華中科技大學

课 程 实 验 报 告

课程名称：并行编程原理与实践

院　　系：计算机科学与技术学院

姓　　名：陈　薇

专业班级：信息安全 1503 班

学　　号：U201514858

指导教师：金　海

报告日期：2018 年 7 月 14 日

1.实验一

1.1 实验目的与要求

实验目的：熟悉并行开发环境，掌握并行编程的基本原理和方法，了解 Linux 系统下 pthread、OpenMP 和 OpenMPI 等工具和框架的优化性能。

实验要求：使用最简单的任务划分方法——每个线程（进程）完成循环体中一次循环的工作，共有 n 个线程同时计算，从而实现对基本向量加法程序的优化。向量加法程序如下所示：

```
for(int i = 0;i < n; i ++)
    C[i] = A[i] + B[i];
```

1.2 实验内容

1.2.1 使用 pthread 做向量加法

算法描述：

```
i = 0, j = 0;
for i < n pthread_create;   //创建线程,并将 i 传递给线程函数 plus_pthread
for j < n pthread_join;     //等待线程 j 结束
```

定义三个全局变量 vector_a[]、vector_b[]和 vector_result[]分别表示相加向量和结果向量，线程函数 plus_pthread 做 vector_result[i] = vector_a[i]+vector_b[i]加法操作。

1.2.2 使用 OpenMP 做向量加法

使用特殊的编译引导语句，OpenMP 会自动将 for 循环分解为多个线程，源程序修改成如下形式，即可实现 OpenMP 并行加速：

```
#pragma omp parallel for
for(i = 0; i < 10; ++i)
    vector_result[i] = vector_a[i] + vector_b[i];
```

1.2.3 使用 OpenMPI 做向量加法

向量加法可以看成是一对多的通信机制，因此采用 MPI_Scatter 散发机制实现进程间通信。算法描述如下：

```
MPI_Init(&argc, &argv); //初始化，启动 MPI 环境
MPI_Comm_rank(MPI_COMM_WORLD, &rankID);  //获取进程标识符
MPI_Comm_size(MPI_COMM_WORLD, &totalNumTasks);//获取进程数
MPI_Scatter(sendBuf, sendCount, MPI_FLOAT,
            recvBuf, recvCount, MPI_FLOAT, source, MPI_COMM_WORLD);
MPI_Finalize();//结束 MPI 环境
```

MPI_Scatter()函数接口中，sendBuf 表示发送缓冲区，即定义的由两个 n*1 维的向量所组成的 n*2 维的矩阵数组，每行两个元素表示要执行加法操作的数据，sendCount 表示发送数据时的数据块的大小，MPI_FLOAT 表示发送的数据类型，recvBuf 表示接收缓冲区，recvCount 表示接收数据时的数据块大小，source 表示根进程的进程号。在使用 mpirun 运行程序时，–np 参数大小为向量长度 n。

1.2.4 使用 CUDA 做向量加法

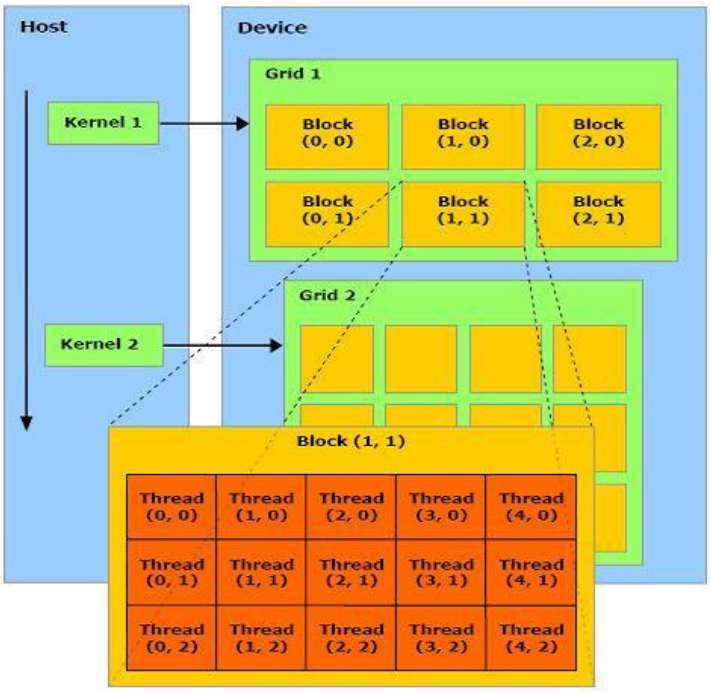

图 1.1 CUDA 内部机制

本实验中，定义了四个的向量 host_a、host_b、host_c 和 host_c2，分别表示主机端的 A、B 和 C 向量，host_c2 是串行计算结果，用于检验并行计算结果是否正确。Kernel 函数配置如下：

```
#define BLOCKSIZE 4
int gridsize = (int)ceil(sqrt(ceil(n / (BLOCKSIZE * BLOCKSIZE))));
dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
dim3 dimGrid(gridsize, gridsize, 1);
add<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
```

初始化 dimBlock 为 4*4*1 的 dim3 类型，执行线程块的三个维度，这里第三维是 1，即退化为 4*4 的二维线程块。为了最大化并行，安排每一个线程负责一次向量加法，那么需要$\text{blockDim} = \left\lceil \frac{n}{\text{BLOCKSIZE}*\text{BOLCKSIZE}} \right\rceil$个线程块，即 block 的维数大小。设置线程网络 grid，grid 大小为$\text{gridDim} = \left\lceil \sqrt{\left\lceil \frac{n}{\text{BLOCKSIZE}*\text{BOLCKSIZE}} \right\rceil} \right\rceil$，即 grid 的维度，Grid 只能是二维以下，第三个维度设置默认忽略。设置中采用向上取整是为了保证至少有一个线程完成向量每对元素的相加，那么这样设置可能会导致线程数多于向量长度，因此在 Kernel 函数中需要让这些线程直接退出，避免数组下标越界。

将线程块号为 blockIdx、线程号为 threadIdx 的线程映射到向量计算的数组下标：

```
块内地址：threadIdx.y *blockDim.x + threadIdx.x
块内地址区间：[0, blockDim.x*blockDim.y-1]
线程块地址：blockIdx.y*gridDim.x+blockIdx.x
线程块地址区间：[0, gridDim.x*gridDim.y-1]
因此线程号为 threadIdx 对应的数组下标为：
i = (blockIdx.y*gridDim.x+blockIdx.x)* blockDim.x* blockDim.y
                        +( threadIdx.y *blockDim.x + threadIdx.x)
```

因此，向量加法 Kernel 函数中，先计算出线程操作数组下标 i，若 i < n 则计算，否则该线程直接退出。Kernel 函数定义如下：

```
__global__ void add(const int *a, const int *b, int *c, int n) {
    int i = (blockIdx.x * gridDim.x + blockIdx.y) * blockDim.x * blockDim.y
+ threadIdx.x * blockDim.x + threadIdx.y;
    if (i < n)    c[i] = a[i] + b[i];
}
```

程序流程图如图 1-2 所示，先将数据从主机内存拷贝到 GPU 内存设备上，然后主机调用向量加法 Kernel 函数让设备异步并行执行，由于 CPU 启动的 Kernel 函数是异步的，并不会阻塞等到 GPU 执行完 kernel 才执行后续的 CPU 部分，因此显示设置同步障来阻塞 CPU 程序。最后验证执行结果，统计执行时间。

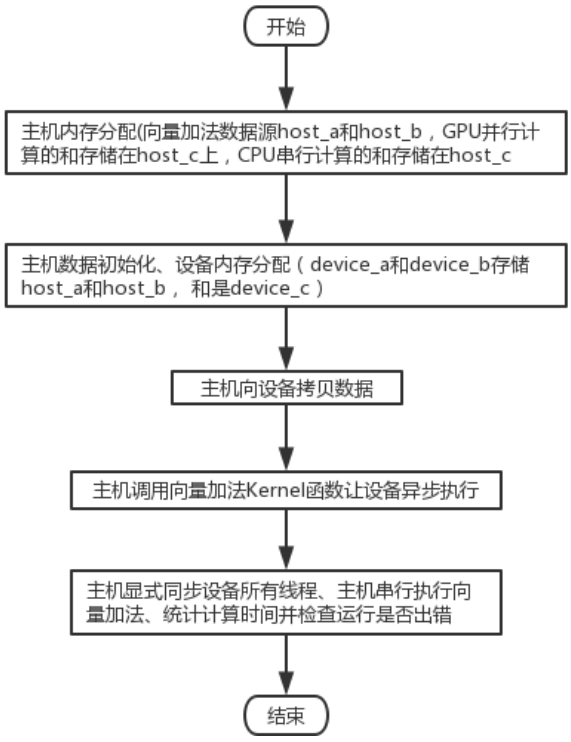

图 1.2 CUDA 做向量加法算法流程图

1.3 实验结果

1.3.1 pthread

1、代码流程

设置向量 vector_a 和 vector_b 的维度为 10，首先利用 rand()%100 随机生成向量的内容为小于 100 的正整数，随后循环创建线程执行 plus_thread 函数进行一次加法操作，传入索引 i，并将线程 id 存放在 tid 数组中，最后调用

pthread_join 函数，传入线程 id，循环等待线程结束，代码如下：

```
1.  //头文件
2.  int vector_a[10], vector_b[10], vector_result[10];
3.  //执行加法的线程函数
4.  void *plus_pthread(void *arg) {
5.      int i = *(int *)arg;
6.      vector_result[i] = vector_a[i] + vector_b[i];
7.    //输出加法结果
8.  }
9.  int main() {
10.     int ret, i;
11.     pthread_t tid[10];
12.     int attr[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
13.     for (i = 0; i < 10; i++) {
14.         vector_a[i] = rand() % 100;
15.         vector_b[i] = rand() % 100;
16.     }
17.     for (i = 0; i < 10; i++) {
18.         ret = pthread_create(&tid[i], NULL, &plus_pthread, &attr[i]);
19.     }
20.     for (i = 0; i < 10; i++) {
21.         pthread_join(tid[i], NULL);
22.     }
23. //输出向量结果
24. }
```

值得注意的是，起初直接将 i 的地址作为 pthread_create 函数的参数传递至线程 plus_thread，但由于循环过程中 i 的值在不断变化，导致某些向量的加法未执行 i 的值已加一，因此传入 attr 数组，保证每个元素会都被执行加法操作。

2、运行结果

头文件：pthread.h

编译：gcc –o lab1_1 lab1_1.c –lpthread

运行：./lab1_1

测试结果如图 1.3 所示。

```
[parallel_exp@node56 exp1]$ gcc -o lab1_1 lab1_1.c -lpthread
[parallel_exp@node56 exp1]$ ./lab1_1
Process 0: vector_result[0] = vector_a[0] + vector_b[0]
Process 4: vector_result[4] = vector_a[4] + vector_b[4]
Process 2: vector_result[2] = vector_a[2] + vector_b[2]
Process 3: vector_result[3] = vector_a[3] + vector_b[3]
Process 1: vector_result[1] = vector_a[1] + vector_b[1]
Process 5: vector_result[5] = vector_a[5] + vector_b[5]
Process 6: vector_result[6] = vector_a[6] + vector_b[6]
Process 7: vector_result[7] = vector_a[7] + vector_b[7]
Process 8: vector_result[8] = vector_a[8] + vector_b[8]
Process 9: vector_result[9] = vector_a[9] + vector_b[9]
The Vector_a is: 83 77 93 86 49 62 90 63 40 72
The Vector_b is: 86 15 35 92 21 27 59 26 26 36
The Vector_result is: 169 92 128 178 70 89 149 89 66 108
[parallel_exp@node56 exp1]$
```

图 1.3 pthread 方法运行截图

图中可以看到一共创建了 10 个进程，每个线程分别做了一次加法运算，由于线程并行，所以打印的结果随机，对比计算结果可知计算结果正确。

1.3.2 OpenMP 方法

1、代码流程

同 pthread 实验，本实验的向量维度为 10，值是随机产生的。实现并行处理只需在 for 循环前添加编译引导语句，会自动将 for 循环分解为多个线程并行计算，核心代码如下：

```
1.  #pragma omp parallel for
2.      for (i = 0; i < 10; i++) {
3.          vector_result[i] = vector_a[i] + vector_b[i];
4.          printf("Process %d: vector_result[%d] = vector_a[%d] +
    vector_b[%d]\n", omp_get_thread_num(), i, i, i);
5.      }
```

2、运行结果

头文件：omp.h

编译：gcc –o lab1_2_1 lab1_2_1.c –fopenmp

运行：./Lab1_2

运行结果如图 1.4 所示。

```
[parallel_exp@node55 exp2]$ gcc -o lab1_2_1 lab1_2_1.c -fopenmp
[parallel_exp@node55 exp2]$ ls
lab1_2_1  lab1_2_1.c  lab1_2_2.c  lab1_2_3.c
[parallel_exp@node55 exp2]$ ./lab1_2_1
Process 0: vector_result[0] = vector_a[0] + vector_b[0]
Process 6: vector_result[6] = vector_a[6] + vector_b[6]
Process 5: vector_result[5] = vector_a[5] + vector_b[5]
Process 3: vector_result[3] = vector_a[3] + vector_b[3]
Process 1: vector_result[1] = vector_a[1] + vector_b[1]
Process 4: vector_result[4] = vector_a[4] + vector_b[4]
Process 2: vector_result[2] = vector_a[2] + vector_b[2]
Process 9: vector_result[9] = vector_a[9] + vector_b[9]
Process 8: vector_result[8] = vector_a[8] + vector_b[8]
Process 7: vector_result[7] = vector_a[7] + vector_b[7]
The Vector_a is: 83 77 93 86 49 62 90 63 40 72
The Vector_b is: 86 15 35 92 21 27 59 26 26 36
The Vector result is: 169 92 128 178 70 89 149 89 66 108
```

图 1.4 Openmp 运行结果

为显示 Openmp 的并行效果，分别计算串行加法和并行加法的时间，利用 gettimeofday 函数记录加法开始前的时间和开始后的时间，差值即为运行时间，代码如下：

```
1. gettimeofday(&start, NULL);
2. for (i = 0; i < 10; i++) {
3.     vector_result_serial[i] = vector_a[i] + vector_b[i];
4. }
5. gettimeofday(&end, NULL);
6. serial_timeuse = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec -
   start.tv_usec;
7. serial_timeuse /= 1000;
```

编译运行结果显示如下图，可以看到此时由于计算量较小，出现负优化，由于并行过程中线程的开启和销毁占用大部分时间，因此并行计算运行的时间反而过长。

```
[parallel_exp@node55 exp2]$ ./lab1_2_2
The Vector_a is: 83 77 93 86 49 62 90 63 40 72
The Vector_b is: 86 15 35 92 21 27 59 26 26 36
The Vector_result is: 169 92 128 178 70 89 149 89 66 108
The Openmp const time is 0.604000 ms
The Serial const time is 0.001000 ms
```

图 1.5 串行和并行时间（维度为 10）

考虑把向量长度 n 增加为 100000，计算结果如图 1.6 所示。虽然已经增大了 n 的级数，但是多次运行的结果可以发现二者执行速度差别很小，说明仅做一次简单的 for 循环，OpenMP 的加速情况并不是特别明显。

```
[parallel_exp@node55 exp2]$ gcc -o lab1_2_3 lab1_2_3.c -fopenmp
[parallel_exp@node55 exp2]$ ./lab1_2_3

The Openmp const time is 0.848000 ms
The Serial const time is 1.092000 ms
[parallel_exp@node55 exp2]$
```

图 1.6 串行和并行运行时间（维度为 100000）

1.3.3 OpenMPI 方法

1、代码流程

OpenMPI 从一开始即为多进程运行，共有 size 个进程，不同程序之间根据 rank 值进行区分，一般 0 号进程为主进程。本程序中，0 号进程负责初始化维度为 n 的向量 vector_a 和 vector_b，由于数组只在 0 号进程中进行赋值，若要其他进程参与计算，需要将操作数发送至其他进程。

实验中利用 MPI_Scatter 函数，将要参与加法运算的操作数分发给每个进程，0 号进程利用向量的值初始化发送缓冲区 sendBuf，该缓冲区利用 n 行 2 列的数组矩阵表示两个向量，其余进程将接收到的数据块存放至 recvBuf 中，每个数据块包含同行向量元素，进程执行加法操作后输出结果。

若要得到计算后的 vector_result，还需将各进程的计算结果进行汇总，因为每个进程仅将自己进程空间中指定 rank 处元素赋值。此处仅输出每个进程的计算结果，核心代码如下：

```
1.  MPI_Scatter(sendBuf, 2, MPI_INT, recvBuf, 2, MPI_INT, 0, MPI_COMM_WORLD);
2.  /*结果输出*/
3.  printf("-------------------------------------\nrank=%d\n", rank);
4.  printf("vector_a[%d] = %d  vector_b[%d] = %d\n", rank, recvBuf[0], rank,
recvBuf[1]);
5.  vector_result[rank] = recvBuf[0] + recvBuf[1];
6.  printf("vector_result[%d] = %d\n", rank, vector_result[rank]);
7.  printf("-------------------------------------\n");
```

2、运行结果

头文件：mpi.h

编译：mpicc –o lab1_3 lab1_3.c

运行：mpirun –np 4 ./lab1_3

运行效果如图 1-5 所示，运行进程的个数 size 是由运行时-np 后的数字决定的，此处为 4，说明两个向量为 4 维，每个进程执行一次加法操作，可以看到进程的先后顺序是不固定的，并且计算结果正确。

```
[parallel_exp@node55 exp3]$ mpicc -o lab1_3 lab1_3.c
[parallel_exp@node55 exp3]$ mpirun -np 4 ./lab1_3
The Vector_a is: 83 77 93 86
The Vector_b is: 86 15 35 92
------------------------------------------------
rank=0
vector_a[0] = 83  vector_b[0] = 86
vector_result[0] = 169
------------------------------------------------
------------------------------------------------
rank=1
vector_a[1] = 77  vector_b[1] = 15
vector_result[1] = 92
------------------------------------------------
------------------------------------------------
rank=3
vector_a[3] = 86  vector_b[3] = 92
vector_result[3] = 178
------------------------------------------------
------------------------------------------------
rank=2
vector_a[2] = 93  vector_b[2] = 35
vector_result[2] = 128
------------------------------------------------
```

图 1.7 OpenMPI 方法计算向量加法

1.3.4 CUDA 方法

1、代码流程

由于一个 GPU 线程计算一次向量加法，因此规定第 i 个线程执行向量中第 i 项的加法。由于 GPU 内存和 CPU 不同，因此在借助 GPU 运算时需要将数据从 CPU 内存空间拷贝至 GPU 内存空间，并在计算结束后再次拷贝回 CPU 内存空间以做剩余处理，涉及到的 CUDA 函数如下：

```
1.  //在 GPU 内存上分配空间
2.  error = cudaMalloc((void **)&device_a, sizeof(int) * n);
3.  ...
4.  //将数据从 CPU 内存拷贝至 GPU 内存
5.  cudaMemcpy(device_a, host_a, sizeof(int) * n, cudaMemcpyHostToDevice);
6.  ...
7.  //kernel 函数
8.  add<<<dimGrid, dimBlock>>>(device_a, device_b, device_c, n);
9.  //线程同步
10. cudaThreadSynchronize();
11. ...
12. //将数据从 GPU 内存空间拷贝至 CPU 内存
13. cudaMemcpy(host_c, device_c, sizeof(int) * n, cudaMemcpyDeviceToHost);
```

2、运行结果

头文件：cuda.h

编译：nvcc –o lab1_4 lab1_4.cu

运行：./lab1_4

在图 1.8 中，设置向量长度默认长度为 512，若有来自命令行的输入则向量长度为命令行参数，块大小 blocksize=16，验证计算结果正确，但是起初执行效率远不如 CPU 线性执行，随着数据量的增大，CUDA 方法的计算效率逐渐增加，最终在$n = 10^8$时效率超过了 CPU 串行。

```
[parallel_exp@node333 exp4]$ nvcc -o lab1_4 lab1_4.cu
[parallel_exp@node333 exp4]$ ./lab1_4
The Cuda const time is 0.183000 ms
The Serial const time is 0.005000 ms
Successfully run on GPU and CPU!
[parallel_exp@node333 exp4]$ ./lab1_4 10000
The Cuda const time is 0.300000 ms
The Serial const time is 0.182000 ms
Successfully run on GPU and CPU!
[parallel_exp@node333 exp4]$ ./lab1_4 1000000
The Cuda const time is 8.501000 ms
The Serial const time is 4.362000 ms
Successfully run on GPU and CPU!
[parallel_exp@node333 exp4]$ ./lab1_4 100000000
The Cuda const time is 377.149000 ms
The Serial const time is 406.022000 ms
Successfully run on GPU and CPU!
[parallel_exp@node333 exp4]$
```

图 1.8 CUDA 方法计算向量加法,blocksize=16

2.实验二

2.1 实验目的与要求

1、掌握使用 pthread 的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、对程序执行结果进行简单的分析和总结。

2.2 算法描述

本次所有实验均实现二值图像的腐蚀操作，结构元素为 3×3 的矩阵，矩阵内容如下，其中中心点为 origin 点，因此在进行卷积操作时，判断中心点的上下左右四个像素点是否均为 1 即可。若符合条件，则保留中心点的像素值为 1（255），否则设置中心点的像素值为 0。

0	1	0
1	1	1
0	1	0

CPU 方法：（串行）

程序首先调用 imread 读取图片，并调用 threshold 函数将原图其转化为二值图 binary_dst，即图片中像素点的值只有 0（黑色）或者 255（白色）。然后对除边界点外的每个像素点，取其上下左右四个点和像素值和该点的像素值，判断是否均为 255 即非 0，若是则将目标图片 erosion_dst 的该点像素值设置为 255，否则该点像素值设置为 0。对于目标图片的边界点，直接将其像素值设置为 0，程序结束。核心代码如下，set_pixel 和 get_pixel 函数分别设置和获取指定位置的像素值：

```
1.  for (int i = 0; i < binary_dst.rows; i++) {
2.      for (int j = 0; j < binary_dst.cols; j++) {
3.          if((i == 0) || (i == binary_dst.rows - 1)
4.           || (j == 0) || (j == binary_dst.cols - 1)){
5.              set_pixel(erosion_dst, i, j, 0);
6.              continue;
7.          }
8.          int origin = get_pixel(binary_dst, i, j);
9.          int upper = get_pixel(binary_dst, i, j - 1);
```

```
        int left = get_pixel(binary_dst, i - 1, j);
        int lower = get_pixel(binary_dst, i, j + 1);
        int right = get_pixel(binary_dst, i + 1, j);
        if (upper && origin && left && lower && right) {
            set_pixel(erosion_dst, i, j, 255);
        }
        else {
            set_pixel(erosion_dst, i, j, 0);
        }
    }
}
```

Pthread 方法：（并行）

程序执行逻辑同串行方法，只是在 for 循环处考虑开启多个线程处理。假设图片为 row 行 col 列的矩阵，开启 n 个线程，则每个线程需要处理 row*col/n 个像素点。以行为单位进行分割，因此每个进程处理 row/n 行像素。设计 n 以命令行参数的形式传入程序，主进程中创建线程和销毁线程的代码如下：

```
for (int i = 0; i < n; i++) {
    attr[i] = i;
    pthread_create(&tid[i], NULL, &Erosion, &attr[i]);
}
for(int i = 0; i < n; i++){
    pthread_join(tid[i], NULL);
}
```

线程函数 Erosion 负责按行处理各像素点，for 循环的核心代码如下，每个线程处理 binary_dst.rows*k/n - binary_dst.rows*(k+1)/n 范围内的像素点。

```
int k = *(int *)args;
int step = ceil(binary_dst.rows / n);
for (int i = k * step; (i < binary_dst.rows) && (i < (k * step + step));
i++) {
    for (int j = 0; j < binary_dst.cols; j++) {
    ...
    }
}
```

for 循环体的语句同串行计算，只是循环范围由线程数 n 的改变而改变。为判断并行加速的效果，分别输出串行计算和并行计算消耗的时间，并且，为校验并行计算的正确性，将并行计算的结果同串行计算结果逐像素点对比，输出对比结果。计算时间的代码如下，结果以 ms 为单位显示：

```
gettimeofday(&start, NULL);
for (int i = 0; i < binary_dst.rows; i++) {
```

```
3.      for (int j = 0; j < binary_dst.cols; j++) {
4.      ...
5.      }
6.  }
7.  gettimeofday(&end, NULL);
8.  serial_timeuse = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec -
    start.tv_usec;
9.  serial_timeuse /= 1000;
10. printf("The Serial const time is %f ms\n", serial_timeuse);
```

2.3 实验方案

开发环境：ubuntu16.04 虚拟机+visual studio code+opencv3.2.0

运行环境：ubuntu16.04 虚拟机+opencv3.2.0

2.4 实验结果与分析

Opencv 程序的运行需要使用 cmake 生成 Makefile 文件，而后利用 make 生成可执行文件，CmakeLists 内容如下，包含生成可执行文件的名称、依赖链接库等信息：

```
1   cmake_minimum_required(VERSION 2.8)
2   project( lab2 )
3   find_package( OpenCV REQUIRED )
4   find_package( Threads REQUIRED )
5   add_executable( lab2 lab2_pthread.cpp )
6   target_link_libraries( lab2 Threads::Threads)
7   target_link_libraries( lab2 ${OpenCV_LIBS} )
8   set(THREADS_PREFER_PTHREAD_FLAG ON)
```

图 2.1 CmakeLists.txt 文件内容

编译生成可执行程序 lab2，传入并行的线程数 4，运行结果如下图 2.2，控制台显示串并行的运行时间以及并行结果正确的提示，同时弹出二值图、串行腐蚀处理后的图像和并行腐蚀处理后的图像，如图 2.3 所示，直观效果为腐蚀后图像字母变瘦，边界点被消除。

```
→  lab2 ls
CMakeCache.txt  cmake_install.cmake  lab2.cpp          Makefile
CMakeFiles      CMakeLists.txt       lab2_pthread.cpp  test.png
→  lab2 make
Scanning dependencies of target lab2
[ 50%] Building CXX object CMakeFiles/lab2.dir/lab2_pthread.cpp.o
[100%] Linking CXX executable lab2
[100%] Built target lab2
→  lab2 ./lab2 4
The Pthread const time is 1.962000 ms
The Serial const time is 5.480000 ms
Pthread compute successfully
```

图 2.2 编译生成可执行文件 lab2 并运行

图 2.3 不同方法腐蚀图像前后的效果图

为查看并行线程个数对运行时间的影响，传入不同的参数，为使得效果更加明显，处理较大的图片，运行情况如下，发现开启 2-8 个线程均呈现加速一倍的效果，且运行时间差别不大。

```
→  lab2 ./lab2 2
The Pthread const time is 74.327000 ms
The Serial const time is 137.901000 ms
Pthread compute successfully
→  lab2 ./lab2 4
The Pthread const time is 71.731000 ms
The Serial const time is 137.625000 ms
Pthread compute successfully
→  lab2 ./lab2 6
The Pthread const time is 70.477000 ms
The Serial const time is 132.768000 ms
Pthread compute successfully
→  lab2 ./lab2 8
The Pthread const time is 74.626000 ms
The Serial const time is 137.927000 ms
Pthread compute successfully
→  lab2
```

图 2.4 线程个数对并行时间的影响

3.实验三

3.1 实验目的与要求

1、掌握使用 OpenMP 进行并行编程设计和性能优化的基本原理和方法；

2、使用 OpenMP 实现形态学图像处理操作的并行算法；

3、进行程序执行结果的简单分析和总结；

4、将它与实验二的结果进行比较。

3.2 算法描述

Open MP 方法：（并行）

程序执行逻辑同实验二中描述的 CPU 串行方法，只是需要在 for 循环体前利用特殊编译引导语句将其自动划分为多个线程并行计算。核心代码如下：

```
#pragma omp parallel for num_threads(n)
    for (int i = 0; i < binary_dst.rows; i++) {
        for (int j = 0; j < binary_dst.cols; j++) {
        ...
        }
    }
```

num_threads(n)指定并行处理中开启的线程数，n 从命令行参数获得，目的是查看开启线程数对并行加速的效果。

同 pthread 实验，本实验比较串并行计算时间的长短和计算结果的一致性，以显示并行加速效果和并行计算的正确性。

3.3 实验方案

开发环境：ubuntu16.04 虚拟机+visual studio code+opencv3.2.0

运行环境：ubuntu16.04 虚拟机+opencv3.2.0

3.4 实验结果与分析

本实验的 CmakeLists 内容如下。

```
cmake_minimum_required(VERSION 2.8)
project( lab3 )
find_package( OpenCV REQUIRED )
find_package( OpenMP REQUIRED)
add_executable( lab3 lab3.cpp )
target_link_libraries( lab3 ${OpenCV_LIBS} )
if(OPENMP_FOUND)
message("OPENMP FOUND")
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")
endif()
```

图 3.1 OpenMP 的 CmakeLists

编译生成可执行程序 lab3，传入并行的线程数 2，运行结果如下图 3.2，控制台显示串并行的运行时间以及并行结果正确的提示，弹出腐蚀处理前后的图像如图 3.3，直观效果为腐蚀后图像字母变瘦，边界点被消除。

```
→  lab3 make
Scanning dependencies of target lab3
[ 50%] Building CXX object CMakeFiles/lab3.dir/lab3.cpp.o
[100%] Linking CXX executable lab3
[100%] Built target lab3
→  lab3 ./lab3 2
The Openmp const time is 2.861000 ms
The Serial const time is 3.391000 ms
Openmp compute successfully
```

图 3.2 编译运行可执行程序

图 3.3 腐蚀前后图片

传入不同的命令行参数，发现开启 2-8 个线程均呈现将近加速一倍的效

果，且运行时间差别不大。同实验二相比，运行时间同样相似，此两种加速方法均是线程并行的方式加速 for 循环，因此效果类似。

```
→  lab3 ./lab3 2
The Openmp const time is 83.304000 ms
The Serial const time is 134.033000 ms
Openmp compute successfully
→  lab3 ./lab3 4
The Openmp const time is 70.600000 ms
The Serial const time is 132.959000 ms
Openmp compute successfully
→  lab3 ./lab3 6
The Openmp const time is 73.582000 ms
The Serial const time is 138.367000 ms
Openmp compute successfully
→  lab3 ./lab3 7
The Openmp const time is 78.098000 ms
The Serial const time is 134.595000 ms
Openmp compute successfully
→  lab3 ./lab3 8
The Openmp const time is 70.750000 ms
The Serial const time is 135.321000 ms
Openmp compute successfully
```

图 3.4 线程个数对并行运行时间的影响

4.实验四

4.1 实验目的与要求

1、掌握使用 MPI 进行并行编程设计和性能优化的基本原理和方法；

2、使用 MPI 实现形态图像处理操作的并行算法；

3、进行程序执行结果的简单分析和总结；

4、将其与实验二和实验三的结果进行比较。

4.2 算法描述

Open MPI 方法：（并行）

不同于前两个实验开启多个线程进行并行加速，OpenMPI 的并行单位为进程。相应的就会产生进程间通信的问题，不同进程的进程空间是相互隔离的，在进程 A 中对变量 C 的改变，在进程 B 中无法察觉。因此若计算数据生成于主进程，当其他进程参与并行运算时，主进程需要将计算数据发送至其余进程，并在计算结束后，其余进程将计算结果发送至主进程，主进程汇总后输出或显示。

MPI 的划分思想同 pthread 实验，每个进程负责计算 step=row/n 行像素点，范围为 rank*step – (rank+1)*step 行，rank 为当前进程的编号，进程的个数 n 来源于命令行参数，每个进程的 for 循环范围如下，取上整是为了每行像素点都会被遍历到：

```
1.  int step = ceil(binary_dst.rows / size);
2.  char *sendBuf = new char[binary_dst.cols * step];
3.  memset(sendBuf, 0, binary_dst.cols * step);
4.  for(int i = step * rank; (i < (step * rank + step)) && (i <
    binary_dst.rows); i++){
5.      for (int j = 1; j < binary_dst.cols - 1; j++) {
6.          ...
7.          if (upper && origin && left && lower && right) {
8.              sendBuf[(i - step * rank) * binary_dst.cols + j] = 255;
9.          }
10.     }
11. }
```

每个进程将结果存放在 sendBuf 缓冲区中，该缓冲区的大小为 col*row/n 个字节，即每个进程需要计算的像素点的个数。计算完成后，各进程将 sendBuf

缓冲区中的数据发送至 0 号主进程，主进程接受后汇总至大小为 col*row 的缓冲区 recvBuf，再将其拷贝到目标图像的数据区域。发送和接受计算结果的代码如下：

```
MPI_Send(sendBuf, step*binary_dst.cols, MPI_CHAR, 0, rank, MPI_COMM_WORLD);
if(rank == 0){
    Mat erosion_dst_ompi = binary_dst.clone();
    memset(erosion_dst_ompi.data, 0, binary_dst.rows * binary_dst.cols);
    char *recvBuf = new char[binary_dst.rows * binary_dst.cols];
    for(int i = 0; i < size; i++){
    MPI_Recv(recvBuf + i * step * binary_dst.cols, step * binary_dst.cols,
MPI_CHAR, i, i, MPI_COMM_WORLD, &status);
    }
memcpy(erosion_dst_ompi.data, recvBuf, binary_dst.rows * binary_dst.cols);
```

本实验计算并行计算时间的方法有所不同，其余实验均是在并行开始之前开始计时，并行结束后停止计时，但由于本实验多个进程从一开始便同时运行，直到 MPI_Finalize 函数，主进程之外的进程才结束，并且计算结果最终在 rank 为 0 的主进程处汇总，因此将主进程开始时间和汇总完成之间的时间差作为并行计算的时间。为证明并行计算的正确性，还需对比串并行计算结果。

4.3 实验方案

开发环境：ubuntu16.04 虚拟机+visual studio code+opencv 3.2.0

运行环境：ubuntu16.04 虚拟机+opencv 3.2.0+openmpi 1.10.2

4.4 实验结果与分析

本实验使用到的 CMakeLists 内容如下：

```
cmake_minimum_required(VERSION 2.8)
project( lab4 )
find_package( OpenCV REQUIRED )
add_executable( lab4 lab4.cpp )
target_link_libraries( lab4 ${OpenCV_LIBS} )
set(CMAKE_C_COMPILER mpicc)
set(CMAKE_CXX_COMPILER mpicxx)
target_link_libraries( lab4 mpi )
```

图 4.1 openmpi 使用到的 CmakeList

编译生成可执行程序 lab4，传入并行的线程数 2，运行结果如下图 4.2，控制台显示串并行的运行时间以及并行结果正确的提示，可以看出并行的运行时间稍小于串行运行时间，同时控制台显示两个进程输出的信息。

```
➜  lab4 make
Scanning dependencies of target lab4
[ 50%] Building CXX object CMakeFiles/lab4.dir/lab4.cpp.o
[100%] Linking CXX executable lab4
[100%] Built target lab4
➜  lab4 mpirun -np 2 ./lab4
-------------------------------------------------
rank=1
The Serial const time is 4.432000 ms
-------------------------------------------------
rank=0
The Openmpi const time is 4.021000 ms
Openmpi compute successfully
```

图 4.2 编译运行可执行程序

弹出腐蚀处理前后的图像如图 4.3，边界被消除。

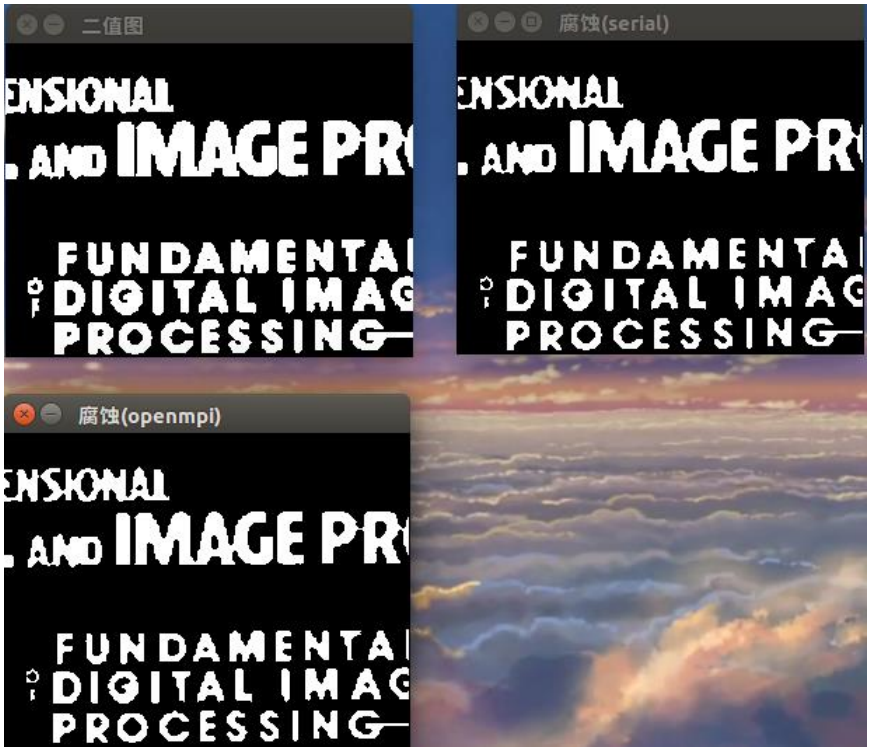

图 4.3 腐蚀处理前后的图像

改变运行的进程个数，执行结果如下，可以看出 openmpi 的并行运行时间略大于实验二和实验三，这是由于创建和销毁进程的开销大于线程，操作系统需要创建新的进程空间，并复制代码段、数据段等信息，由于不同进程之间的进程空间不一致，进程间通信成为另一个比较耗时的问题，若数据量不够大，很可能出现负优化的情况。

```
➜  lab4 mpirun -np 2 ./lab4
The Serial const time is 7.300000 ms
The Openmpi const time is 3.546000 ms
Openmpi compute successfully
➜  lab4 mpirun -np 4 ./lab4
The Serial const time is 8.591000 ms
The Openmpi const time is 3.923000 ms
Openmpi compute successfully
```

图 4.4 改变线程个数的运行情况

5.实验五

5.1 实验目的与要求

1、深入理解 GPGPU 的架构并掌握 CUDA 编程模型；

2、使用 CUDA 实现形态学图像处理操作的并行算法；

3、进行程序执行结果的简单分析和总结；

4、提出基于执行结果和硬件环境的优化解决方案；

5、将其与实验二、三和实验四的结果进行比较。

5.2 算法描述

类似于实验一每个线程执行一次加法操作，本实验考虑每个线程处理一行像素点。假设图片为 rows 行 cols 列的矩阵，则共需要 rows 个线程。初始化线程块的大小为 4*4，那么共需要$\text{blockDim} = \left\lceil \frac{\text{row}}{4*4} \right\rceil$个线程块，即 block 的维数。设置线程网络大小为$\text{gridDim} = \left\lceil \sqrt{\left\lceil \frac{\text{n}}{4*4} \right\rceil} \right\rceil$，即 grid 的维度。设置中采用向上取整是为了保证线程数多于图片的行数，因此在 Kernel 函数中需要让行数越界的线程直接退出，避免数组下标越界。

```
#define BLOCKSIZE 4
int gridsize = (int)ceil(sqrt(ceil(binary_dst.rows / (BLOCKSIZE * BLOCKSIZE))));
dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
dim3 dimGrid(gridsize, gridsize, 1);
```

本实验中，定义处于 CPU 内存的二值图片 binary_dst、串行腐蚀结果 erosion_dst 和并行腐蚀结果 erosion_dst_cuda，同时定义 GPU 内存区域 device_a 和 device_b，分别存放 GPU 计算的操作数和结果。Kernel 函数负责处理 device_a 中的每一行，行号则是由线程块号 blockIdx 和线程号 threadIdx 映射得到的，公式如下：

```
/* 块内地址：threadIdx.y *blockDim.x + blockIdx.x
块内地址区间：[0, blockDim.x*blockDim.y-1]
线程块地址：blockIdx.y*gridDim.x+blockIdx.y
线程号为 threadIdx 对应的行号为： */
int i = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x * blockDim.y
    + threadIdx.y * blockDim.x + threadIdx.x;
```

Kernel 函数中，先计算出线程对应的操作行 i，若 i < 图片的行数 rows 则计算，否则该线程直接退出。Kernel 函数定义如下，每个线程仅执行一次循环，循环体内逻辑同串行运算。

```
//a 存放操作数 b 存放结果 rows 为图片行数 cols 为列数
__global__ void erosion(char *a, char *b, int rows, int cols) {
    int i = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x;
    if(i < rows){
        for (int j = 0; j < cols; j++) {
            if((i == 0) || (i == rows - 1) || (j == 0) || (j == cols - 1)){
                *(b + i * cols + j) = 0;  //边界
                continue;
            }
            int origin = *(a + i * cols + j);
            int upper = *(a + i * cols + j - 1);
            int left = *(a + (i - 1) * cols + j);
            int lower = *(a + i * cols + j + 1);
            int right = *(a + (i + 1) * cols + j);
            if (upper && origin && left && lower && right) {
                *(b + i * cols + j) = 255;
            }
        }
    }
}
```

main 函数中，首先将二值图片 binary_dst 从主机内存拷贝到 GPU 内存设备 device_a 上，然后调用腐蚀操作 Kernel 函数让设备异步并行执行，执行完成后将 device_b 中的数据拷贝至腐蚀图像 erosion_dst_cuda，并计算此过程花费的时间。最后利用串行结果验证执行结果，证明并行计算的正确性。main 函数核心代码如下，下述代码开始之前开始计时，结束后停止计时，差值作为并行操作花费的时间。

```
int gridsize = (int)ceil(sqrt(ceil(binary_dst.rows / (BLOCKSIZE * BLOCKSIZE))));
dim3 dimBlock(BLOCKSIZE, BLOCKSIZE, 1);
dim3 dimGrid(gridsize, gridsize, 1);
//腐蚀操作
erosion<<<dimGrid, dimBlock>>>(device_a, device_b, binary_dst.rows, binary_dst.cols);
cudaThreadSynchronize();
//将结果从 GPU 内存拷贝至 CPU 内存
```

```
8.  cudaMemcpy(erosion_dst_cuda.data, device_b, binary_dst.rows *
    binary_dst.cols, cudaMemcpyDeviceToHost);
```

5.3 实验方案

开发环境：ubuntu16.04 虚拟机+visual studio code+opencv3.2.0

运行环境：Windows Xshell 远程连接到 Linux 服务器

5.4 实验结果与分析

本实验的 CmakeLists 内容如下。

```
1   cmake_minimum_required(VERSION 2.8)
2   project( lab5 )
3   INCLUDE(/usr/local/share/cmake-2.8/Modules)
4   find_package( OpenCV REQUIRED )
5   find_package( CUDA REQUIRED )
6   CUDA_ADD_EXECUTABLE( lab5 lab5.cu )
7   target_link_libraries( lab5 ${OpenCV_LIBS} )
8   INCLUDE_DIRECTORIES(${CUDA_INCLUDE_DIRS})
9   SET(CMAKE_C_COMPILER g++)
10  add_definitions("$(pkg-config opencv --libs)")
11  target_link_libraries(lab5 ${CUDA_LIBRARIES})
12
```

图 5.1 Cuda 的 CmakeLists

不同于实验 2-4 在本机上运行，由于节点上 opencv 的版本较老，因此头文件从 opencv2/opencv.hpp 修改为 cv.h 和 highgui.h。

编译生成可执行程序 lab5，运行结果如下图 5.2，控制台显示串并行的运行时间以及并行结果正确的提示。

```
[parallel_exp@node333 lab5]$ make
[100%] Built target lab5
[parallel_exp@node333 lab5]$ ./lab5
The Cuda const time is 0.957000 ms
The Serial const time is 3.610000 ms
Successfully run on GPU and CPU!
```

图 5.2 cuda 程序运行结果

由于远程节点无法直接显示处理后的图片，因此利用 imwrite 函数将图片以文件形式存储在节点中，利用 sz filename 命令可以将图片文件从远程节点传输到本机，本实验处理前后的图像如图 5.3，直观效果为腐蚀后图像字母变瘦，边界点被消除。

图 5.3 腐蚀前后图片文件

可以从图 5.2 中看出，cuda 编程的并行加速效果最为明显，在未达到并行最大化的情况下，运行时间仍然比 pthread、openmp 和 openmpi 少一个数量级，这是由于前三个实验需要 CPU 花费时间创建和销毁进程或线程，而 Cuda 实验利用了 GPU 渲染图像的并行特点，尽管存在 CPU 内存和 GPU 内存的交互，但也是直接通过 memcpy 实现，速度很快。因此在海量数据处理例如机器学习中，由于其强大而高效的并行计算能力，GPU 常被用于优化运行时间，训练深度神经网络。

6.Project 2

6.1 实验目的

1、掌握两个典型的并行程序开发工具 OpenMP 和 MPI；

2、了解两个工具在并行程序设计和优化过程中的异同；

3、由于优化，调整和分析生成的并行算法的并行粒度；

4、进一步了解并行编程的原理以及应该注意什么。

6.2 实验假设

本节主要对 bfs 算法做简单介绍并导出实验中的相关的设计原理。

给定一个特定的源点 s，广度优先搜索(BFS)系统的探索了从 s 可达的图 G 的每个顶点。假设 V 和 E 分别表示图 G 的顶点集和边集，它们的个数分别为 n=|V|，m=|E|。假设要遍历的图是无权值的，等价于 E 中的每条边的权值为 1。一条路径的长度是指这条路径上所有边的权值之和。用 d(s,t)表示顶点 s 和 t 的距离，或点 s 和 t 之间的最短路径长度。BFS 指距离为 k 的所有顶点（即第 k 层顶点）应该比距离为 k+1 的所有顶点先访问。从 s 到每个可达的顶点的距离就是最后的输出。在广度优先搜索图的遍历的应用程序中，当第一次访问一个顶点的时候，可能会执行辅助的计算。此外，也可能会得到一棵广度优先生成树，这棵树的根为 s，且包含所有可达的顶点。

6.3 实验方案

本实验首先实现串行 bfs 算法，然后基于串行算法进行修改，使其并行化运行，并对其进行分析。

1、基于单个队列的 BFS 串行算法及其并行化（open MP）

基于单个队列的 BFS 算法是十分简单的 BFS 算法，利用队列储存临近的顶点，每访问到相邻的一个顶点，就将其加入队列中，当该顶点的所有相邻点都访问之后，从队列中 pop 出该顶点，直到该队列为空。本算法十分容易理解，其 C++代码如下：

```
1. int bfs_serial(int start_index, int end_index)
2. {
3.     queue < int >node_queue;
4.     int flag[N] = {0};
```

```
5.     node_queue.push(start_index);
6.     flag[start_index] = 1;
7.     int head, i;
8.     while (!node_queue.empty()) {
9.     head = node_queue.front();
10.     node_queue.pop();
11.     for (i = 0; i < N; i++) {
12.         if (adjacency_test_small[head][i] == 1 && flag[i] == 0) { //联通且未
    访问
13.         flag[i] = 1;
14.         node_queue.push(i);
15.         }
16.     }
17.     }
18.     return 0;
19. }
```

本次实验中使用到的第一个简单的 BFS 并行算法是基于上述算法的修改。修改的主要思想是将图的遍历循环并行化，即上述的 for 循环。这样可以同时处理多个元素。但是在这种算法中，点的访问以及队列的更新需要是原子操作。因此并行算法如下：

```
1. int bfs_openMP(int start_index, int end_index)
2. {
3. ...
4.     node_queue.push(start_index);
5.     flag[start_index] = 1;
6. ...
7.     while (!node_queue.empty()) {
8.     head = node_queue.front();
9.    node_queue.pop();
10. #pragma omp parallel for
11.     for (i = 0; i < N; i++) {
12.         if (adjacency_test_small[head][i] == 1) {
13. #pragma omp critical
14.         {
15.             if (flag[i] == 0) {
16.                 flag[i] = 1;
17.                 node_queue.push(i);}
18.         }
19.    }
20.  ...
```

2、基于两个队列的 BFS 算法串行算法及其并行化（open MP）

基于两个队列这种写法的 bfs 和前面的最大区别在于它对队列的处理，之前的简单 bfs 是每次从队列中取出当前的访问节点后，之后就将它的邻接节点加入到队列中，这样明显不利于并行化，

为此，这里使用了两个队列，第一个队列上当前同一层的节点，第二个队列用来存储当前层节点的所有邻接节点，等到当前层的节点全部访问完毕后，再将第一个队列与第二个队列进行交换，即可。

这样做的优势在于便于以后的并行化。同一层的节点可以一起运行，不会受到下一层节点的干扰。其串行算法如下：

```
int bfs_serial_two_queue(int start_index, int end_index) {
    queue < int >node_queue;//当前层的顶点
    queue < int >node_queue_two;//下一层的顶点
   ...
    node_queue.push(start_index);
    flag[start_index] = 1;
    ...
    while (!node_queue.empty()) {
    int s = node_queue.size();
     for (int i = 0; i < s; ++i) {
         head = node_queue.front();
         node_queue.pop();
         for (int i = 0; i < N; i++) {
//邻接且未访问
         if (adjacency_test_small[head][i] == 1 && flag[i] == 0) {
             flag[i] = 1;
             node_queue_two.push(i);}
         }
    }
    swap(node_queue, node_queue_two);
    }
    return 0;
}
```

该算法的并行化思想依然是将图的遍历并行化，使得可以同时判多个顶点。值得注意的是由于顶点的分层，该算法中，同步操作的冲突明显变少。算法的 C++代码如下所示：

```
int bfs_serial_two_queue_MP(int start_index, int end_index) {
    queue < int >node_queue;
    queue < int >node_queue_two;
   ...
    node_queue.push(start_index);
    flag[start_index] = 1;
    int head, i;
    while (!node_queue.empty()) {
    int s = node_queue.size();
#pragma omp parallel for
     for (int i = 0; i < s; ++i) {
#pragma omp critical
        {
        head = node_queue.front();
        node_queue.pop();
        }
        for (int i = 0; i < N; i++) {
        if (adjacency_test_small[head][i] == 1 && flag[i] == 0) {
            flag[i] = 1;
#pragma omp critical
        {
            node_queue_two.push(i);
            }
        }   }
    }
    swap(node_queue, node_queue_two);
    }
    return 0;
}
```

3、基于 MPI 的并行 BFS 算法

从上述算法可以看出，基于单个队列的 BFS 算法不利于并行执行，因为队列存储之间的关联性较强，而 BFS 遍历其实是可以分层执行的。层次化 BFS 算法具有先天的并行化优势。基于这个事实，以下提出一个适用于 MPI 的 BFS 算法。

对于一个顶点个数为 N 的无向图，其邻接矩阵的大小 N*N。矩阵的每一行可以看作是一个大小为 N 的数组。利用 MPI 启动 N 个进程，各个进程依次分得矩阵中的一行。各个进程得到该数组后，利用该数组生成该顶点的访问队列 adjacency_queue。最后根进程将其汇总并剔除重复部分，则可以得到广度优先生成树。其主要代码如下：

```
    MPI_Init(nullptr, nullptr);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     //分发数据
    MPI_Scatter(adjacency_matrix, N, MPI_INT, adjacency_row, N, MPI_INT,
0,MPI_COMM_WORLD);

    int index = 0;
    if (rank >= start_index) {
for (int i = 0; i < N; i++) {
     if (adjacency_row[i] == 1)
            adjacency_queue[index++] = i;
     }
    }
    MPI_Barrier(MPI_COMM_WORLD);
//汇总数据
    MPI_Gather(adjacency_queue, N, MPI_INT, bfs_traversal,
        N, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        bool end = false;
        for (int i = 0; i < N * N && !end; ++i) {
            if (areAllVisited(visited, N))  break;
            if (bfs_traversal[i] != -1) {
                if (visited[bfs_traversal[i]] == 0)
                    visited[bfs_traversal[i]] = 1;
            } else continue;
        }
    }
    MPI_Finalize();
```

6.4 实验结果

1、OPENMP 并行实验结果

为了保证并行算法的正确性，本实验设计了一系列测试用例。实验中，会遍历生成顶点个数为 N 的所有无向图的矩阵，然后对 bfs 算法进行测试，如下所示：

```
for (uint i = 0; i < (0x1 << total_case(N)); ++i) {
    init_test_case(i);//初始化测试用例
    bfs_serial(0, N - 1);//单个队列的串行 bfs
    bfs_openMP(0, N - 1);//单个队列的并行 bfs
    bfs_serial_two_queue(0, N - 1);//两个队列的串行 bfs
    bfs_serial_two_queue_MP(0, N - 1);//两个队列的并行 bfs
}
```

当 N==5 时，会生成 2^10 共 1024 个测试用例。源文件中使用 DEBUG 宏来控制是否测试。一旦 DEBUG 宏开启，那么就会进入测试状态，在测试状态中，每个 BFS 算法将储存自己的遍历路径以便与后续的结果校验。不开启该宏，那么程序将直接运行结束并显示时间。

```
➜  pro2 g++ -o common_bfs common_bfs.cpp -fopenmp -std=c++11
➜  pro2 ./common_bfs
共计1024个图
bfs_serial花费了0.002252秒
bfs_openMP花费了0.008268秒
bfs_serial_two_queue花费了0.001824秒
bfs_serial_two_queue_MP花费了0.006008秒
➜  pro2
```

图 6.1 未开启 DEBUG 宏

为了能够验证并行算法的正确性，源码中还设置了 SORT 宏，该宏一旦开启，程序会对并行算法的结果进行部分排序，将排序结果与串行算法比较，可以验证并行算法是否正确。因此，需要同时开启 DEBUG 和 SORT 进行实验，结果如下图：

```
➜  pro2 g++ -o common_bfs common_bfs.cpp -fopenmp -std=c++11
➜  pro2 ./common_bfs
共计1024个图
bfs_serial花费了0.001567秒
bfs_openMP花费了0.009648秒
bfs_serial_two_queue花费了0.002568秒
bfs_serial_two_queue_MP花费了0.007593秒
测试一:路径长度一致(ok)
测试一:路径一致(Yes)
```

图 6.2 同时开启 DEBUG 和 SORT 宏

在源代码中仅开启 DEBUG 宏后，将输出如下测试信息，说明程序确实是并发执行的。

```
➜  pro2 g++ -o common_bfs common_bfs.cpp -fopenmp -std=c++11
➜  pro2 ./common_bfs
共计1024个图
bfs_serial花费了0.001524秒
bfs_openMP花费了0.012754秒
bfs_serial_two_queue花费了0.001985秒
bfs_serial_two_queue_MP花费了0.006264秒
测试一:路径长度一致(ok)
测试一:路径一致(No)
```

图 6.3 仅开启 DEBUG 宏的运行结果

当 DEBUG 宏开启时，可以看到程序输出“路径长度一致(ok)”和“路径一致(No)”。该测试结果表明，这四种 BFS 算法的遍历图的路径长度一致但是路径不一致，这是符合预期的。**因为对于并行算法而言，其遍历的路径并不唯**

一，所以会产生路径长度一致而路径不一致。

2、MPI 并行实验结果

在本实验中，为了验证 MPI 下 BFS 的正确性，同样设计了测试用例。该测试用例是一个 N=5 的无向图，其矩阵在下图中打印出来。为了能够正常运行，需要指定 np 为 5(与 N 相等)。从下图可以看出，共有五个进程运行，分别打印出了 Rank 信息。最后的结果为 0->1->4->3 为广度优先遍历结果。

```
➜  pro2 mpic++ -o mpi_bfs mpi_bfs.cpp
➜  pro2 mpirun -np 5 --oversubscribe ./mpi_bfs
In rank 2
In rank 3
In rank 1
In rank 4
In rank 0
0|1|0|0|1|
1|0|0|1|0|
0|0|0|0|0|
0|1|0|0|0|
1|0|0|0|0|
--------------
0 -> 1 -> 4 -> 3
➜  pro2
```

图 6.4 MPI 实验结果

6.5 思考与分析

1、openMP 与 MPI

OpenMP 利用编译指示、运行时库函数和环境变量来说明共享内存结构的并行机制。程序员使用编译指示与编译器通信说明并行结构、并行方式及数据的共享/私有等属性信息；OpenMP 实现的是线程级并行，线程间通信是隐含的，通过共享变量来实现。

OpenMP 具有良好的可移植性，其实现相对简单，充分利用了共享内存体系结构的特点，避免了消息传递的开销，既可实现粗粒度并行也可实现细粒度并行。其设计目标之一是实现串行程序并行化，将串行程序转变为 OpenMP 并行程序，不需对源代码做大的改动。在本次实验中，可以明显看出 OpenMP 使用时只需要找到并行位置，然后指定临界区即可。

然而 OpenMP 程序只能在共享内存机器上运行而且数据的放置可能会带来问题，粗粒度并行需考虑核 MPI 同样的并行策略问题，使得实现更加复杂，而且需要明确的同步说明。

MPI 实现进程级并行，其优势在于：MPI 允许静态任务调度，提供明确的

并行机制，通常可获得较高的性能；提供了多种点对点和组通信库函数，可根据实际问题的需要选择最优的通信模式；理论上讲，MPI 并行程序中可实现通信和计算的重叠；同步通过消息通信来实现，减少了额外的同步开销。 应用 MPI 进行并行程序设计也有许多不利的地方：

（1）MPI 程序设计中需要程序员完成任务和数据的划分及分布，应用的开发和调试难度较大；

（2）消息传递的开销非常大，需考虑尽量减少通信延迟；

（3）串行程序到并行程序的转化难度较大，需做大量的代码修改等。

2、粒度与性能问题

实现出色并行性能的关键是选择适合应用的粒度。粒度是指并行任务的实际工作量。如果粒度太细，则并行性能会因通信开销增加而受到影响。如果粒度太粗，则并行性能会因负载不均衡而受到影响。为确保实现最佳并行性能，开发人员应确定适合并行任务的粒度（通常粒度越大越好，在本次实验中，使用双队列的实现，增大了并行粒度，明显比单队列的实现性能要好）；同时还应避免负载不均衡和通信开销增加的情况发生。

7.实验小结

本次并行与编程实验分别实现 pthread、OpenMP、OpenMPI 和 Cuda 并行框架，以向量加法为示例逐步理解各并行框架的实现原理，并以图片腐蚀为背景实现四种并行框架，在 project 中对 openmp 和 openmpi 的应用加深。在进行并行计算时，个人认为重点是如何将一个串行的任务合理的均分为各个细小的工作，将这些工作分发给线程、进程或者 GPU 进行运算，值得注意的是并行单元之间数据的通信问题。论并行加速的效果，GPU 并行计算的效果最好。在实验过程中，不免遇到了各种各样的问题，pthread 和 openmp 由于之前或多或少接触过，因此实现起来比较容易，但 openmpi 和 cuda 就需要一定时间边做边理解其并行思想，尤其是 openmpi 进程之间分发和汇总数据的问题。另外，由于 opencv 程序需要利用 cmake 进行编译，因此通过上网搜集资料的形式对 CmakeList 的编写有了一定了解。对于 opencv 的使用，需要了解图片如何存储在 Mat 这一数据类型中以及像素点的排列规则，以便通过坐标访问像素点。