

# 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 成思霖

专业班级 CS1503

学 号 U201514551

指导教师 陆枫

报告日期 2018年7月19日

目录

[实验一 5](#_Toc519847087)

[1.1 实验目的与要求 5](#_Toc519847088)

[1.1.1实验目的 5](#_Toc519847089)

[1.1.2实验要求 5](#_Toc519847090)

[1.2 实验内容 5](#_Toc519847091)

[1.2.1 使用pthread做向量加法 5](#_Toc519847092)

[1.2.2 使用OpenMP做向量加法 6](#_Toc519847093)

[1.2.3 使用OpenMPI做向量加法 6](#_Toc519847094)

[1.2.4 使用CUDA做向量加法 9](#_Toc519847095)

[1.3 实验结果 14](#_Toc519847096)

[1.3.1 pthread 14](#_Toc519847097)

[1.3.2 OpenMP方法 15](#_Toc519847098)

[1.3.3 OpenMPI方法 15](#_Toc519847099)

[1.3.4 CUDA方法 16](#_Toc519847100)

[实验二 17](#_Toc519847101)

[2.1 实验目的与要求 17](#_Toc519847102)

[2.2 算法描述 17](#_Toc519847103)

[2.2.1 图像的锐化 17](#_Toc519847104)

[2.2.2 拉普拉斯算子进行图像的锐化 18](#_Toc519847105)

[2.2.3 设计思路 18](#_Toc519847106)

[2.3 实验方案 21](#_Toc519847107)

[2.4 实验结果与分析 21](#_Toc519847108)

[实验三 22](#_Toc519847109)

[3.1 实验目的与要求 22](#_Toc519847110)

[3.2 算法描述 22](#_Toc519847111)

[3.2.1 OpenMP 22](#_Toc519847112)

[3.2.2 设计思路 23](#_Toc519847113)

[3.3 实验方案 25](#_Toc519847114)

[3.4 实验结果与分析 25](#_Toc519847115)

[实验四 26](#_Toc519847116)

[4.1 实验目的与要求 26](#_Toc519847117)

[4.2 算法描述 26](#_Toc519847118)

[4.2.1 MPI编程思想 26](#_Toc519847119)

[4.2.3 设计思路 27](#_Toc519847120)

[4.3 实验方案 28](#_Toc519847121)

[4.4 实验结果与分析 29](#_Toc519847122)

[实验五 30](#_Toc519847123)

[5.1 实验目的与要求 30](#_Toc519847124)

[5.2 算法描述 30](#_Toc519847125)

[5.2.1 主函数设计思想 30](#_Toc519847126)

[5.2.3 图像锐化函数设计思想 30](#_Toc519847127)

[5.3 实验方案 31](#_Toc519847128)

[5.4 实验结果与分析 31](#_Toc519847129)

[实验六（选做） 33](#_Toc519847130)

[6.1 实验目的与要求 33](#_Toc519847131)

[6.2 算法描述 33](#_Toc519847132)

[6.2.1聚类 33](#_Toc519847133)

[6.2.2基于划分的聚类方法 33](#_Toc519847134)

[6.2.3 K-Means算法 33](#_Toc519847135)

[6.2.4 MapReduce并行化聚类算法设计思路 34](#_Toc519847136)

[6.3 代码实现 34](#_Toc519847137)

[6.3.1 Instance 34](#_Toc519847138)

[6.3.2 Cluster 34](#_Toc519847139)

[6.3.3 EuclideanDistance 35](#_Toc519847140)

[6.3.4 RandomClusterGenerator 35](#_Toc519847141)

[6.3.5 KMeans 35](#_Toc519847142)

[6.3.6 KmeansCluster 35](#_Toc519847143)

[6.3.7 KMeansDriver 35](#_Toc519847144)

[6.4 实验方案 36](#_Toc519847145)

[6.5 实验结果与分析 36](#_Toc519847146)

[项目：并行优化广度优先搜索 37](#_Toc519847147)

[7.1实验目的 37](#_Toc519847148)

[7.2实验假设 37](#_Toc519847149)

[7.3实验方案 38](#_Toc519847150)

[7.3.1 广度优先搜索图的基本原理 38](#_Toc519847151)

[7.3.2 使用并行方法进行广度优先搜索图 39](#_Toc519847152)

[7.4 实验结果 39](#_Toc519847153)

[7.5讨论与总结 40](#_Toc519847154)

[附录 40](#_Toc519847155)

[实验六 40](#_Toc519847156)

[项目 50](#_Toc519847157)

# 实验一

## 1.1 实验目的与要求

### 1.1.1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。

### 1.1.2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

linux操作系统使用符合POSIX线程作为系统标准线程，该POSIX线程标准定义了一整套操作线程的API。在本次实验中，我主要使用Linux下的pthread来实现向量加法的并行计算，其实现思路比较简单，也即每个线程执行一个特定的加法任务，结果在主线程中进行汇总累加再输出。

算法描述：

array = { {1,2,3,4,5,6,7,8,9,10}, {10000,20000,30000,40000,50000,60000,70000,80000,90000,100000}, … }

sum = 0

for each element in array:

pthread\_create //为每个数组元素的加法创建一个线程

for each thread:

pthread\_join //连接各个线程

sum = sum + result

print sum

调用先pthread 函数创建10个线程，各个线程分别对向量中特定的元素执行加法运算。然后使用pthread\_join函数连接各个线程，并将加法运算的的结果汇总到变量sum当中

### 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<10;i++)

{

vector\_result[i] = vector\_a[i] + vector\_b[i];

}

### 1.2.3 使用OpenMPI做向量加法

**①MPI编程涉及的一些概念**

* 程序代码：

这里的程序不是指以文件形式存在的源代码、可执行代码等，而是指为了完成一个计算任务而进行的一次运行过程。

* 进程(Process)

一个 MPI 并行程序由一组运行在相同或不同计算机 /计算节点上的进程或线程构成。为统一起见，我们将 MPI 程序中一个独立参与通信的个体称为一个进程。

* 进程组：

一个 MPI程序的全部进程集合的一个有序子集。进程组中每个进程都被赋予一个在改组中唯一的序号（rank），用于在该组中标识该进程。序号范围从 0 到进程数－1。

* 通信器（communicator）:

有时也译成通信子，是完成进程间通信的基本环境，它描述了一组可以互相通信的进程以及它们之间的联接关系等信息。MPI所有通信必须在某个通信器中进行。通信器分域内通信器（intracommunicator）和域间通信器（intercommunicator）两类，前者用于同一进程中进程间的通信，后者则用于分属不同进程的进程间的通信。

* 序号（rank）：

即进程的标识，是用来在一个进程组或一个通信器中标识一个进程。MPI 的进程由进程组/序号或通信器/序号唯一确定。

* 消息（message）：

MPI 程序中在进程间传递的数据。它由通信器、源地址、目的地址、消息标签和数据构成。

* 通信（communication）：

通信是指在进程之间进行消息的收发、同步等操作。

②MPI编程用到的API

本次实验中用到的MPI函数以及各个参数的含义如表1.1、1.2。

**表1.1 初等例程**

|  |  |  |
| --- | --- | --- |
| 函数名 | 参数 | 动作 |
| int MPI\_Init(int \*argc, char \*\*argv[]) | 来自 main()的参数 来自 main()的参数 | 初始化 MPI 环境 |
| int MPI\_Finalize(void) | 终止 MPI 执行环境 |  |
| int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank) | 通信子 序号（返回的） | 确定在通信子中进程的序号 |
| int MPI\_Comm\_size(MPI\_Comm comm, int \*size) | 通信子 组的大小（返回的） | 确定与通信子关联的组的大小 |

**表1.2点对点消息传递**

|  |  |  |
| --- | --- | --- |
| 函数名 | 参数 | 动作 |
| int MPI\_Send(void \*buf, int count, MPI\_Datatype datatype int dest, int tag, MPI\_Comm comm) | 发送缓冲区 缓冲区内的项数 项的数据类型 目的进程序号 消息标识 通信子 | 发送消息（锁定） MPI\_Datatype 与 C 对应 MPI\_CHAR signed char MPI\_INT signed int MPI\_FLOAT float |
| int MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status) | 接收缓冲区（已装载） 缓冲区内最大项数 项的数据类型 源进程序号 消息标识 通信子 状态（返回的） | 接收消息（锁定） |

③为了实现两个向量的加法运算，可以指定一个进程A使用MPI\_Send函数发送两个源向量的信息，另一个进程B则调用MPI\_Recv负责收集两个源向量的信息，并完成两个源向量的加法运算。与此同时，进程B将结果通过MPI\_Send函数发送给进程A，进程A将搜集到的结果保存起来，并输出打印。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

If(rank==0)

MPI\_Send (vector\_a+2\*i-2, 2, MPI\_INT, i, 0, MPI\_COMM\_WORLD)

MPI\_Send (vector\_b+2\*i-2, 2, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

MPI\_Recv (vector\_result+2\*i-2, 2, MPI\_INT, i, 2, MPI\_COMM\_WORLD, &stat);

Else

MPI\_Recv (a, 2, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &stat);

MPI\_Recv (b, 2, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, &stat);

c[0]=a[0]+b[0];

c[1]=a[1]+b[1];

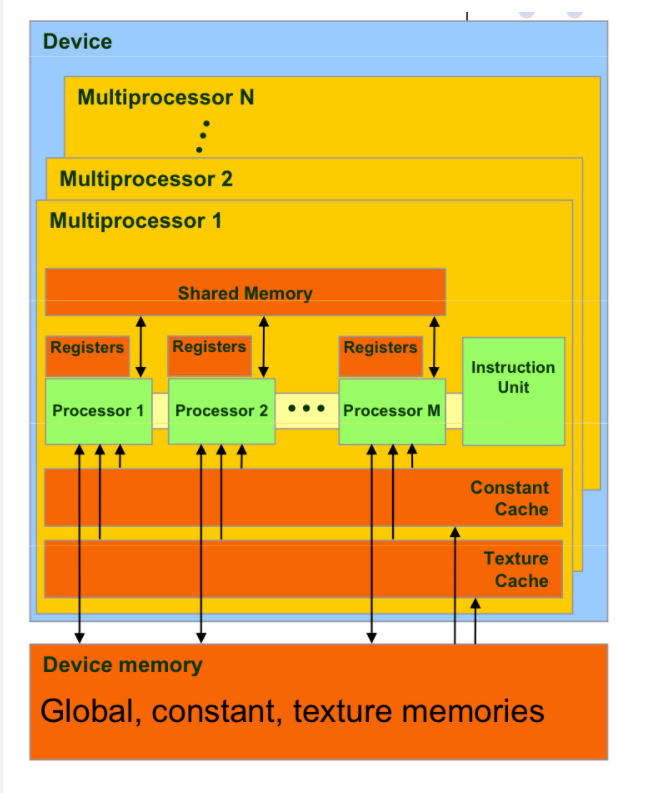
MPI\_Send (c, 2, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

MPI\_Finalize();//结束MPI环境

### 1.2.4 使用CUDA做向量加法

①CUDA编程背景知识

CUDA（Compute Unified Device Architecture）是显卡厂商NVIDIA推出的运算平台。 CUDA是一种由NVIDIA推出的通用并行计算架构，该架构使GPU能够解决复杂的计算问题。 它包含了CUDA指令集架构（ISA）以及GPU内部的并行计算引擎。 开发人员现在可以使用C语言来为CUDA架构编写程序，C语言是应用最广泛的一种高级编程语言。所编写出的程序于是就可以在支持CUDA的处理器上以超高性能运行



（1）GPU架构

主流的GPU架构是以NVIDIA GeForce系列为代表流处理器阵列架构，GPU上集成了上百个流式多处理器（Stream Multiprocessor，SM），如GeForce 8800GTX包含了128个流处理器。每个SM包含若干独立受控于指令的可同步执行的纯量处理器（Scalar Processor，SP）、Cache（指令、常数、纹理）、多线程指令发射单元、特殊功能单元（Special Function Unit，SFU）以及供线程块共享数据的Shared Memory。

（2）存储器层次结构

流式多处理器片上内存包括每个线程独立拥有的寄存器以及局部内存（Local Memory）、片上共享内存（On-chip Shared Memory，供线程块内共享数据）。每个流式多处理器有自己常量Cache和纹理Cache。

GPU设备内存（Device Memory）包括全局内存（Global Memory）以及只读的常量内存（Constant Memory）和纹理内存（Texture Memory）。所有流式多处理器均可访问GPU设备内存。

主机内存（Host Memory）是一般意义的内存，GPU设备不能直接访问主机内存，需要通过拷贝API传递数据（见后）。

（3）线程组织结构

* CUDA中线程也可以分成三个层次：线程、线程块和线程网络。
* 线程是CUDA中基本执行单元，由硬件支持、开销很小，每个线程执行相同代码。线程块（Block）是若干线程的分组，Block内一个块至多512个线程，线程块可以是一维、二维或者三维的
* 线程网络（Grid）是若干线程块的网格，Grid是一维和二维的。

线程用ID索引，线程块内用局部ID标记threadID，配合blockDim和blockID可以计算出全局ID，用于SIMD分配任务。

（4）CUDA程序结构

CUDA程序的结构大体是：{主机串行->GPU并行}+ -> 主机串行，这样的

串并交叉结构。主机串行过渡到GPU并行时需要将数据从主机内存上拷贝GPU设备内存上，GPU执行完毕时也需要把数据拷贝回来。

（5）CUDA内核函数   
主机调用设备代码的唯一接口就是Kernel函数，使用限定符:\_\_global\_\_(见后)。

调用内核函数需要在内核函数名后添加<<<>>>指定内核函数配置，如<<<DimGrid, DimBlock>>>指定线程网络和线程块维度。若当前硬件无法满足用户配置，则内核函数不会被执行，直接返回错误。

（6）CUDA限定符   
函数限定符：（默认host、global异步、主机不能调device；设备上执行的函数参数数目固定、不能声明静态变量且不支持递归调用）



变量限定符：（shared共享一致性必须由显式线程同步保证）

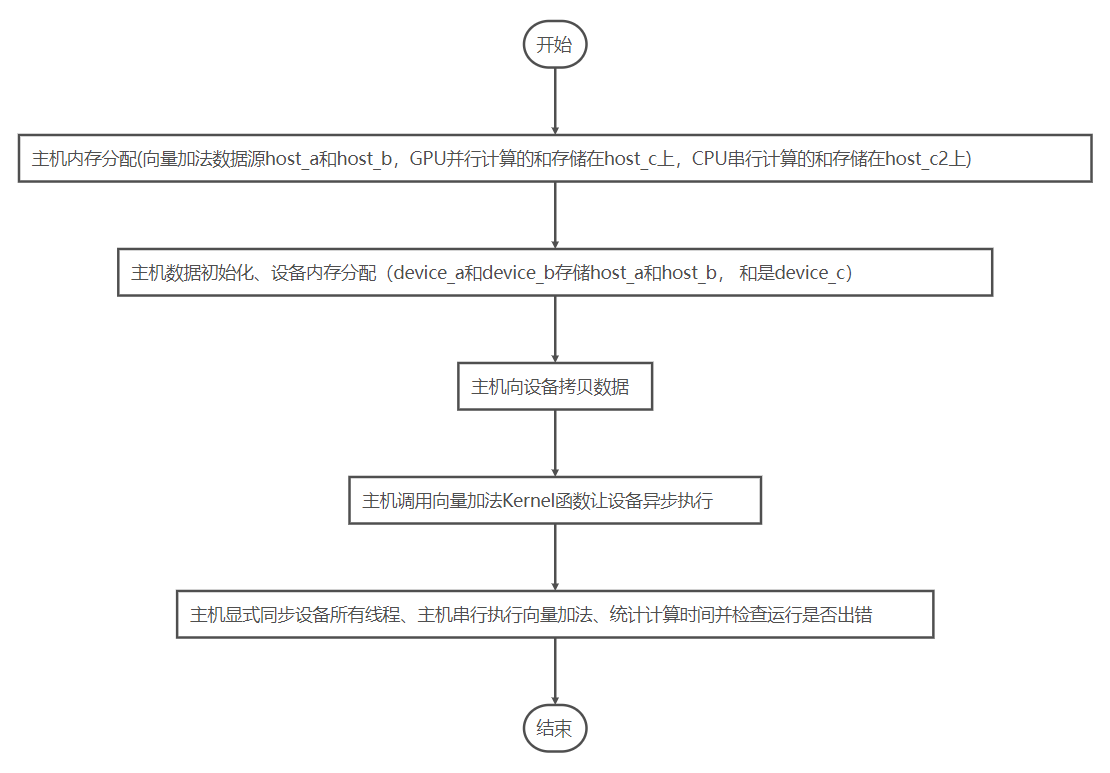


（7）同步

* CPU启动kernel函数是异步的，它并不会阻塞等到GPU执行完kernel函数才执行后面的CPU部分，因此如果后续程序立即需要用到上一个kernel函数的结果我们需要显式设置同步障来阻塞CPU程序。
* 一个线程块内需要同步共享存储器的共享变量（\_\_shared\_\_｀）时，需要在使用前显式调用\_\_syncthreads()`同步块内所有线程。

②算法实现

算法实现流程图如下图：



Kernel函数配置如下：

\_\_global\_\_ void VecAdd(int\* A, int\* B, int\* C, int N) {

int i = blockDim.x \* blockIdx.x + threadIdx.x;

if (i < N) {

A[i] = B[i] + C[i];

}

}

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.下面来推导块号为blockIdx、线程号为threadIdx的线程完成向量计算的数组下标:。推导过程如下：

块内地址为 threadIdx.x \* blockDim.x + threadIdx.y，把线程号映射到[0, blockDim.x \* blockDim.y - 1]。线程块地址为 blockIdx.x \* gridDim.x + blockIdx.y，把块号映射到[0, gridDim.x \* gridDim.y - 1]。 所以该线程的地址为i = (blockIdx.x \* gridDim.x + blockIdx.y) \* blockDim.x \* blockDim.y + threadIdx.x \* blockDim.x + threadIdx.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：

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Kernel

int \*dstA, \*dstB, \*dstC;

// 分配设备内存

cudaMalloc((void\*\*) &dstA, N \* sizeof(int));

cudaMalloc((void\*\*) &dstB, N \* sizeof(int));

cudaMalloc((void\*\*) &dstC, N \* sizeof(int));

int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// 将数据拷贝到设备

cudaMemcpy(dstB, B, sizeof(int) \* N, cudaMemcpyHostToDevice);

cudaMemcpy(dstC, C, sizeof(int) \* N, cudaMemcpyHostToDevice);

// 调用kernel

VecAdd<<<blocksPerGrid, threadsPerBlock>>>(dstA, dstB, dstC, N);

// 将计算结果拷贝回host

cudaMemcpy(A, dstA, N \* sizeof(int), cudaMemcpyDeviceToHost);

// 释放设备内存

cudaFree(dstA);

cudaFree(dstB);

cudaFree(dstC);

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。

## 1.3 实验结果

### 1.3.1 pthread

编译：gcc shiyan1\_1.c -o shiyan1\_1 –lpthread

运行：./Lab1\_1

测试结果如图1-3所示。

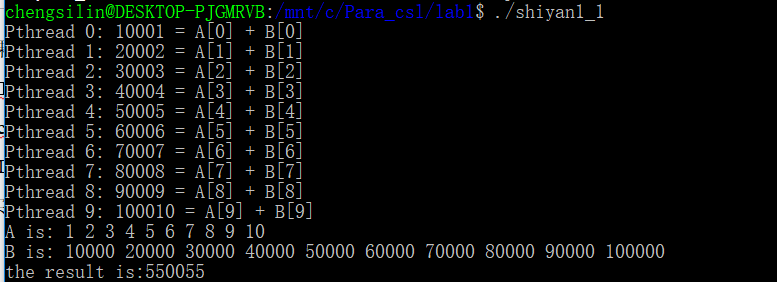


图1-3 pthread方法示例

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，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1-4所示。因此我们把向量长度n增加为100000，计算结果如图1-5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。

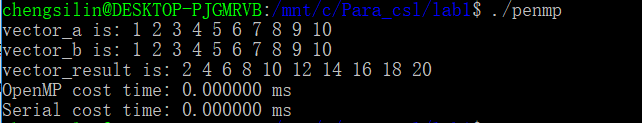


图1-4 OpenMP计算向量加法样例，n=10

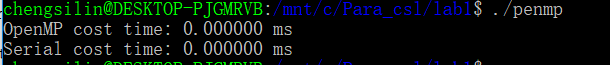


图1-5 OpenMP计算向量加法样例，n=

### 1.3.3 OpenMPI方法

编译：mpicc shiyan1\_3.c –o shiyan1\_3

运行：mpirun –np 4 ./Lab1\_3

代码执行效果如图1-5所示。

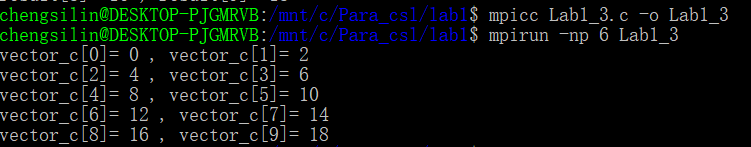


图1-6 OpenMPI方法计算向量加法

### 1.3.4 CUDA方法

编译：nvcc Lab1\_4.cu –o Lab1\_4

运行：./Lab1\_4

在图1-7中，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。

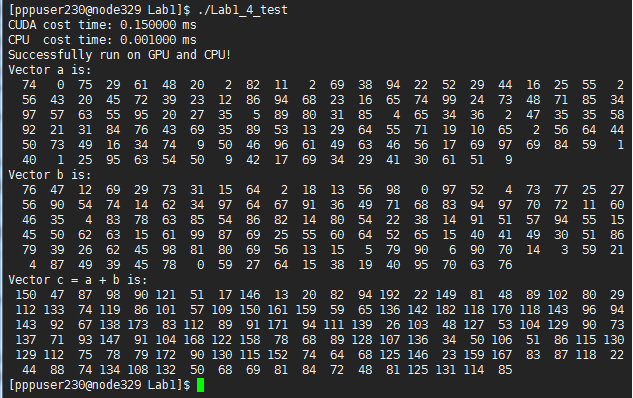


图1-7 CUDA方法计算向量加法，n=128，blocksize=4

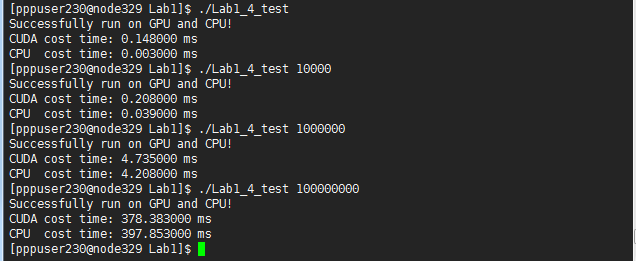


图1-8 CUDA方法计算向量加法，blocksize=16，n显示设置

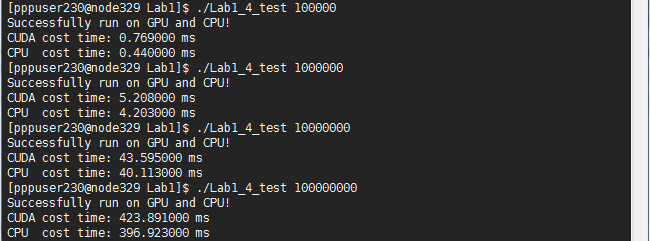


图1-9 CUDA方法计算向量加法，blocksize=32，n显示设置

# 实验二

## 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

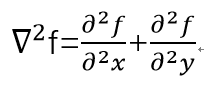
### 2.2.1 图像的锐化

 在图像增强过程中，通常利用各类图像平滑算法消除噪声，图像的常见噪声主要有加性噪声、乘性噪声和量化噪声等。一般来说，图像的能量主要集中在其低频部分，噪声所在的频段主要在高频段，同时图像边缘信息也主要集中在其高频部分。这将导致原始图像在平滑处理之后，图像边缘和图像轮廓模糊的情况出现（平滑可以认为是去除噪声，这样也就模糊了图像的边缘信息）。为了减少这类不利效果的影响，就需要利用图像锐化技术，使图像的边缘变得清晰。

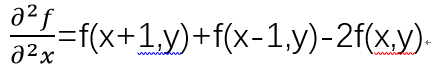
图像锐化处理的目的是为了使图像的边缘、轮廓线以及图像的细节变得清晰，经过平滑的图像变得模糊的根本原因是因为图像受到了平均或积分运算，因此可以对其进行逆运算(如微分运算)就可以使图像变得清晰。微分运算是求信号的变化率，由傅立叶变换的微分性质可知，微分运算具有较强高频分量作用。从频率域来考虑，图像模糊的实质是因为其高频分量被衰减，因此可以用高通滤波器来使图像清晰。但要注意能够进行锐化处理的图像必须有较高的性噪比，否则锐化后图像性噪比反而更低，从而使得噪声增加的比信号还要多，因此一般是先去除或减轻噪声后再进行锐化处理。

### 2.2.2 拉普拉斯算子进行图像的锐化

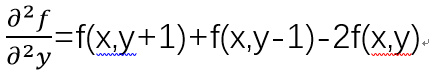
拉普拉斯算子是一个是n维[欧几里德空间](http://baike.baidu.com/item/%E6%AC%A7%E5%87%A0%E9%87%8C%E5%BE%B7%E7%A9%BA%E9%97%B4)中的一个二阶微分算子，它的定义如下：



在x方向上



在y方向上



合起来就是

https://images2015.cnblogs.com/blog/1078653/201704/1078653-20170422174743759-1539091839.png

拉普拉斯强调的是图像中灰度的突变，并不强调图像的灰度缓变（灰度缓变由一阶微分，也就是梯度，图像应用是sobel算子，具体下面介绍）

根据上边的表达式，可以确定拉普拉斯算子的模板

例如：

          [ 0 1 0

            1 -4 0

            0  1  0]

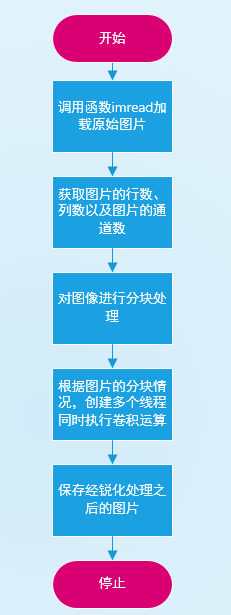
这是以90度增量旋转的拉普拉斯算子。

本次实验采用的就是以90度增量旋转的拉普拉斯算子来对图像进行处理。

### 2.2.3 设计思路

（1）总体设计对应的流程图如图所示。

在使用拉普拉斯算子对图像进行锐化的过程中，采用的卷积核大小为3\*3，其考虑到每一个像素点的计算仅仅取决于原图的像素点与卷积核矩阵，而与其他像素点的计算无关，因此可以将图片划分成一个个子区域，并对这些子区域同时执行卷积运算，最后将计算结果合并起来就能形成锐化的图片。



（2）拉普拉斯算子的代码实现思路。

在使用拉普拉斯算子对图片进行卷积操作前，首先要获取图片的大小以及图片的通道个数等信息。图片的大小信息可以通过图片的行数、列数来反映，图片的通道个数可以直接调用opencv的库函数得到。

对于每个线程而言，其只负责计算图片某个特定区域进行卷积运算。由于卷积核大小为3\*3，且将原始图像第一行与最后一行的像素值置为0，因此卷积核设置为从图片第二行开始进行卷积。由于图片为png图片，有三个通道，因此输出图像的遍历指针除了与当前行的指针同步递增之外, 还应当以每行的每一个像素点的每一个通道值为一个递增量，这样方可完成对每个通道的卷积。

核心代码如下：

void \*parallel\_pthread(void\* pp)

{

int row\_start = (\*(int\*)pp) \* countt + 1;

int row\_end = row\_start + countt;

int i,j;

int channel = imagein.channels();//通道个数

//处理除最外围一圈外的所有像素值

for(i = row\_start; (i < row\_end)&&(i < imagein.rows - 1) ;i++) // 第一行和最后一行无法计算

{

const uchar\*pre = imagein.ptr<uchar>(i-1); // 上一行数据的指针

const uchar\*cur = imagein.ptr<uchar>(i);//当前行，第i行

const uchar\*next = imagein.ptr<uchar>(i+1);//下一行

uchar\*outData = imageout.ptr<uchar>(i);//输出图像的第i行

int startCol = channel;//每一行的开始处理点

int endCol = (imagein.cols-1)\* channel;//每一行的处理结束点

for(j=startCol; j < endCol; j++)

//输出图像的遍历指针与当前行的指针同步递增, 以每行的每一个像素点的每一个通道值为一个递增量, 因为要考虑到图像的通道数

{

outData[j] = saturate\_cast<uchar>( 5\*cur[j]-pre[j]-next[j]-cur[j-channel]-cur[j+channel]); //saturate\_cast<uchar>保证结果在uchar范围内

}

}

}

## 2.3 实验方案

开发环境：windows10+unbuntu16.04+opencv3.4.0

运行环境：unbuntu16.04

## 2.4 实验结果与分析

运行编译指令g++ shiyan2.cpp -lpthread -o shiyan2 `pkg-config --libs --cflags opencv`，生成可执行文件Lab2。

输入指令./shiyan2运行，输入线程数，进行卷积操作，输出图像卷积所的时间。

首先输入线程数为4,运行时间为0.757533ms。



图2-2 线程数为4时的运算时间

当输入线程数为1时可以认为是串行运算，运行时间为1.36425ms，相比线程数为4的运算时间，慢了将近1倍的时间。



图2-3线程数为1时的运算时间

进行卷积运算的原图如图2-4所示，程序运行生成的图片如图2-5所示。对比发现，相较于原图，生成的图片有了明显的锐化，实验结果正确。



图2-4 进行卷积运算的原图



图2-5 程序输出的图像

# 实验三

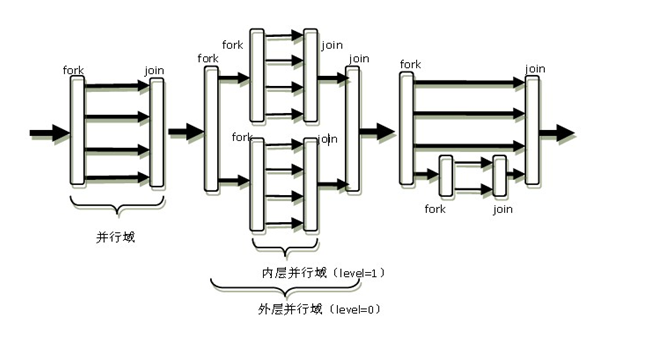
## 3.1 实验目的与要求

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. 然后对程序执行结果进行简单的分析和总结。

## 3.2 算法描述

### 3.2.1 OpenMP

OpenMP的执行模型采用fork-join的形式，其中fork创建新线程或者唤醒已有线程；join即多线程的会合。fork-join执行模型在刚开始执行的时候，只有一个称为“主线程”的运行线程存在。主线程在运行过程中，当遇到需要进行并行计算的时候，派生出线程来执行并行任务。在并行执行的时候，主线程和派生线程共同工作。在并行代码执行结束后，派生线程退出或者阻塞，不再工作，控制流程回到单独的主线程中。



OpenMP的编程者需要在可并行工作的代码部分用制导指令向编译器指出其并行属性，而且这些并行区域可以出现嵌套的情况，如上图所示。

### 3.2.2 设计思路

和单线程方法相同，只是在循环体上加了#pragma omp parallel for宏，让编译器自动进行多线程执行优化。

Mat imgConv(Mat imagein)

{

int row = imagein.rows;

int column = imagein.cols;

int channel = imagein.channels();

#pragma omp parallel for

for (int i = 1; i < row - 1; i++) {

const uchar\*pre = imagein.ptr<uchar>(i-1); // 上一行数据的指针

const uchar\*cur = imagein.ptr<uchar>(i);//当前行，第i行

const uchar\*next = imagein.ptr<uchar>(i+1);//下一行

uchar\*outData = imgout.ptr<uchar>(i);//输出图像的第i行

int startCol = channel;//每一行的开始处理点

int endCol = (imagein.cols-1)\* channel;//每一行的处理结束点

// #pragma omp parallel for

for(int j=startCol; j < endCol; j++)

//输出图像的遍历指针与当前行的指针同步递增, 以每行的每一个像素点的每一个通道值为一个递增量, 因为要考虑到图像的通道数

{

outData[j] = saturate\_cast<uchar>( 5\*cur[j]-pre[j]-next[j]-cur[j-channel]-cur[j+channel]); //saturate\_cast<uchar>保证结果在uchar范围内

}

}

return imgout;

}

}

## 3.3 实验方案

开发/运行环境：Windows10+ unbuntu16.04+opencv3.4.0

## 3.4 实验结果与分析

执行程序的演示如图 3.1所示，试验运行结果只有0.90909ms。



图 3.1 执行omp优化的图像卷积处理程序的结果

对比线程数为16的pthread方法进行卷积运算的运算时间，发现快了将近4倍。

进行卷积运算的原图如图3-2所示，程序运行生成的图片如图3-3所示。对比发现，相较于原图，生成的图片有了明显的锐化，实验结果正确。



图3-2进行卷积运算的原图



图3-3 OpenMP方法进行卷积运算输出的图片

# 实验四

## 4.1 实验目的与要求

1. 掌握使用MPI的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. 然后对程序执行结果进行简单的分析和总结。

## 4.2 算法描述

### 4.2.1 MPI编程思想

本实验实现算法与前几次实验基本相同，需要注意的地方主要是MPI接口函数的使用。主要核心接口有6个，可以简单的分为三类：

开始和结束MPI的接口：MPI\_Init、 MPI\_Finalize

获取进程状态的接口：MPI\_Comm\_rank、MPI\_Comm\_size

传输数据的接口：MPI\_Send、MPI\_Recv

每个接口的具体定义和使用方法如下：

* MPI\_Init(&argc, &argv) ：

初始化MPI执行环境，建立多个MPI进程之间的联系，为后续通信做准备。

* MPI\_Comm\_rank(communicator, &myid) ：

用来标识各个MPI进程的，给出调用该函数的进程的进程号,返回整型的错误值。两个参数：MPI\_Comm类型的通信域，标识参与计算的MPI进程组； &rank返回调用进程中的标识号。

* MPI\_Comm\_size(communicator, &numprocs) ：

用来标识相应进程组中有多少个进程。

* MPI\_Finalize() ：

结束MPI执行环境。

* MPI\_Send(buf,counter,datatype,dest,tag,comm) ：

buf：发送缓冲区的起始地址，可以是数组或结构指针；

count：非负整数，发送的数据个数；

datatype：发送数据的数据类型；

dest：整型，目的的进程号；

tag：整型，消息标志；comm：MPI进程组所在的通信域

含义:向通信域中的dest进程发送数据，数据存放在buf中，类型是datatype，个数是count，这个消息的标志是tag，用以和本进程向同一目的进程发送的其它消息区别开来。

* MPI\_Recv(buf,count,datatype,source,tag,comm,status) ：

source:整型，接收数据的来源，即发送数据进程的进程号；

status：MPI\_Status结构指针，返回状态信息。

### 4.2.3 设计思路

使⽤ MPI 进⾏图像处理的主要思想是：主线程读出图中的像素点信息，然后根据线程数将图按⾏均匀分割得到每个线程处理的最小行和最大行，然后调用MPI\_Send() 函数将像素点信息分发到对应的线程。子线程调⽤ MPI\_Recv() 函数接受主线程发送的像素点信息，根据像素点个数计算⾏数n，只需要处理从第 1 行到第 n-1 行的像素点信息。处理完之后将处理后的像素点信息整合到send\_buf 中，调⽤ MPI\_Send() 函数传给主线程。主线程接受⼦线程传来的处理之后的像素点信息，整合成新图。

程序设计的流程图如图所示：



这里要特别注意的是，主进程（进程号为0）一方面负责将图片分块发给各个结点，另一方面主进程也负责收集有各个节点发送过来的像素值，并利用这些像素值生成一张新的图片；而对于各个结点而言，主要负责对各自的图像区域进行卷积处理。

## 4.3 实验方案

开发/运行环境：Windows10+ CMake +Visual Studio2017

启用MPI的关键CMake代码段如下：

## 4.4 实验结果与分析

实验结果如图所示。

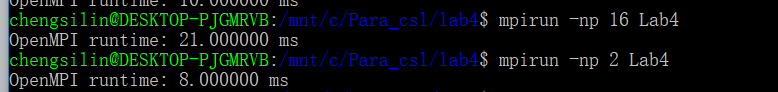


图4.1 进行卷积运算的原图

运行编译指令mpic++ -o Lab4 Lab4.cpp `pkg -config --libs --cflags opencv`，生成可执行文件Lab4。

输入指令mpirun -np 16 ./mpi运行文件，得到输出图像，其中16为线程数。当线程数为16时，利用MPI进行卷积运算的运行时间21ms。

根据执行结果可以发现，通过多次测试当开启的进程数为2的时候，并行执行的时间最短，仅约为8ms。

将输出的图片下载到本地后查看，原图如图4-1所示，输出图片如图4-2所示，可以看到有了明显的锐化，实验结果正确。



图4.2 进行卷积运算的原图



图4.3 MPI方法进行卷积运算的输出图片

通过分别对 pthread、 OpenMP 和 MPI 并⾏编程算法的实现，可以发现，这三种方式的基本思想都是对⼀个可并行的任务进行划分，将其分成多个小任务，分配给不同的线程并⾏执⾏以达到提⾼程序执⾏效率和性能的⽬的。不同的是，不同的实现方案在底层调用线程方式有所差异，导致三种并行编程实现对于程序的执行效率不尽相同。根据程序结构以及计算机硬件性能选择合适的并行编程方式，旨在能最大程度上地提高程序执行的效率。

# 实验五

## 5.1 实验目的与要求

（1）掌握使用CUDA进行并行编程设计和性能优化的基本原理和方法

（2）使用CUDA实现图像卷积运算的并行算法

（3）对程序执行结果进行简单的分析和总结；

（4）将其与实验二、实验三与实验四的结果进行比较。

## 5.2 算法描述

### 5.2.1 主函数设计思想

①读取源图片。定义imagein指针指向待处理的源图片，同时获取源图片的相关参数，并创建出一张新的图片。

②调用CudaMalloc函数在GPU中开辟一片内存区域，并通过cudaMemcpy经图片传入GPU。

③通过下面一行代码调用图像锐化函数完成对源图片的锐化处理。parallel\_cuda<<<grid,block>>>(dev\_src,dev\_dst,imagein.rows,imagein.cols,thread\_num,channel)

④将图片传回cpu，这一结果通过cudaMemcpy来实现。

### 5.2.3 图像锐化函数设计思想

利用实验二设计的卷积计算的算法，结合 CUDA 框架来实现计算的并行执 行。具体实现使用256个Block，每个Block内有512个Threads进行CUDA运算，采用上述算法分别进行简单算法测试及优化算法测试，将测试结果分别进行对于及同其他CPU上的并行算法效率进行对比，得出结论。

算法设计流程如下：

①计算线程在 Grid 中的索引。

int thread\_id1 = (blockIdx.x \* gridDim.x + blockIdx.y) \* blockDim.x \* blockDim.y + threadIdx.x \* blockDim.x + threadIdx.y

②初始化生成图中对应的元素

③读取卷积核原点 (xIndex,yIndex) 坐标周围的点，经过拉普拉斯算子后

\_\_global\_\_ void parallel\_cuda(uchar \*dev\_src,uchar \*dev\_dst,int row,int col,int NUM,int channel)

{

int thread\_id1 = (blockIdx.x \* gridDim.x + blockIdx.y) \* blockDim.x \* blockDim.y + threadIdx.x \* blockDim.x + threadIdx.y;

int block\_row = row / NUM + 1;

int i, j, row\_start, row\_end;

if(thread\_id1 < NUM){

row\_start = thread\_id1 \* block\_row + 1;

row\_end = row\_start + block\_row;

for(i = row\_start; (i < row\_end)&&(i<row-1) ;i++){

const uchar \*pre = dev\_src+(i-1)\*channel\*col;

const uchar \*cur = dev\_src+(i)\*channel\*col;

const uchar \*next = dev\_src+(i+1)\*channel\*col;

uchar \*outData = dev\_dst+(i)\*channel\*col;

for (j = channel; j < (col-1)\*channel; j++) {

int tmp = ( cur[j]\*9 - pre[j- channel]- pre[j]- pre[j + channel]-cur[j- channel]-cur[j + channel] -next[j- channel]-next[j]-next[j + channel]);

outData[j] = tmp;

}

}

}

}

## 5.3 实验方案

开发环境：windows10+visual studio2017+opencv3.0.0

运行环境：Xshell远程连接到Linux服务器

## 5.4 实验结果与分析

运行编译指令nvcc -o Lab5 Lab5.cu `pkg-config --libs --cflags opencv`，生成可执行文件Lab5，如图5-3所示。由图5-3可知，利用CUDA进行图像卷积运算的运行时间为0.703ms。



图5-3 利用CUDA进行卷积运算的运行时间

进行卷积运算的原图如图5-4所示，输出图片如图5-5所示，可以看到有了明显的锐化，实验结果正确。



图5-4 进行卷积运算的原图



图5-5 CUDA方法进行卷积运算的输出图片

在利用 CUDA 并行化后的执行时间大大减少，说明利用 CUDA 编程实现计算的并行化可以大大提高计算效率，性能较CPU上并行算法有很大提升。而且和 Pthread、OpenMP 以及 MPI 相比， CUDA 的执行时间要快于线程数为16时的Pthread ，但是慢于OpenMP和MPI，所以 CUDA 的执行效率要高于 Pthread，低于OpenMP和MPI。

# 实验六（选做）

## 6.1 实验目的与要求

（1）掌握使用MaoReduce技术实现K-means的方法

（2）掌握MapReduce模型的原理以及使用方法。

（3）掌握云计算环境下的分布式编程方法

## 6.2 算法描述

### 6.2.1聚类

将给定的多个对象分成若干组，组内的各个对象是相似的，组间的对象是不相似的。进行划分的过程就是聚类过程，划分后的组称为簇(cluster)。

几种聚类方法：

* 基于划分的方法；
* 基于层次的方法；
* 基于密度的方法；

### 6.2.2基于划分的聚类方法

给定N个对象，构造K个分组，每个分组就代表一个聚类。K个分组满足以下条件：(1)每个分组至少包含一个对象；(2)每个对象属于且仅属于一个分组。

K-Means算法是最常见和典型的基于划分的聚类方法

### 6.2.3 K-Means算法

**输入：**待聚类的N个数据点，期望生成的聚类的个数K

**输出：**K个聚类

**算法描述:**

选出K个点作为初始的cluster center

Loop:

对输入中的每一个点p：

｛

计算p到各个cluster的距离；

将p归入最近的cluster;

｝

重新计算各个cluster的中心

如果不满足停止条件，goto Loop; 否则，停止

### 6.2.4 MapReduce并行化聚类算法设计思路

将所有的数据分布到不同的MapReduce节点上，每个节点只对自己的数据进行计算。

每个Map节点能够读取上一次迭代生成的cluster centers，并判断自己的各个数据点应该属于哪一个cluster。

Reduce节点综合每个属于每个cluster的数据点，计算出新的cluster centers。

每一个节点需要访问如下的全局文件：当前的迭代计数和K个表示不同聚类中心的如下的数据结构：

｛

cluster id

cluster center

属于该cluster center的数据点的个数

｝

这是唯一的全局文件。

## 6.3 代码实现

### 6.3.1 Instance

该类对应文件中原始数据点的格式，以ArrayList存放数据点的各个分量。

需要编写加法，乘法，除法函数，用于计算簇中心：簇中所有点相加/簇中数据点的个数。

如果是在原有的簇中追加一个数据点，则用（簇中心点\*簇中数据点个数+新的数据点）/（簇中数据点个数+1）。

### 6.3.2 Cluster

该类记录簇的信息<簇的id，属于该簇的数据点的个数，簇中心>

各个节点共享的全局信息，记录每次划分的簇的信息。

### 6.3.3 EuclideanDistance

计算欧式距离，通过计算数据点与各个簇中心的距离，选择最小的欧式距离对应的簇中心作为该数据点属于的簇。

### 6.3.4 RandomClusterGenerator

该类用于初始时生成k个簇中心，一开始将读入的每个节点作为簇中心，并为其分配一个ID，当达到k个时，以1/(1+k)的概率返回一个[0,k-1]中的正整数，将其对应的簇ID的簇中心替换。

### 6.3.5 KMeans

该类即是kmeans算法的实现。

它的mapper类应该读入所有的簇中心的信息——kclusters。应在setup()时将之前计算的簇中心读入，作为全局文件。map()所要做的是，读取每个数据点寻找离该点最近的簇id，通过计算欧式距离，选择距离最小的那个簇中心，输出的格式是<clusterID，Instance>。

它的combiner类将同一个节点的数据点各个簇计算新的簇中心。簇中所有点相加/簇中数据点的个数=新的簇中心。

它的reducer类将不同节点的计算结果汇总，计算全局的新的簇中心。

### 6.3.6 KmeansCluster

该类做的工作其实与KMeans类差不多，只不过是没有KMeans的reducer类。

在收敛条件满足且所有簇中心的文件最后产生后，再对输入文件中的所有实例进行划分簇的工作，最后把所有实例按照(实例,簇id)的方式写进结果文件。

它的Mapper类也是在setup()时将之前计算的簇中心读入，作为全局文件。map()所要做的是，读取每个数据点寻找离该点最近的簇id，通过计算欧式距离，选择距离最小的那个簇中心，输出的格式是<Instance，clusterID >。

### 6.3.7 KMeansDriver

该类是用来启动整个MapReduce，启动参数包括k: 簇中心数，iteration num: 迭代数，input path: 输入路径，output path: 输出路径。

首先调用RandomClusterGenerator类的generateInitialCluster()初始化簇中心，然后用KMeans类进行簇中心的迭代计算，最后用KMeansCluster类为每个数据点选择最终确定的距离最近的簇。

## 6.4 实验方案

开发/运行环境：windows10+Eclipse

## 6.5 实验结果与分析

设置簇个数为5，迭代次数设置为10。源数据点如下图：

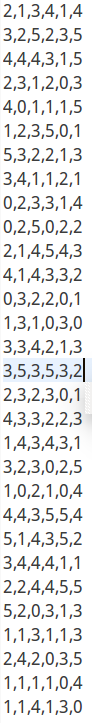


图6.1 源数据

聚类结果如下图

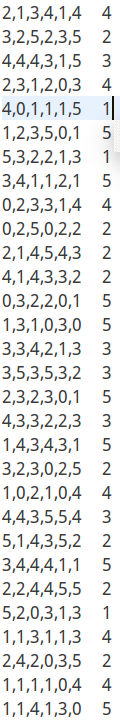


图6.2 聚类结果

k-means算法对初始cluster centers的选取会影响到最终的聚类结果，能得到局部最优解，不保证得到全局最优解。

# 项目：并行优化广度优先搜索

## 7.1实验目的

* 使用掌握两个典型的并行程序开发工具（OpenMP和MPI）对广度优先搜索进行优化
* 探索了解两个工具在并行程序设计和优化过程中的异同
* 调整并行算法的并行粒度，并分析优化结果
* 进一步了解并行编程的原理以及应该注意什么

## 7.2实验假设

在选取的粒度恰当时，使用并行程序开发工具OpenMp和MPI均能改善广度优先搜索的效率。MPI的改善或更佳。

## 7.3实验方案

在本项目中，采用的是图的广度优先搜索来测试OpenMp和MPI对广度优先搜索效率的改善。

图的数据结构采用的是邻接矩阵，即：

int adjac[N][N] = { 0 }，其中N是收订输入的，用于确定图中的节点个数。图的生成采用了随机生成的方式，即：

for (i = 0; i<N; i++)

for (j = i; j<N; j++){

weight=myRand(i,j);

adjacency[i][j]=weight;

adjacency[j][i]=weight;

### 7.3.1 广度优先搜索图的基本原理

所谓广度，就是一层一层的，向下遍历，层层堵截，还是这幅图，我们如果要是广度优先遍历的话，我们的结果是V1 V2 V3 V4 V5 V6 V7 V8。

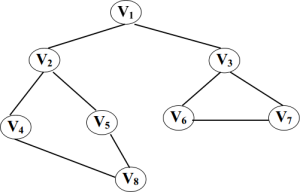


图7.1 广度优先搜索

广度优先搜索的思想：

① 访问顶点vi ；

② 访问vi 的所有未被访问的邻接点w1 ,w2 , …wk ；

③ 依次从这些邻接点（在步骤②中访问的顶点）出发，访问它们的所有未被访问的邻接点; 依此类推，直到图中所有访问过的顶点的邻接点都被访问；

说明：

为实现③，需要保存在步骤②中访问的顶点，而且访问这些顶点的邻接点的顺序为：先保存的顶点，其邻接点先被访问。 这里我们就想到了用标准模板库中的queue队列来实现这种先进现出的服务。

### 7.3.2 使用并行方法进行广度优先搜索图

在本项目中分别采用了openMP和MPI的方式对图进行了并行搜索优化。OpenMP的方法较为简单，只需在广度优先搜索便利每个点的for语句前加上一句并行标识语即可：

#pragma omp parallel for

for(i=0; i<N; i++)

{

if(adjacency[head][i]==1&&flag[i]==0)

{

flag[i]=1;

vertex.push(i);

}

}

而使用MPI的方式对图进行并行搜索优化，则用到了MPI中的相关通信。在使用MPI方式时，主进程每一次将当前i的标记信息flag和当前的出发点head发送给子进程。

MPI\_Send(head, 1,MPI\_INT,i+1, 0, MPI\_COMM\_WORLD);

MPI\_Send(flag+i-1, 1,MPI\_INT,i+1, 0, MPI\_COMM\_WORLD);

而子进程在接受到来自MPI的信息后，判断该点是否需要加入到队列中，再将判断信息返回给主进程。

MPI\_Send(&isNeed, 1,MPI\_INT,i+1, 0, MPI\_COMM\_WORLD);

经过上述步骤，便能成功地实现广度优先搜索的并行。

## 7.4 实验结果

选用不同的矩阵大小，分别采用OpenMP模式和MPI模式进行广度优先图搜索的执行。结果如表7.1所示。

表7.1 OpenMP和MPI执行结果对比

|  |  |  |
| --- | --- | --- |
| 节点个数 | OpenMP | MPI |
| 400 |  |  |
| 2500 |  |  |
| 6400 |  |  |

由于本机资源有限，再继续增加节点个数使得资源一处，造成无法计数执行。

## 7.5讨论与总结

项目结果表明，OpenMP能够对BFS算法进行并行的优化。

从上述结果中可以看出，当节点个数较少时， OpenMP和MPI的速度差不多，但随着节点个数的增长，在节点个数到达6400时，OpenMP的效率开始超过顺序执行的效率。而MPI的速度却一直要大于前者。

出现上述结果的原因可能是不同并行编程方法的特点所决定的。

OpenMP是一套支持跨平台共享内存方式的多线程并发的编程API，其采用可移植的、可扩展的模型，为程序员提供了一个简单而灵活的开发平台。OpenMP相对于MPI而言更容易使用,其对原串行代码改动较小，可以保护代码原貌，使得代码更容易理解和维护，并且允许渐进式并行化。但是所有线程共享内存空间，硬件制约较大 ，因此目前主要针对循环并行化

而MPI无论硬件是否共享内存空间，都可以使用。（但是线程间不共享内存空间）。其与OpenMP相比，可以处理规模更大的问题 。

因此我们在选用并行算法时，要考虑所使用的平台。就单主机而言，OpenMP会是更优的选择，可以用其来对BFS算法进行优化。

# 附录

实验六和项目由于占空间大，就没有打包，而是把代码粘贴在附录中了。

## 实验六

public class Instance implements Writable{

ArrayList<Double> value;

public Instance(){

value = new ArrayList<Double>();

}

public Instance(String line){

String[] valueString = line.split(",");

value = new ArrayList<Double>();

for(int i = 0; i < valueString.length; i++){

value.add(Double.parseDouble(valueString[i]));

}

}

public Instance(Instance ins){

value = new ArrayList<Double>();

for(int i = 0; i < ins.getValue().size(); i++){

value.add(new Double(ins.getValue().get(i)));

}

}

public ArrayList<Double> getValue(){

return value;

}

public Instance add(Instance instance){

if(value.size() == 0)

return new Instance(instance);

else if(instance.getValue().size() == 0)

return new Instance(this);

else if(value.size() != instance.getValue().size())

try {

throw new Exception("can not add! dimension not compatible!" + value.size() + ","

+ instance.getValue().size());

} catch (Exception e) {

e.printStackTrace();

return null;

}

else{

Instance result = new Instance();

for(int i = 0;i < value.size(); i++){

result.getValue().add(value.get(i) + instance.getValue().get(i));

}

return result;

}

}

public Instance multiply(double num){

Instance result = new Instance();

for(int i = 0; i < value.size(); i++){

result.getValue().add(value.get(i) \* num);

}

return result;

}

public Instance divide(double num){

Instance result = new Instance();

for(int i = 0; i < value.size(); i++){

result.getValue().add(value.get(i) / num);

}

return result;

}

@Override

public void write(DataOutput out) throws IOException {

out.writeInt(value.size());

for(int i = 0; i < value.size(); i++){

out.writeDouble(value.get(i));

}

}

@Override

public void readFields(DataInput in) throws IOException {

int size = 0;

value = new ArrayList<Double>();

if((size = in.readInt()) != 0){

for(int i = 0; i < size; i++){

value.add(in.readDouble());

}

}

}

}

public class Cluster implements Writable{

private int clusterID;

private long numOfPoints;

private Instance center;

public Cluster(){

this.setClusterID(-1);

this.setNumOfPoints(0);

this.setCenter(new Instance());

}

public Cluster(int clusterID,Instance center){

this.setClusterID(clusterID);

this.setNumOfPoints(0);

this.setCenter(center);

}

public Cluster(String line){

String[] value = line.split(",",3);

clusterID = Integer.parseInt(value[0]);

numOfPoints = Long.parseLong(value[1]);

center = new Instance(value[2]);

}

@Override

public void write(DataOutput out) throws IOException {

out.writeInt(clusterID);

out.writeLong(numOfPoints);

center.write(out);

}

@Override

public void readFields(DataInput in) throws IOException {

clusterID = in.readInt();

numOfPoints = in.readLong();

center.readFields(in);

}

}

public interface Distance<T> {

double getDistance(List<T> a,List<T> b) throws Exception;

}

public class EuclideanDistance<T extends Number> implements Distance<T> {

public double getDistance(List<T> a, List<T> b) throws Exception {

if(a.size() != b.size())

throw new Exception("size not compatible!");

else{

double sum = 0.0;

for(int i = 0;i < a.size();i++){

sum += Math.pow(a.get(i).doubleValue() - b.get(i).doubleValue(), 2);

}

sum = Math.sqrt(sum);

return sum;

}

}

}

public final class RandomClusterGenerator {

private int k;

private FileStatus[] fileList;

private FileSystem fs;

private ArrayList<Cluster> kClusters;

private Configuration conf;

public RandomClusterGenerator(Configuration conf,String filePath,int k){

this.k = k;

try {

fs = FileSystem.get(URI.create(filePath),conf);

fileList = fs.listStatus((new Path(filePath)));

kClusters = new ArrayList<Cluster>(k);

this.conf = conf;

} catch (IOException e) {

e.printStackTrace();

}

}

public void generateInitialCluster(String destinationPath){

Text line = new Text();

FSDataInputStream fsi = null;

try {

for(int i = 0;i < fileList.length;i++){

fsi = fs.open(fileList[i].getPath());

LineReader lineReader = new LineReader(fsi,conf);

while(lineReader.readLine(line) > 0){

Instance instance = new Instance(line.toString());

makeDecision(instance);

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

fsi.close();

} catch (IOException e) {

e.printStackTrace();

}

}

writeBackToFile(destinationPath);

}

public void makeDecision(Instance instance){

if(kClusters.size() < k){

Cluster cluster = new Cluster(kClusters.size() + 1, instance);

kClusters.add(cluster);

}else{

int choice = randomChoose(k);

if(!(choice == -1)){

int id = kClusters.get(choice).getClusterID();

kClusters.remove(choice);

Cluster cluster = new Cluster(id, instance);

kClusters.add(cluster);

}

}

}

public int randomChoose(int k){

Random random = new Random();

if(random.nextInt(k + 1) == 0){

return new Random().nextInt(k);

}else

return -1;

}

public void writeBackToFile(String destinationPath){

Path path = new Path(destinationPath + "cluster-0/clusters");

FSDataOutputStream fsi = null;

try {

fsi = fs.create(path);

for(Cluster cluster : kClusters){

fsi.write((cluster.toString() + "\n").getBytes());

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

fsi.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

public class KMeans {

public static class KMeansMapper extends Mapper<LongWritable,Text,IntWritable,Cluster>{

private ArrayList<Cluster> kClusters = new ArrayList<Cluster>();

protected void setup(Context context) throws IOException,InterruptedException{

super.setup(context);

FileSystem fs = FileSystem.get(context.getConfiguration());

FileStatus[] fileList = fs.listStatus(new Path(context.getConfiguration().get("clusterPath")));

BufferedReader in = null;

FSDataInputStream fsi = null;

String line = null;

for(int i = 0; i < fileList.length; i++){

if(!fileList[i].isDir()){

fsi = fs.open(fileList[i].getPath());

in = new BufferedReader(new InputStreamReader(fsi,"UTF-8"));

while((line = in.readLine()) != null){

Cluster cluster = new Cluster(line);

cluster.setNumOfPoints(0);

kClusters.add(cluster);

}

}

}

in.close();

fsi.close();

}

public void map(LongWritable key, Text value, Context context)throws

IOException, InterruptedException{

Instance instance = new Instance(value.toString());

int id;

try {

id = getNearest(instance);

if(id == -1)

throw new InterruptedException("id == -1");

else{

Cluster cluster = new Cluster(id, instance);

cluster.setNumOfPoints(1);

context.write(new IntWritable(id), cluster);

}

} catch (Exception e) {

e.printStackTrace();

}

}

public int getNearest(Instance instance) throws Exception{

int id = -1;

double distance = Double.MAX\_VALUE;

Distance<Double> distanceMeasure = new EuclideanDistance<Double>();

double newDis = 0.0;

for(Cluster cluster : kClusters){

newDis = distanceMeasure.getDistance(cluster.getCenter().getValue()

, instance.getValue());

if(newDis < distance){

id = cluster.getClusterID();

distance = newDis;

}

}

return id;

}

}

public static class KMeansCombiner extends Reducer<IntWritable,Cluster,IntWritable,Cluster>{

public void reduce(IntWritable key, Iterable<Cluster> value, Context context)throws

IOException, InterruptedException{

Instance instance = new Instance();

int numOfPoints = 0;

for(Cluster cluster : value){

numOfPoints += cluster.getNumOfPoints();

instance = instance.add(cluster.getCenter().multiply(cluster.getNumOfPoints()));

}

Cluster cluster = new Cluster(key.get(),instance.divide(numOfPoints));

cluster.setNumOfPoints(numOfPoints);

System.out.println("combiner emit cluster:" + cluster.toString());

context.write(key, cluster);

}

}

public static class KMeansReducer extends Reducer<IntWritable,Cluster,NullWritable,Cluster>{

public void reduce(IntWritable key, Iterable<Cluster> value, Context context)throws

IOException, InterruptedException{

Instance instance = new Instance();

int numOfPoints = 0;

for(Cluster cluster : value){

numOfPoints += cluster.getNumOfPoints();

instance = instance.add(cluster.getCenter().multiply(cluster.getNumOfPoints()));

}

Cluster cluster = new Cluster(key.get(),instance.divide(numOfPoints));

cluster.setNumOfPoints(numOfPoints);

context.write(NullWritable.get(), cluster);

}

}

}

public class KMeansCluster {

public static class KMeansClusterMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

private ArrayList<Cluster> kClusters = new ArrayList<Cluster>();

protected void setup(Context context) throws IOException,InterruptedException{

super.setup(context);

FileSystem fs = FileSystem.get(context.getConfiguration());

FileStatus[] fileList = fs.listStatus(new Path(context.getConfiguration().get("clusterPath")));

BufferedReader in = null;

FSDataInputStream fsi = null;

String line = null;

for(int i = 0; i < fileList.length; i++){

if(!fileList[i].isDir()){

fsi = fs.open(fileList[i].getPath());

in = new BufferedReader(new InputStreamReader(fsi,"UTF-8"));

while((line = in.readLine()) != null){

System.out.println("read a line:" + line);

Cluster cluster = new Cluster(line);

cluster.setNumOfPoints(0);

kClusters.add(cluster);

}

}

}

in.close();

fsi.close();

}

public void map(LongWritable key, Text value, Context context)throws

IOException, InterruptedException{

Instance instance = new Instance(value.toString());

int id;

try {

id = getNearest(instance);

if(id == -1)

throw new InterruptedException("id == -1");

else{

context.write(value, new IntWritable(id));

}

} catch (Exception e) {

e.printStackTrace();

}

}

public int getNearest(Instance instance) throws Exception{

int id = -1;

double distance = Double.MAX\_VALUE;

Distance<Double> distanceMeasure = new EuclideanDistance<Double>();

double newDis = 0.0;

for(Cluster cluster : kClusters){

newDis = distanceMeasure.getDistance(cluster.getCenter().getValue()

, instance.getValue());

if(newDis < distance){

id = cluster.getClusterID();

distance = newDis;

}

}

return id;

}

}

}

public class KMeansDriver {

private int k;

private int iterationNum;

private String sourcePath;

private String outputPath;

private Configuration conf;

public KMeansDriver(int k, int iterationNum, String sourcePath, String outputPath, Configuration conf){

this.k = k;

this.iterationNum = iterationNum;

this.sourcePath = sourcePath;

this.outputPath = outputPath;

this.conf = conf;

}

public void clusterCenterJob() throws IOException, InterruptedException, ClassNotFoundException{

for(int i = 0;i < iterationNum; i++){

Job clusterCenterJob = new Job();

clusterCenterJob .setJobName("clusterCenterJob" + i);

clusterCenterJob .setJarByClass(KMeans.class);

clusterCenterJob.getConfiguration().set("clusterPath", outputPath + "/cluster-" + i +"/"); clusterCenterJob.setMapperClass(KMeans.KMeansMapper.class);

clusterCenterJob.setMapOutputKeyClass(IntWritable.class);

clusterCenterJob.setMapOutputValueClass(Cluster.class); clusterCenterJob.setCombinerClass(KMeans.KMeansCombiner.class);

clusterCenterJob.setReducerClass(KMeans.KMeansReducer .class);

clusterCenterJob.setOutputKeyClass(NullWritable.class);

clusterCenterJob.setOutputValueClass(Cluster.class); FileInputFormat.addInputPath(clusterCenterJob, new Path(sourcePath));

FileOutputFormat.setOutputPath(clusterCenterJob, new Path(outputPath + "/cluster-" + (i + 1) +"/"));

clusterCenterJob.waitForCompletion(true);

System.out.println("finished!");

}

}

public void KMeansClusterJob() throws IOException, InterruptedException, ClassNotFoundException{

Job kMeansClusterJob = new Job();

kMeansClusterJob.setJobName("KMeansClusterJob");

kMeansClusterJob.setJarByClass(KMeansCluster.class);

kMeansClusterJob.getConfiguration().set("clusterPath", outputPath + "/cluster-" + (iterationNum - 1) +"/");

kMeansClusterJob.setMapperClass(KMeansCluster.KMeansClusterMapper.class);

kMeansClusterJob.setMapOutputKeyClass(Text.class);

kMeansClusterJob.setMapOutputValueClass(IntWritable.class); kMeansClusterJob.setNumReduceTasks(0); FileInputFormat.addInputPath(kMeansClusterJob, new Path(sourcePath));

FileOutputFormat.setOutputPath(kMeansClusterJob, new Path(outputPath + "/clusteredInstances" + "/"));

kMeansClusterJob.waitForCompletion(true);

System.out.println("finished!");

}

public void generateInitialCluster(){

RandomClusterGenerator generator = new RandomClusterGenerator(conf, sourcePath, k);

generator.generateInitialCluster(outputPath + "/");

}

public static void main(String[] args) throws IOException, InterruptedException, ClassNotFoundException{

System.out.println("start");

Configuration conf = new Configuration();

int k = Integer.parseInt(args[0]);

int iterationNum = Integer.parseInt(args[1]);

String sourcePath = args[2];

String outputPath = args[3];

KMeansDriver driver = new KMeansDriver(k, iterationNum, sourcePath, outputPath, conf);

driver.generateInitialCluster();

System.out.println("initial cluster finished");

driver.clusterCenterJob();

driver.KMeansClusterJob();

}

}

## 项目

#include <iostream>

#include <queue>

#include <stdlib.h>

#include <math.h>

#include <sys/time.h>

#include <stdio.h>

#include <time.h>

#include <sys/time.h>

using namespace std;

int N;

int adjacency[105][105]={0};

/\*

\*功能：随机生成邻接矩阵，25%的概率两个顶点之间不存在一条边

\*

\*/

int myRand(int row, int col)

{

if(row==col)

return 0;

int temp=rand()%4;

if(temp<3)

return 1;

else

return 0;

}

/\*

\*功能：随机生成邻接矩阵

\*

\*/

void initMap()

{

int i,j,weight;

cout<<"Please input the number of vertices:";

cin>>N;

for(i=0; i<N; i++)

for(j=i; j<N; j++)

{

weight=myRand(i,j);

adjacency[i][j]=weight;

adjacency[j][i]=weight;

}

//调试状态下输出邻接矩阵

}

/\*

\*功能：宽度优先搜索

\*

\*/

void boradSearch(int start)

{

queue<int> vertex;;

int flag[N]={0};

vertex.push(start);

flag[start]=1;

int head,i ;

while(!vertex.empty())

{

head=vertex.front();

vertex.pop();

#pragma omp parallel for

for(i=0; i<N; i++)

{

if(adjacency[head][i]==1&&flag[i]==0)

{

flag[i]=1;

vertex.push(i);

}

}

}

cout<<endl;

}

double get\_time()//获取当前时间

{

double time[2];

struct timeval time\_tmp;

gettimeofday(&time\_tmp, NULL);

time[0]=time\_tmp.tv\_sec;//秒数

time[1]=time\_tmp.tv\_usec;//微秒数

return time[0]+time[1]\*1.0e-6;

}

int main(int argc, char\*\* argv)

{

initMap();

double start = get\_time();

boradSearch(0);//从顶点0开始遍历随机图

double stop = get\_time();

double Parallel\_runTime = stop - start;

cout << "time in OPENMP:" << Parallel\_runTime << endl;

return 0;

}