

# 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

姓 名 朱锦辉

专业班级 ACM1501

学 号 U201514582

指导教师 金海

报告日期 2018年7月9日

1.实验一 熟悉并行编程环境

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]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：

#pragma omp parallel for

for(i=0;i<5;++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，即我们定义的由两个n\*1维的向量所组成的n\*2维的矩阵数组，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 时，–np参数大小应该为向量长度n。

**1.2.4 使用CUDA做向量加法**

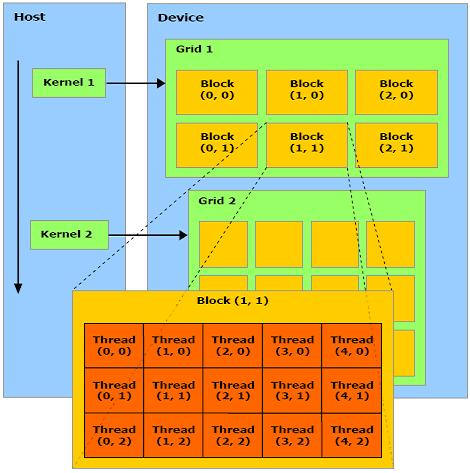


图1-1 CUDA内部机制

我们定义了四个128维的向量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用于检验计算结果是否正确。

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，那么需要个线程块，即block的维数大小。设置线程网络grid，grid大小为，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.x \*blockDim.x + blockIdx.y

块内地址区间：[0, blockDim.x\*blockDim.y-1]

线程块地址：blockIdx.x\*gridDim.x+blockIdx.y

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.x\*gridDim.x+blockIdx.y)\* blockDim.x\* blockDim.y

+( threadIdx.x \*blockDim.x + blockIdx.y)

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。

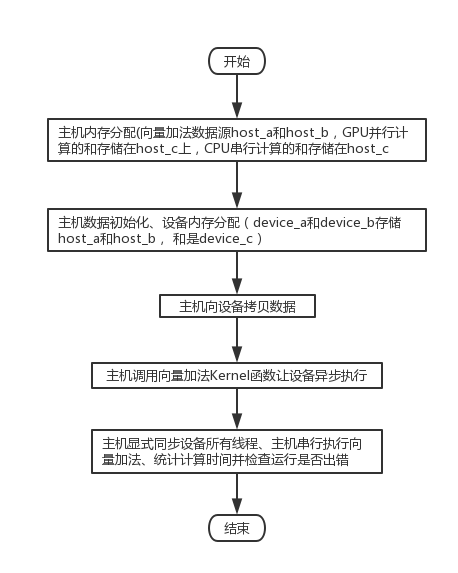


图1-2 CUDA做向量加法算法流程图

1.3 实验结果

**1.3.1 pthread**

编译：gcc Lab1\_1.c -o Lab1\_1 -lpthread

运行：./Lab1\_1

测试结果如图1-3所示。

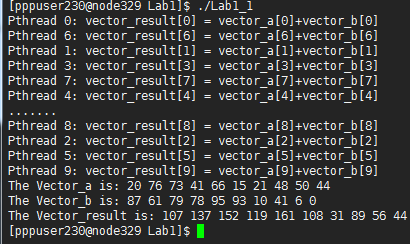


图1-3 pthread方法示例

由于将向量维度n设置为10，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**

编译：gcc Lab1\_2.c -o Lab1\_2 –openmp

运行：./Lab1\_2

由于该实验是通过OpenMP特殊的编译引导语句自动将for循环分解为多个线程并行的，测试结果不是十分直观，如图1-3所示。因此我们把向量长度n增加为100000，计算结果如图1-4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，若果仅仅只做一次简单的for循环，OpenMP的加速情况并不是特别明显。

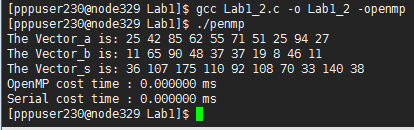


图1-4 OpenMP计算向量加法样例，n=10

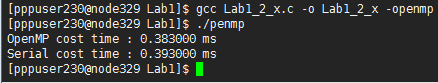


图1-5 OpenMP计算向量加法样例，n=

**1.3.3 OpenMPI方法**

编译：mpic Lab1\_3.c –o Lab1\_3

运行：mpirun –np 4 ./Lab1\_3

用一个n\*2维数组矩阵表示两个向量，通过MPI\_Scatter接口每次分发相同大小的数据块，每个数据块包含同行向量元素，每个进程执行一次加法运算。运行效果如图1-5所示。

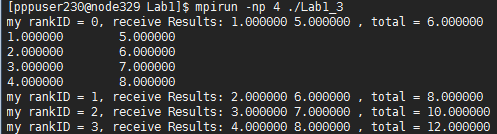


图1-6 OpenMPI方法计算向量加法

**1.3.4 CUDA方法**

编译：nvcc Lab1\_4.cu –o Lab1\_4

运行：./Lab1\_4

在图1-7中，我们在程序中设置向量长度n=128，块大小blocksize=4，验证计算结果正确，但是执行效率远不如CPU线性执行，而且测试到时二者效率几乎相同。图1-8为修改blocksize=16后的测试结果，我们看到随着数据量的增大，CUDA方法的计算效率逐渐增加，最终在时效率超过了CPU。当我们将blocksize设置为32时发现效率又降下来了，查阅资料才知道每个线程块（Block）一般最多可以创建512个并行线程，即blocksize<=16。

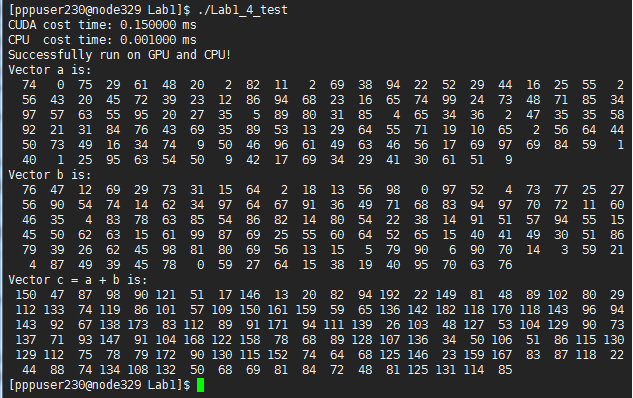


图1-7 CUDA方法计算向量加法，n=128，blocksize=4

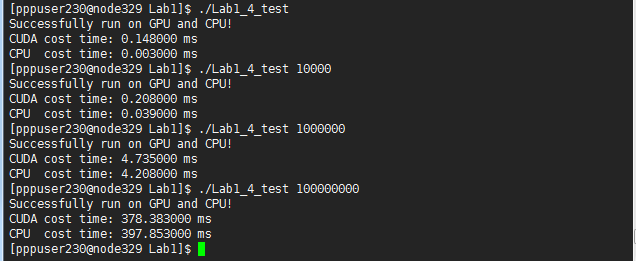


图1-8 CUDA方法计算向量加法，blocksize=16，n显示设置

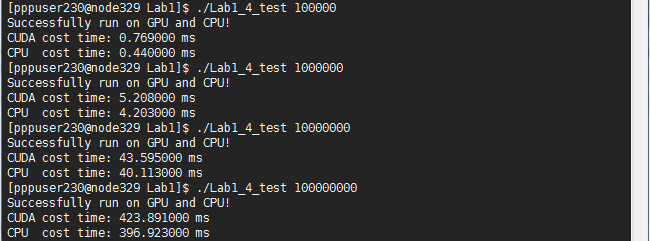


图1-9 CUDA方法计算向量加法，blocksize=32，n显示设置

实验二 pthread

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. 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

**CPU方法：**

程序开始;

Mat image = imread(); //载入图像

for遍历image除边界外的所有像素点：

取中心点周围卷积核大小的像素块point\_ROI;

Convolute(point\_ROI, kernel); //卷积操作

对边界点赋值为0;

程序结束;

**Pthread方法：**

程序开始;

Mat image = imread; //载入图像

;

创建n个pthread;

每个线程执行一次point\_ROI\*kernel矩阵乘并修改image像素值；

等待线程结束，显示image;

程序结束;

2.3 实验方案

开发环境：windows7+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

**CPU方法：**

for (int i=0; i<n; i++)

for(int j=0; j<m; j++)

{

对每一个像素点执行腐蚀操作

}

**Pthread方法：**

for (int i=0; i<4; i++)

{

pthread\_create; //创建线程，

}

创建4个线程,并将处理的开始行号和行数传递给线程函数erode\_line

每个线程独立的对执行腐蚀操作

for (int i=0; i<4; i++)

{

pthread\_join; //等待线程p[i][j]结束

}

主要设计方案是将一个n×m的图片按行分割成4块，前三块每块的行数为n/4，第四块的行数为(n-n/4)，创建4个线程，每个线程独立执行线程函数。

具体设计方案为：

载入图像，将图像转换为灰度图，再将灰度图二值化成一个n×m的图片。构建一个3×3的结构元素element做为核，默认锚点位置为中心点。

开始计时。

将图片按行分割成四块，创建4个线程（不能等分时，最后一个线程处理剩余的行），每个线程独立的执行线程函数erode\_line。

线程函数erode\_line：遍历每一个像素点，进行腐蚀处理。

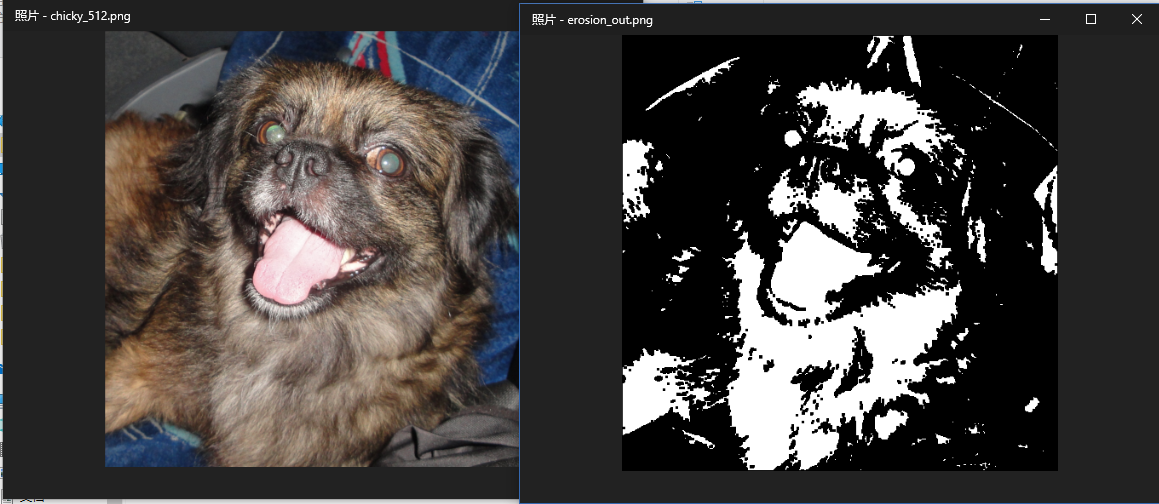
腐蚀处理函数erode\_center：首先判断边界并对边界做处理，把element放到原图上，如果原图没有包含element，则为边界，函数返回值为0。如果不是边界，计算element是否和原图fit，如果fit，返回值就设为255，否则设为0。

等待每一个线程执行结束。

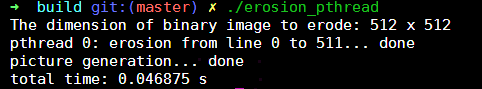
结束计时，打印程序运行时间。

2.4 实验结果与分析

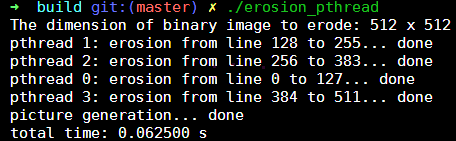
选取一张512x512的图片作为输入，使用pthread进行的并行化形态学腐蚀处理的结果对比如下：



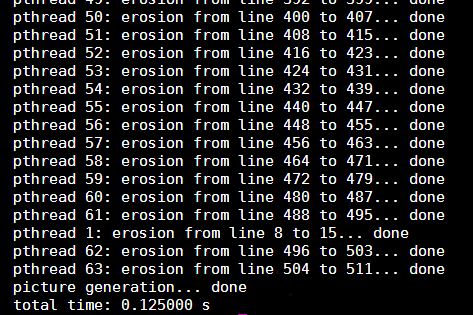
单线程即串行运行结果如下：



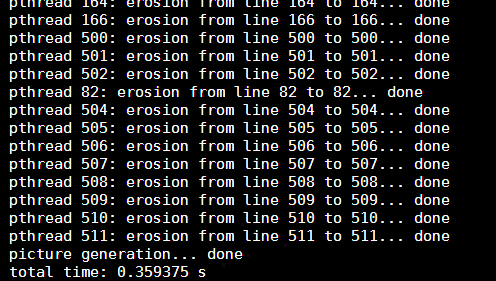
当线程数(THREAD)为4时，程序在终端的输出如下：



当线程数为64时，程序在终端的输出如下：



当线程数为512，即图片的每一行像素建立一个线程时，程序在终端的输出如下：



对比以上不同粒度的pthread并行化腐蚀程序的结果，可以发现单线程即串行的腐蚀操作花费的时间是最少的，且随着线程数的增加，腐蚀操作花费的时间逐渐上升，当线程数达到512个，即对原图像中每一行像素点都建立一个线程时，并行程序运行时间甚至达到了串行运行时间的10倍！这说明当每个线程承担的工作量较小时，建立线程和线程切换的开销是巨大的，甚至会在程序的总开销中占据主导地位。

实验三 openMP

3.1 实验目的与要求

1. 掌握OpenMP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. 对程序执行结果进行简单分析总结

3.2 算法描述

3.3 实验方案

开发环境：windows10+visual studio2017+opencv3.4.2+OpenMP

运行环境：windows10+visual studio2017+opencv3.4.2+OpenMP

主要设计方案是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，每个线程独立执行线程函数。其他操作同实验二。

具体设计方案为：

载入图像，将图像转换为灰度图，再将灰度图二值化成一个n×m的图片。构建一个3×3的结构元素element做为核，默认锚点位置为中心点。

开始计时。

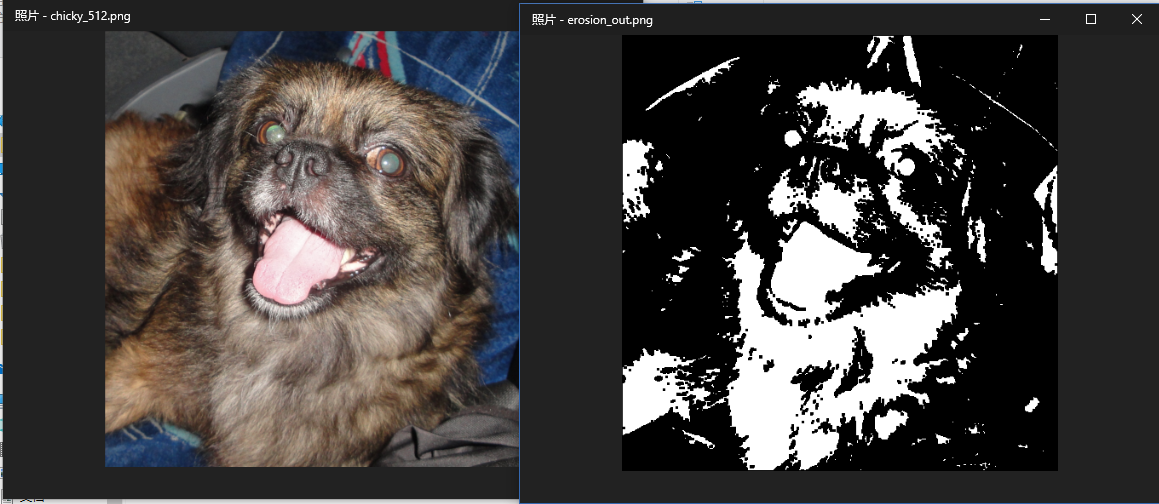
遍历每一个像素点，OpenMP自动将for循环分解成多个线程每个线程独立的执行线程函数erode\_ center。

线程函数erode\_center：首先判断边界并对边界做处理，把element放到原图上，如果原图没有包含element，则为边界，函数返回值为0。如果不是边界，计算element是否和原图fit，如果fit，返回值就设为255，否则设为0。

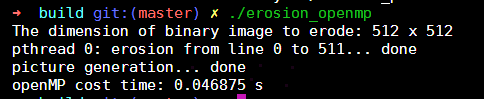
结束计时，打印程序运行时间。

3.4 实验结果与分析

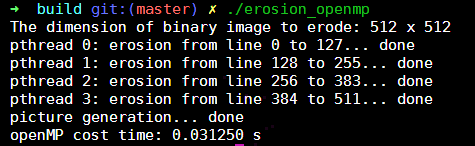
程序执行后得到的腐蚀图片与原图对比如下。可以发现，腐蚀效果与pthread实验中得到的效果完全一致。



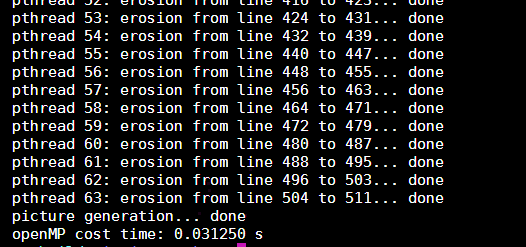
单线程即串行运行结果如下：



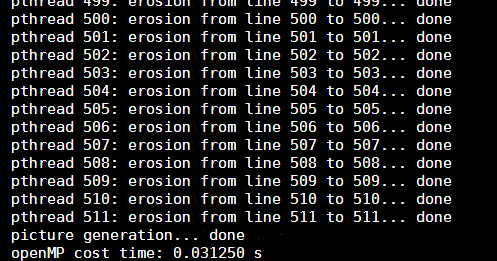
当线程数(THREAD)为4时，程序在终端的输出如下：



当线程数为64时，程序在终端的输出如下：



当线程数为512，即图片的每一行像素建立一个线程时，程序在终端的输出如下：



可见，使用openMP对程序并行化确实能使程序的性能得到提升。当建立的线程数逐渐提升时，建立线程和线程切换的开销在程序的总开销中逐渐占据主导地位，程序并行化带来的速度提升效果下降，建立大量线程（512个）和建立少量（4个）线程的效果基本相同。而且可以预见的是，当输入的数据很大时，即输入的图片像素点数量很多时，建立大量线程的开销会占据主导地位，甚至超过线程本身工作的开销，使得建立大量线程的并行程序性能反而不如建立少量线程，甚至可能不如串行程序。

使用pthread、openMP和MPI并行策略的图像腐蚀处理程序在512x512的相同图像上不同并行粒度的运行时间如下表所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 4 | 64 | 512 |
| Pthread | 0.046875s | 0.0625s | 0.125s | 0.359375s |
| openMP | 0.046815s | 0.03125s | 0.03125s | 0.03125s |

对比openMP方法和pthread方法，可以发现openMP方法中编译器除了对for循环进行并行化处理，对每次循环的循环体都建立一个线程外，在建立线程和线程优化的处理上也做了相应优化，使得这部分的开销非常的小，当线程数增加时，线程建立和线程切换的开销也不会占据主导地位，使得并行程序的性能维持在一个稳定而且较好的水平，即使线程数非常巨大时，并行程序的性能仍然会由于串行程序。

实验四 MPI

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实现形态学图像处理操作的并行算法
3. 对程序执行结果进行简单分析总结

4.2 算法描述

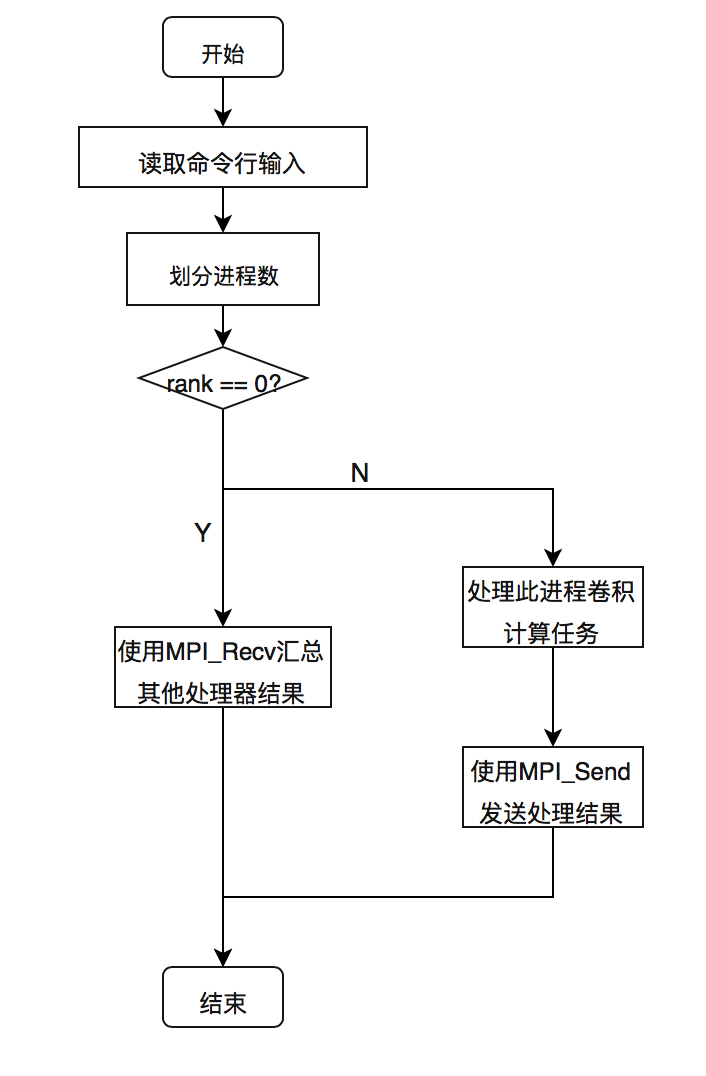
基本思路仍然与pthread和openMPI方法一致：将原图像的像素点按行进行切分，对每一部分的像素点进行并行化处理。

不同之处在于，pthread和openMP方法并行处理的单位是线程，即对每一部分像素点都建立一个线程，每一个线程分别独立地在这些像素点上进行腐蚀操作。由于程序的所有线程共享数据区，每个线程都可以独立地从数据区读取输入，向数据区写回处理的结果，减少了线程之间通信的开销，在处理上也非常方便。

而MPI方法并行处理的单位是进程，即对每一部分像素点都建立一个进程，各个进程独立地在自己负责的像素点上进行腐蚀操作。这时候各个工作进程都需要与主进程进行通信，以获取需要自己处理的像素点（输入），并将处理得到的结果发送到主进程，主进程最后将所有结果拼凑起来得到完成的腐蚀图像。主进程在程序开始时需要将二值图图像发送到每个工作进程，也需要从每个工作进程上接收处理的结果。

4.3 实验方案

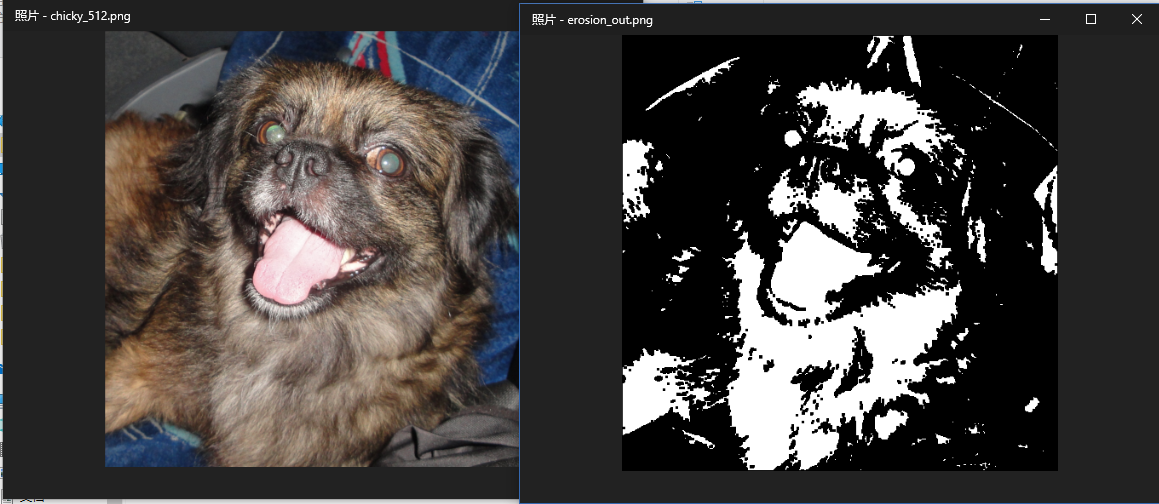
使用MPI并行化的流程图描述如下：



1. 初始化MPI环境，划分进程数，读取当前进程号和总进程数。
2. 主进程读取命令行输入，从命令行输入中解析得到原图片路径，或者使用默认参数，从对应路径中读取待腐蚀的图片。如果读取失败，打印错误信息，退出程序。如果读取成功，对图片进行预处理，将彩色图片转换成灰度图（opencv），对灰度图进行阈值处理（threshold）得到二值图（binary image）。如果总进程数是1，即串行腐蚀操作，由主进程直接串行地对所有像素点进行腐蚀处理，导出腐蚀图像，程序结束。如果总进程数大于1，则说明有工作进程可以使用，即可以进行并行化操作，向所有工作进程发送原图像（数据量不大，直接发送全部数据），然后主进程挂起，等待接收工作进程的执行结果。当收到所有工作进程的执行结果后，汇总（reduce）所有的执行结果，产生完整的腐蚀图像，关闭MPI环境，程序结束。
3. 工作进程接收主进程发送来的原图像数据，对自己负责的那一部分像素点进行腐蚀操作，处理完毕后将处理结果发送到主进程。

4.4 实验结果与分析

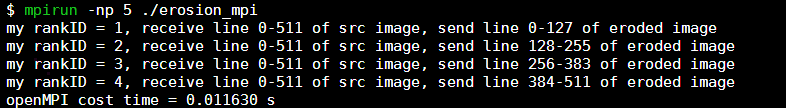
程序执行后得到的腐蚀图片与原图对比如下。可以发现，腐蚀效果与pthread实验和openMPI实验中得到的效果完全一致。



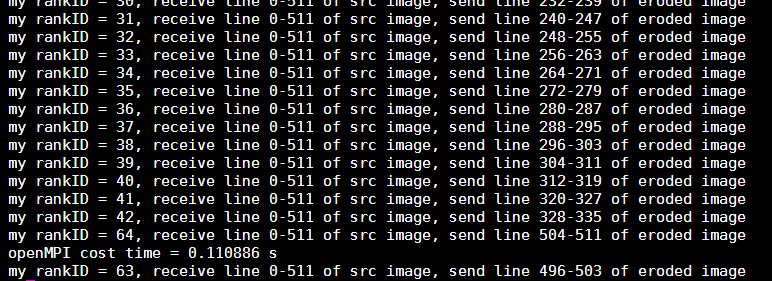
只有一个主进程的单进程串行运行结果如下：



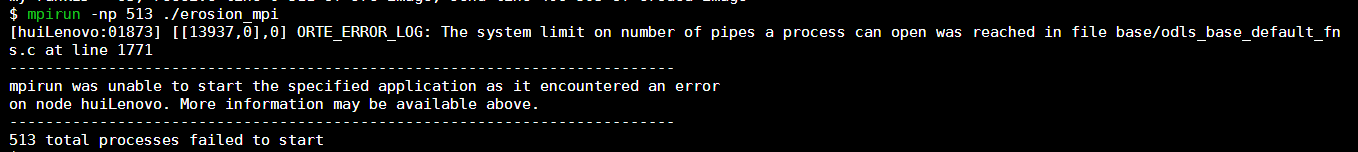
当进程数(np)为5，即1个主进程和4个并行的工作进程时，程序在终端的输出如下：



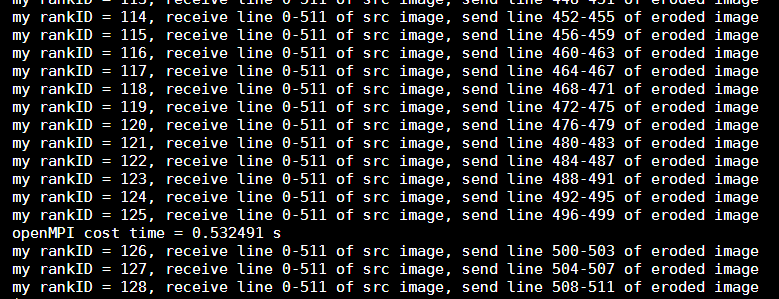
当线程数为65，即1个主进程和64个并行的工作进程时，程序在终端的输出如下：



当进程数为513，由于系统资源限制，无法创建那么多数量的进程，报错信息如下：



这时将进程数量调整为129，即1个主进程和128个工作进程，程序在终端的输出如下：



可见，使用MPI对程序并行化确实能使程序的性能得到提升。当openMPI建立的进程数适中时（在本例中是1个主进程和4个工作进程），程序的性能达到最优（0.01163s），利用进程对程序进行并行化的效果得到最大化。然而随着进程数的继续上升，每个进程承担的工作量减小，每个进程实际工作的时间减少，而建立进程和CPU花费在进程切换上的时间逐渐增加，并开始成为程序运行时间的主要部分，程序性能急剧下降。当进程数达到129个（1个主进程和128个工作进程）时，程序的运行时间更是达到了0.532491s，比串行程序的速度（0.02855s）还要慢将近20倍。

使用pthread、openMP和MPI并行策略的图像腐蚀处理程序在512x512的相同图像上不同并行粒度的运行时间如下表所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 4 | 64 | 512 |
| Pthread | 0.046875s | 0.0625s | 0.125s | 0.359375s |
| openMP | 0.046815s | 0.03125s | 0.03125s | 0.03125s |
| MPI | 0.02855s | 0.01163s | 0.110886s | >0.532491s |

在并行的粒度较小（总进程数较少）时，MPI的并行策略非常有效，无论是单进程的串行化程序还是进程数适中（4个）的并行化程序性能都明显优于pthread和openMP。然而当进程数量继续增加时，建立进程、进程切换和进程间通信的开销开始占据主导地位，这些开销上升带来的对程序性能的影响非常明显，当进程数达到512个时，程序运行时间远远大于pthread方法和openMP方法。而且由于系统资源的限制，过于庞大的进程数量往往是不会被接受的，通常会在MPI环境初始化阶段就会被操作系统拒绝。

实验五 CUDA

5.1 实验目的与要求

1. 深刻理解GPGPU架构，掌握CUDA编程模型。
2. 通过CUDA实现图像的形态学图像处理。
3. 对实现结果进行简单的分析。
4. 基于执行结果和硬件环境提出优化策略。
5. 与pthread、OpenMP、MPI实验结果进行比较。

5.2 算法描述

**CUDA方法：**

|  |
| --- |
| 程序开始;  初始化CUDA环境；  Mat image = imread(); //载入图像  转换成灰度图gray；  for遍历grayImage的所有像素点：//二值化（阈值设定为128）  将灰度图进行二值化；  显示二值图；  开始计时；  cudaMalloc（）；//申请GPU存储空间  cudaMemcpy(cudaMemcpyHostToDevice);//复制待处理的数据到GPU中  g\_erosion <<<1, 4 >>>(参数);//使用GPU进行腐蚀操作  g\_dilation <<<1, 4 >>>(参数);// 使用GPU进行膨胀操作  cudaMemcpy(cudaMemcpyDeviceToHost);//取出处理好的数据  显示腐蚀、膨胀图像；  cudaFree（）；//释放空间  停止计时；  程序结束; |

5.3 实验方案

**5.3.1 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" \t "_blank" \o "大型网站架构知识库)，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" \t "_blank" \o "算法与数据结构知识库)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

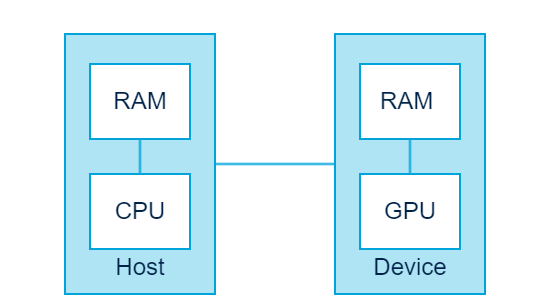


图5.1 cuda架构

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

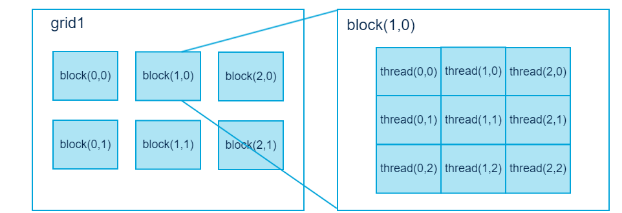


图5.2 thread-block-grid 结构

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

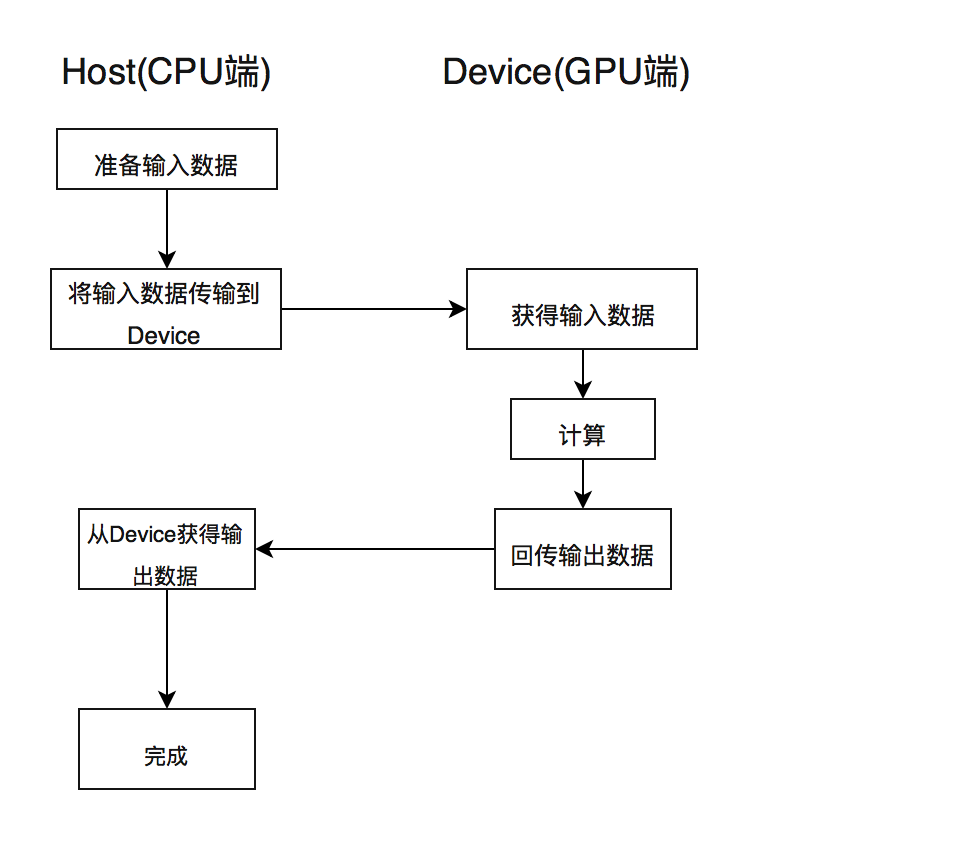


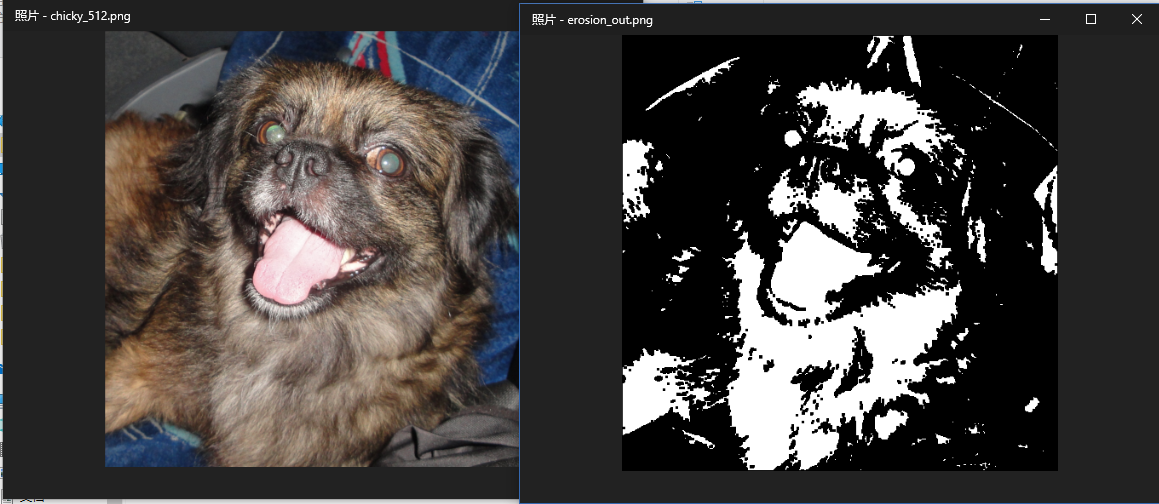
图5.3 所示处理流程

**5.3.2 算法实现**

1. 初始化CUDA编译环境。
2. 解析命令行参数，读取对应路径的图片，预处理图片，将彩色图片转化成灰度图，使用阈值操作（threshold）处理成二值图（binary image）。
3. 在显存中为计算对象和结果对象开辟空间。
4. 将二值图数据传输进显存。
5. 调用kernel函数，根据显存地址、自身块号和线程号对二值图中对应像素点进行腐蚀操作，并把结果存储到显存的结果对象中。
6. 在内存中为计算对象开辟空间，从显存获取处理结果，产生最终的腐蚀图像。
7. 释放显存。

5.4 实验结果与分析

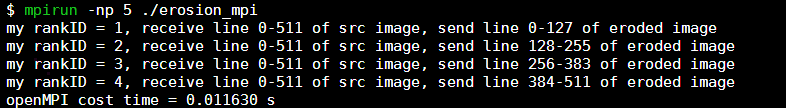
程序执行后得到的腐蚀图片与原图对比如下。可以发现，腐蚀效果与pthread实验、openMPI实验和MPI实验中得到的效果完全一致。



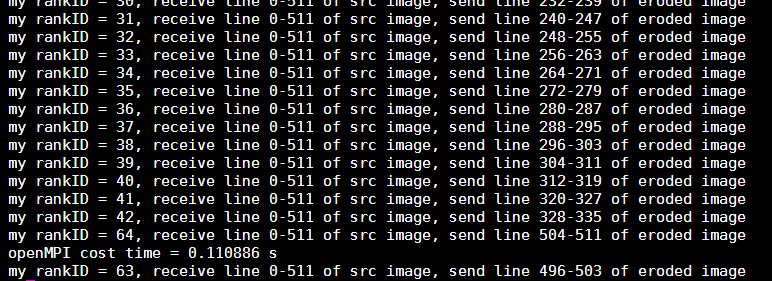
只有一个主进程的单进程串行运行结果如下：



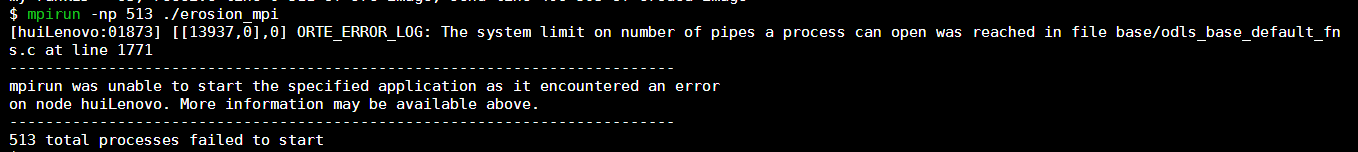
当进程数(np)为5，即1个主进程和4个并行的工作进程时，程序在终端的输出如下：



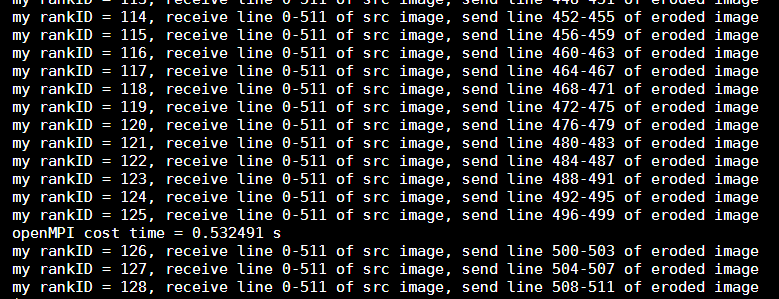
当线程数为65，即1个主进程和64个并行的工作进程时，程序在终端的输出如下：



当进程数为513，由于系统资源限制，无法创建那么多数量的进程，报错信息如下：



这时将进程数量调整为129，即1个主进程和128个工作进程，程序在终端的输出如下：



可见，使用MPI对程序并行化确实能使程序的性能得到提升。当openMPI建立的进程数适中时（在本例中是1个主进程和4个工作进程），程序的性能达到最优（0.01163s），利用进程对程序进行并行化的效果得到最大化。然而随着进程数的继续上升，每个进程承担的工作量减小，每个进程实际工作的时间减少，而建立进程和CPU花费在进程切换上的时间逐渐增加，并开始成为程序运行时间的主要部分，程序性能急剧下降。当进程数达到129个（1个主进程和128个工作进程）时，程序的运行时间更是达到了0.532491s，比串行程序的速度（0.02855s）还要慢将近20倍。

使用pthread、openMP和MPI并行策略的图像腐蚀处理程序在512x512的相同图像上不同并行粒度的运行时间如下表所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 4 | 64 | 512 |
| Pthread | 0.046875s | 0.0625s | 0.125s | 0.359375s |
| openMP | 0.046815s | 0.03125s | 0.03125s | 0.03125s |
| MPI | 0.02855s | 0.01163s | 0.110886s | >0.532491s |

在并行的粒度较小（总进程数较少）时，MPI的并行策略非常有效，无论是单进程的串行化程序还是进程数适中（4个）的并行化程序性能都明显优于pthread和openMP。然而当进程数量继续增加时，建立进程、进程切换和进程间通信的开销开始占据主导地位，这些开销上升带来的对程序性能的影响非常明显，当进程数达到512个时，程序运行时间远远大于pthread方法和openMP方法。而且由于系统资源的限制，过于庞大的进程数量往往是不会被接受的，通常会在MPI环境初始化阶段就会被操作系统拒绝。

Project1 Sudoku problem

6.1 实验目的

1. 掌握并行化和提升程序性能的方法。
2. 理解并行粒度和性能之间的关系。
3. 掌握如何去划分数据和分解复杂算法的任务。

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。

6.2 实验假设

数独问题可以并行化。

6.3 实现方法

**6.3.1背景知识**

数独的求解步骤如下图所示：

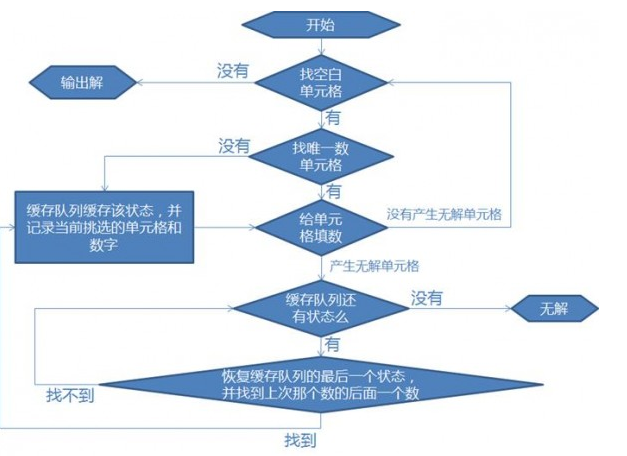


图1.1 求解步骤

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。

**6.3.2 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。

6.4 实验结果与分析

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：

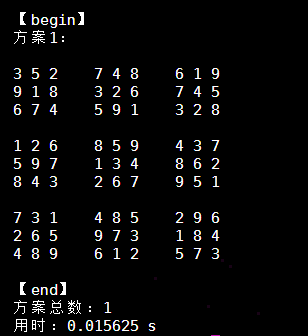


图1.3 数独输出

6.5 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 Parallel BFS

7.1 实验目的

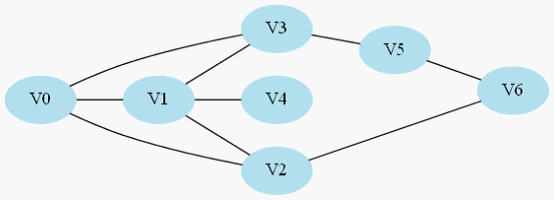
1. 掌握两种典型的并行程序开发工具（openMP或MPI）。
2. 理解在并行程序设计和优化过程中两种工具的异同点。
3. 调整和分析因为优化而产生的并行算法的并行粒度。
4. 进一步理解并行编程原理和应该注意的地方。

7.2 实验假设

广度优先搜索在每棵子树上完成的任务相互独立，可以并行化。

7.3 实现方法

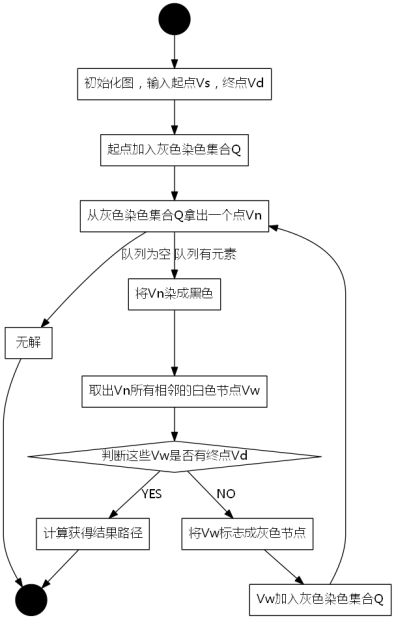
给定连通图，



下面对这个图进行广度优先搜索。

**7.3.1 串行的广度优先算法**

传统的串行化的广度优先搜索流程图如下所示。



**7.3.2 并行的广度优先算法**

在本次实验中，使用基于队列实现的广度优先算法。

1. 将所有顶点标记为未访问。
2. 起始节点s入队，将s标记为已访问。
3. 若队列非空，记队首元素为head，队首元素出队，否则算法结束。
4. 记head的邻居集合为N(head)，将N(head)中所有未被访问过的顶点入队，并标记为已访问。
5. 回到步骤3。

仔细分析以上广度优先算法，可以发现，如果在第4步中对所有顶点的遍历操作是互不相关的，即可并行化的，就可以利用openmp或MPI建立多个线程或进程分别处理每个邻居节点，即对根节点的每个儿子都建立一个线程或进程，对根节点的所有子树进行并行处理。

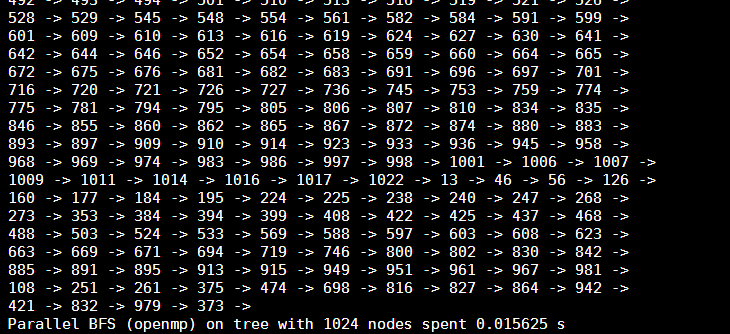
7.4 实验结果与分析

在本实验中，使用openmp对广度优先搜索并行化。

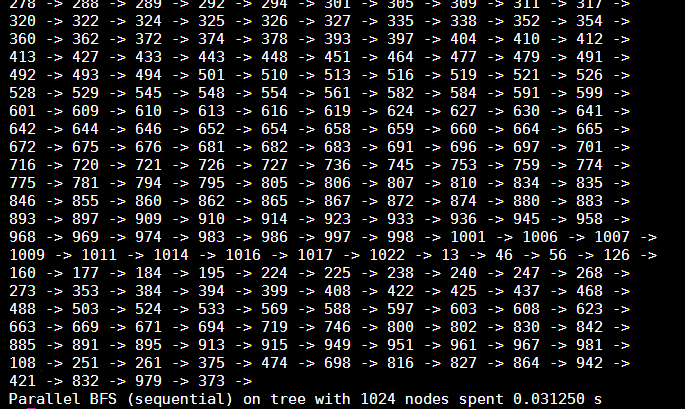
编译：g++ bfs\_openmp.cc –o bfs\_openmp –openmp

运行：./bfs\_openmp

在程序中建立了一个有1024个节点的稠密连通图，两个不同顶点之间以75%的概率连接一条边。使用openmp的广度优先搜索在该图上的运行效果如下所示。



在相同的稠密连通图上运行串行的广度优先搜索运行效果如下所示。



可见，传统的串行广度优先搜索在1024个节点的稠密连通图上运行时间是0.031250s，而使用openmp实现的并行广度优先搜索在同样的连通图上运行时间是0.15625s，广度优先搜索的性能提升了将近一倍。

7.5 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。

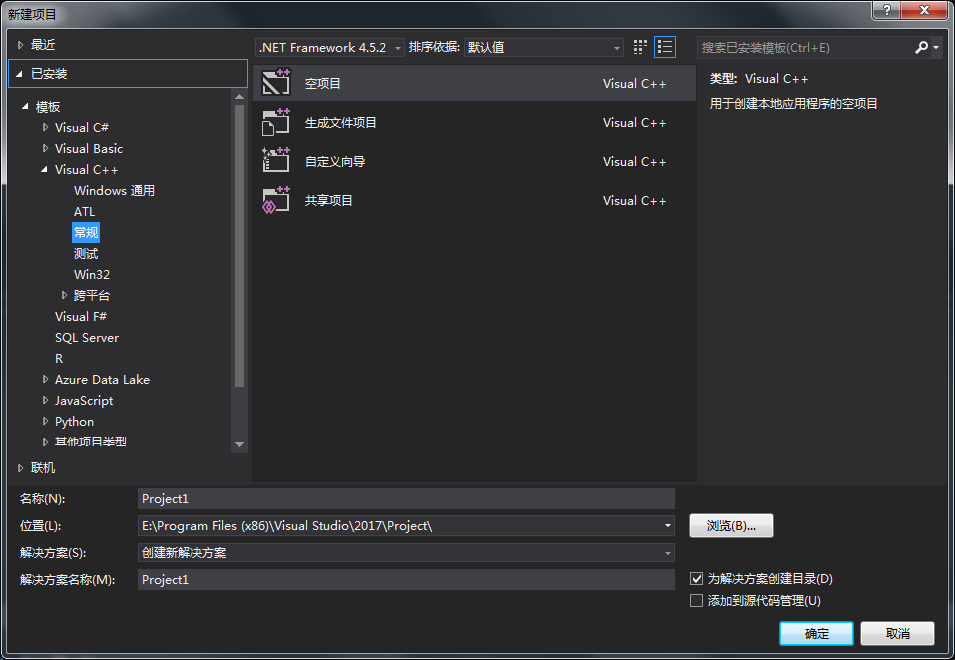
实验已经接近尾声了，但它给我的不只是程序设计上的满足，更重要的是对自己编程思想的一次更新，以及对算法的一个全新的认识！

附录

在OpenCV3.0.0 + Visual Studio 2017环境下创建工程相关配置。

1. 新建项目

点击文件→新建→Visual C++ 空项目→opencv\_conv



1. 右键工程文件夹下的“源文件”→新建项→C++文件→conv\_func.cpp→确定
2. 右键工程文件夹“opencv\_conv”→属性→“VC++目录”下

“包含目录”添加：

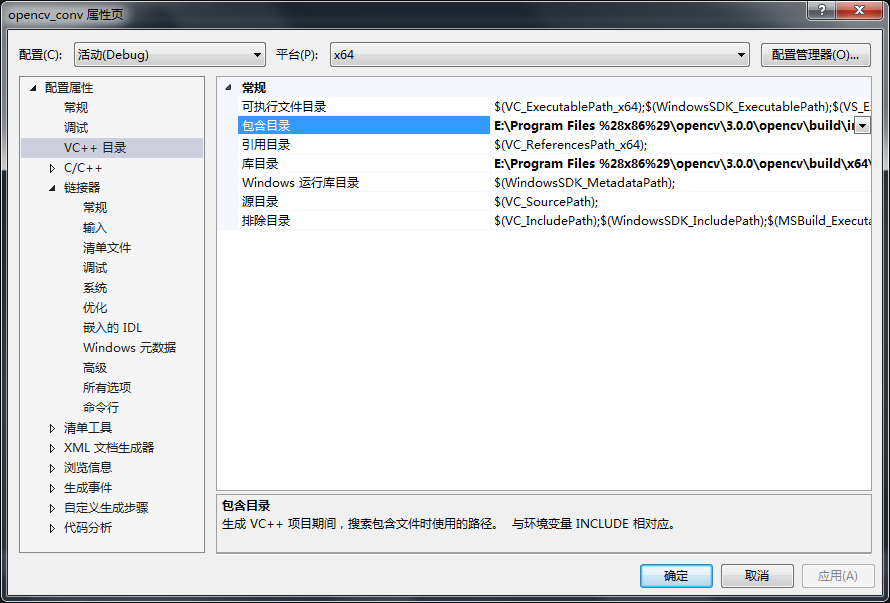
${OpenCV解压路径}\opencv\build\include

${OpenCV解压路径}\opencv\build\include\opencv

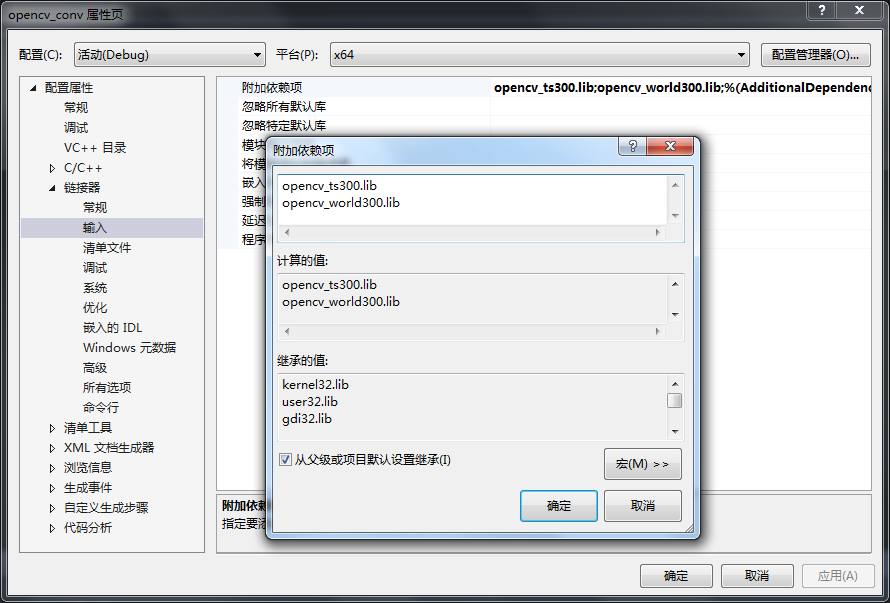
${OpenCV解压路径}\opencv\build\include\opencv2

“库目录”添加：

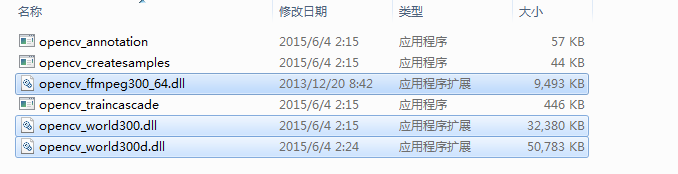
${OpenCV解压路径}\opencv\build\x64\vc12\lib



1. 链接器→附加依赖项→添加“opencv\_ts300.lib”和“opencv\_world300.lib”（一定要手动输入！）



1. 将“${OpenCV解压路径}\opencv\build\x64\vc12\bin”中的动态链接文件复制到“C:\Windows\System32”中，该文件夹为Windows7系统的动态链接文件夹，可以省去每次执行“opencv程序.exe”之前需把这三个动态链接文件复制到“opencv.exe”相同文件夹下。



1. 编制程序→编译链接→执行“.exe”文件

