**华中科技大学计算机科学与技术学院**

**课程实验报告**

课程名称： 计算机科学与技术

专 业： 计算机科学与技术

班 级： 计算机ACM2014班

报 告 人： 李昊洋

学 号： U201414713

指导教师： 金海、陆枫、邵志远

**完成日期：** 2017**年 7 月 10日**

目录

[Lab1熟悉并行编程环境 4](#_Toc488742042)

[一、 实验目的与要求 4](#_Toc488742043)

[二、 实验内容 4](#_Toc488742044)

[三、 实验结果 4](#_Toc488742045)

[Lab2 pthread 7](#_Toc488742046)

[一、 实验目的与要求 7](#_Toc488742047)

[二、 算法描述 7](#_Toc488742048)

[三、 实验方案 10](#_Toc488742049)

[四、 实验结果及分析 13](#_Toc488742050)

[Lab3 OpenMP 15](#_Toc488742051)

[一、 实验目的与要求 15](#_Toc488742052)

[二、 算法描述 15](#_Toc488742053)

[三、 实验方案 16](#_Toc488742054)

[四、 实验结果及分析 18](#_Toc488742055)

[Lab4 MPI 20](#_Toc488742056)

[一、 实验目的与要求 20](#_Toc488742057)

[二、 算法描述 20](#_Toc488742058)

[三、 实验方案 22](#_Toc488742059)

[四、 实验结果及分析 26](#_Toc488742060)

[Lab5 CUDA 28](#_Toc488742061)

[一、 实验目的与要求 28](#_Toc488742062)

[二、 算法描述 28](#_Toc488742063)

[三、 实验方案 32](#_Toc488742064)

[四、 实验结果及分析 35](#_Toc488742065)

[Project1 Sudoku problem 36](#_Toc488742066)

[一、 实验目的 36](#_Toc488742067)

[二、 实验假设 36](#_Toc488742068)

[三、 实现方法 36](#_Toc488742069)

[四、 设计与实现 38](#_Toc488742070)

[五、 思考与总结 38](#_Toc488742071)

[Project2 Monte-Carl algorithm 40](#_Toc488742072)

[一、 实验目的 40](#_Toc488742073)

[二、 实验假设 40](#_Toc488742074)

[三、 实现方法 40](#_Toc488742075)

[四、 设计与实现 42](#_Toc488742076)

[五、 结果比较与分析 45](#_Toc488742077)

[六、 思考与总结 45](#_Toc488742078)

# Lab1熟悉并行编程环境

## 实验目的与要求

1. 熟悉并行编程环境
2. 熟悉基本的并行编程理论和方法
3. 熟悉在Linux系统下使用类似pthread、OpenMp、MPI等类似工具对性能的优化

## 实验内容

使用pthread、OpenMp将向量加法并行化。MPI体会简单的通信机制。MPI在lab4具体设计了，这里就不赘述了。

## 实验结果

串行执行程序：

|  |
| --- |
| #include <stdio.h>  #define SIZE 2048  int main()  {  int i = 0;  int A[SIZE];  int B[SIZE];  int C[SIZE];  memset(A,1,SIZE);  memset(B,2,SIZE);  for(i = 0;i < SIZE;i++)  C[i] = A[i] + B[i];  return 0;  } |

Pthread程序：

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #define SIZE 2048  int A[SIZE];  int B[SIZE];  int C[SIZE];  void add(void \*arg)  {  int i = \*(int\*)arg;  C[i] = A[i] + B[i];  }  int main()  {  int i = 0;  memset(A,1,SIZE);  memset(B,2,SIZE);  for(i = 0;i < SIZE;i++)  {  pthread\_t t;  pthread\_create(&t, NULL, add, &i);  pthread\_join(t, NULL);  }  return 0;  } |

Openmp:

|  |
| --- |
| #include <stdio.h>  #include "omp.h"  #define SIZE 2048  int main()  {  int i = 0;  int A[SIZE];  int B[SIZE];  int C[SIZE];  memset(A,1,SIZE);  memset(B,2,SIZE);  #pragma omp parallel for  for(i = 0;i < SIZE;i++)  C[i] = A[i] + B[i];  return 0;  } |

MPI详见lab4

执行时间如表1.1所示

表1.1 时间对比

|  |  |
| --- | --- |
| 方式 | 时间 |
| 串行 | 6263 |
| Pthread | 5623 |
| OpenMP | 6523 |

结果分析：我们可以看到，穿行时间介于中间，线程经过产生时间最小，讲道理应该是open MP时间最少，但是是最多，追其根本原因是因为开辟新的进程扼要花费大量的时间。

# Lab2 pthread

## 实验目的与要求

* 1. master the basic principles and methods of parallel programming design and performance optimization using pthread
  2. understand the basic method for data partition and task decomposition in parallel programming
  3. implement the parallel algorithm of image convolution operation using pthread
  4. then carries on the simple analysis and summary of the program execution results

简单的来讲就是利用pthread实现卷积操作。

## 算法描述

#### 2.2.1背景知识

线性滤波可以说是图像处理最基本的方法，它可以允许我们对图像进行处理，产生很多不同的效果。做法很简单。首先，我们有一个二维的滤波器矩阵（有个高大上的名字叫卷积核）和一个要处理的二维图像。然后，对于图像的每一个像素点，计算它的邻域像素和滤波器矩阵的对应元素的乘积，然后加起来，作为该像素位置的值。这样就完成了滤波过程。如图2.1所示：

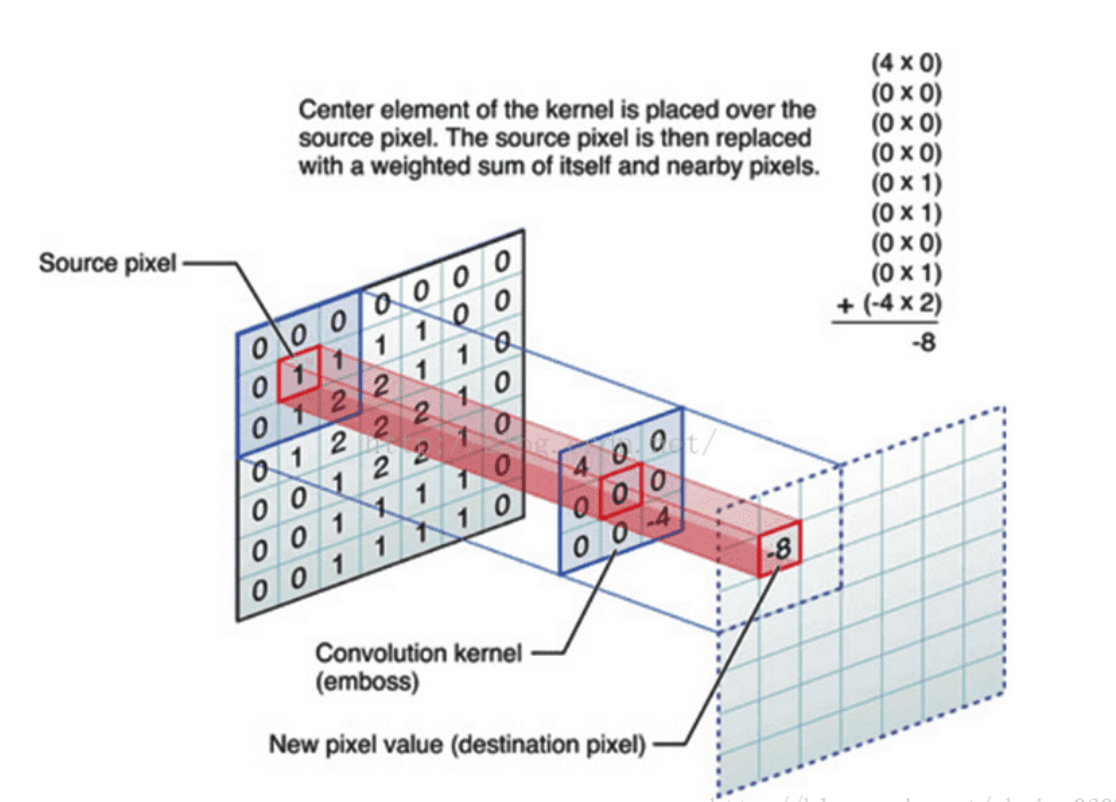


图2.1 图像卷积操作

对图像和滤波矩阵进行逐个元素相乘再求和的操作就相当于将一个二维的函数移动到另一个二维函数的所有位置，这个操作就叫卷积或者协相关。卷积和协相关的差别是，卷积需要先对滤波矩阵进行180的翻转，但如果矩阵是对称的，那么两者就没有什么差别了。

      Correlation 和 Convolution可以说是图像处理最基本的操作，但却非常有用。这两个操作有两个非常关键的特点：它们是线性的，而且具有平移不变性shift-invariant。平移不变性指我们在图像的每个位置都执行相同的操作。线性指这个操作是线性的，也就是我们用每个像素的邻域的线性组合来代替这个像素。这两个属性使得这个操作非常简单，因为线性操作是最简单的，然后在所有地方都做同样的操作就更简单了。实际上，在信号处理领域，卷积有广泛的意义，而且有其严格的数学定义，但在这里不关注这个。  2D卷积需要4个嵌套循环4-double loop，所以它并不快，除非我们使用很小的卷积核。这里一般使用3x3或者5x5。而且，对于滤波器，也有一定的规则要求：

      1）滤波器的大小应该是奇数，这样它才有一个中心，例如3x3，5x5或者7x7。有中心了，也有了半径的称呼，例如5x5大小的核的半径就是2。

      2）滤波器矩阵所有的元素之和应该要等于1，这是为了保证滤波前后图像的亮度保持不变。当然了，这不是硬性要求了。

      3）如果滤波器矩阵所有元素之和大于1，那么滤波后的图像就会比原图像更亮，反之，如果小于1，那么得到的图像就会变暗。如果和为0，图像不会变黑，但也会非常暗。

      4）对于滤波后的结构，可能会出现负数或者大于255的数值。对这种情况，我们将他们直接截断到0和255之间即可。对于负数，也可以取绝对值。

#### 2.2.2 实现方式

我们使用平滑处理图像，方法是，我们可以将当前像素和它的邻域的像素一起取平均，然后再除以9，或者直接在滤波器的9个地方取1/9的值即可，如下图2.2所示：



图2.2 处理

#### 2.2.3 实现流程

1. 使用pthread实现图像卷积操作流程图如图2.2所示

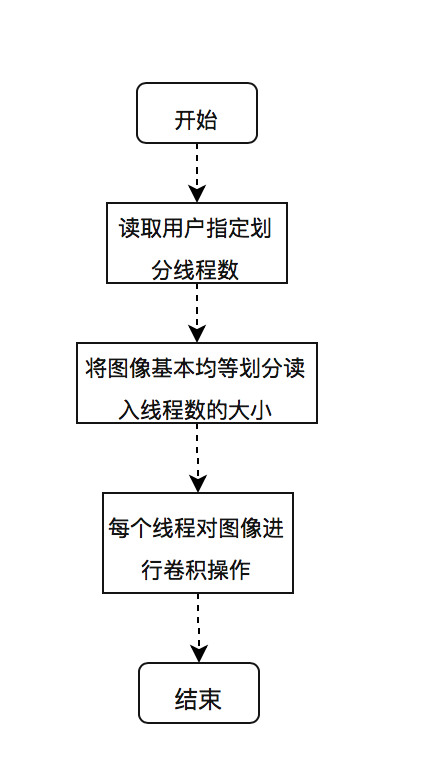


图2.3 pthread实现图像卷积操作

## 实验方案

#### 2.3.1 实验环境

1. 操作系统:Linux，内核版本:2.6.32
2. 编译器：gcc
3. CPU:x86

#### 2.3.2 实验代码

对读取图像的形成的矩阵行数进行分离通道 ，每一个通道的计算任务为一个线程；比较pthread与串行程序执行的时间相比较，理解并行程序设计的思想与方案，以及并行程序设计对程序性能的影响

代码如下：

|  |
| --- |
| #include <opencv2/opencv.hpp>  #include <opencv2/highgui/highgui.hpp>  #include <opencv2/imgproc/imgproc.hpp>  #include <stdlib.h>  #include <stdio.h>  #include <time.h>  #include <sys/time.h>  #include <pthread.h>  #include <vector>  using namespace cv;  void \*smooth(void \*num);  Mat mv[3];  Mat x;  Mat mv1[3];  int main(int argc, char \* argv[]){  int i;  int a[3];  pthread\_t pid[3];  Mat image = imread(argv[1]);  Mat result;  printf("%s 111\n",argv[0] );  if(image.data)  printf("success！\n");  else printf("fail!\n");  const int nchannels = image.channels();  printf("5555!\n");  split(image,mv);  printf("666!\n");  x.create(mv[1].size(),mv[1].type());  for(i=0;i<3;i++){  printf("777!\n");  mv1[i].create(mv[1].size(),mv[1].type());  printf("888!\n");  }  printf("999!\n");  result.create(image.size(),image.type());  struct timeval start,finish;  double cost\_time;  gettimeofday(&start, NULL);  printf("1111\n");  for (i = 0; i < 3; ++i)  {  a[i] = i;  pthread\_create(&pid[i],NULL,smooth,(void\*)&a[i]);  }    for (i = 0; i < 3; ++i)  {  pthread\_join(pid[i],NULL);  }  merge(mv1,3,result);  printf("222\n");  if(!imwrite(argv[2], result)){  printf("write file failed!\n");  }  gettimeofday(&finish, NULL);  cost\_time = 1000000 \* (finish.tv\_sec - start.tv\_sec) + finish.tv\_usec - start.tv\_usec;  printf("the time interval is : %lf \n",cost\_time);  return 0;  }  void \*smooth(void \*num){  int j,k;  int\* m = (int\*) num;  int i = \*m;  printf("%d\n",i);  for(j=1;j<(mv[i].rows-1);j++)  {  const uchar\* previous = mv[i].ptr<uchar>(j-1);  const uchar\* current = mv[i].ptr<uchar>(j);  const uchar\* next = mv[i].ptr<uchar>(j+1);  uchar\* output = mv1[i].ptr<uchar>(j);  output++;  for (k = 1; k< (mv[i].cols-1); ++k)  {  \*output = saturate\_cast<uchar>((previous[k-1]+previous[k]+previous[k+1]+current[k-1]+current[k]+current[k+1]+next[k-1]+next[k]+next[k+1])/9);  output ++;  }  }  printf("44444\n");  mv1[i].row(0).setTo(Scalar(0));  mv1[i].row(mv1[i].rows-1).setTo(Scalar(0));  mv1[i].col(0).setTo(Scalar(0));  mv1[i].col(mv1[i].cols-1).setTo(Scalar(0));  } |

编译命令如下：

|  |
| --- |
| gcc CNN\_Phread.cpp -lpthread `pkg-config opencv --libs --cflags opencv` -o t  ./t 1.jpeg 2.jpeg |

## 实验结果及分析

在本机上串行执行，图片卷积结果如图2.4所示，执行时间如图2.5所示



图2.4 图像卷积原图



图2.5 图像卷积结果

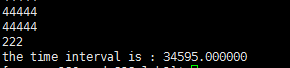


图2.2 串行结果

比较串行和pthread执行时间，通过将图片划分为一个task，，能使卷积操作的效率明显提高. 线程并发时，进行线程切换是有代价的，从lab2实验结果来看，如果线程数过少，不能最大限度地利用资源；如果进行细粒度的划分，线程切换开销太大，反而会降低程序执行效率

# Lab3 OpenMP

## 实验目的与要求

1. master the basic principles and methods of parallel programming design and performance optimization using OpenMP
2. implement the parallel algorithm of image convolution operation using OpenMP
3. carries on the simple analysis and summary of the program execution results
4. compare it with the results of Lab2

简而言之，本次实验是利用OpenMP来实现图像的卷积操作，然后分析性能差异。

## 算法描述

1. 使用OpenMp实现图像卷积操作流程图如图3.1所示

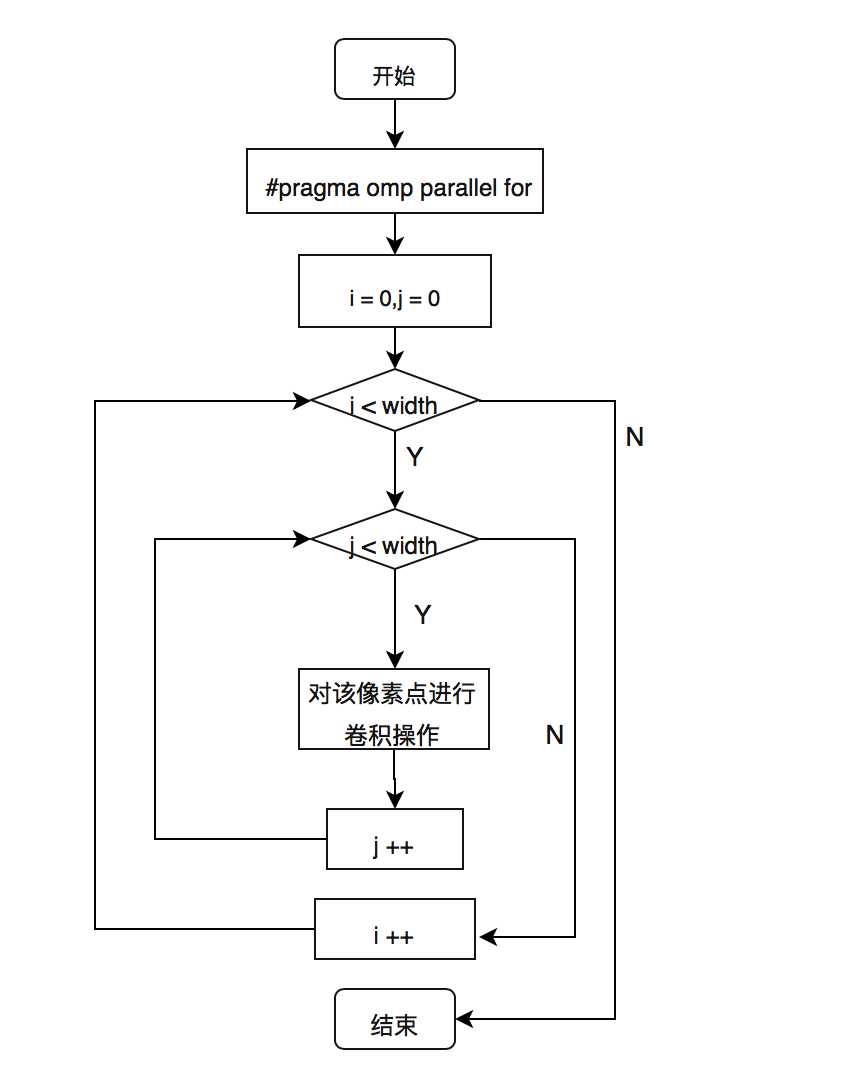


图3.1 流程图

## 实验方案

#### 3.3.1 实验环境

1. 操作系统:Linux，内核版本:2.6.32
2. 编译器：gcc
3. CPU:x86

#### 3.3.2 实验代码

对读取图像的形成的矩阵行数进行分离通道 ，每一个通道的计算任务为一个线程；比较OpenMP与串行程序执行的时间相比较，理解并行程序设计的思想与方案，以及并行程序设计对程序性能的影响。通过与lab2的结果进行比较，理解OpenMP并行实现卷积操作与pthread的设计方法与实现的不同点，理解OpenMP对卷积处理性能的影响

代码如下：

|  |
| --- |
| #include <opencv2/opencv.hpp>  #include <opencv2/highgui/highgui.hpp>  #include <opencv2/imgproc/imgproc.hpp>  #include <stdlib.h>  #include <stdio.h>  #include <time.h>  #include <sys/time.h>  #include <pthread.h>  #include <vector>  #include <omp.h>  using namespace cv;  void \*smooth(void \*num);  Mat mv[3];  int main(int argc, char \* argv[]){  int i,k;    Mat image = imread(argv[1]);  Mat result;  printf("%s 111\n",argv[0] );  if(image.data)  printf("success！\n");  else printf("fail!\n");  const int nchannels = image.channels();    result.create(image.size(),image.type());  struct timeval start,finish;  double cost\_time;  gettimeofday(&start, NULL);  printf("1111\n");  #pragma omp parallel for  for(i=1;i<(image.rows-1);i++)  {  printf("2222\n");  const uchar\* previous = image.ptr<uchar>(i-1);  const uchar\* current = image.ptr<uchar>(i);  const uchar\* next = image.ptr<uchar>(i+1);  uchar\* output = result.ptr<uchar>(i);  output++;  for (k = (nchannels+1); k< ((image.cols-1)\*nchannels); ++k)  {  \*output = saturate\_cast<uchar>((previous[k-nchannels]+previous[k]+previous[k+nchannels]+current[k-nchannels]+current[k]+current[k+nchannels]+next[k-nchannels]+next[k]+next[k+nchannels])/9);  output ++;  }  }  printf("3333\n");  result.row(0).setTo(Scalar(0));  printf("4444\n");  result.row(result.rows-1).setTo(Scalar(0));  printf("5555\n");  result.col(0).setTo(Scalar(0));  result.col(1).setTo(Scalar(0));  result.col(2).setTo(Scalar(0));  printf("6666\n");  result.col(result.cols-1).setTo(Scalar(0));  result.col(result.cols-2).setTo(Scalar(0));  result.col(result.cols-3).setTo(Scalar(0));  printf("7777\n");  if(!imwrite(argv[2], result)){  printf("write file failed!\n");  }  printf("8888\n");  gettimeofday(&finish, NULL);  cost\_time = 1000000 \* (finish.tv\_sec - start.tv\_sec) + finish.tv\_usec - start.tv\_usec;  printf("the time interval is : %lf \n",cost\_time);  return 0;  } |

## 实验结果及分析

实验结果如图3.2所示：



图2.4 图像卷积结果

结点串行执行时间、pthread、penMp执行时间，结果比较如表3.1所示

表3.1 结果比较

|  |  |
| --- | --- |
|  | 执行时间 |
| 串行 | 56066 |
| Pthread并行 | 34595 |
| OpenMP并行 | 32569 |
| 分析结果：串行执行明显不如并行卷积运算的效率 | |

这里我们发现多线程和OpenMP时间差不多，但是远远小于串行时间。

# Lab4 MPI

## 实验目的与要求

1. 掌握使用MPI实现并行程序设计的基本原理与方法
2. 通过MPI实现图像的卷积操作
3. 对实现结果进行简单的分析
4. 与pthread、OpenMP实验结果进行比较

## 算法描述

#### 4.2.1背景知识

MPI采用简单暴力的多进程来实现并行，即将同样的程序拷贝给所有的处理器执行，提供进程间的通信。通信方式分为阻塞式通信和非阻塞式通信。阻塞式通信包含标准通信模式、同步通信模式、准备通信模式、缓存通信模式

1. 采用标准通信模式

标准通信模式下，如果每个处理器有系统缓存，处理如图4.2所示，通信API如图4.1所示

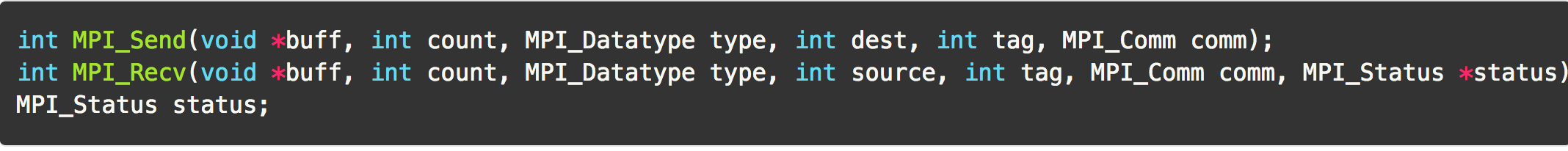


图4.1 通信API

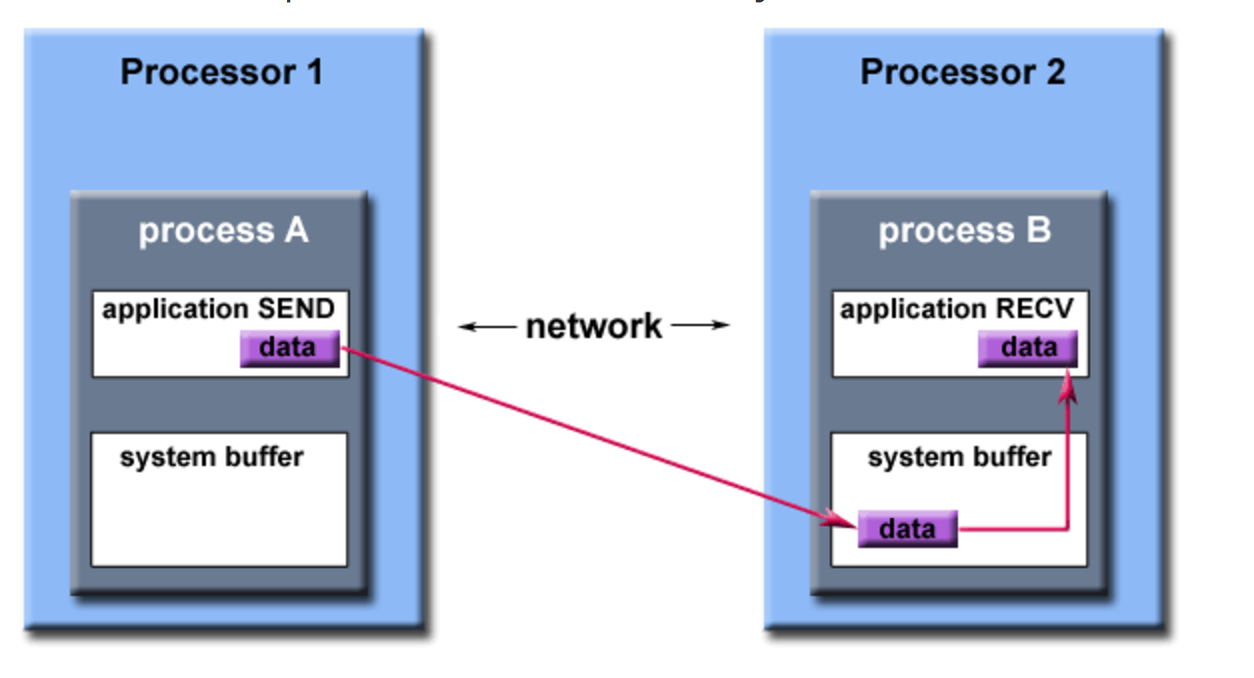


图4.2 通过系统缓存通信

1. 采用同步通信模式

这种通信的开始不依赖于相应的接收操作是否起动，但是同步发送必须等相应的接收进程开始后才可正确返回。这个返回意味着发送缓冲区中的数据被系统缓冲区缓存并且开始发送，就是说，返回时发送缓冲区可以被覆盖了。

1. 采用准备通信模式

只有当接收操作起动时，发送操作才能开始，否则发送操作会出错。对于非阻塞发送操作的正确返回，不等于发送完成，但对于阻塞发送的正确返回，发送缓冲区可被覆盖。这是一个特别的通信模式，所以，一般在使用时，会先起动接收操作，然后再向发送进程发消息，发送进程收到这个消息后，才会开始进行发送操作。

1. 采用缓存通信模式

用户必须保证发送前必须有缓冲区可用，否则失败返回。阻塞发送后，缓冲区可覆盖，但非阻塞发送正确返回后，缓冲区不能立刻被覆盖。这种通信模式下发送调用是否正确返回只和有无缓冲区有关。另外，请注意区别MPI的发送/接收缓冲区与系统缓冲区。

#### 4.4.2实现算法

实现流程描述如图4.3所示

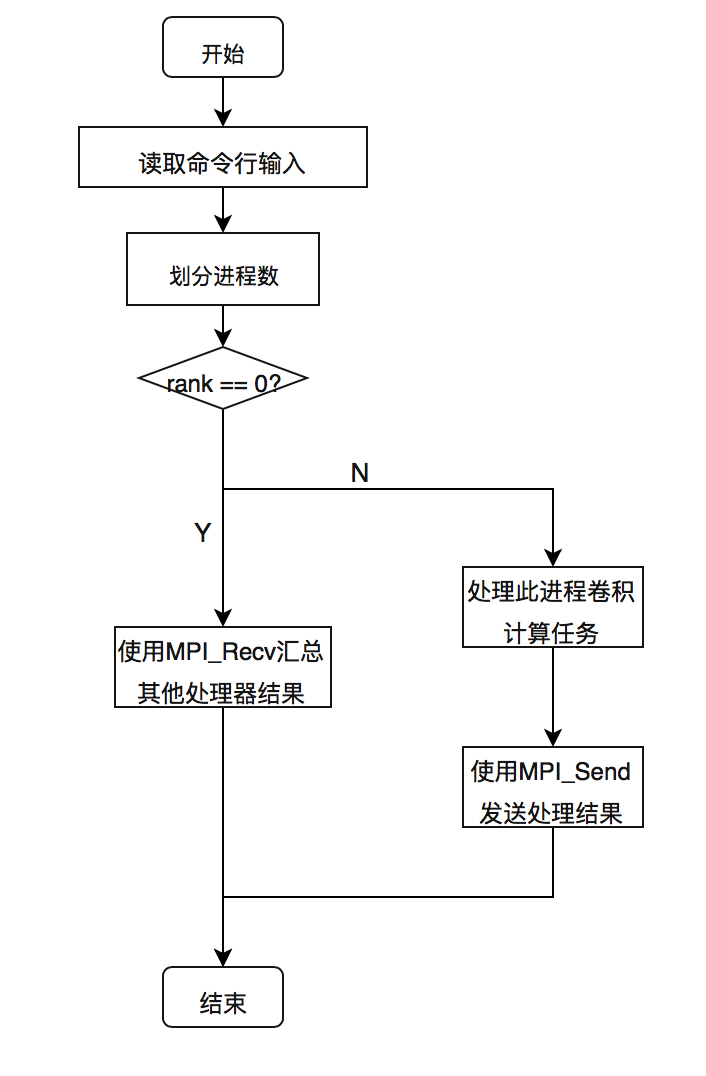


图4.3 MPI卷积处理描述

## 实验方案

#### 4.3.1实验环境

1. 操作系统:Linux，内核版本:2.6.32
2. 编译器：mpicc
3. CPU:x86

#### 4.3.2实验方案

（1）比较MPI划分为不同进程执行效率

通过与lab2、lab3的结果进行比较，理解MPI并行实现卷积操作与pthread、OpenMP的设计方法与实现的不同点，理解MPI对卷积处理性能的影响，代码如下:

|  |
| --- |
| #include <opencv2/opencv.hpp>  #include <opencv2/highgui/highgui.hpp>  #include <opencv2/imgproc/imgproc.hpp>  #include <stdlib.h>  #include <stdio.h>  #include <time.h>  #include <sys/time.h>  #include <pthread.h>  #include <vector>  #include <mpi.h>  using namespace cv;  void \*smooth(void \*num);  int ave\_row,final\_row;  Mat image ;  int main(int argc, char \* argv[]){  int i,j,k;    int my\_rank,comm\_sz;    image = imread(argv[1]);    printf("%s 111\n",argv[0] );  if(image.data)  printf("success！\n");  else printf("fail!\n");  const int nchannels = image.channels();  struct timeval start,finish;  double cost\_time;  gettimeofday(&start, NULL);  printf("1111\n");  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  ave\_row = (image.rows-2)/comm\_sz;  final\_row = image.rows-2 - ave\_row\*(comm\_sz-1);  printf("image.rows :%d comm\_sz: %d ave\_row: %d\n",image.rows,comm\_sz,ave\_row );  if(my\_rank==0)  {  Mat result;  int i,j,k;  result.create(image.size(),image.type());  //printf("111");  printf("ave\_row:%d\n",ave\_row );  for(i=0;i<ave\_row;i++)  {  const uchar\* previous = image.ptr<uchar>(my\_rank\*ave\_row+i);  const uchar\* current = image.ptr<uchar>(1+my\_rank\*ave\_row+i);  const uchar\* next = image.ptr<uchar>(2+my\_rank\*ave\_row+i);  uchar\* output = result.ptr<uchar>(1+my\_rank\*ave\_row+i);  printf("i:%d\n",i );  for(k=nchannels;k<((image.cols-1)\*nchannels);k++)  {  \*output = saturate\_cast<uchar>((previous[k-nchannels]+previous[k]+previous[k+nchannels]+current[k-nchannels]+current[k]+current[k+nchannels]+next[k-nchannels]+next[k]+next[k+nchannels])/9);  output ++;  }  }  printf("8 \n");  for(i=1;i<comm\_sz;i++)  {  printf("jia %d\n", i);  printf("lia %d\n", comm\_sz);  uchar\* output = result.ptr<uchar>(i\*ave\_row+1);  MPI\_Recv(output, nchannels\*(i==(comm\_sz-1)?final\_row:ave\_row)\*image.cols, MPI\_UNSIGNED\_CHAR , i, i, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  }  printf("3333\n");  result.row(0).setTo(Scalar(0));  printf("4444\n");  result.row(result.rows-1).setTo(Scalar(0));  printf("5555\n");  result.col(0).setTo(Scalar(0));  printf("6666\n");  result.col(result.cols-1).setTo(Scalar(0));  if(!imwrite(argv[2], result)){  printf("write file failed!\n");  }  }  else  {  int i,k,j;  int temp = nchannels\*(my\_rank==(comm\_sz-1)?final\_row:ave\_row)\*image.cols;  uchar\* output = (uchar\*)malloc( nchannels\*(my\_rank==(comm\_sz-1)?final\_row:ave\_row)\*image.cols \*sizeof(uchar));  uchar\* x = output;  printf("9\n");  for(i=0;i<(my\_rank==(comm\_sz-1)?final\_row:ave\_row);i++)  {  const uchar\* previous = image.ptr<uchar>(my\_rank\*ave\_row+i);  const uchar\* current = image.ptr<uchar>(1+my\_rank\*ave\_row+i);  const uchar\* next = image.ptr<uchar>(2+my\_rank\*ave\_row+i);  printf(" my\_rank : %d\n",my\_rank);  for(k=0;k<(image.cols\*nchannels);k++)  {  if(k<3||k>(image.cols\*nchannels-4))  \*output = 0;  else \*output = saturate\_cast<uchar>((previous[k-nchannels]+previous[k]+previous[k+nchannels]+current[k-nchannels]+current[k]+current[k+nchannels]+next[k-nchannels]+next[k]+next[k+nchannels])/9);    output ++;  }  }  MPI\_Send (x,temp,MPI\_UNSIGNED\_CHAR ,0,my\_rank,MPI\_COMM\_WORLD);    }    MPI\_Finalize();  printf("8888\n");  gettimeofday(&finish, NULL);  cost\_time = 1000000 \* (finish.tv\_sec - start.tv\_sec) + finish.tv\_usec - start.tv\_usec;  printf("the time interval is : %lf \n",cost\_time);  return 0;  } |

## 实验结果及分析

1. MPI划分为1、10、20、50、100执行时间如图4.4所示,时间如表4.1所示



图4.3 图像卷积结果

表4.1 执行时间

|  |  |
| --- | --- |
| 进程数 | 执行时间 |
| 1 | 45263 |
| 10 | 56323 |
| 20 | 66235 |
| 50 | 65686 |
| 100 | 86545 |

时间分析：MPI采用简单暴力的多进程来实现并行，即将同样的程序拷贝给所有的处理器执行，当进程数过多时，花费在拷贝，进程通信的开销明显增大。而且时间花销超过穿行时间

1. MPI与前两个实验比较，如表4.2所示

表4.2 实验结果对比

|  |  |
| --- | --- |
|  | 执行时间(ms) |
| 串行 | 56066 |
| Pthread并行 | 34595 |
| OpenMP并行 | 32569 |
| MPI并行 | 45263 |

实验分析：因为多进程复制的原因，时间运行结果不如前面两种并行算法运行时间好，但是效果优于串行执行。

# Lab5 CUDA

## 实验目的与要求

1. 掌握使用CUDA实现并行程序设计的基本原理与方法
2. 通过CUDA实现图像的卷积操作
3. 对实现结果进行简单的分析
4. 与pthread、OpenMP、MPI实验结果进行比较

## 算法描述

#### 5.2.1背景知识

1. CUDA是什么

CUDA(Compute Unified Device Architecture)，是显卡厂商NVIDIA推出的运算平台。是一种通用并行计算[架构](http://lib.csdn.net/base/architecture)，该架构使GPU能够解决复杂的计算问题。说白了就是我们可以使用GPU来并行完成像神经网络、图像处理[算法](http://lib.csdn.net/base/datastructure)这些在CPU上跑起来比较吃力的程序。通过GPU和高并行，我们可以大大提高这些算法的运行速度。

有的同学可能知道，在CPU和GPU上跑同一个神经网络，由于其大量的浮点数权重计算以及可高并行化，其速度的差距往往在10倍左右，原本需要睡一觉才能看到的训练结果也许看两集动漫就OK了。

GPU并行在图像处理方面更是应用广泛，大家知道图像处理实际上是对图像的二维矩阵进行处理，图像的尺寸都是几百乘几百的，很容易就是上万个像素的操作，随便搞个什么平滑算法，匹配算法等等的图像算法在CPU上跑个几十秒都是很正常的，对于图像处理，神经网络这种大矩阵计算，往往是可以并行化的，通过GPU并行化处理往往能够成倍的加速。

综上所述，去学习一下怎么在GPU上开个几千个线程过把优化瘾还是一件很惬意的事情，更何况CUDA为我们提供了这么优秀的计算平台，可以直接使用C/C++写出在显示芯片上执行的程序，还是一件很赞的事情。

1. CUDA架构

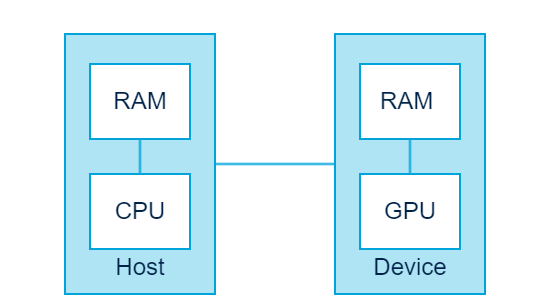


图5.1 cuda架构

在 CUDA 的架构下，一个程序分为两个部份：host 端和 device 端。Host 端是指在 CPU 上执行的部份，而 device 端则是在显示芯片上执行的部份。Device 端的程序又称为 “kernel”。通常 host 端程序会将数据准备好后，复制到显卡的内存中，再由显示芯片执行 device 端程序，完成后再由 host 端程序将结果从显卡的内存中取回。

1. thread-block-grid 结构

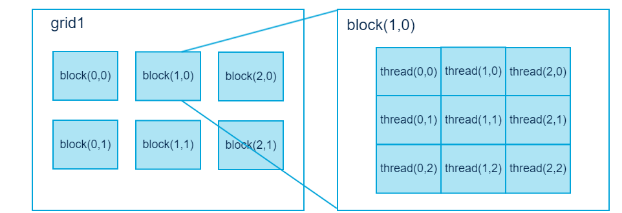


图5.2 thread-block-grid 结构

在 CUDA 架构下，显示芯片执行时的最小单位是thread。数个 thread 可以组成一个block。一个 block 中的 thread 能存取同一块共享的内存，而且可以快速进行同步的动作。

每一个 block 所能包含的 thread 数目是有限的。不过，执行相同程序的 block，可以组成grid。不同 block 中的 thread 无法存取同一个共享的内存，因此无法直接互通或进行同步。因此，不同 block 中的 thread 能合作的程度是比较低的。不过，利用这个模式，可以让程序不用担心显示芯片实际上能同时执行的 thread 数目限制。例如，一个具有很少量执行单元的显示芯片，可能会把各个 block 中的 thread 顺序执行，而非同时执行。不同的 grid 则可以执行不同的程序（即 kernel）。

每个 thread 都有自己的一份 register 和 local memory 的空间。同一个 block 中的每个thread 则有共享的一份 share memory。此外，所有的 thread（包括不同 block 的 thread）都共享一份 global memory、constant memory、和 texture memory。不同的 grid 则有各自的 global memory、constant memory 和 texture memory

1. CUDA处理流程

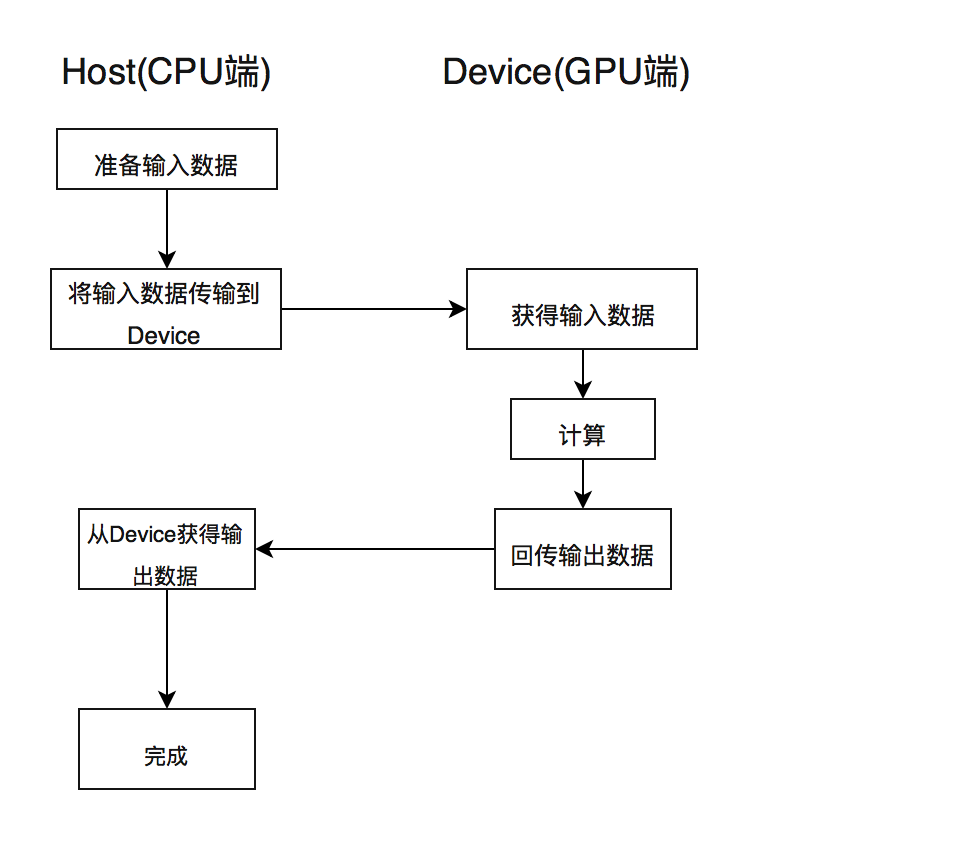


图5.3 所示处理流程

#### 5.2.2 算法实现

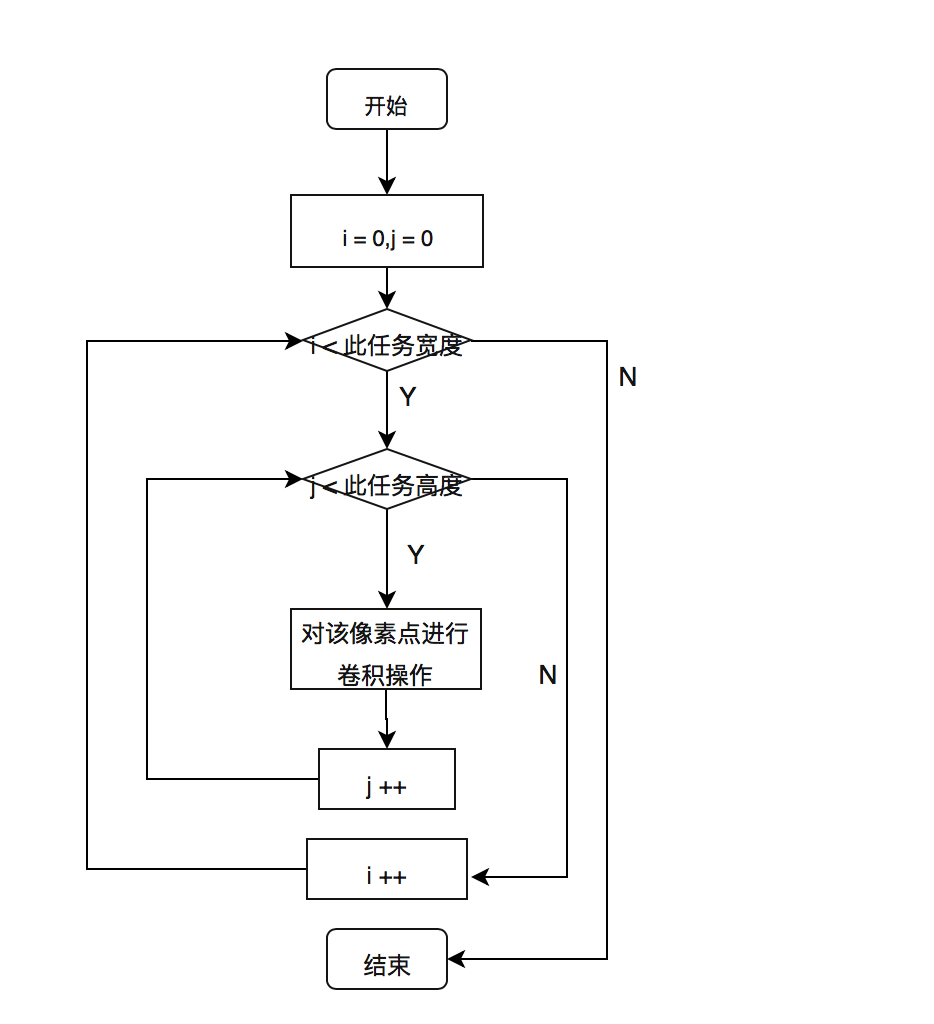


图5.4 卷积计算流程

## 实验方案

#### 5.3.1 实验环境

1. 操作系统:Linux，内核版本:2.6.32
2. 编译器：NVCC
3. CPU:x86

#### 5.3.2 实验方案

通过与前三个实验的结果进行比较，理解cuda并行实现卷积操作与pthread、OpenMp、MPI的设计方法与实现的不同点，理解cuda对卷积处理性能的影响,实验代码如下：

|  |
| --- |
| #include <opencv2/opencv.hpp>  #include <opencv2/highgui/highgui.hpp>  #include <opencv2/imgproc/imgproc.hpp>  #include <stdlib.h>  #include <stdio.h>  #include <time.h>  #include <sys/time.h>  #include <cuda.h>  using namespace cv;  \_\_global\_\_ void add(const uchar \*image, uchar \*result,const int rows,const int cols,const int nchannels, int num) {  int i,k;  int t;  int len = cols\*nchannels;  int id = threadIdx.x+blockIdx.x \* blockDim.x;  int temp;  if (id < num) {  int ave\_row = (rows-2)/num;  int final\_row = (rows-2)%num + ave\_row;  if(id==(num-1))  temp = final\_row;  else temp = ave\_row;  for( i=0;i<temp;i++)  {  const uchar\* previous = image + id\*ave\_row\*len+i\*len;  const uchar\* current = image + 1\*len+id\*ave\_row\*len+i\*len ;  const uchar\* next = image + (2\*len+id\*ave\_row\*len+i\*len);  uchar\* output = result + (1\*len+id\*ave\_row\*len+i\*len);    for(k=nchannels;k<((cols-1)\*nchannels);k++)  {  t = ((previous[k-nchannels]+previous[k]+previous[k+nchannels]+current[k-nchannels]+current[k]+current[k+nchannels]+next[k-nchannels]+next[k]+next[k+nchannels])/9);  if(t<0)  \*output = 0;  else if(t>255)  \*output = 255;  else \*output = t;  output ++;  }  }  }  }  int main(int argc, char \*\*argv) {    cudaError\_t flag1,flag2;  Mat image = imread(argv[1]);  Mat result;  printf("%s 111\n",argv[0] );  if(image.data)  printf("success！\n");  else printf("fail!\n");  const int nchannels = image.channels();  result.create(image.size(),image.type());  int M,N;  sscanf(argv[3],"%d",&M);  sscanf(argv[4],"%d",&N);    uchar \*mirror, \*result1;  flag1 = cudaMalloc((void \*\*)&mirror, sizeof(uchar) \* image.rows\*image.cols\*nchannels);  flag2 = cudaMalloc((void \*\*)&result1, sizeof(uchar) \* (image.rows-2)\*image.cols\*nchannels);  if (flag1!= cudaSuccess || flag2!=cudaSuccess) {  printf(" cudaMalloc failed!");  return -1;  }  printf("999\n");  struct timeval start,finish;  double cost\_time;  gettimeofday(&start, NULL);  uchar\* buf = image.ptr<uchar>(0);  cudaMemcpy(mirror, buf, sizeof(uchar) \* image.rows\*image.cols\*nchannels, cudaMemcpyHostToDevice);  add<<<M, N>>>(mirror, result1, image.rows,image.cols,nchannels,M\*N);  cudaThreadSynchronize();    buf = result.ptr<uchar>(1);  cudaMemcpy(buf, result1, sizeof(uchar) \*(image.rows-2)\*image.cols\*nchannels , cudaMemcpyDeviceToHost);    result.row(0).setTo(Scalar(0));  result.row(result.rows - 1).setTo(Scalar(0));  gettimeofday(&finish, 0);  cost\_time = 1000000 \* (finish.tv\_sec - start.tv\_sec) + finish.tv\_usec - start.tv\_usec;  printf("time interval is : %lf \n",cost\_time);  if(!imwrite(argv[2], result)){  printf("write file failed!\n");  }  cudaFree(mirror);  cudaFree(result1);  return 0;  } |

## 实验结果及分析

1. 实验结果如图5.5所示



图5.3 图像卷积结果

1. 结果分析

使用Cuda进行并行处理时基于GPU的并行处理，处理问题时，可以采用线程并行和块并行，两者的差别在与并行的粒度。对于一个问题，进行线程并行和块并行性能可能会有较大差异

# Project1 Sudoku problem

## 实验目的

* 1. master the methods to parallelize and improve a program
  2. understand the relationship between parallel granularity and performance
  3. master how to partition data and decompose tasks of a complex algorithm

根据上述要求我们使用回溯法求解数独问题。以达到实验目的。

## 实验假设

数独问题可以并行化.

## 实现方法

#### 1.3.1背景知识

数独的求解步骤

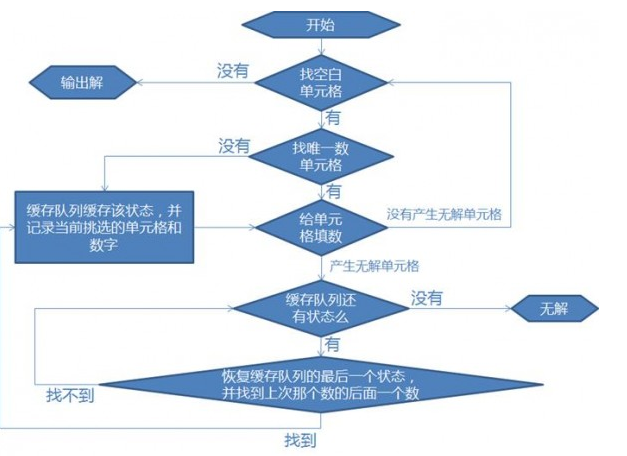


图1.1 求解步骤

 传统的数独游戏是将一个大正方形划成3×3的九个九宫格，每个九宫格又由3行3列共9个小方格构成，这样整个大正方形形成一个9×9的方格群。在这个大正方形内填满1-9的数字,要求大正方形每一行、每一列及每个九宫格内均必须包括1到9的每一个数字，既不能遗漏也不能重复

#### 数独实现算法

采取深搜加回溯加剪枝的算法。

1. 对于选择的数字，有3种限制，我们通过这三种限制来对回溯过程进行剪枝。

行不重复  r[9][9]

列不重复  c[9][9]

块不重复  b[9][9]

对使用3个9×9的数组来描述这三种限制。

对于行 n 来说，所有之前填过的数字nums，都有r[n][nums] = 1

同理，对与列和块来说，之前每个填过的数字都相应标记为1.

1. 当尝试向一个格子[row][col] (块序号为bi)填充数据i  （0<=i<=8 ,的时候，如果对应r[row][i] =1 或 c[col][i] = 1 或 b[bi][i] =1，说明无法满足条件，是非法的。这时再尝试i = i+1.
2. 当向坐标为[row][col] 填入一个满足条件的数字i的时候，我们更新限制数组如下

r[row][i] = 1;

c[col][i] = 1;

b[bi][i] = 1;

1. 然后坐标移到下一个，这里选用的顺序是右移动。对于一个格子CX的后续CY，如果我们遍历0～9完成，那么返回。尝试CX的其他可能性。
2. 当填入的格子是 [8][8]时，整个问题处理完毕

## 设计与实现

如图1.2是数独输入：

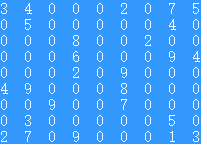


图1.2 数独输入

图1.3是数独输出所示：

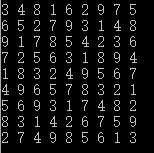


图1.3 数独输出

## 思考与总结

随着时间的推移，课程设计也即将结束了，但这个学期并行编程的学习还是具有相当大的意义，它从一个程度上改变了我们的编程思想，如何将一个程序快速而又准备的进行编写，进行编译，都成为了我们思考的重点，也通过这一个学期的学习，我们将数据结构的思想带入到了我们以后的编程学习中去。在这个阶段，我也明白了，好的思想，不能提留于字面上的认知，还需要的是平时多练多写一些相关的程序，并且通过修改，加入新的算法去尝试改变自己的一些编程思想。保持更新算法的速度，这才是关键。

实验已经接近尾声了，但它给我的不只是程序设计上的满足，更重要的是对自己编程思想的一次更新，以及对算法的一个全新的认识！

# Project2 Monte-Carl algorithm

## 实验目的

* 1. master the two typical parallel program development tools (OpenMP and MPI)
  2. understand the similarities and differences between the two tools during the process of parallel program design and optimization
  3. adjust and analyze the parallel granularity of the generated parallel algorithms because of optimization
  4. further understand the principles of parallel programming and what should pay attention to

根据上述要求我们使用Monte-Carl algorithm求圆周率。以达到实验目的。

## 实验假设

计算机可以产生随机概率。

## 实现方法

以 概率和统计理论方法为基础的一种计算方法。将所求解的问题同一定的概率模型相联系，用计算机实现统计模拟或抽样，以获得问题的近似解。比如，给定 x=a ，和 x=b ，你要求某一曲线 f 和这两竖线，及 x 轴围成的面积，你可以起定 y 轴一横线 y=c 其中 c>=f(a) and c>=f(b) ，很简单的，你可以求出 y=c,x=a,x=b, 及 x 轴围成的矩形面积，然后利用随机参生生大量在这个矩形范围之类的点，统计出现在曲线上部点数和出现在曲线下部点的数目，记为： doteUpCount,nodeDownCount, 然后所要求的面积可以近似为 doteDownCounts 所占比例 \* 矩形面积。

在数值积分法中，利用求单位圆的 1/4 的面积来求得 Pi/4 从而得到 Pi 。单位圆的 1/4 面积是一个扇形，它是边长为 1 单位正方形的一部分。只要能求出扇形面积 S1 在正方形面积 S 中占的比例 K=S1/S 就立即能得到 S1 ，从而得到 Pi 的值。怎样求出扇形面积在正方形面积中占的比例 K 呢？一个办法是在正方形中随机投入很多点，使所投的点落在正方形中每一个位置的机会相等看其中有多少个点落在扇形内。将落在扇形内的点数 m 与所投点的总数 n 的比 m/n 作为 k 的近似值。 P 落在扇形内的充要条件是 x^2+y^2<=1 。

如何用蒙特卡罗方法计算圆周率π？正方形内部有一个相切的圆，它们的面积之比是π/4。如下图 8.1：

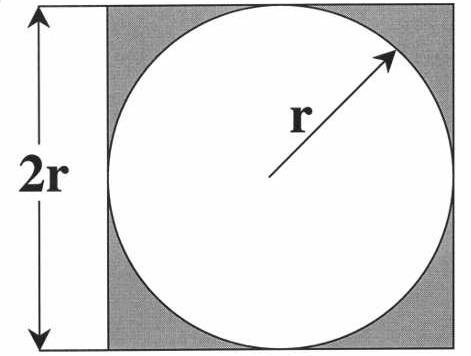
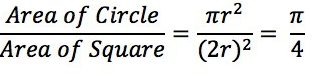


图8.1 面积图

所以我们可以得到如下公式：



这个公式是我们估计的原理，现在，在这个正方形内部，随机产生10000个点（即10000个坐标对 (x, y)），计算它们与中心点的距离，从而判断是否落在圆的内部。这也就是Monte-Carl algorithm，如图8.2所示;

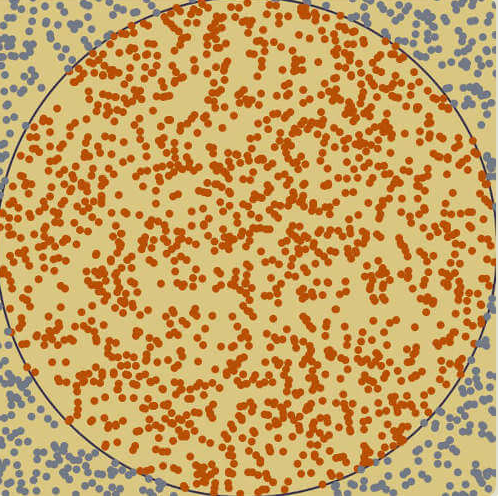


图8.2 原理图

## 设计与实现

蒙特卡洛方法实现计算圆周率的方法比较简单，其思想是假设我们向一个正方形的标靶上随机投掷飞镖，靶心在正中央，标靶的长和宽都是2 英尺。同时假设有一个圆与标靶内切。圆的半径是1英尺，面积是π平方英尺。如果击中点在标靶上是均匀分布的（我们总会击中正方形），那么飞镖击中圆的数量近似满足等式飞镖落在圆内的次数/飞镖落在标靶内的总次数=π/4，因为环包含的面积与正方形面积的比值是π/4，因为环所包含的面积与正方形面积的比值是π/4。我们可以用这个公式和随机数产生器来估计π的值。

这种采用了随机（随机投掷）的方法称为蒙特卡洛（Monte Carlo）方法。编写了一个采用蒙特卡洛方法的MPI， OpenMP程序估计π的值。使用MPI编写时，进程0读取总的投掷次数，并把它们广播给各个进程。使用MPI\_Reduce求出局部变量number\_in\_cycle的全局总和，并让进程0打印它。使用OpenMP编写时，在开启任何线程前读取总的投掷次数。使用reduction子句计算飞镖集中环内的次数。在合并所有的线程后，打印结果。

代码如下：

|  |
| --- |
| #include <opencv2/opencv.hpp>  #include <opencv2/highgui/highgui.hpp>  #include <opencv2/imgproc/imgproc.hpp>  #include <stdlib.h>  #include <stdio.h>  #include <time.h>  #include <sys/time.h>  #include <pthread.h>  #include <vector>  #include <omp.h>  #include <mpi.h>  using namespace cv;  int main(int argc, char\* argv[]){  int my\_rank,comm\_sz;  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);  if(my\_rank==0)  {  long count1 = 0;  long count2 = 0;  int i;  double result;  //#pragma omp parallel for reduction(+:count1,count2)  for(i=1;i<comm\_sz;i++)  {  int len[2];  MPI\_Recv(len, 2, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  count1 += len[0];  count2 += len[1];  }  result = count1\*4.0/(count1+count2);  printf("count1: %d count2 : %d\n", count1,count2);  printf(" pai: %lf\n", result);  }  else  {  int i;  time\_t t;  int len[2];  int count1= 0;  int count2 = 0;  srand((unsigned) time(&t)+my\_rank);  #pragma omp parallel for reduction(+:count1,count2)  for(i=0;i<10000000;i++)  {  unsigned int seed = (unsigned) time(&t)\*10000/99+my\_rank + i;  double x = 2.0\*rand\_r(&seed)/RAND\_MAX - 1;  double y = 2.0\*rand\_r(&seed)/RAND\_MAX - 1;  //printf("%lf\n", sqrt(x\*x+y\*y));  if(sqrt(x\*x+y\*y)<=1)  count1 ++;  else count2++;  }  len[0] = count1;  len[1] = count2;  printf("%d %d %d\n",my\_rank, count1,count2 );  MPI\_Send (len,2,MPI\_INT ,0,0,MPI\_COMM\_WORLD);  }  MPI\_Finalize();  return 0;  } |

编译命令如下;

|  |
| --- |
| mpicc m.cpp -openmp `pkg-config opencv --libs --cflags opencv` -o t  mpirun -np 10 ./t |

实现结果如下图8.3：

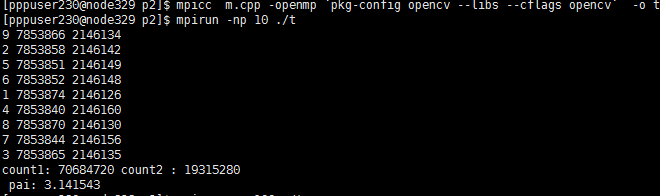


图8.3 实现结果

## 结果比较与分析

编译成功后，我们把节点数设置为1、5、10、15、20，进行结果比较如下表8.1所示：

|  |  |
| --- | --- |
| 节点数 | 圆周率 |
| 1 | 3.141520 |
| 5 | 3.141522 |
| 10 | 3.141532 |
| 15 | 3.141546 |
| 20 | 3.141550 |

我们知道真实的圆周率大小是3.14159265左右，我们对比上表发现，节点数越多越接近圆周率的大小。这个是符合结果预期。

## 思考与总结

计算机技术的发展，使得蒙特卡罗方法在最近10年得到快速的普及。现代的蒙特卡罗方法，已经不必亲自动手做实验，而是借助计算机的高速运转能力，使得原本费时费力的实验过程，变成了快速和轻而易举的事情。它不但用于解决许多复杂的科学方面的问题，也被项目管理人员经常使用。借助计算机技术，蒙特卡罗方法实现了两大优点：

一是简单，省却了繁复的数学报导和演算过程，使得一般人也能够理解和掌握；

二是快速。简单和快速，是蒙特卡罗方法在现代项目管理中获得应用的技术基础。

　 蒙特卡罗方法有很强的适应性，问题的几何形状的复杂性对它的影响不大。该方法的收敛性是指概率意义下的收敛，因此问题维数的增加不会影响它的收敛速度，而且存贮单元也很省，这些是用该方法处理大型复杂问题时的优势。因此，随着电子计算机的发展和科学技术问题的日趋复杂，蒙特卡罗方法的应用也越来越广泛。它不仅较好地解决了多重积分计算、微分方程求解、积分方程求解、特征值计算和非线性方程组求解等高难度和复杂的数学计算问题，而且在统计物理、核物理、真空技术、系统科学 、信息科学 、公用事业、地质、医学，可靠性及计算机科学等广泛的领域都得到成功的应用。

基于蒙特卡罗模拟的风险分析,对于工程实际应用具有较强的参考价值。随机模拟500000次,如果仅靠人的大脑进行计算,这在现实世界中是不可能的,但考虑到系统决策支持功能, 算法设计为由使用者自己设计方案, 采用[人机交互](http://wiki.mbalib.com/wiki/%E4%BA%BA%E6%9C%BA%E4%BA%A4%E4%BA%92), 这样可以发挥使用者的经验判断;系统实现模拟运算——系统对每一个设定的投资项目期投资、寿命期、残值以及各年的收入、支出,以及[应付税金](http://wiki.mbalib.com/wiki/%E5%BA%94%E4%BB%98%E7%A8%8E%E9%87%91)的税率、项目的资本成本等随机变量及他们的[概率密度函数](http://wiki.mbalib.com/wiki/%E6%A6%82%E7%8E%87%E5%AF%86%E5%BA%A6%E5%87%BD%E6%95%B0),通过蒙特卡罗模拟方法,得出了项目在不同概率发生的情况下净现值模拟计算结果。为人们解决不确定性项目的决策提供了简单的方法,节约了人们的工作量和时间。但是利用蒙特卡罗模型分析问题时,收集数据是非常关键的