

**课 程 实 验 报 告**

**课程名称： 并行编程原理与实践**

**专业班级：计算机科学与技术1409班**

**学 号： U201414815**

**姓 名： 胡超**

**指导教师： 金海**

**报告日期： 2017.07.12**

**计算机科学与技术学院**

## 实验一

1.1 实验目的与要求

1.1.1 实验目的

了解并行编程的目的，思路和方法，并且熟悉并行计算的分析和解决问题的技巧。

1.1.2 实验要求

1） 使用通用工具和程序模型（如pthread，OpenMP，MPI，MapReduce和CUDA）的基本原理和方法来并行现有算法并开发新的并行算法

2） 分析并行过程和结果，让学生在并行化理论的更深层次，编译和操作系统的基础上了解并行化的目的和性能改进的原因

1.2 实验内容

1.2.1 pThread并行加法

1. 原理

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

基本的API：

int pthread\_create( pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*func) (void \*), void \*arg);

thread表示线程ID，与线程中的pid概念类似

attr表示设定线程的属性，可以暂时不用考虑

func表示新创建的线程会从这个函数指针处开始运行

arg表示这个函数的参数指针

返回值为0代表成功，其他值为错误编号

int pthread\_join( pthread\_t thread, void \*\*retval );

thread表示线程ID，与线程中的pid概念类似

retval用于存储等待线程的返回值

1. 实践

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

#include <pthread.h>

#include <stdio.h>

#include <iostream>

using namespace::std;

int A[10] = {0,0,0,0,0,0,0,0,0,0};

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

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

void \*\_add(void \* num) {

int i = \*(int \*)num;

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

printf("result:%d\n",A[i]);

return (void \*)0;

}

int main() {

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

pthread\_t t\_id[10];

int i = 0;

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

if(pthread\_create(&t\_id[i],NULL,\_add,&num[i]) != 0) {

cout<<"线程创建出错";

}

}

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

if(pthread\_join(t\_id[i],NULL) != 0) {

cout<<"线程出错";

}

}

return 0;

}

1.2.2 OpenMP并行加法

1. 原理

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

特殊的编译语句为#pragma omp parallel for。

1. 实践

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

#include <stdio.h>

#include <omp.h>

int main() {

int A[10] = {0,0,0,0,0,0,0,0,0,0};

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

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

#pragma omp parallel for

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

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

printf("result:%d\n",A[i]);

}

return 0;

}

1.2.3 MPI并行加法

1. 原理

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

基本的API：

int MPI\_Init(int \*argc, char \*\*argv)

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

int MPI\_Finalize(void)

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

int MPI\_Comm\_size(MPI\_Comm comm, int \*size);

获取进程个数p。

int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank);

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

int MPI\_Send(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag,MPI\_Comm comm);

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

其中datatype有MPI\_INT, MPI\_FLOAT等常用类型

Tag的作用是用于区分一对进程之间发送的不同信息

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

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

1. 实践

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

#include <stdio.h>

#include <mpi.h>

int main(int argc,char \*argv[]) {

int A[10] = {0,0,0,0,0,0,0,0,0,0};

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

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

int my\_rank = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

if(my\_rank <= 9) {

A[my\_rank] = B[my\_rank] + C[my\_rank];

printf("result:%d\n",A[my\_rank]);

}

MPI\_Finalize();

return 0;

}

1.2.4 Cuda并行加法

1. 原理

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

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

将任务合理的分配到grid和thread中，有助于提升程序的性能

基本API：

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

在设备端分配size大小的空间，起始地址为devPtr。

cudaError\_t cudaMemcpy (void \* dst, const void \* src,size\_t count,enum cudaMemcpyKind kind);

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

cudaError\_t cudaFree (void \*devPtr);

在设备端清理以devPtr为起始地址的内存空间

1. 实践

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

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

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

\_\_global\_\_ void addKernel(int \*c, const int \*a, const int \*b) {

int i = threadIdx.x;

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

}

int main() {

const int arraySize = 10;

const int a[arraySize] = {0,1,2,3,4,5,6,7,8,9};

const int b[arraySize] = {0,1,2,3,4,5,6,7,8,9};

int c[arraySize] = {0,0,0,0,0,0,0,0,0,0};

// Add vectors in parallel.

addWithCuda(c, a, b, arraySize);

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",c[0], c[1], c[2], c[3], c[4]);

// cudaThreadExit must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaThreadExit();

return 0;

}

// Helper function for using CUDA to add vectors in parallel.

void addWithCuda(int \*c, const int \*a, const int \*b, size\_t size) {

int \*dev\_a = 0;

int \*dev\_b = 0;

int \*dev\_c = 0;

// Choose which GPU to run on, change this on a multi-GPU system.

cudaSetDevice(0);

// Allocate GPU buffers for three vectors (two input, one output) .

cudaMalloc((void\*\*)&dev\_c, size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_a, size \* sizeof(int));

cudaMalloc((void\*\*)&dev\_b, size \* sizeof(int));

printf("%d\n",cudaStatus);

// Copy input vectors from host memory to GPU buffers.

cudaMemcpy(dev\_a, a, size \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, b, size \* sizeof(int), cudaMemcpyHostToDevice);

// Launch a kernel on the GPU with one thread for each element.

addKernel<<<1, size>>>(dev\_c, dev\_a, dev\_b);

// cudaThreadSynchronize waits for the kernel to finish, and returns

// any errors encountered during the launch.

cudaThreadSynchronize();

// Copy output vector from GPU buffer to host memory.

cudaMemcpy(c, dev\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

cudaFree(dev\_c);

cudaFree(dev\_a);

cudaFree(dev\_b);

}

1.3 实验结果

1.3.1 pThread并行加法

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

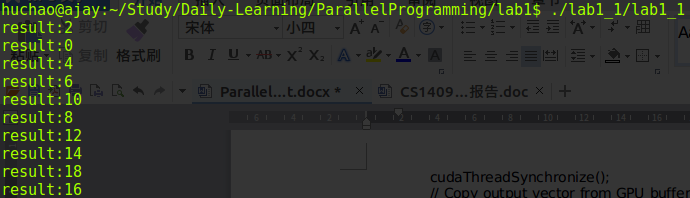


图1-1 pthread并行加法结果图

1.3.2 OpenMp并行加法

数据与前述方法相同，得到结果如下图1-2所示，可知并行运行结果正确。

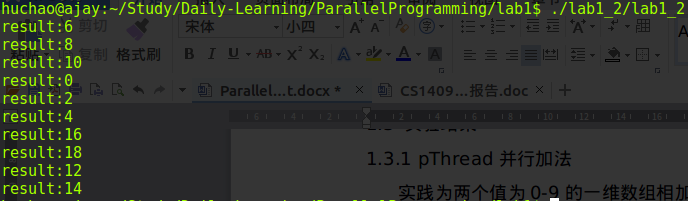


图1-2 openmp并行加法结果图

1.3.3 MPI并行加法

数据与前述方法相同，得到结果如下图1-3所示，可知并行运行结果正确。



图1-3 mpi并行加法结果图

1.3.4 Cuda并行加法

数据与前述方法相同，得到结果如下图1-4所示，可知并行运行结果正确。

lab1_4

图1-4 cuda并行加法结果图

## 实验二

2.1 实验目的与要求

1）掌握使用pthread并行编程设计和性能优化的基本原理和方法

2）了解并行编程中数据分区和任务分解的基本方法

3）使用pthread实现图像卷积运算的并行算法

4）然后对程序执行结果进行简单的分析和总结

2.2 算法描述

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

算法流程图如下图2-1所示：

开始

读入图片

计时开始

图片像素行分解

开启线程

等待线程结束

计时结束

保存结果

结束

边缘操作

图2-1 pthread并行卷积算法流程图

2.3 实验方案

2.3.1 开发与运行环境

系统：ubuntu 16.04 Desktop

kernel：linux-4.4.0-83-generic

编译器：g++5.4.0

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

内存：4G+4G

2.3.2实验步骤

首先引入opencv操作环境，方便能够对图片进行处理。

图片读取为Mat格式，是现在较新的opencv推行的图片格式。

按照边缘处理卷积并行处理算法编写实验代码。

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

在执行程序查看结果。

2.4 实验结果与分析

实验原图由于较大，不好进行展示，截取部分如下图2-2所示。



图2-2 实验原图部分截取示意图

此次实验结果部分截取如下图2-3所示：



图2-3 pthread并行卷积结果图部分截图示意图

## 实验三

3.1 实验目的与要求

3.1.1 实验目的

3.1.2 实验要求

3.2 算法描述

3.3 实验方案

3.4 实验结果与分析

## 实验四

4.1 实验目的与要求

4.1.1 实验目的

4.1.2 实验要求

4.2 算法描述

4.3 实验方案

4.4 实验结果与分析

## 实验五

5.1 实验目的与要求

5.1.1 实验目的

5.1.2 实验要求

5.2 算法描述

5.3 实验方案

5.4 实验结果与分析

## 实验小结