

# 华中科技大学

# 课程实验报告

课程名称： 并行编程原理与实践

专业班级： 计算机科学与技术 1409 班

学 号： U201414815

姓 名： 胡超

指导教师： 金海

报告日期： 2017.07.12

计算机科学与技术学院

目录

- 1 实验一.....4
  - 1.1 实验目的与要求 .....4
  - 1.2 实验内容 .....4
  - 1.3 实验结果 .....9
- 2 实验二.....11
  - 2.1 实验目的与要求 .....11
  - 2.2 算法描述 .....11
  - 2.3 实验方案 .....12
  - 2.4 实验结果与分析 .....12
- 3 实验三.....14
  - 3.1 实验目的与要求 .....14
  - 3.2 算法描述 .....14
  - 3.3 实验方案 .....14
  - 3.4 实验结果与分析 .....15
- 4 实验四.....17
  - 4.1 实验目的与要求 .....17
  - 4.2 算法描述 .....17
  - 4.3 实验方案 .....17
  - 4.4 实验结果与分析 .....18
- 5 实验五.....20
  - 5.1 实验目的与要求 .....20
  - 5.2 算法描述 .....20
  - 5.3 实验方案 .....21
  - 5.4 实验结果与分析 .....21
- 6 实验六.....23
  - 6.1 实验环境 .....23
  - 6.2 实验目的 .....23
  - 6.3 设计思路 .....23
  - 6.4 实验结果 .....24
- 7 附录.....27

7.1	pthread 并行卷积源代码 .....	27
7.2	openmp 并行卷积源代码 .....	28
7.3	MPI 并行卷积源代码 .....	29
7.4	Cuda 并行卷积源代码.....	31
7.5	PageRank 源代码.....	33

# 1 实验一

## 1.1 实验目的与要求

### 1.1.1 实验目的

了解并行编程的目的，思路和方法，并且熟悉并行计算的分析 and 解决问题的技巧。

### 1.1.2 实验要求

- 1) 使用通用工具和程序模型（如 pthread, OpenMP, MPI, MapReduce 和 CUDA）的基本原理和方法来并行现有算法并开发新的并行算法
- 2) 分析并行过程和结果，让学生在并行化理论的更深层次，编译和操作系统的基础上了解并行化的目的和性能改进的原因

## 1.2 实验内容

### 1.2.1 pthread 并行加法

#### 1) 原理

使用 pthread 创建多个线程，用来完成一个基本的向量加法任务。pthread 相关头文件为 pthread.h，编译时需要加上 -lpthread。

基本的 API:

① `int pthread_create( pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *), void *arg);`

thread 表示线程 ID，与线程中的 pid 概念类似

attr 表示设定线程的属性，可以暂时不用考虑

func 表示新创建的线程会从这个函数指针处开始运行

arg 表示这个函数的参数指针

返回值为 0 代表成功，其他值为错误编号

② `int pthread_join( pthread_t thread, void **retval );`

thread 表示线程 ID，与线程中的 pid 概念类似

retval 用于存储等待线程的返回值

## 2) 实践

本实践实现两个向量数组相加，分别需要实现一个主函数 `main`，和线程函数 `_add`。由于数组大小为 10，因此创建了十个线程用来计算。这其中需要注意的是由于线程参数是指针，很容易会犯在创建线程时参数的值和线程执行时的值不一样的情况，因此在创建线程传递参数时要避免传递同一变量的地址给线程，以避免造成线程中取到的值与我们设想的不一樣。具体代码实现如下：

```
#include <pthread.h>
#include <stdio.h>
#include <iostream>
using namespace::std;

int A[10] = {0,0,0,0,0,0,0,0,0,0};
int B[10] = {0,1,2,3,4,5,6,7,8,9};
int C[10] = {0,1,2,3,4,5,6,7,8,9};

void *_add(void * num) {
    int i = *(int *)num;
    A[i] = B[i] + C[i];
    printf("result:%d\n",A[i]);
    return (void *)0;
}

int main() {
    int num[10] = {0,1,2,3,4,5,6,7,8,9};
    pthread_t t_id[10];
    int i = 0;

    for(i = 0;i <= 9;i++) {
        if(pthread_create(&t_id[i],NULL,_add,&num[i]) != 0) {
            cout<<"线程创建出错";
        }
    }

    for(i = 0;i <= 9;i++) {
        if(pthread_join(t_id[i],NULL) != 0) {
            cout<<"线程出错";
        }
    }

    return 0;
}
```

### 1.2.2 OpenMP 并行加法

#### 1) 原理

使用特殊的编译引导语句来实现一个基本的向量加法任务。Openmp 会自动将 `for` 循环分解为多个线程并行执行，Openmp 相关头文件为 `omp.h`，编译时需要加上 `-fopenmp`。

特殊的编译语句为 `#pragma omp parallel for`。

#### 2) 实践

在 `for` 循环语句前加上 `#pragma omp parallel for`，编译执行即可。具体实现代码如下：

```

#include <stdio.h>
#include <omp.h>

int main() {
    int A[10] = {0,0,0,0,0,0,0,0,0,0};
    int B[10] = {0,1,2,3,4,5,6,7,8,9};
    int C[10] = {0,1,2,3,4,5,6,7,8,9};

    #pragma omp parallel for
    for(int i = 0; i <= 9; i++) {
        A[i] = B[i] + C[i];
        printf("result:%d\n", A[i]);
    }

    return 0;
}

```

### 1.2.3 MPI 并行加法

#### 1) 原理

使用 MPI 通过多个进程来完成一个基本的向量加法任务。与 OpenMP 并行程序不同，MPI 是一种基于信息传递的并行编程技术。消息传递接口是一种编程接口标准，而不是一种具体的编程语言。简而言之，MPI 标准定义了一组具有可移植性的编程接口。因此可以借助其将程序分散到多个机器上进行并行执行，以提高程序执行效率。MPI 相关头文件为 `mpi.h`，编译时需要用专门的编译工具 `mpicc`，运行时也有运行环境 `mpirun`。

基本的 API:

#### ① `int MPI_Init(int *argc, char **argv)`

`MPI_Init` 是 MPI 程序的第一个调用，它完成 MPI 程序的所有初始化工作，启动 MPI 环境，标志并行代码的开始。

#### ② `int MPI_Finalize(void)`

`MPI_Finalize` 是 MPI 程序的最后一个调用，它结束 MPI 程序的运行，标志并行代码的结束，结束除主进程外其它进程。其之后串行代码仍可在主进程(rank = 0)上继续运行。

#### ③ `int MPI_Comm_size(MPI_Comm comm, int *size);`

获取进程个数 p。

#### ④ `int MPI_Comm_rank(MPI_Comm comm, int *rank);`

MPI 获取当前进程的 RANK，rank 值取址范围是 0~p-1，RANK 值唯一的表示了进程的 ID，其中 Rank=0 的为主进程

#### ⑤ `int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`

发送函数：当前进程将以 buf 为初始地址，长度为 count 且元素类型为 datatype 的信息发动给 rank 值为 dest 的进程，这条消息的标识符为 tag。

其中 datatype 有 MPI\_INT, MPI\_FLOAT 等常用类型

Tag 的作用是用于区分一对进程之间发送的不同信息

⑥int MPI\_Recv(void\* buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status);

接受函数：从 rank 值为 source 的进程接受标识符为 tag 的信息，存入以 buf 为初始地址，长度为 count 的存储区域中，类型为 datatype。

## 2) 实践

由于这里只需计算，我没有考虑结果的传回，因此没有进程间的相互通信，只是分为十个进程，进行计算然后将结果输出，编译执行。具体实现代码如下：

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

int main(int argc, char *argv[]) {
    int A[10] = {0,0,0,0,0,0,0,0,0,0};
    int B[10] = {0,1,2,3,4,5,6,7,8,9};
    int C[10] = {0,1,2,3,4,5,6,7,8,9};

    int my_rank = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if(my_rank <= 9) {
        A[my_rank] = B[my_rank] + C[my_rank];
        printf("result:%d\n", A[my_rank]);
    }
    MPI_Finalize();

    return 0;
}
```

### 1.2.4 Cuda 并行加法

#### 1) 原理

CUDA 在执行的时候是让 host 里面的一个一个的 kernel 按照线程网格(Grid)的概念在显卡硬件 (GPU) 上执行。

每一个线程网格又可以包含多个线程块(block)，每一个线程块中又可以包含多个线程(thread)。

将任务合理的分配到 grid 和 thread 中，有助于提升程序的性能

基本 API:

①cudaError\_t cudaMalloc (void \*\*devPtr, size\_t size );

在设备端分配 size 大小的空间，起始地址为 devPtr。

② `cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind);`

将以 `src` 为地址长度为 `count` 的数据赋值到 `dst` 为起始地址的内存区域中，常用的 `kind` 有 `cudaMemcpyHostToDevice`，`cudaMemcpyDeviceToHost`。

③ `cudaError_t cudaFree (void *devPtr);`

在设备端清理以 `devPtr` 为起始地址的内存空间

## 2) 实践

用 `cuda` 实现向量加法，首先将数据从主机复制到显卡内存，计算后再取出即可。具体实现代码如下：

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

void addWithCuda(int *c, const int *a, const int *b, size_t size);

__global__ void addKernel(int *c, const int *a, const int *b) {
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    const int arraySize = 10;
    const int a[arraySize] = {0,1,2,3,4,5,6,7,8,9};
    const int b[arraySize] = {0,1,2,3,4,5,6,7,8,9};
    int c[arraySize] = {0,0,0,0,0,0,0,0,0,0};

    // Add vectors in parallel.
    addWithCuda(c, a, b, arraySize);
    printf("{ 1,2,3,4,5 } + { 10,20,30,40,50 } = { %d,%d,%d,%d,%d }\n", c[0], c[1], c[2], c[3], c[4]);
    // cudaThreadExit must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaThreadExit();
    return 0;
}

// Helper function for using CUDA to add vectors in parallel.
void addWithCuda(int *c, const int *a, const int *b, size_t size) {
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaSetDevice(0);
    // Allocate GPU buffers for three vectors (two input, one output)
    cudaMalloc((void**)&dev_c, size * sizeof(int));
    cudaMalloc((void**)&dev_a, size * sizeof(int));
    cudaMalloc((void**)&dev_b, size * sizeof(int));
    printf("%d\n", cudaStatus);
    // Copy input vectors from host memory to GPU buffers.
    cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<1, size>>>>(dev_c, dev_a, dev_b);
```



```

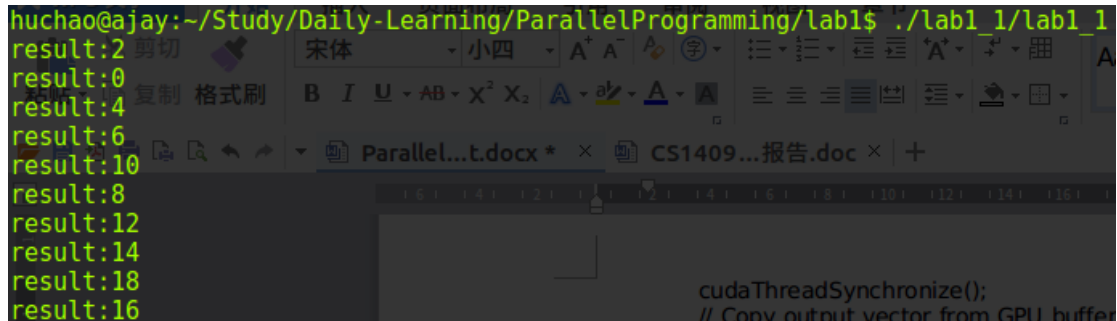
// cudaThreadSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaThreadSynchronize();
// Copy output vector from GPU buffer to host memory.
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
}

```

## 1.3 实验结果

### 1.3.1 pthread 并行加法

实践为两个值为 0-9 的一维数组相加，结果如下图 1-1 所示，可知并行运行结果正确。



```

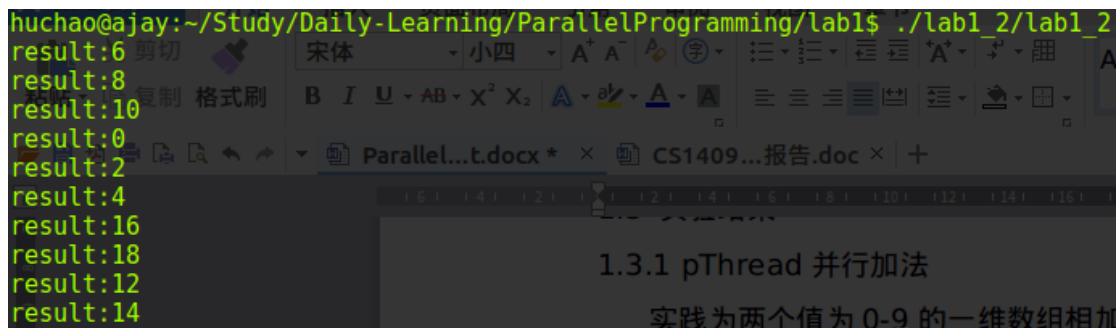
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab1$ ./lab1_1/lab1_1
result:2
result:0
result:4
result:6
result:10
result:8
result:12
result:14
result:18
result:16

```

图 1-1 pthread 并行加法结果图

### 1.3.2 OpenMp 并行加法

数据与前述方法相同，得到结果如下图 1-2 所示，可知并行运行结果正确。



```

huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab1$ ./lab1_2/lab1_2
result:6
result:8
result:10
result:0
result:2
result:4
result:16
result:18
result:12
result:14

```

图 1-2 openmp 并行加法结果图

### 1.3.3 MPI 并行加法

数据与前述方法相同，得到结果如下图 1-3 所示，可知并行运行结果正确。

```
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab1/lab1_3$ mpirun -np 10 lab1_3
result:10
result:0
result:6
result:8
result:2
result:4
result:12
result:14
result:16
result:18
```

1.3.2 OpenMp 并行加法

数据与前述方法相同，得到结果如下图 1-2 所示，

```
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab1/lab1_3$ mpirun -np 10 lab1_3
result:6
result:8
result:10
result:0
result:2
```

图 1-3 mpi 并行加法结果图

### 1.3.4 Cuda 并行加法

数据与前述方法相同，得到结果如下图 1-4 所示，可知并行运行结果正确。

```
pppuser230@node19:~/U2014114815/lab1/lab1_4$ ./lab1_4
{0,1,2,3,4,5,6,7,8,9} + {0,1,2,3,4,5,6,7,8,9} = {0,2,4,6,8,10,12,14,16,18}
```

图 1-4 cuda 并行加法结果图

## 2 实验二

### 2.1 实验目的与要求

- 1) 掌握使用 pthread 并行编程设计和性能优化的基本原理和方法
- 2) 了解并行编程中数据分区和任务分解的基本方法
- 3) 使用 pthread 实现图像卷积运算的并行算法
- 4) 然后对程序执行结果进行简单的分析和总结

### 2.2 算法描述

本次实验采用多线程来并行实现图像的卷积操作，具体为边缘操作。主要的工作重点在线程执行函数上，首先将图像像素点分为 10 部分，然后在创建线程时将该线程应该执行计算的开始行数与结束行数传到线程函数中。由于只有一个参数，因此采用结构体来对多个数据进行传递。接着只需对每个像素点做相应计算即可。这里需要注意在行数分解时可能不会完全均分，因此要对最后一组做区别对待。剩下的便是等待执行结束，保存执行结果。

算法流程图如下图 2-1 所示：

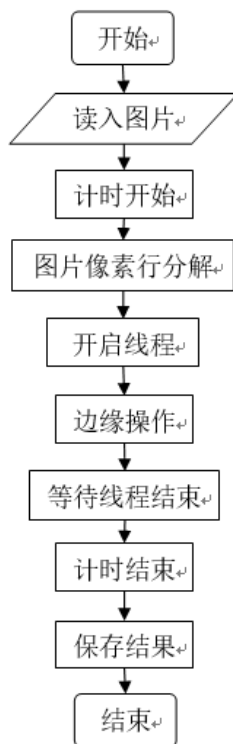


图 2-1 pthread 并行卷积算法流程图

## 2.3 实验方案

### 2.3.1 开发与运行环境

系统：ubuntu 16.04 Desktop

kernel：linux-4.4.0-83-generic

编译器：g++5.4.0

opencv：opencv-2.4.9.1

CPU：Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz

内存：4G+4G

### 2.3.2 实验步骤

首先引入 opencv 操作环境，方便能够对图片进行处理。

图片读取为 Mat 格式，是现在较新的 opencv 推行的图片格式。

按照边缘处理卷积并行处理算法编写实验代码。

用命令“g++ lab2.cpp -o lab2 -lpthread `pkg-config --libs --cflags opencv`”编译程序。

在执行程序查看结果。

## 2.4 实验结果与分析

实验原图由于较大，不好进行展示，截取部分如下图 2-2 所示。



图 2-2 实验原图部分截取示意图

此次实验结果图部分截取如下图 2-3 所示：

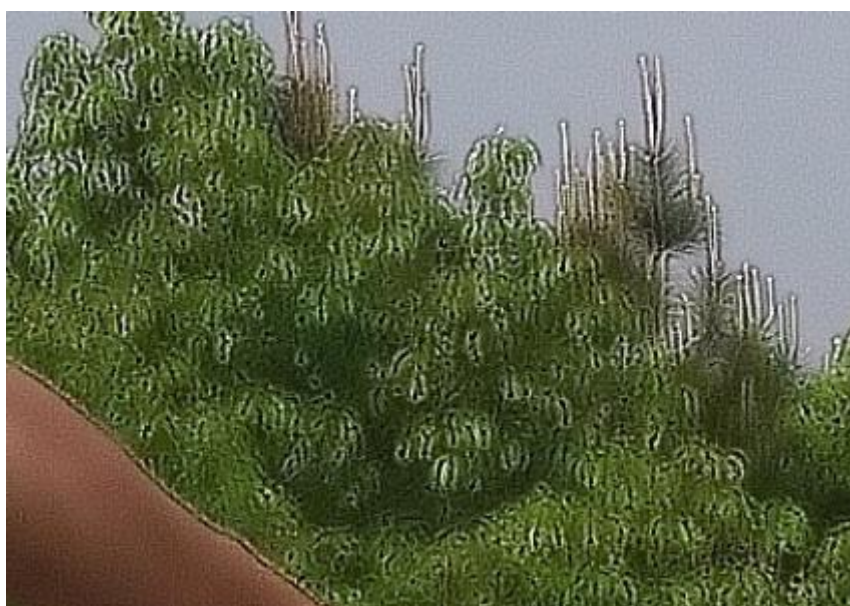


图 2-3 pthread 并行卷积结果图部分截图示意图

此次实验程序执行结果如图 2-4 所示。

```
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.273348s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.263791s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.266681s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.239812s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.272529s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.271503s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab2$ ./lab2
time=1.266070s
```

图 2-4 pthread 并行卷积执行结果图

经过多次执行测试，并行执行时间大概在 1.27s 左右。图片边缘操作效果较为明显。这里执行时间包括从创建线程到线程执行结束的时间。

## 3 实验三

### 3.1 实验目的与要求

- 1) 掌握使用 OpenMP 的并行编程设计和性能优化的基本原理和方法
- 2) 使用 OpenMP 实现图像卷积运算的并行算法
- 3) 进行程序执行结果的简单分析和总结
- 4) 将其与 Lab2 的结果进行比较

### 3.2 算法描述

本次实验采用 OpenMp 来并行实现图像的卷积操作，具体为边缘操作。在顺序执行每一行的像素操作 for 循环上加上 OpenMp 提供的特殊编译声明语句“`#pragma omp parallel for num_threads(10)`”，使得编译器自动将这部分 for 循环编译为 10 个线程的操作，openmp 并行与线程并行相似，要注意循环体内变量的共享性问题，例如在此处 for 循环里面就嵌套的有多个循环，若将内部循环的循环条件变量定义在 for 循环外，就会是所有线程都可以访问的变量，会导致循环条件不正确的问题。剩下的便是等待程序的执行结束，保存程序的执行结果。

### 3.3 实验方案

#### 3.3.1 开发与运行环境

系统: ubuntu 16.04 Desktop

kernel: linux-4.4.0-83-generic

编译器: g++5.4.0

opencv: opencv-2.4.9.1

CPU: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz

内存: 4G+4G

#### 3.3.2 实验步骤

首先引入 opencv 操作环境，方便能够对图片进行处理。

图片读取为 Mat 格式，是现在较新的 opencv 推行的图片格式。

编写 for 循环单线程执行每一个像素边缘操作的卷积操作



在 for 循环开头加上 openmp 特殊多线程编译语句。  
执行程序，查看程序执行打印结果和程序运行结果图。

### 3.4 实验结果与分析

实验原图如图 2-2 所示，实验结果图如图 3-1 所示，图片边缘操作效果正确而且明显，说明 openmp 并行卷积操作运行流程正确。



图 3-1 openmp 并行卷积结果图部分截图示意图

此次实验程序执行结果如图 3-2 所示。

```
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.263054s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.248930s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.276464s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.264921s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.257090s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.273805s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.258557s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab3$ ./lab3
4160 3120
time=1.256577s
```

图 3-2 openmp 并行卷积执行结果图

经过多次执行测试，并行执行时间大概在 1.25s 左右。对比用 pthread 来实现并行卷积操作，两者对图片的处理效果都是一样的，都表现出了较好的边缘操作处理效果。在耗时方面，这里计算了整个 for 循环执行所需的时间，用 openmp 大约比 pthread 提高了 1.5% 左右，两者相差不大，我想应该是 openmp 实际上也是通过将 for 循环编译成多个线程执行本质上与自己创建线程分配任务相似的原因。



## 4 实验四

### 4.1 实验目的与要求

- 1) 掌握使用 MPI 并行编程设计和性能优化的基本原理和方法
- 2) 使用 MPI 实现图像卷积运算的并行算法
- 3) 进行程序执行结果的简单分析和总结
- 4) 将其与 Lab2 和 Lab3 的结果进行比较

### 4.2 算法描述

本次实验采用 MPI 来并行实现图像的卷积操作，具体为边缘操作。MPI 是将程序分为多个子进程来分别执行计算任务，最后子进程将结果发回给主进程。这里卷积操作的核心部分与 pthread 很像，并行部分是通过检测进程号来判断当前进程应该执行那一部分的计算任务，然后将结果发送至 0 号主进程，共分为 11 个进程，1 个主进程不执行计算任务，10 个子进程按行分配计算任务。剩下的便是等待程序的执行结束，保存程序的执行结果。

算法流程图如下图 4-1 所示：

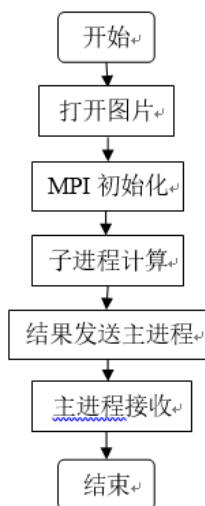


图 4-1 MPI 并行卷积算法流程图

### 4.3 实验方案

#### 4.3.1 开发与运行环境

系统: ubuntu 16.04 Desktop

kernel: linux-4.4.0-83-generic

编译器: mpic++ (g++5.4.0)

opencv: opencv-2.4.9.1

CPU: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz

内存: 4G+4G

运行环境: mpirun-1.10.2

#### 4.3.2 实验步骤

首先引入 opencv 操作环境, 方便能够对图片进行处理。

图片读取为 Mat 格式, 是现在较新的 opencv 推行的图片格式。

MPI 运行环境初始化。

根据进程号判断当前进程所需要计算的图像像素行数。

将计算结果发送给 0 号进程。

0 号进程接收数据结果。

用 mpic++ 编译, 编译命令为“mpic++ lab4.cpp -o lab4 `pkg-config --libs --cflags opencv`”。

用 mpirun 执行程序, 总共分配 11 个进程, 执行命令为“mpirun -np 11 lab4”。

得到实验结果, 并对其进行分析。

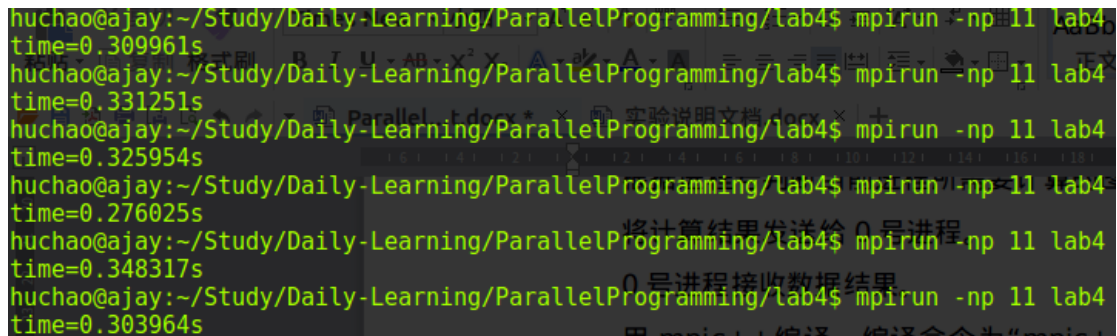
## 4.4 实验结果与分析

实验原图如图 2-2 所示, 实验结果图如图 4-2 所示, 图片边缘操作效果正确而且明显, 说明 MPI 并行卷积操作运行流程正确。



图 4-2 MPI 并行卷积结果图部分截图示意图

此次实验程序执行结果如图 4-3 所示。



```
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab4$ mpirun -np 11 lab4
time=0.309961s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab4$ mpirun -np 11 lab4
time=0.331251s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab4$ mpirun -np 11 lab4
time=0.325954s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab4$ mpirun -np 11 lab4
time=0.276025s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab4$ mpirun -np 11 lab4
time=0.348317s
huchao@ajay:~/Study/Daily-Learning/ParallelProgramming/lab4$ mpirun -np 11 lab4
time=0.303964s
```

图 4-3 MPI 并行卷积执行结果图

经过多次执行测试,并行执行时间大概在 0.3s 左右。对比用 pthread 和 openmp 来实现并行卷积操作,三者对图片的处理效果都是一样的,都表现出了较好的边缘操作处理效果。在耗时方面,计算了从初始化 MPI 环境到接收结果完毕的时间,用 MPI 大约比 pthread 和 openmp 提高了 76%左右。我想应该是进程的并行效果比线程强。

## 5 实验五

### 5.1 实验目的与要求

- 1) 深入了解 GPGPU 的架构，掌握 CUDA 编程模型
- 2) 使用 CUDA 实现图像卷积运算的并行算法
- 3) 进行程序执行结果的简单分析和总结
- 4) 根据执行结果和硬件环境提出优化解决方案
- 5) 将其与 Lab2, Lab3 和 Lab4 的结果进行比较

### 5.2 算法描述

本次实验采用 Cuda 来并行实现图像的卷积操作，具体为边缘操作。Cuda 利用 Gpu 强大的计算能力，来解决实际中计算需求较大的问题。算法的关键在于首先在 Gpu 中分配空间，然后将源数据复制到 Gpu 中，分为 10 个部分，利用 10 个线程来进行计算，最后将结果复制到内存中即可。Cuda 编程有一个核函数，该函数运行在 Gpu 中，与主机函数相分离。剩下的便是等待程序的执行结束，保存程序的执行结果。

算法流程图如下图 5-1 所示：

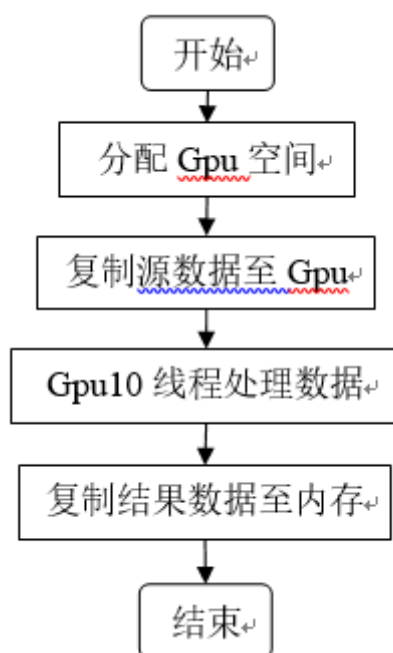


图 5-1 Cuda 并行卷积算法流程图

## 5.3 实验方案

### 5.3.1 开发与运行环境

kernel: linux-4.4.0-83-generic

编译器: nvcc-8.0.44

opencv: opencv-2.4.9

CPU: Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz

运行环境: NVIDIA Corporation GM204 [GeForce GTX 980]

### 5.3.2 实验步骤

按照算法描述编写实验代码。

用 nvcc 编译, 编译命令为“nvcc lab5.cu -o lab5 `pkg-config --libs --cflags opencv`”。

多次直接执行代码, 查看输出时间结果, 以及图片卷积操作结果。

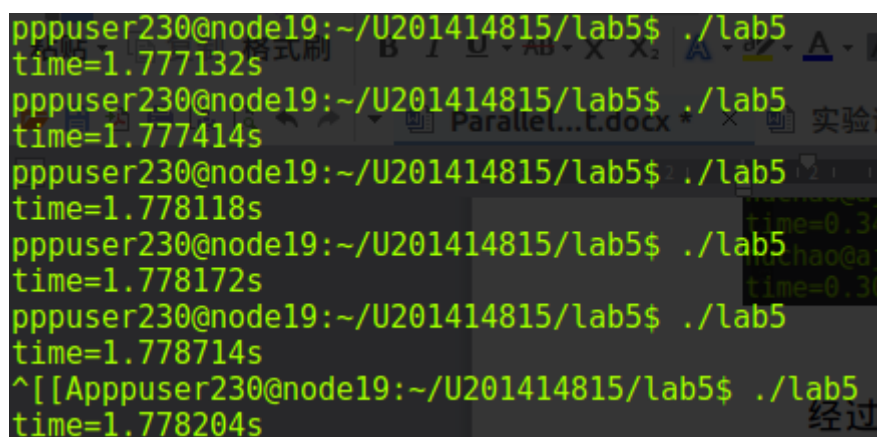
## 5.4 实验结果与分析

实验原图如图 2-2 所示, 实验结果图如图 5-2 所示, 图片边缘操作效果正确而且明显, 说明 Cuda 并行卷积操作运行流程正确。



图 5-2 Cuda 并行卷积结果图部分截图示意图

此次实验程序执行结果如图 5-3 所示。

A terminal window screenshot showing the execution of a program named 'lab5'. The user 'pppuser230' is at 'node19' in the directory '~/U201414815/lab5'. The program is executed multiple times, and the output shows the execution time for each run. The times are: 1.777132s, 1.777414s, 1.778118s, 1.778172s, 1.778714s, and 1.778204s. The terminal also shows some window titles like 'Parallel...t.docx' and '实验'.

```
pppuser230@node19:~/U201414815/lab5$ ./lab5
time=1.777132s
pppuser230@node19:~/U201414815/lab5$ ./lab5
time=1.777414s
pppuser230@node19:~/U201414815/lab5$ ./lab5
time=1.778118s
pppuser230@node19:~/U201414815/lab5$ ./lab5
time=1.778172s
pppuser230@node19:~/U201414815/lab5$ ./lab5
time=1.778714s
^[[Apppuser230@node19:~/U201414815/lab5$ ./lab5
time=1.778204s
```

图 5-3 Cuda 并行卷积执行结果图

经过多次执行测试，并行执行时间大概在 1.77s 左右。对比用 pthread、openmp 和 MPI 来实现并行卷积操作，三者对图片的处理效果都是一样的，都表现出了较好的边缘操作处理效果。在耗时方面，计算了从开始核函数到等待核函数运行结束的时间，用 cuda 比 pthread 并行都更慢，这里应该是计算任务太小，Gpu 计算的其他方面耗时已经超过了它本省计算的时间才会出现这样的现象。



## 6 实验六

### 6.1 实验环境

本次实验在 ubuntu 16.04 LTS 64-bit 桌面版操作系统环境下进行，内存为 7.7GiB，处理器为 Intel® Core™ i5-4210U CPU @ 1.70GHz ×4。

采用的hadoop版本为2.8.0,采用Intelij IDEA 2016.1.3进行代码的编写, JAVA版本为openjdk\_1.8.0\_121。

### 6.2 实验目的

本次基于平台课程的前半部分讲解了开源系统 hadoop 的背景、发展以及使用，这里就要求我们采用 MapReduce 开源系统 Hadoop 实现 SSSP 算法或者 PageRank 算法，来对课堂上的知识进行一个巩固，已达到此次教学的应有的本质目的。

### 6.3 设计思路

#### 1.3.1 mapreduce 介绍

MapReduce 是面向大数据并行处理的计算模型、框架和平台，它隐含了以下三层含义：

1) MapReduce 是一个基于集群的高性能并行计算平台（Cluster Infrastructure）。它允许用市场上普通的商用服务器构成一个包含数十、数百至数千个节点的分布和并行计算集群。

2) MapReduce 是一个并行计算与运行软件框架（Software Framework）。它提供了一个庞大但设计精良的并行计算软件框架，能自动完成计算任务的并行化处理，自动划分计算数据和计算任务，在集群节点上自动分配和执行任务以及收集计算结果，将数据分布存储、数据通信、容错处理等并行计算涉及到的很多系统底层的复杂细节交由系统负责处理，大大减少了软件开发人员的负担。

3) MapReduce 是一个并程序计模型与方法（Programming Model & Methodology）。它借助于函数式程序设计语言 Lisp 的设计思想，提供了一种简便的并程序计方法，用 Map 和 Reduce 两个函数编程实现基本的并行计算任

务，提供了抽象的操作和并行编程接口，以简单方便地完成大规模数据的编程和计算处理。

### 1.3.2 pagerank 算法介绍

一个页面的“得票数”由所有链向它的页面的重要性来决定，到一个页面的超链接相当于对该页投一票。一个页面的 PageRank 是由所有链向它的页面（“链入页面”）的重要性经过递归算法得到的。一个有较多链入的页面会有较高的等级，相反如果一个页面没有任何链入页面，那么它没有等级。

假设一个由 4 个页面组成的小团体：A，B，C 和 D。如果所有页面都链向 A，那么 A 的 PR（PageRank）值将是 B，C 及 D 的 Pagerank 总和。

$$PR(A) = PR(B) + PR(C) + PR(D)$$

继续假设 B 也有链接到 C，并且 D 也有链接到包括 A 的 3 个页面。一个页面不能投票 2 次。所以 B 给每个页面半票。以同样的逻辑，D 投出的票只有三分之一算到了 A 的 PageRank 上。

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}$$

换句话说，根据链出总数平分一个页面的 PR 值。

$$PR(A) = \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)}$$

最后，所有这些被换算为一个百分比再乘上一个系数。由于“没有向外链接的页面”传递出去的 PageRank 会是 0，所以，Google 通过数学系统给了每个页面一个最小值：

$$PR(A) = \left( \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \frac{PR(D)}{L(D)} + \dots \right) d + \frac{1-d}{N}$$

### 1.3.3 算法设计实现

此算法基于 hadoop 的实现主要就是实现一个 Map 函数和一个 Reduce 函数，Map 函数对任务进行细分，而 Reduce 函数讲小任务的结构合并起来，最终当结果趋于稳定时，结束任务。

## 6.4 实验结果

### (1) 用伪分布式打开 hadoop 集群



命令: `./sbin/start-all.sh`

结果:

```
huchao@ajay:~/Study/hadoop-2.8.0$ ./sbin/start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/huchao/Study/hadoop-2.8.0/logs/hadoop-huchao-namenode-ajay.out
localhost: starting datanode, logging to /home/huchao/Study/hadoop-2.8.0/logs/hadoop-huchao-datanode-ajay.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/huchao/Study/hadoop-2.8.0/logs/hadoop-huchao-secondarynamenode-ajay.out
starting yarn daemons
starting resourcemanager, logging to /home/huchao/Study/hadoop-2.8.0/logs/yarn-huchao-resourcemanager-ajay.out
localhost: starting nodemanager, logging to /home/huchao/Study/hadoop-2.8.0/logs/yarn-huchao-nodemanager-ajay.out
huchao@ajay:~/Study/hadoop-2.8.0$ jps
6480 DataNode
6706 SecondaryNameNode
7333 Jps
6887 ResourceManager
6311 NameNode
7020 NodeManager
```

(2) 将网站链接文件上传至 hdfs

命令: `./bin/hadoop fs -put input/test.txt /usr/input`

```
huchao@ajay:~/Study/hadoop-2.8.0$ ./bin/hadoop fs -ls /usr/input/*
-rw-r--r-- 1 huchao supergroup 162 2017-05-26 15:35 /usr/input/test.txt
huchao@ajay:~/Study/hadoop-2.8.0$ ./bin/hadoop fs -cat /usr/input/*
1 8 2 4 5 6 7 8 9
2 10 3 4 7 8 9
3 9 1 4 6 8 10
4 8 2 3 5 8 9 10
5 9 1 2 3 6 8
6 7 2 3 4 7 9
7 5 1 2 3 5 6 8 9
8 6 1 3 5 6 9 10
9 5 2 3 4 5 7
10 8 1 3 4 5 6 8 9
```

(3) 执行打包的 jar 程序

命令: `./bin/hadoop jar lab/pr.jar PageRank`

```
huchao@ajay:~/Study/hadoop-2.8.0$ ./bin/hadoop fs -cat /usr/output/*
1 6.457143
2 13.847619
3 24.238094
4 32.72381
5 39.05714
6 46.65714
7 52.199997
8 62.133327
9 70.86667
10 75.00001
```

## 实验小结

从当初开始选课的时候，我发现这一门是有金海老师教授的课程，所以我毫不犹豫的选择了他。金海老师是谁，绝对是我到这个学院之后听到过次数最多的人名之一，由此可知，其实力绝对是非常之厉害的。

经过将近有一个学期的学习，果然没有让任何人失望，这门课就是那么的优秀，让人沉浸其中。

关于这个实验，让我们了解并行程序的多种方法，比如说 `pthread`、`openmp`、`MPI`、`cuda` 等等，具体的内容都是一些较为容易的实践，让我们能够真正认识并了解并行程序，以及对它所能达到的结果有一个较为清晰的认识。通过这个实验，我觉得我是熟悉了一门的新的编程技巧，也是当前大环境下所不得不会的一门编程技巧，我很感谢这门课。

## 7 附录

### 7.1 pthread 并行卷积源代码

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;
using namespace cv;

struct _con_arg {
    Mat *img;
    Mat *result;
    int start;
    int end;
};

void *convolution(void *con_arg) {
    int i = 0, j = 0, k = 0;
    struct _con_arg *temp = (struct _con_arg *)con_arg;
    const int n = temp->img->channels();
    //printf("start:%d-end:%d\n", temp->start, temp->end);
    for(i = temp->start; i <= temp->end; i++) {
        const uchar *previous = temp->img->ptr<const uchar>(i-1);
        const uchar *current = temp->img->ptr<const uchar>(i);
        const uchar *next = temp->img->ptr<const uchar>(i+1);
        uchar *output = temp->result->ptr<uchar>(i);
        for(j = 1; j < temp->result->cols - 1; j++) {
            for(k = 0; k < n; k++) {
                //锐化操作
                //output[j*n+k] = saturate_cast<uchar>(9*current[j*n+k] - previous[(j-1)*n+k]
                - previous[j*n+k] - previous[(j+1)*n+k] - current[(j-1)*n+k] - current[(j+1)*n+k] - next[(j-1)*n+k]
                - next[j*n+k] - next[(j+1)*n+k]);
                //边缘操作
                output[j*n+k] = saturate_cast<uchar>((-7)*current[j*n+k] +
                previous[(j-1)*n+k] + previous[j*n+k] + previous[(j+1)*n+k] + current[(j-1)*n+k] +
                current[(j+1)*n+k] + next[(j-1)*n+k] + next[j*n+k] + next[(j+1)*n+k]);
            }
        }
    }
}

int main() {
    int i = 0;
    int m = 0;
    clock_t start, end;
    pthread_t t_id[10];
```

```

struct _con_arg con_arg[10];
Mat img = imread("/home/huchao/Study/hehe.jpg");
Mat result;

if(img.empty()) {
    cout<<"open img failed";
    return -1;
}
result.create(img.size(),img.type());
m = ceil((img.rows-2)/10.0);

start = clock();
for(i = 0;i < 10;i++) {
    con_arg[i].img = &img;
    con_arg[i].result = &result;
    con_arg[i].start = 1+i*m;
    if(i == 9) con_arg[i].end = img.rows-2;
    else con_arg[i].end = (i+1)*m;
    if(pthread_create(&t_id[i],NULL,convolution,&con_arg[i])) {
        cout<<"线程创建出错\n";
    }
}

for(i = 0;i < 10;i++) {
    if(pthread_join(t_id[i],NULL) != 0) {
        cout<<"线程出错\n";
    }
}
end = clock();
printf("time=%fs\n",((double)(end-start)/CLOCKS_PER_SEC));

//result.row(0).setTo(Scalar(0,0,0));
//result.row(result.rows-1).setTo(Scalar(0,0,0));
//result.col(0).setTo(Scalar(0,0,0));
//result.col(result.cols-1).setTo(Scalar(0,0,0));

imwrite("/home/huchao/Study/pppp.jpg",result);
return 0;
}

```

## 7.2 openmp 并行卷积源代码

```

#include <omp.h>
#include <time.h>
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace std;
using namespace cv;

int main() {
    int i = 0;
    clock_t start,end;

```

```

Mat img = imread("/home/huchao/Study/hehe.jpg");
Mat result;

if(img.empty()) {
    cout<<"open img failed";
    return -1;
}

result.create(img.size(),img.type());

const int rows = img.rows;
const int cols = img.cols;
const int n = img.channels();
cout<<rows<<"\t"<<cols<<"\n";

start = clock();
#pragma omp parallel for num_threads(10)
for(i = 1;i < rows - 1;i++) {
    const uchar *previous = img.ptr<const uchar>(i-1);
    const uchar *current = img.ptr<const uchar>(i);
    const uchar *next = img.ptr<const uchar>(i+1);
    uchar *output = result.ptr<uchar>(i);
    for(int j = 1;j < cols - 1;j++) {
        for(int k = 0; k < n; k++) {
            //锐化操作
            output[j*n+k] = saturate_cast<uchar>(9*current[j*n+k] - previous[(j-1)*n+k]
- previous[j*n+k] - previous[(j+1)*n+k] - current[(j-1)*n+k] - current[(j+1)*n+k] - next[(j-1)*n+k]
- next[j*n+k] - next[(j+1)*n+k]);
            //边缘操作
            output[j*n+k] = saturate_cast<uchar>((-7)*current[j*n+k] +
previous[(j-1)*n+k] + previous[j*n+k] + previous[(j+1)*n+k] + current[(j-1)*n+k] +
current[(j+1)*n+k] + next[(j-1)*n+k] + next[j*n+k] + next[(j+1)*n+k]);
        }
    }
    //printf("rows:%d\n",i);
}
end = clock();
printf("time=%fs\n",((double)(end-start)/CLOCKS_PER_SEC));

//result.row(0).setTo(Scalar(0,0,0));
//result.row(result.rows-1).setTo(Scalar(0,0,0));
//result.col(0).setTo(Scalar(0,0,0));
//result.col(result.cols-1).setTo(Scalar(0,0,0));

imwrite("/home/huchao/Study/pppp.jpg",result);

return 0;
}

```

## 7.3 MPI 并行卷积源代码

```

#include <stdlib.h>
#include <stdio.h>

```

```

#include <time.h>
#include <mpi.h>
#include <opencv2/opencv.hpp>
using namespace cv;

void convolution(Mat *img, Mat *result, int start, int end) {
    int i = 0, j = 0, k = 0;
    const int n = img->channels();
    //printf("start:%d-end%d\n", start, end);
    for(i = start; i <= end; i++) {
        const uchar *previous = img->ptr<const uchar>(i-1);
        const uchar *current = img->ptr<const uchar>(i);
        const uchar *next = img->ptr<const uchar>(i+1);
        uchar *output = result->ptr<uchar>(i);
        for(j = 1; j < result->cols-1; j++) {
            for(k = 0; k < n; k++) {
                //锐化操作
                //output[j*n+k] = saturate_cast<uchar>(9*current[j*n+k] - previous[(j-1)*n+k]
- previous[j*n+k] - previous[(j+1)*n+k] - current[(j-1)*n+k] - current[(j+1)*n+k] - next[(j-1)*n+k]
- next[j*n+k] - next[(j+1)*n+k]);
                //边缘操作
                output[j*n+k] = saturate_cast<uchar>((-7)*current[j*n+k] +
previous[(j-1)*n+k] + previous[j*n+k] + previous[(j+1)*n+k] + current[(j-1)*n+k] +
current[(j+1)*n+k] + next[(j-1)*n+k] + next[j*n+k] + next[(j+1)*n+k]);
            }
        }
    }
}

int main(int argc, char *argv[]) {
    int my_rank = 0, comm_sz = 0;
    int i = 0, m = 0, count1 = 0, count2 = 0, count3 = 0;
    clock_t start, end;

    Mat img = imread("/home/huchao/Study/hehe.jpg");
    Mat temp, result;

    if(img.empty()) {
        printf("open image failed\n");
        return -1;
    }
    result.create(img.size(), img.type());
    temp.create(img.size(), img.type());
    m = (img.rows-2)/10;

    count1 = img.cols*sizeof(uchar)*3;
    count2 = m*count1;
    count3 = (img.rows-2-9*m)*count1;

    start = clock();
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    if(my_rank != 0) {
        if(my_rank == 10) {
            convolution(&img, &temp, 1+(my_rank-1)*m, img.rows-2);

```

```

        MPI_Send(temp.data
+(1+(my_rank-1)*m)*count1,count3,MPI_CHAR,0,my_rank,MPI_COMM_WORLD);
        //printf("send%d\n",my_rank);
    }
    else {
        convolution(&img,&temp,1+(my_rank-1)*m,my_rank*m);
        MPI_Send(temp.data
+(1+(my_rank-1)*m)*count1,count2,MPI_CHAR,0,my_rank,MPI_COMM_WORLD);
        //printf("send%d\n",my_rank);
    }
}
else if(my_rank == 0) {
    for(i = 0;i <= 8;i++) {

        MPI_Recv(result.data+(1+i*m)*count1,count2,MPI_CHAR,i+1,i+1,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        //printf("recv%d\n",i+1);
    }

    MPI_Recv(result.data+(1+i*m)*count1,count3,MPI_CHAR,i+1,i+1,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    //printf("recv%d\n",i+1);
    end = clock();
    printf("time=%fs\n",((double)(end-start)/CLOCKS_PER_SEC));
    imwrite("/home/huchao/Study/pppp.jpg",result);
}

MPI_Finalize();

return 0;
}

```

## 7.4 Cuda 并行卷积源代码

```

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <time.h>
#include <opencv2/opencv.hpp>
using namespace cv;

void conWithCuda(const Mat *img, Mat *result, size_t size);

__global__ void conKernel(const uchar *img, uchar *result, const int *_size, const int *_rows,
const int *_cols) {
    int t = threadIdx.x;
    int i = 0,j = 0,k = 0,n = 3;
    int size = *_size;
    int rows = *_rows;
    int cols = *_cols;
    int start = 1 + t * (rows-2) / size;
    int end = (t+1 == size) ? rows-2 : (t+1)*(rows-2)/size;
    int temp = 0;

```

```

    for(i = start; i <= end; i++) {
        const uchar *previous = img+(i-1)*cols*3;
        const uchar *current = img+i*cols*3;
        const uchar *next = img+(i+1)*cols*3;
        uchar *output = result+i*cols*3;
        for(j = 1; j < cols-1; j++) {
            for(k = 0; k < n; k++) {
                //锐化操作
                //output[j*n+k] = saturate_cast<uchar>(9*current[j*n+k] - previous[(j-1)*n+k]
- previous[j*n+k] - previous[(j+1)*n+k] - current[(j-1)*n+k] - current[(j+1)*n+k] - next[(j-1)*n+k]
- next[j*n+k] - next[(j+1)*n+k]);
                //边缘操作
                temp = (-7)*current[j*n+k] + previous[(j-1)*n+k] + previous[j*n+k] +
previous[(j+1)*n+k] + current[(j-1)*n+k] + current[(j+1)*n+k] + next[(j-1)*n+k] + next[j*n+k] +
next[(j+1)*n+k];
                if(temp < 0) output[j*n+k] = 0;
                else if(temp > 255) output[j*n+k] = 255;
                else output[j*n+k] = (uchar)temp;
            }
        }
    }
}

```

```

int main() {
    int size = 10;
    Mat img = imread("hehe.jpg");
    Mat result;

    if(img.empty()) {
        printf("open image failed\n");
        return -1;
    }
    result.create(img.size(), img.type());

    // convolution in parallel.
    conWithCuda(&img, &result, size);
    imwrite("pppp.jpg", result);
    // cudaThreadExit must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaThreadExit();
    return 0;
}

```

// Helper function for using CUDA to convolution in parallel.

```

void conWithCuda(const Mat *img, Mat *result, size_t size) {
    clock_t start, end;
    int *dev_size = 0;
    int *dev_rows = 0;
    int *dev_cols = 0;
    uchar *dev_img = 0;
    uchar *dev_result = 0;
    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaSetDevice(0);
    // Allocate GPU buffers for three vectors (two input, one output)
    cudaMalloc((void**)&dev_size, sizeof(int));
    cudaMalloc((void**)&dev_rows, sizeof(int));
    cudaMalloc((void**)&dev_cols, sizeof(int));
}

```



```

    cudaMalloc((void**)&dev_img, img->rows * img->cols * sizeof(uchar) * 3);
    cudaMalloc((void**)&dev_result, img->rows * img->cols * sizeof(uchar) * 3);
    // Copy input vectors from host memory to GPU buffers.
    cudaMemcpy(dev_size, &size, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_rows, &(img->rows), sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_cols, &(img->cols), sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_img, img->ptr<uchar>(0), img->rows * img->cols * sizeof(uchar) * 3,
cudaMemcpyHostToDevice);
    start = clock();
    // Launch a kernel on the GPU with one thread for each element.
    conKernel<<<1, size>>>>(dev_img, dev_result, dev_size, dev_rows, dev_cols);
    // cudaThreadSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaThreadSynchronize();
    // Copy output vector from GPU buffer to host memory.
    end = clock();
    cudaMemcpy(result->ptr<uchar>(0), dev_result, img->rows * img->cols * sizeof(uchar) * 3,
cudaMemcpyDeviceToHost);
    cudaFree(dev_size);
    cudaFree(dev_rows);
    cudaFree(dev_cols);
    cudaFree(dev_img);
    cudaFree(dev_result);
    printf("time=%fs\n",((double)(end-start)/CLOCKS_PER_SEC));
}

```

## 7.5 PageRank 源代码

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;
import java.util.StringTokenizer;

public class PageRank {
    public static class Map extends Mapper<Object,Text,IntWritable,FloatWritable> {
        private final IntWritable word = new IntWritable();
        private String pr;
        public void map(Object key,Text value,Context context) throws IOException,
InterruptedException{
            StringTokenizer itr = new StringTokenizer(value.toString());
            if(itr.hasMoreTokens()) {String id = itr.nextToken();}
            else return;
            pr = itr.nextToken();
            int count = itr.countTokens();
            float average_pr = Float.parseFloat(pr)/count;
            while(itr.hasMoreTokens()){
                word.set(Integer.parseInt(itr.nextToken()));
            }
        }
    }
}

```

```

        context.write(word, new FloatWritable(average_pr));
    }
}

public static class Reduce extends
Reducer<IntWritable,FloatWritable,IntWritable,FloatWritable> {
    float sum;
    public void reduce(IntWritable key,Iterable<FloatWritable>values,Context context)
throws IOException, InterruptedException{
        for(FloatWritable val:values){
            sum += val.get();
        }
        context.write(key,new FloatWritable(sum));
    }
}

public static void main(String[] args) throws IOException, ClassNotFoundException,
InterruptedException {
    // TODO Auto-generated method stub
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "pagerank");
    job.setJarByClass(PageRank.class);
    job.setNumReduceTasks(1);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(FloatWritable.class);

    FileInputFormat.addInputPath(job, new Path("/usr/input"));
    FileOutputFormat.setOutputPath(job, new Path("/usr/output"));

    System.exit(job.waitForCompletion(true)? 0 : 1);
}
}

```