

# 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 周俊辰

专业班级 CS1402

学 号 U201414608

指导教师 金海

报告日期

## 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：对于每个工具，挑选一种简单的可以并行化的算法，配置框架、编写程序并进行运行演示。

例如，可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std::thread做向量加法**

由于在Windows平台下进行实验，在这里选择C++11标准库提供的std::thread进行实验。std::thread可以提供和pthread完全等价的功能。

为了演示std::thread的功能，实现一个简单的“并行加法”程序：每个进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：

data = { {1, 2, 3, 4, 5}, {2, 3, 4, 5, 6}, … }

sum = 0

for each array in data:

spawn worker thread add(array)

for each worker thread:

join thread and get result

sum = sum + result

print sum

使用std::thread创建线程。为了将数据从worker线程传递到main线程，使用了<future>头文件提供的std::promise。

为了更好的展示效果，在每个worker进程执行结束后，会单独打印其计算结果。这里为了让不同线程打印的内容不相互干扰，设置了一个静态的互斥锁变量std::mutex m，并在打印时占用该锁。

**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，可以将基本向量加法程序的源程序修改成如下形式：

#pragma omp parallel for

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

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

OpenMP不需要专门的库，而是内建在编译器中的。使用CMake进行项目构建时，可以使用FindOpenMP库来检测OpenMP支持，以及设置OpenMP编译相关的flags。

**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, &world\_rank); //获取进程标识符

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);//获取进程数

elems\_per\_proc = N / world\_size; // 每个进程处理的元素数

MPI\_Scatter(B, elems\_per\_proc, MPI\_INT,

B\_recv, elems\_per\_proc, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(C, elems\_per\_proc, MPI\_INT,

C\_recv, elems\_per\_proc, MPI\_INT, 0, MPI\_COMM\_WORLD);

for (i in 0 until 100)

B\_recv[i] = B\_recv[i] + C\_recv[i]

MPI\_Gather(B\_recv, elems\_per\_proc, MPI\_INT, A, elems\_per\_proc, MPI\_INT, 0, MPI\_COMM\_WORLD) // 收集计算结果

if world\_rank == 0 then print all elements in A // 在主进程打印结果

MPI\_Finalize();//结束MPI环境

将要进行计算的B与C向量划分为等大小（elems\_per\_proc）的数据块，通过MPI\_Scatter分布到每个子进程中进行计算，再通过MPI\_Gather函数将数据汇总到主进程，并在主进程输出结果。

**1.2.4 使用CUDA做向量加法**

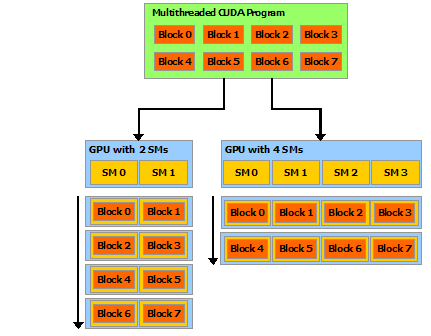


图 1.1 CUDA编程模型

CUDA的编程模型如图 1.1所示。CUDA程序为Host，在调用Kernel函数的时候访问Device。Device中的计算是并行执行，最小的执行单元是线程（Thread），每个Block可同时执行多个Thread，每个Grid中可同时执行多个Block。

通过两个三维索引blockIdx和threadIdx来获取当前函数调用所在的block/thread信息。Kernel函数的定义如下：

\_\_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];

}

}

每个thread执行一个索引的加法运算。通过blockIdx和threadIdx计算出该thread对应计算的索引号，并执行计算。

调用kernel的过程如下：

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);

1.3 实验结果

**1.3.1 std::thread方法**

编译运行：

* 在lab1/exp1-thread/文件夹中执行cmake .创建工程
* 使用平台相关的编译方法进行编译运行

测试结果如图 1.2所示。

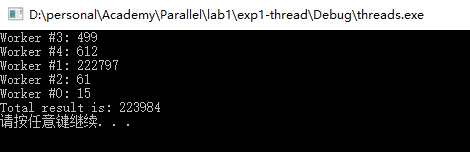


图 1.2 pthread方法示例

设置了5个数组分别执行加法工作。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**

编译运行：

* 在lab1/exp2-openmp文件夹中cmake .创建工程
* 使用平台相关的构建方法进行构建运行

为了让实验结果更为直观，在循环体中可以通过omp\_get\_thread\_num()函数获取每次执行工作时的进程id并打印出来。测试结果如图 1.3所示。

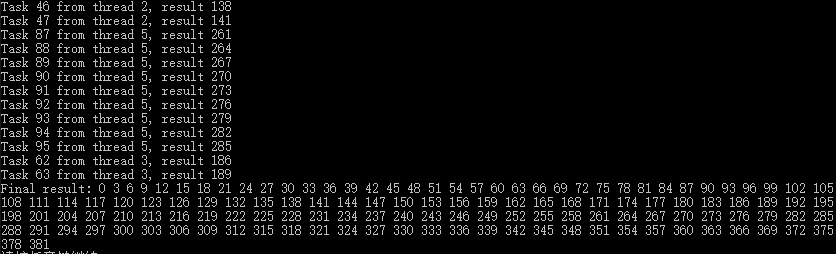


图 1.3 OpenMP计算向量加法样例, n=128

**1.3.3 OpenMPI方法**

编译运行：

* 在lab1/exp3-mpi文件夹中cmake .创建工程
* 使用平台相关的构建方法进行构建运行

将B和C数组划分成等大小的数据块，并分布到每个进程。每个进程对获取的数据块执行加法计算。运行效果如图 1.4所示。

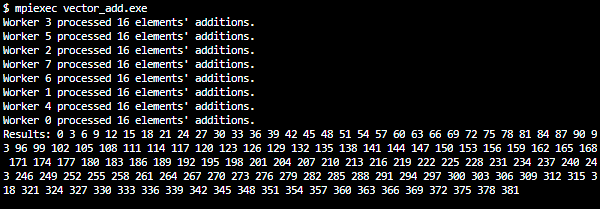


图 1.4 OpenMPI方法计算向量加法, n=128

**1.3.4 CUDA方法**

编译运行：

* 在lab1/exp4-cuda中cmake -A x64 .创建工程
  + 如果使用windows：需要先安装CUDA。目前CUDA 9.2对Visual Studio 2017的支持有问题。如果安装了VS2017，需先额外安装msvc140 toolset，并且使用cmake -A x64 -G “Visual Studio 14 2015”指令生成vs2015编译器的项目。
* 使用平台相关的构建方法进行构建运行

运行结果如图 1.5所示。可以看到加法计算结果正确。

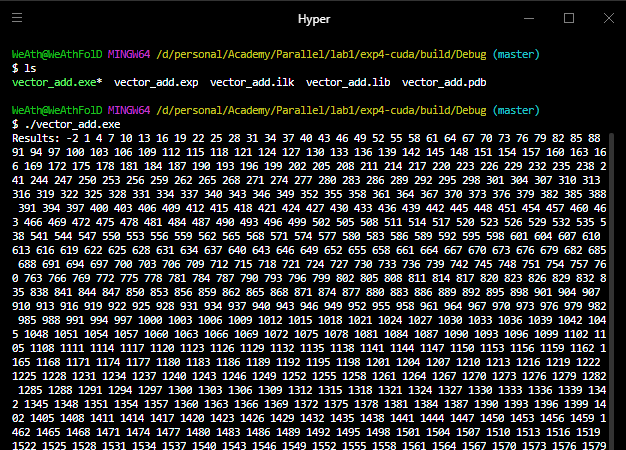


图 1.5 CUDA向量加法运行结果

## 实验二

2.1 实验目的与要求

1. 掌握使用thread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thread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：**

在这里选择dilation（扩张）算法进行实现。后续所有涉及形态学图像处理处皆使用dilation卷积核，不再重复说明。

function morph\_dilation(Image source, Image dest):

for each pixel position (x, y) in source:

hit = false

for (dx, dy) in (-1, -1) … (1, 1):

if source pixel is set (x + dx, y + dy):

hit = true

break

set dest pixel (x, y) to hit ? 1 : 0

**std::thread方法：**

使用thread进行并行化本质上是将上述的顺序算法分解成多个线程。在这里将图像数据考虑为一个大小为width \* height的连续数组，将数组尽量均匀的划分到thread\_count（可调整）个线程，每个线程负责一块连续区域。算法如下：

function worker(Image source, Image dest, int offset, int count):

for i in offset until offset + count:

y = i / source.width // 反算出像素坐标

x = i / source.width

bool hit = false

for (dx, dy) in (-1, -1) … (1, 1):

if source pixel is set (x + dx, y + dy):

hit = true

break

set dest pixel (x, y) to hit ? 1 : 0

function morph\_dilation(Image source, Image dest):

thread\_count = ??? // 设置线程数

size = source.width \* source.height

per\_thread = size / thread\_count // 计算单个线程负责像素的数量

for i in 0 until thread\_count: // 生成线程

spawn thread worker(source, dest, per\_thread \* i, per\_thread)

join all child threads // 等待所有线程执行结束

2.3 实验方案

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

由于后续实验lab2-lab5中都要以不同的方式实现形态学图像处理，且都有相同的读取图像、写图像、计算运行时间的需求，先编写了一个简单的库morphlib来执行上述操作。它定义了一个morph命名空间，其中包含：

* BinaryImage类，可以读取二进制图数据和写出数据
* 函数类型test\_func，是执行形态学图像处理操作的核心函数。每个lab中只需要重新实现一个test\_func即可。
* 函数exec\_test，它接受程序的命令行参数和一个test\_func，并读取文件、调用test\_func并测量执行时间，最后输出运算后的图像文件。

为了方便C++代码的编写，定义了一种简单的二值图像的二进制格式，它的构成是：

1. 两个32位无符号整数，分别代表width和height；
2. 余下长度为(width\*height)字节的图像数据。每个字节非0即1，代表该像素是否为白。

又通过Kotlin（它可以调用java的图像读取、写函数，对各种图片格式都有较好的支持）编写了一个简单的命令行工具，来将一般图片格式和这种二值格式相互转换。其用法如下：

* Java -jar imgcvt.jar b image.png binary-image.bin // 将png转换为二值图
* Java -jar imgcvt.jar p binary-image.bin image.png // 将二值图转换为png

在周边工具编写完成之后，首先编写了一个线性执行dilation操作的程序，用于参考和后续数据比较，最后才用thread的形式实现了dilation操作。

2.4 实验结果与分析



图 2.1 dilation形态学图像处理程序运行效果

对大小为900\*600的二值图片进行dilation操作的效果如图 2.1所示。由于后续lab2-lab5的执行结果都完全相同（在运行正确的前提下），运行结果将不会重复说明。

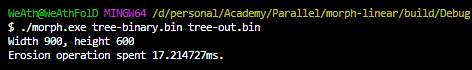


图 2.2 线性计算程序的执行示例

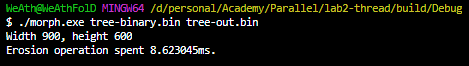


图 2.3 std::thread多线程计算的执行示例

## 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

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。

function morph\_dilation(Image source, Image dest):

#pragma omp parallel for

for each pixel position (x, y) in source:

hit = false

#pragma omp parallel for

for (dx, dy) in (-1, -1) … (1, 1):

if source pixel is set (x + dx, y + dy):

hit = true

break

set dest pixel (x, y) to hit ? 1 : 0

3.3 实验方案

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

让编译器开启OpenMP的CMake关键代码如下：

include (FindOpenMP)

if (OPENMP\_FOUND)

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} ${OpenMP\_CXX\_FLAGS}")

else()

message("WARNING: OpenMP not supported!")

endif()

C++代码方面，只需要在实验二中所写的顺序执行代码的基础上，在两个循环体的上方添加#pragma omp parallel for宏即可。

3.4 实验结果与分析

执行程序的演示如图 3.1所示。

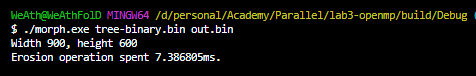


图 3.1 执行omp优化的形态学图像处理程序的结果

## 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

将要处理的图像纵向划分为MPI的world\_size个子区域，分配到每个rank单独进行处理。需要注意的是，在卷积采样时上边界和下边界要多采样一个像素，因此每个rank的接受buffer大小是width \* (2 + grid\_height)，但结果buffer大小只有width \* grid\_height。

对于每个rank，首先通过MPI\_Bcast将图像元信息（width，grid\_height）广播到所有节点，再使用MPI\_Send将每个rank处理的数据发送。整体执行逻辑的伪代码如下：

MPI\_Init(nullptr, nullptr)

world\_size, rank = Get MPI Comm size and rank

if rank == 0:

image = read image

width = image.width

grid\_height = ceil(image.height / world\_size) // 计算每个rank负责区域

padded\_height = grid\_height \* world\_size // 数组长度对齐

buf = allocate [(2 + padded\_height) \* width] bytes

set row 0 of buf to 0’s

set row 1…image.height of buf to copy of image’s buffer

set rest of buf to 0’s

for i in 0…world\_size:

use MPI\_Send to send part of buf

use MPI\_Recv to receive processing data

use MPI\_Bcast to broadcast width and grid\_height

perform convolution on part of the data

use MPI\_Send to send the result to root process

if rank == 0:

use MPI\_Recv to receive all results

write to target image

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：

include (FindMPI)

if (NOT MPI\_FOUND)

message(FATAL\_ERROR "No MPI implementation has been found.")

endif ()

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} ${MPI\_CXX\_COMPILER\_FLAGS}")

set(LINK\_FLAGS "${LINK\_FLAGS} ${MPI\_CXX\_LINK\_FLAGS}")

add\_executable(morph …) // 添加源文件，在此略过

target\_link\_libraries(morph ${MPI\_LIBRARIES})

include\_directories(${MPI\_INCLUDE\_PATH})

编译运行：

* 使用cmake创建Visual Studio 2017工程
* 在VS2017工程中build，生成可执行文件
* 在Debug/文件夹中，执行：
* mpiexec morph.exe <输入文件> <输出文件>

4.4 实验结果与分析

执行程序的演示如图 4.1所示。

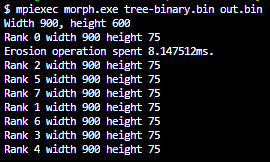


图 4.1 执行MPI优化的图像处理程序结果

在执行程序时发现一个问题——在处理的图像较小时，可以正常的完成执行；但较大时，则会卡死失去响应。经过仔细的检查，发现代码中存在一个竞态条件：当数据量较大时，MPI\_Send会阻塞，我们在向每个rank发送源数据的时候用到了MPI\_Send。在这之后紧接着是使用MPI\_Bcast广播文件的元信息，然后是每个非0的子rank通过MPI\_Recv接受信息。可是对于根进程来说，如果它在MPI\_Send处阻塞了，就会导致它永远无法调用到Bcast，而其他的子rank此时已经调用了Bcast，在等待根进程调用Bcast。这样，根进程和子进程分别对两个永远不会得到释放的资源进行等待，形成了一个死锁。

通过使用非阻塞的MPI\_Isend函数，在Bcast执行完之后，再通过MPI\_Waitall手动的延迟阻塞，问题就解决了。

## 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. 对程序的运行结果进行简单的分析和总结
4. 基于运行结果和硬件环境提出优化的解决方案
5. 将结果与Lab2、Lab3和Lab4的结果进行比对

5.2 算法描述

CUDA Kernel代码如下所示。每个thread负责一个像素的采样操作。通过thread反算像素位置的公式为：

**pos** = **blockIdx** \* **blockDim** + **threadIdx**

\_\_global\_\_ void dilation(char \*src, char\*dst, int width, int height) {

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

int y = blockIdx.y \* blockDim.y + threadIdx.y;

if (x >= width || y >= height)

return;

int set = 0;

for (int i = 0; i < 9; ++i) {

int x2 = x + (i / 3) - 1;

int y2 = y + (i % 3) - 1;

if (x2 >= 0 && y2 >= 0 && x2 < width && y2 < height) {

int idx2 = y2 \* width + x2;

if (src[idx2]) {

set = 1;

}

}

}

int idx = y \* width + x;

dst[idx] = set;

}

调用CUDA Kernel的代码如下所示。

char \*devSrc, \*devDst;

size\_t size = src.width \* src.height;

// 分配设备端内存

cudaMalloc((void\*\*) &devSrc, size);

cudaMalloc((void\*\*) &devDst, size);

dim3 threadsPerBlock = dim3(src.width, src.height);

dim3 blocksPerGrid = dim3(1);

// 图像 memory -> device

cudaMemcpy(devSrc, src.bytes, size, cudaMemcpyHostToDevice);

// 调用Kernel

dilation<<<blocksPerGrid, threadsPerBlock>>>(devSrc, devDst, src.width, src.height);

// 结果 device -> memory

cudaMemcpy(dst.bytes, devDst, size, cudaMemcpyDeviceToHost);

// 释放内存

cudaFree(devSrc);

cudaFree(devDst);

5.3 实验方案

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

启用CUDA的关键CMake代码段如下（需CMake 3.8+）：

enable\_language(CUDA)

编译运行：

* 使用cmake创建Visual Studio 2017工程
  + 对于Visual Studio 2017，需要使用 cmake -A x64 -G “Visual Studio 14 2015”创建VS2015的工程，否则会出现编译问题
* 在VS2017工程中build，生成可执行文件
* 在Debug/文件夹中，执行：
* morph.exe <输入文件> <输出文件>

5.1 实验结果与分析

首次执行CUDA并行程序的结果如图 5.1所示。整个执行花费了377ms，远远超过预期的时间。经调查，发现第一个cuda函数调用会加载cuda runtime，带来大量的性能开销。

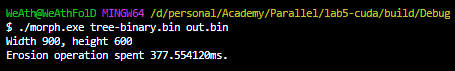


图 5.1 CUDA并行优化执行结果1

通过在main函数里添加一对cudaMalloc和cudaFree的函数调用，预先加载CUDA运行时，再运行测试程序，结果如图 5.2所示。



图 5.2 CUDA并行优化执行结果2

最后，尝试修改blocksPerGrid和threadsPerBlock来达到更高的并行度，优化性能。将threadsPerBlock改为dim3(256, 256)，blocksPerGrid改为dim3(ceil(width / 32), ceil(height / 32))，执行时间从33ms降低到了12ms。可见将任务分配到多个block有益于提高并行度，优化性能。

5.2 所有实现的性能综合比较分析

接下来对实验中所有（5个）的形态学图像处理算法的性能进行统一的测试，并进行性能分析。

我们通过Photoshop的“渲染-云彩”滤镜，分别生成了5张长宽为256、512、1024、2048、4096的图片，并使用image-convert.jar生成二值图，来对各个程序在不同负载情况下的性能进行测试。

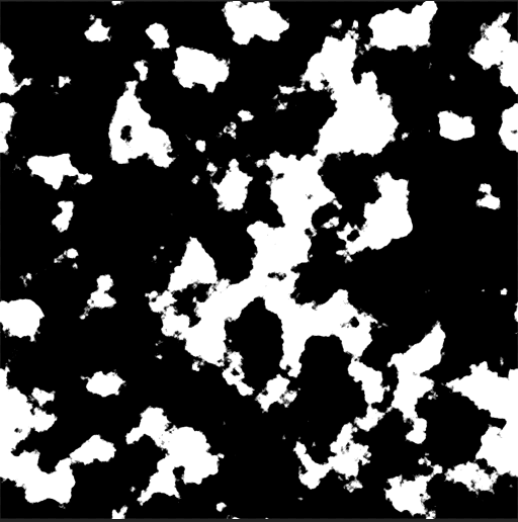


图 5.3 生成的“云彩”图片

各个程序的测量数据如表 5.1所示。

表 5.1 各个程序在不同图像大小下的运行时间（单位：ms）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **pixel count** | **linear** | **thread** | **omp** | **mpi** | **cuda** |
| 256 | 2.69 | 5.95 | 1.49 | 1.99 | 3.30 |
| 512 | 16.24 | 6.85 | 5.16 | 4.71 | 3.06 |
| 1024 | 52.53 | 18.93 | 16.78 | 17.74 | 3.95 |
| 2048 | 285.11 | 75.83 | 67.88 | 94.80 | 7.03 |
| 4096 | 1127.20 | 279.18 | 259.42 | 360.31 | 23.07 |

从该表绘成的折线图——图 5.4可以对各个程序的运行时间做简要的对比和分析。

图 5.4 程序执行时间和图像大小的关系

首先，显而易见的是所有程序都随图片长度的增长运行时间增加，且基本都是呈抛物线（平方）式增加——这符合理论推测，因为处理的像素数是图片长度的平方。

无优化线性执行方式的性能是最差的，执行时间增长最快，且在4096px时执行时间超过1秒；thread、omp、mpi三种优化方式的曲线十分靠近，这是因为它们都用了类似的方式——在CPU上并行的执行任务来进行优化。执行时间的些微差异是因为具体实现方式的不同，开销不一样带来的。如mpi所用的时间在三者中最长，可能是因为跨进程通信的开销比跨线程更大。

在所有优化方式中，CUDA以极大的优势胜出。尽管在256px时性能稍差，但在那之后执行时间的增长都很小。即便处理4096\*4096大小的图像，也能在数十毫秒级别的时间内完成。

总而言之：

1. 所有优化方式，执行时间都大致随图片长度呈平方增长；
2. 在图片大小较小时，几种优化方式执行时间相仿；
3. 在图片大小增加时，线性方式以极快的速度增长；多线程/进程（thread,omp,openmpi）方式增长大致相同，比线性方式慢很多；
4. CUDA以极大的优势打败了所有竞争者，在4096图片长度时仍能保持数十毫秒级别的执行速度。

## Project: 广度优先搜索的并行优化

6.1 实验目标

1. 提出使用并行计算方式优化广度优先搜索的方法；
2. 理解各种工具在进行并行优化过程中的特点与区别；
3. 通过调整并行计算的并行粒度，进行计算速度的优化；
4. 更进一步的理解在并行计算中主要的注意事项。

6.2 实验方法

在本实验中，通过多线程的方法对广度优先索索进行并行优化。为了方便起见，我们仅对一些生成的树（而非图）进行并行搜索。

经典的对树的广度优先搜索算法伪代码如下所示：

# Input: Root node

# Output: List of nodes in BFS order which they are visited

func dfs(root):

result := new empty list

Q := new empty queue

Q.enqueue(root)

while (Q is not empty):

next = Q.poll()

result.add(next)

for all V adjacent to next:

Q.enqueue(V)

end { result is BFS order of the tree }

算法的核心是维护了一个FIFO队列，每次将队列头的节点加入访问列表并将所以和该节点邻接的节点加入队列Q，不断处理直到Q为空为止。

我们现在尝试将这个算法改造为多线程的：

const THREAD\_COUNT = 16 # How many thread we divide the task to

# Input: Root node

# Output: List of nodes in BFS order which they are visited

func dfs\_parallel(root):

result := new empty list

Q := new empty queue

Q.enqueue(root)

while (Q is not empty):

levelCount = Q.size()

perThread = ceil(levelCount / THREAD\_COUNT)

spawn THREAD\_COUNT threads, each does:

for perThread times:

next = Q.poll()

result.add(next)

for all V adjacent to next:

Q.add(V)

join all previous threads

end { result is BFS order of the tree }

我们将对Q**头部元素的访问**并行化。对于深度优先搜索来说，每个深度中的所有节点可以以不同的顺序被访问，该优化就基于这个特点。每次最外层的while循环代表深度推进一层，在当前层的所有节点划分为THREAD\_COUNT（最合适设置为当前CPU的核心数）个线程分别进行读取和处理。

需要注意的是每个线程都需要对共享资源Q进行读写操作，因此Q必须是线程安全的。

6.3 实验结果

使用Kotlin语言进行本次实验代码的编写，Intellij IDEA作为开发环境。线程使用java提供的java.concurrent.Thread类进行创建。关键的代码段如下所示，使用ConcurrentLinkedQueue来保证队列是多线程安全的。

val queue = ConcurrentLinkedQueue<Node>()

queue.add(root)

while (!queue.isEmpty()) {

val curCount = queue.count()

val threadCount = if (curCount <= MaxThreads) curCount else MaxThreads

val perThread = (curCount + threadCount - 1) / threadCount

(0 until threadCount).map {

val x = perThread \* it

val toProcess = Math.min(perThread, curCount - x)

thread { **// 创建线程**

for (i in 0 until toProcess) {

val next = queue.poll()

results.add(next)

next.children.forEach { queue.add(it) }

}

}

}

.forEach { it.join() }

}

代码做的只是简单的将元素添加到结果列表里，而不做其他事（由于元素有几十万个，在这里也不打印了）。执行的结果如图 6.1所示。

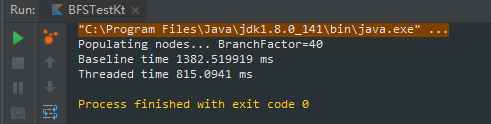


图 6.1 BFS多线程优化的执行结果

根据实验并行优化的特点，猜测当BranchFactor（树中每个非叶节点的子节点数）越高时，并行优化带来的性能提升越大。构造BranchFactor不同的树分别对算法进行测试，得出单线程和多线程的执行时间分别如表 6.1所示。

表 6.1 不同BranchFactor下的执行性能

|  |  |  |
| --- | --- | --- |
| BranchFactor | Baseline | MultiThread |
| 20 | 26 | 51 |
| 30 | 149 | 187 |
| 40 | 1273 | 805 |
| 50 | 2799 | 1561 |
| 60 | 4505.9 | 4403.3 |

图 6.2的结果更为直观：

图 6.2 不同BranchFactor下的执行性能

可以看到，在BranchFactor较低时，多线程优化的结果稍逊于单线程执行；在BranchFactor达到40-50时，多线程优化体现出明显优于单线程的性能；但在BranchFactor达到60时，两者的性能又趋于一致。

6.4 讨论和结论

BFS的多线程优化中，实现要保证Q是线程安全的，这会带来额外的同步开销，导致在任务数较低时其执行效率较低。而当BranchFactor升高时，个人认为是JVM对执行代码进行了额外的优化。由于单线程的代码比较简单，不涉及锁也不涉及同步问题，所以JVM可以对字节码进行**更为彻底的优化**，这导致在执行次数较高之后两者性能再次趋同。如果通过C++进行实现，应该会观察到多线程实现在一开始体现出比较大的开销，但之后速度稳定的快于多线程。

BFS的多线程优化效果没有形态学图像处理理想。这是因为BFS中对共享资源的操作占据了大量的时间。为了解决竞态条件，必须花大量的时间在资源的同步（加锁和等待上）。如果我们需要对每个节点，搜索到以后进行一些额外的计算，这些计算可以被均匀的分布在各个线程中，优化效果将更为明显。

## 附录

lab2-thread, lab3-openmp的环境配置

这两个项目不需要任何外部库的支持。只需要在项目文件夹下执行 “cmake .” 就可以生成对应的项目文件并执行编译。

lab4-mpi的环境配置（Windows）

需要先下载并安装Microsoft MPI ([msdn.microsoft.com/en-us/library/bb524831.aspx](https://msdn.microsoft.com/en-us/library/bb524831.aspx))，再使用cmake创建工程。cmake会自动检测Microsoft MPI的位置并设置编译器和linker的flag。

lab5-cuda的环境配置（Windows）

首先在NVIDIA官网下载CUDA（[developer.nvidia.com/cuda-downloads](https://developer.nvidia.com/cuda-downloads)），再使用cmake创建工程。

cmake创建工程时，一定需要-A x64标志。如果是使用Visual Studio 2017，一定需要-G “Visual Studio 14 2015”创建VS2015的工程（并且在VS的安装器中，添加VC140工具集支持），因为CUDA对VS2017的编译器支持尚有问题。

image-convert工具的使用

image-convert工具将普通的图片和实验中专门的二进制二值图格式进行相互转换。使用Intellij IDEA打开util/image-convert文件夹即可打开工程。通过Build -> Generate Artifact可以生成可执行的jar文件。使用说明请参考实验二。

## 参考资料

1. Using std::cout in multiple threads, [https://stackoverflow.com/questions/18277304](https://stackoverflow.com/questions/18277304/using-stdcout-in-multiple-threads).
2. C++: Simple return value from std::thread?, <https://stackoverflow.com/questions/7686939/>.
3. Thread – C++ reference, <http://www.cplusplus.com/reference/thread/thread/>.
4. Cmake cannot find OpenMP, <https://stackoverflow.com/questions/17633513>.
5. Microsoft MPI, <https://msdn.microsoft.com/en-us/library/bb524831.aspx>.
6. How to compile an MPI included program using Cmake, <https://stackoverflow.com/questions/23163075>.
7. MPI Hello World, <http://mpitutorial.com/tutorials/mpi-hello-world/>.
8. MPI Scatter, Gather, and Allgather, <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>.
9. FindMPI – CMake 3.10.3 Documentation, <https://cmake.org/cmake/help/v3.10/module/FindMPI.html>.
10. Programming Guide :: CUDA Toolkit Documentation, <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
11. CUDA 9.0 does not work with the latest VS 2017 update, <https://devtalk.nvidia.com/default/topic/1027209>.
12. Morphology – Dilation, https://homepages.inf.ed.ac.uk/rbf/HIPR2/dilate.htm.
13. Java – How to Swave a BufferedImage as a File, <https://stackoverflow.com/questions/12674064>.
14. MPI Send and Receive, <http://mpitutorial.com/tutorials/mpi-send-and-receive/>.
15. MPI Broadcast and Collective Communication, <http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/>.
16. First cudaMalloc() takes long time?, <https://devtalk.nvidia.com/default/topic/392429>.
17. Deadlock with MPI – Stack Overflow, <https://stackoverflow.com/questions/20448283/deadlock-with-mpi>.