ抽出アルゴリズム

本研究では表 1 に示す 4 台の計算機を用いて画素を対象とした並列画像処理を行う。近傍データに対する局所的な積和演算処理の代表として、画像からエッジを抽出する処理がある。本研究では、その中でも一次微分としてよく利用されるソーベル(Sobel)オペレータのプログラムの並列化を行う。
エッジ抽出アルゴリズムは以下の式を計算することがアルゴリズムの中心になる。

$$f(i,j) = \Sigma f(i+m,j+n)*W(m,n)$$

ただし、f(i,j)は n*n サイズの画像、W(m,n)は m*m サイズのマスクである。また、m は 3~11 程度の奇数値であり、本研究では m=3 としている。
以下に逐次積和演算アルゴリズムを示す。

**逐次積和演算アルゴリズム**

```
for i= 1 to n do
        for j = 1 to n do
                g(i,j) = 0
                for m =1 to m do
                        for n = 1 to m do
                                g(i,j) = g(i,j) + f(i + m,j +n)*W(m,n)
                        end
                end
        end
end
```

このアルゴリズムは、O(1)個の CPU を用いて $O(n^2m^2)$時間で積和演算を行える。これは最も効率のよい逐次アルゴリズムである。以上の事をふまえて並列化の方法について述べていく。

2.3 並列エッジ抽出アルゴリズム

逐次積和演算アルゴリズムには 4 つのループが存在する。最初の 2 つのループ(制御変数 i,j)は画素を走査するためのものであり、内側の 2 つのループ(m,n)は各画素に対して局所的な積和演算を行うものである。
本研究では最初の 2 つのループを範囲指定することにより、処理の対象画像を複数の部分画像に分割し、各部分画像の処理を CPU に割り当てることにより並列化を行う。

以下に p 台の CPU を用いた並列積和演算アルゴリズムを示す。ただし、各 CPU には 0~p-1 の識別番号が割り当てられているとし、r(0 ≤ r < p)と表記する。

**並列積和演算アルゴリズム**

(1) サイズ n*n 画像 I(i,j)(0 ≤ i, j < n)をサイズ n*(n/p)の部分画像 Ir = (i,j)(0 ≤ i < n, nr/p ≤ j < n(r+1)/p)に分割する。

(2) 各 CPU(0~p-1)において、逐次積和アルゴリズムを用いて部分画像 Ir のエッジ抽出を行う。

上記の並列積和演算アルゴリズムを、MPICH2 を用いてプログラムし、画像データの処理計算にかかる時間(画像の読み込みと処理後の保存を除く時間)を CPU 数別に計測を行う。計測に必要な画像として、容量約 30kb～8MB で大きさがばらばらの PGM 画像(バイナリ)を数枚用意した。実行する際には、各 PC に実行形式ファイルと処理する原画像が必要である。本研究では共有フォルダを作り、そこに入れることにより作業の効率性を図った。付録 1 に本研究で用いた画像を示す。また、付録 2 に本研究で作成した MPICH2 によるエッジ抽出プログラムを示す。

## 3　結果・考察

表 2 に本研究で作成したエッジ抽出プログラムによる画像ごとのエッジ抽出にかかる処理時間を示す。なお、処理時間は 5 回計測した平均値である。また、図 1 に、画像ごとのエッジ抽出にかかる処理時間と CPU 数の関係を示す。CPU 数を 1 から 2 に増やした時は、実行時間が大きく変わったが、(本研究では約 2/3 の短縮)、3~4 と増やしていった時は、大きな変化は見られず、実行時間は CPU の数を増やすにつれて収束していった。

また、CPU 数が 4 で 30kb 程度の容量が軽い画像を処理した場合では、PC1 と PC4 の処理時間が約 0.0345 秒かかり、1 台の時よりも遅くなることがしばしば起きた。これは通信・同期にかかる時間が全体の処理時間の大きな割合を占めていたからだと考えられる。

画像の容量や計算機のスペックによって、処理速度や通信・同期にかかる時間がかわってくるが、処理範囲を分割して画像を並列処理するのは、計測結果を見る限り、CPU 数 2、多くて 3 が実用的だと思われる。

本研究ではソーベルオペレータのプログラムを並列化したが、画像を分割することにより並列化しているので、2 次微分オペレータの(Laplacian)でも同様に並列化できる。そして、画像のフォーマットを PGM で行ったが、画素を対象とした並列画像処理なので、BMP などの他のフォーマットでも無理なく並列化できる。

表 2：エッジ抽出にかかる処理時間(秒)

| 画像容量 | 0.0293M | 0.692M | 3M | 6M | 7.91M |
|---|---|---|---|---|---|
| CPU数1 | 0.0380162 | 0.790179 | 3.373799 | 6.850611 | 8.996721 |
| CPU数2 | 0.0278698 | 0.528397 | 2.331417 | 4.722933 | 6.265224 |
| CPU数3 | 0.0257642 | 0.487474 | 2.167782 | 4.43409 | 5.711888 |
| CPU数4 | 0.0217564 | 0.449219 | 2.035436 | 4.106444 | 5.368646 |

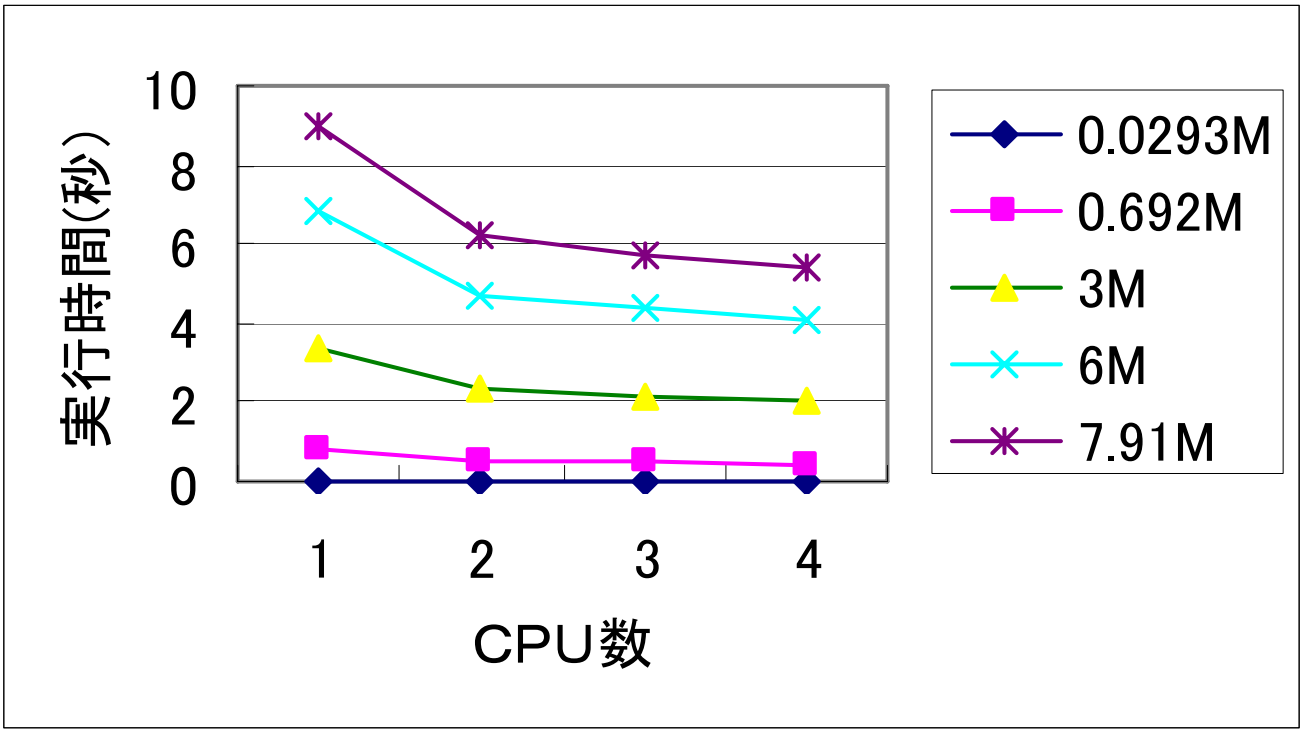


図 1：処理時間と CPU 数の関係

## 4　結論・今後の課題

本研究では、MPI を用いて画像処理の 1 つであるエッジ抽出を行い、その時間を計測した。サイズの大きな画像の場合は MPI を用いて並列化を行うことにより、時間の短縮が得られる。一方、サイズが小さい画像の場合は、1 台の時より遅くなることがあり、MPI の使用が必ずしも効率的だとは言えない。従って、MPI を使用する際には、使用できる CPU、ネットワークの性能やその他の計算機環境に応じてプログラミングする必要がある。今後の課題としては、サイズに応じて使用する CPU 数を変える、また各 CPU のスペックに応じて割り当てられる部分画像サイズを変える等により、より高速な並列化を図ることが考えられる。

## 5　謝辞

本研究を進めるにあたり、石水助手及び情報論理工学研究室の皆様には様々な助言や御指導をいただき、お世話になりました。深く感謝申し上げます。

## 参考文献・サイト

[1] 渡邉真也, MPI による並列プログラミングの基礎
http://mikilab.doshisha.ac.jp/dia/smpp/cluster2000/PDF/chapter02.pdf
[2] 美濃道彦, 並列画像処理, コロナ社, 1999
[3] 安居院猛,長尾智晴, C 言語による画像処理プログラミング入門書, 昭晃堂, 2000

## 付録

・付録１

以下に本研究で用いた画像を示す。

・付録２

以下に本研究で作成した MPICH2 によるエッジ抽出プログラムを示す。

```
#include <stdio.h>
#include "mpi.h"

#include<stdlib.h>
#include<limits.h>
#include"mypgm.h"

void spacial_filtering()
/* 画像の空間フィルタリングを行う.                          */
/* Sobelフィルタ （水平方向微分）                           */
/* 原画像 image1[y][x] ===> 変換後の画像 image2[y][x]    */
{
    /* 適用するフィルタの定義 (Sobel水平方向微分) */
    int weight[3][3] = {
        { -1,  0,  1 },
        { -2,  0,  2 },
        { -1,  0,  1 }  };
    double div_const = 1.0; /* <== 最後に割る値 */
    double pixel_value,s; /* 計算値 */
    double min, max; /* 計算値の最小値, 最大値 */
    int x, y, i, j, xa, xb, ya, yb,rank,procs;  /* ループ変数 */
        MPI_Comm mpi_comm;
        mpi_comm = MPI_COMM_WORLD;
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&procs);

    /* フィルタリング後の階調値の最小値, 最大値を求める */
    printf("原画像のフィルタリングをしています. ¥n");
    min = (double)INT_MAX;
    max = (double)INT_MIN;

        xa=rank*(int)(x_size1/procs);
        xb=(rank+1)*(int)(x_size1/procs);
        ya=rank*(int)(y_size1/procs);
        yb=(rank+1)*(int)(y_size1/procs);

    for ( y = ya; y < yb-1 ; y ++ ){
```

```
        for ( x = xa; x < xb-1 ; x ++ ){
            s= 0.0;
            for ( i = - 1; i < 2; i ++  )
                for ( j = -1; j < 2; j ++ )

s = s + weight[i + 1][j + 1] * image1[y + i][x + j];
            s = s / div_const;
            if ( s < min ) min = s;
            if ( s > max ) max = s;

        }
    }
MPI_Reduce(&s,&pixel_value,1,MPI_DOUBLE,MPI_SUM,0,mpi_comm);

    if ( (int)(max - min) == 0 ) exit(1);
    /* image2[y][x] の初期化(外郭の画素のため) */
    x_size2 = x_size1;  y_size2 = y_size1;
    for ( y = 0; y < y_size2; y ++ )
        for ( x = 0; x < x_size2; x ++ )
            image2[y][x] = 0;
    /* 処理後の値を線形変換してから image2 に代入 */

    for ( y = 1; y <= y_size1-1 ; y ++ ){
        for ( x = 1; x <= x_size1-1 ; x ++ ){
            pixel_value = 0.0;
            for ( i = -1; i < 2; i ++  )
                for ( j = -1; j < 2; j ++ )
                    pixel_value = pixel_value + weight[i + 1][j + 1] * image1[y + i][x + j];

            pixel_value = pixel_value / div_const;
            pixel_value = MAX_BRIGHTNESS /
                        ( max - min ) * ( pixel_value - min );
            image2[y][x] = (unsigned char)pixel_value;
        }
    }
}


int main(int argc,char *argv[])
```

```
{
        int tag=1000,myid,numprocs;
        double s_time,e_time;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

        if(myid==0){
        fprintf(stderr,"I am rank  %d of size %d¥n", myid, numprocs);

        load_image_data( );   /* 画像を読み込んで image1 へ */
        }
        MPI_Barrier(MPI_COMM_WORLD); //←測定前に同期をとる
        s_time = MPI_Wtime();         //この部分から時間計測
    //↓測定プログラム
        spacial_filtering( ); /* 空間フィルタリングして image2 へ */
        //測定プログラム

        MPI_Barrier(MPI_COMM_WORLD);  //足並みをそろえる
        e_time = MPI_Wtime();          //時間計測終了

        if(myid==0){
    save_image_data( );   /* image2 を保存する */
        }
        printf("処理time = %f ¥n",e_time-s_time);
        MPI_Finalize();
        return 0;
}
```