

**课 程 实 验 报 告**

**课程名称： 并行编程原理与实践**

**专业班级： ACM2012**

**指导教师： 金海老师**

**报告日期： 2015/06/20**

**计算机科学与技术学院**

**目录**

[实验一、熟悉并行环境 3](#_Toc423344240)

[1.1实验目的与要求 3](#_Toc423344241)

[1.2实验内容 3](#_Toc423344242)

[1.3实验结果 3](#_Toc423344243)

[实验二、pthread实现矩阵乘法 7](#_Toc423344244)

[2.1实验目的与要求 7](#_Toc423344245)

[2.2实验内容 7](#_Toc423344246)

[实验三、OpenMp实现矩阵乘法 9](#_Toc423344247)

[3.1实验目的与要求 9](#_Toc423344248)

[3.2实验内容 9](#_Toc423344249)

[3.3实验结果 9](#_Toc423344250)

[实验四、MPI实现矩阵乘法 10](#_Toc423344251)

[4.1实验目的与要求 10](#_Toc423344252)

[4.2实验内容 10](#_Toc423344253)

[4.3实验结果 11](#_Toc423344254)

[实验五、CUDA实现矩阵乘法 13](#_Toc423344255)

[5.1实验目的与要求 13](#_Toc423344256)

[5.2实验内容 13](#_Toc423344257)

[5.3实验结果 14](#_Toc423344258)

[实验小结 15](#_Toc423344259)

[Project name：基于蒙特卡罗方法实现的积分求法 16](#_Toc423344260)

[AIM 16](#_Toc423344261)

[HYPOTHESIS 16](#_Toc423344262)

[METHODS 17](#_Toc423344263)

[RESULTS 17](#_Toc423344264)

[DISCUSSION & CONCLUSION 18](#_Toc423344265)

[附录 19](#_Toc423344266)

# 实验一、熟悉并行环境

## 1.1实验目的与要求

熟悉和配置各种并行环境，熟悉并行编程基本原则和并行编程方法，如何有效使用工具进行优化处理。

本次实验主要熟悉如何在linux系统下使用pthread、OpenMp、MPI、CUDA进行编程处理。

本次的实验环境有pthread、OpenMp、MPI、CUDA这四个平台

pthread、openmp使用gcc直接编译即可

mpi需要安装mpich2即可使用mpicc进行编译

cuda需要N卡，先安装驱动，再安装cuda套件和sdk即可。

## 1.2实验内容

配置环境，熟悉实验平台

本次实验主要在东五集群节点node230、231、232、233和329上进行，实验环境之前已经配好，不需要再进行配置。

方案一、

适用于linux用户

使用ssh登录：

1.先登陆到外网代理节点：输入ssh jumptohpcc@211.69.198.206，密码为hustcloudhpcc。

2.登录到跳板机之后，直接登录到集群管理节点：输入ssh pppuser230@202.114.10.172 密码为cgclhpcc2015。

3.登录到计算节点，例如登录到计算节点node230，输入：ssh node230。

4. Node329为cuda计算结点

方案二、

适用于windows用户

使用Xshell登陆，

步骤和方案一一致。

## 1.3实验结果

本次实验在Ubuntu12.04下进行。

使用方案一登陆之后会进入按照lab的示例编译运行代码，显示正常。

所有编译器和运行都是正常执行。

本实验开源在

https://github.com/XinYao1994/Work

使用Makefile书写编译规范：

#compiler

CC = gcc

MPICC = mpicc

MPIRUN = mpirun

CUDACC = nvcc

CFLAGS = -g -O3

#begin to choose

pthread: pthread.c

$(CC) -o pthread pthread.c -lpthread $(CFLAGS)

openmp: openmp.c

$(CC) -o openmp openmp.c -fopenmp $(CFLAGS)

mpi: mpi.c

$(MPICC) -o mpi mpi.c $(CFLAGS)

#$(MPIRUN) -np 16 ./mpi

cuda: cuda.cu

$(CUDACC) -o cuda cuda.cu -lcudart $(CFLAGS) -G

$(CUDACC) -run ./cuda

mc: Monte-Carl.c

$(MPICC) -o mc Monte-Carl.c -fopenmp $(CFLAGS)

clean:

-rm -f pthread openmp mpi cuda mc

Pthread

基本的API:

创建一个新的线程

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

主进程等待线程结束：

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

头文件加入pthread.h

编译命令加入 –lpthread 库

OpenMp

不需要使用MPI，使用特殊的编译引导语句，源程序修改成如下形式：

#pragma omp parallel for

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

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

}

Openmp会自动将for循环分解为多个线程并行执行

头文件加入omp.h

编译时加入 –openmp 参数

MPI

头文件加入<mpi.h>

编译：mpicc –o bin code.c

运行：mpirun –np x bin //其中x为设置的进程数

使用MPI完成这个工作

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

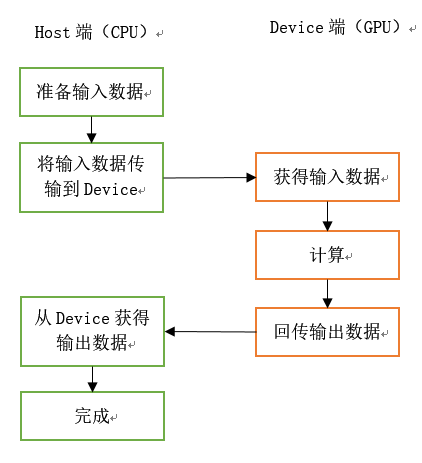
Tag的作用是用于区分一对进程之间发送的不同信息

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

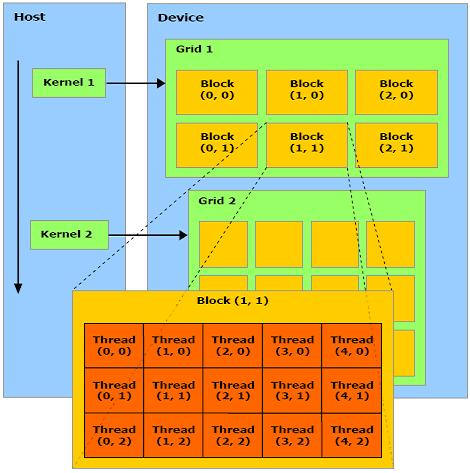
接受函数：从rank值为source的进程接受标识符为tag的信息，存入以buf为初始地址，长度为count的存储区域中，类型为datatype.

CUDA

GPU协处理器的运算流程



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

# 实验二、pthread实现矩阵乘法

## 2.1实验目的与要求

使用pthread实现矩阵乘法加速

## 2.2实验内容

将目标矩阵按行列进行分割为diva \* divc块对矩阵c进行填充。

使用Point记录每一个pthread需要填充矩阵c的起点

分割代码

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

for(j=0; j<divc; j++){//x, y. xlen, ylen

array[i][j].x = i\*adis;

array[i][j].y = i\*cdis;

if(i+1==diva) array[i][j].xlen = la - i\*adis;

else array[i][j].xlen = adis;

if(j+1==divc) array[i][j].ylen = lc - i\*cdis;

else array[i][j].ylen = cdis;

if(pthread\_create(&hThread[i][j], NULL, (void \*)MultMatrix, (void \*)&array[i][j])){

printf("create error!\n");

return 0;

}

}

各线程计算代码

void \*MultMatrix(void \*arg){

int i, j, k;

int ax, alen, cy, clen;

Point \*p = (Point \*)arg;

ax = p->x;

alen = p->xlen;

cy = p->y;

clen = p->ylen;

for(i=ax; i<(ax+alen); i++)

for(j=cy; j<(cy+clen); j++)

for(k=0; k<a->b; k++)

\*(c->m+i\*c->b+j) += (\*(a->m+i\*a->b+k))\*(\*(b->m+k\*b->b+j));

}

Join过程即将原分配的线程数组Join起来即可。

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

for(j=0; j<divc; j++)

pthread\_join(hThread[i][j], NULL);

2.3实验结果

Make pthread

然后用time直接可以进行测量：

time ./pthread [a] [b] [c]

a b c 为可选矩阵参数（a\*b）×（b\*c）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 2\*2 | 3\*3 | 4\*4 | 5\*5 | 不加速 |
| 1000 | 1.387s | 0.995s | 0.758s | 0.681s | 4.024s |
| 2000 | 10.289s | 7.128s | 5.440s | 4.959s | 31.553s |
| 3000 | 48.632s | 37.193s | 27.111s | 0m24.211s | 2m29.718s |
| 5000 | 4m58.950s | 3m30.220s | 2m25.646s | 2m7.825s | 14m23.258s |
| 10000 | 35m57.285s | 26m3.434s | 17m29.707s | 16m20.500s | 103m0.015s |

因为使用time进行的测试，其中包含了许多串行执行的部分和矩阵初始化的时间。而且，我们可以看到随着线程数量的增加，实际花费的时间是会大大减少的，但是由于实际上处理机数量有限，线程增加到一定数量后，加速就不是很明显了。

# 实验三、OpenMp实现矩阵乘法

## 3.1实验目的与要求

使用OpenMp进行矩阵乘法的加速。

## 3.2实验内容

如何使用一般嵌套循环求和的话，会使得Cache的利用率下降，从而使时间浪费掉了。因此在对OpenMp下的矩阵求和换了一种方式：

void openmp\_multmatrix(){

int i,j,k;

int index;

int all = a->a \* b->b;

#pragma omp parallel private(i,j,k)

for(index=0; index<all; index++){

i = index/b->b;

j = index%b->b;

for(k=0; k<a->b; k++){

\*(c->m+i\*c->b+j) += (\*(a->m+i\*a->b+k))\*(\*(b->m+k\*b->b+j));

}

}

}

这里，我们将要求的矩阵c看做一个大棋盘，并且将其下标索使用首地址加偏移量的方式来增加数据局部性。另一方面，index比起正常的嵌套for的量都要大许多，有利于并行调度执行的负载均衡。

## 3.3实验结果

Make openmp

然后用time直接可以进行测量：

time ./openmp [a] [b] [c]

a b c 为可选矩阵参数（a\*b）×（b\*c）

|  |  |  |
| --- | --- | --- |
|  | 加速 | 不加速 |
| 1000 | 1.628s | 4.024s |
| 3000 | 54.660s | 14m23.526s |
| 5000 | 4m6.789s | 65m6.928s |
| 7000 | 11m12.499s | 177m31.527s |
| 10000 | 28m24.213s | 450m3.734s |

单单从加速效果来看，这个效果是显著的，但是算法不值得推荐，因为串行算法本身花费的时间远远超过一般的串行时间。但这只是说明这种思想不适用于这个问题，而不是说这种思想本身不正确。从加速比的情况来看，这种算法更适用于并行，如果处理器的数量更多的情况下，一定可以得到更短的执行时间。

# 实验四、MPI实现矩阵乘法

## 4.1实验目的与要求

使用MPI加速矩阵乘法

## 4.2实验内容

在一般矩阵乘法当中，其他进程所有的信息都是从0号进程得到，计算，并返回各个进程的计算结果。其实这种算法会带来一些负载不均衡的问题，0号进程负载过重。我们只需要对矩阵A、B进行划分，然后每一个进程都从同行的进程中获取矩阵A的信息，从同列的进程中获取矩阵B的信息，就可以计算自己负责的矩阵C的部分。

信息传递过程

void init\_con(int id, int same){

int i,j,k;

posy = id/Wp;//get group

int bb = posy\*Hp, ee = (posy+1)\*Hp;

int con;//send once recive k-1

int \*buffa = (int \*)malloc(sizeof(int)\*ystep\*same);

int \*buffb = (int \*)malloc(sizeof(int)\*xstep\*same);

for(i=bb;i<ee;i++){

//i send

if(i==id){

for(j=bb;j<ee;j++){

if(j==id) continue;

MPI\_Send(bufa, ystep\*same, MPI\_INT, j, 1, MPI\_COMM\_WORLD);//send a

MPI\_Send(bufb, xstep\*same, MPI\_INT, j, 2, MPI\_COMM\_WORLD);//send b

}

}

else{

MPI\_Recv(buffa, ystep\*same, MPI\_INT, i, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(buffb, xstep\*same, MPI\_INT, i, 2, MPI\_COMM\_WORLD, &status);

con = 0;

xB = i \* xstep;

yB = i \* ystep;

xE = (i+1)\*xstep;

yE = (i+1)\*ystep;

for(j=yB;j<yE;j++)

for(k=0;k<same;k++){

\*(a->m+j\*a->b+k) = \*(buffa+con);

con++;

}

con = 0;

for(j=xB;j<xE;j++)

for(k=0;k<same;k++){

\*(b->m+j\*b->b+k) = \*(buffb+con);

con++;

}

}

}

}

0号进程接受矩阵信息：

void Fill(int id, int x, int y, MPI\_Status \*status){

int i,j;

posx = id%Wp;

posy = id/Hp;

xstep = x/Wp;

ystep = y/Hp;

xB = posx \* xstep;

yB = posy \* ystep;

xE = (posx+1)\*xstep;

yE = (posy+1)\*ystep;

//buf = (int \*)malloc(sizeof(int)\*xstep\*ystep);

if(id) MPI\_Recv(buf, xstep\*ystep, MPI\_INT, id, 0, MPI\_COMM\_WORLD, status);

int con = 0;

for(i=xB;i<xE;i++)

for(j=yB;j<yE;j++){

\*(c->m+i\*x+j) = \*(buf+con);

con++;

}

}

## 4.3实验结果

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2\*2 | 5\*5 | 不加速 |
| 1000 | 1.803s | 2.231s | 4.024s |
| 3000 | 19.629s | 15.211s | 2m29.718s |
| 5000 | 1m49.651s | 1m17.825s | 14m23.258s |
| 10000 | 18m12.511s | 5m20.500s | 72m32.015s |

|  |  |  |
| --- | --- | --- |
|  | 3\*3 | 不加速 |
| 1800 | 3.160s | 14.024s |
| 3600 | 17.193s | 2m25.718s |
| 5400 | 1m4.220s | 9m23.258s |
| 8100 | 4m17.956s | 38m25.999s |
| 9000 | 6m54.945s | 62m0.015s |

|  |  |  |
| --- | --- | --- |
|  | 4\*4 | 不加速 |
| 1600 | 2.870s | 14.764s |
| 3200 | 9.383s | 1m56.718s |
| 4800 | 27.646s | 6m46.258s |
| 9600 | 4m57.707s | 78m10.015s |

MPI因为在矩阵较小时，传递消息占用了大量的时间，所以加速比不明显。当矩阵规模变大时，其加速比明显超过了之前使用pthread和OpenMp的效果。虽然大矩阵的动态分配会占用一定的时间，但是我们明显可以看出MPI进程工作互不干扰，一定程度上减少了线程调度的负载。另一方面，由于核数限制，后期的加速比也维持在一个常数附近。

# 实验五、CUDA实现矩阵乘法

## 5.1实验目的与要求

使用CUDA实现矩阵乘法加速

## 5.2实验内容

CUDA当中需要注意的是内核函数，以及内存类型的使用。我们在这里使用共享内存，提高利用率，理解如何使用线程块索引找到计算资源及目标。

内核函数：

template <int BLOCK\_SIZE> \_\_global\_\_ void

matrixMulCuda(int \*c, int \*a, int \*b, int wA, int wB){

int bx = blockIdx.x;

int by = blockIdx.y;

int tx = threadIdx.x;

int ty = threadIdx.y;

int aBegin = wA \* BLOCK\_SIZE \* by;

int aEnd = aBegin + wA - 1;

int aStep = BLOCK\_SIZE;

int bBegin = BLOCK\_SIZE \* bx;

int bStep = BLOCK\_SIZE \* wB;

int Cadd = 0;

for (int i = aBegin, j = bBegin;i <= aEnd; i += aStep, j += bStep)

{

\_\_shared\_\_ float As[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float Bs[BLOCK\_SIZE][BLOCK\_SIZE];

As[ty][tx] = a[i + wA \* ty + tx];

Bs[ty][tx] = b[j + wB \* ty + tx];

\_\_syncthreads();

for (int k = 0; k < BLOCK\_SIZE; ++k)

{

Cadd += As[ty][k] \* Bs[k][tx];

}

\_\_syncthreads();

}

int w = wB \* BLOCK\_SIZE \* by + BLOCK\_SIZE \* bx;

c[w + wB \* ty + tx] = Cadd;

}

## 5.3实验结果

|  |  |
| --- | --- |
|  | 加速 |
| 1024 | 0.377s |
| 2048 | 1.186s |
| 4096 | 6.880s |
| 8192 | 51.316s |
| 12288 | 2m50.932s |
| 16384 | 6m42.798s |
| 20480 | 13m4.617s |

CUDA加速矩阵的效果无疑是以上所有方法当中最快的，其重要原因之一就是资源多。GPU中的核数远远超过我们之间所使用的CPU的核数。这种典型的SIMD的计算模型很适用于GPU进行运算。其并行度会轻松超过我们所使用的MPI、OpenMp、pthread等等。

# 实验小结

之前接触过OpenMp、CUDA等之类的并行编程库或语言，整个实验写起来还算是比较顺利的。但对MPI接触不多，编程时出错进行调试时发现工具不全，而用gdb只能调试0号进程的东西，设计的时候算法也和普通有些不同之处。虽然MPI看上去不是很难，写起来也容易上手，但还是不小心会在消息传递那里发生死锁，最终使用一个强制串行化的方法，每一组进程当中只有一次只有一个进程向其他进程广播消息，终于解决了这个问题，虽然效果还不错，但是至少是还有优化空间的，之后还需要加以改进。

矩阵乘法可以说是并行编程的Hello, World。每一个学习并行编程的程序员都少不了从这里入手。首先矩阵乘法是一个比较有规律的算法，而且优化的着手点也有许多不同之处，如从如何提高并行度，如何利用cache，等等诸多方面。并行优化的程序还有许多需要注意的地方和使用到的方法，如原子操作、锁机制、栅栏等等。

随着GPU计算的发展，GPU+CPU的混合计算模型已逐步登上舞台，其正以丰富的资源计算能力赢得了高性能计算的青睐。同时，rdma等新型远程直接存取技术也纷纷应用到CPU、GPU当中，为传统的通信技术进行加速，这都是值得我们关注的地方。

# Project name：基于蒙特卡罗方法实现的积分求法

## AIM

使用OpenMp和MPI加速蒙特卡罗方法求积分

蒙特·卡罗方法（Monte Carlo method），也称统计模拟方法，是二十世纪四十年代中期由于科学技术的发展和电子计算机的发明，而被提出的一种以概率统计理论为指导的一类非常重要的数值计算方法。是指使用随机数（或更常见的伪随机数）来解决很多计算问题的方法。与它对应的是确定性算法。蒙特·卡罗方法在金融工程学，宏观经济学，计算物理学（如粒子输运计算、量子热力学计算、空气动力学计算）等领域应用广泛。

当所求解问题是某种随机事件出现的概率，或者是某个随机变量的期望值时，通过某种“实验”的方法，以这种事件出现的频率估计这一随机事件的概率，或者得到这个随机变量的某些数字特征，并将其作为问题的解。

工作过程

蒙特卡罗方法的解题过程可以归结为三个主要步骤：构造或描述概率过程；实现从已知概率分布抽样；建立各种估计量。

蒙特卡罗方法解题过程的三个主要步骤：

（1）构造或描述概率过程

对于本身就具有随机性质的问题，如粒子输运问题，主要是正确描述和模拟这个概率过 程，对于本来不是随机性质的确定性问题，比如计算定积分，就必须事先构造一个人为的概率过程，它的某些参量正好是所要求问题的解。即要将不具有随机性质的问题转化为随机性质的问题。

（2）实现从已知概率分布抽样

构造了概率模型以后，由于各种概率模型都可以看作是由各种各样的概率分布构成的，因此产生已知概率分布的随机变量（或随机向量），就成为实现蒙特卡罗方法模拟实验的基本手段，这也是蒙特卡罗方法被称为随机抽样的原因。

（3）建立各种估计量

一般说来，构造了概率模型并能从中抽样后，即实现模拟实验后，我们就要确定一个随机变量，作为所要求的问题的解，我们称它为无偏估计。建立各种估计量，相当于对模拟实验的结果进行考察和登记，从中得到问题的解。

数学应用：

通常蒙特·卡罗方法通过构造符合一定规则的随机数来解决数学上的各种问题。对于那些由于计算过于复杂而难以得到解析解或者根本没有解析解的问题，蒙特·卡罗方法是一种有效的求出数值解的方法。一般蒙特·卡罗方法在数学中最常见的应用就是蒙特·卡罗积分。

本次实验，我们借用并行技术来加速蒙特•卡罗积分的求解。

## HYPOTHESIS

在蒙特卡罗投针实验中，将投针次数划分到多个进程中进行，每个进程中，在实际进行投针时，使用OpenMp进行优化，并求出所在范围内的投针数量，并使用MPI合并到0号进程，最终求出积分的面积。

随着投针次数增大，并行化效果会更加凸显，因为比起进程间通信，实际每个进程进行任务的时间会增大。然而也会有一些弊端，因为在计数投针数量的时候，这个和是一个临界资源，又因为是有条件的求和，不可以简单使用OpenMp的reduce进行求和，因此我选择使用#pragma omp critical解决临界资源问题。

最后将各个进程收集到在区域内的点数聚集到0号进程，求和按比例求出积分面积。

## METHODS

1. 按进程数、实验数量(投针数量)划分任务量到各个进程。
2. 各进程内部使用OpenMp进行模拟实验，并统计积分面积范围内的投针数量。
3. 各进程将统计的投针数量返回给0号进程。
4. 0号进程估计计算积分面积

部分代码：

int i, x, y;

yBord = fun(BORD);

Ycon = 0;

#pragma omp parallel for

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

x = rand()%(BORD+1);

y = rand()%(yBord+1);

if(y <= fun(x)){

#pragma omp critical

Ycon = Ycon + 1;

}

}

## RESULTS

Results and necessary explanation

以 y = x^2函数为例子，x的范围为[0, 100]

注：左上角为MPI的进程数量，每个进程运行过程中使用了OpenMp进行加速

|  |  |  |  |
| --- | --- | --- | --- |
| 4 |  | 不加速 | 面积 |
| 1000 | 11.686s | 2m42.4s | 335029 |
| 3000 | 30.289s | 7m40.4s | 335087 |
| 5000 | 0m59.503s | 14m28.4s | 335165 |
| 10000 | 1m58.910s | 29m27.6s | 335042 |

|  |  |  |  |
| --- | --- | --- | --- |
| 8 |  | 不加速 | 面积 |
| 1000 | 10.211s | 2m43.6s | 335006 |
| 3000 | 37.597s | 9m32.7s | 334908 |
| 5000 | 1m0.673s | 15m37.5s | 335094 |
| 10000 | 2m6.410s | 33m9.1s | 335151 |

|  |  |  |  |
| --- | --- | --- | --- |
| 16 |  | 不加速 | 面积 |
| 1000 | 13.046s | 3m04.4s | 334969 |
| 3000 | 30.047s | 7m40.4s | 335022 |
| 5000 | 1m5.608s | 16m57.4s | 334925 |
| 10000 | 2m42.098s | 40m43.6s | 335042 |

|  |  |  |  |
| --- | --- | --- | --- |
| 25 |  | 不加速 | 面积 |
| 1000 | 11.651s | 2m33.4s | 334847 |
| 3000 | 29.816s | 7m23.4s | 334915 |
| 5000 | 1m8.503s | 17m32.4s | 334713 |
| 10000 | 2m24.910s | 37m52.6s | 334964 |

## DISCUSSION & CONCLUSION

在实现蒙特卡罗算法时，我使用了MPI和OpenMp两种并行方法，其中MPI是进程间并行计算模型，进程间通信可以使用共享内存或者消息机制。而MPI则是选用了后者的消息机制。为了减轻开销，我们不得不尽可能地减少消息的大小和次数。当然我们也需要指定一个消息传达的顺序，从而防止死锁出现。而且对于一个进程来说，我们往往需要分配比较有份额的任务，否则会使得创建进程的开销不划算。对于OpenMp来说则不一样了，OpenMp的调度以线程为基础，线程是比较轻量级的，开销小，但是如果数量多且硬件资源不够时，引起的调度开销则会让人难以接受。所以线程一般都是用来执行一些小任务。

这次实现过程中，MPI的进程数量是通过NumPro和实际执行的进程数量来决定的，而OpenMp则是动态调度线程，一般可以看作它会尽量充分利用CPU的资源。

随着数据规模的增大，我们的加速是十分明显的。但是在同一数据规模下，我们可以看到用户态下执行时间实际上是没有改动的(这是程序运算的串行执行时间)，而核态执行时间会明显增加。这是因为我们使用的进程数量增加，从而会使整个通信过程增加。然而由于硬件资源不足的原因，其并没有对运算过程起到加速作用。最终影响到了并行效果。

而在同一数据规模下，进程少和进程多的时间花费是一样长，这是因为虽然进程数量是我们分配的，OpenMp的并行执行，对线程的创建是动态的，它会尽可能地在充分利用处理器资源的同时，减少调度开销。即，虽然我们分配进程少了，但是OpenMp动态分配的线程会多一些，来加快执行速度。

从上述分析，我们容易看出MPI的控制是需要人为控制，编程会更加麻烦，而且效果不一定会好，但这一定是能得到尽可能大的加速比的方法。而OpenMp是编译器自由化，在运行时自己进行动态调整优化，程序员只需要之处需要优化的地方即可，这种方法虽然编程简单，加速稳定，但是不易进行优化。如之前的实验当中，我们通过巧妙的算法设计，使得MPI计算矩阵乘法的性能大大超过了OpenMp的性能。

# 附录

//pthread.c

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define LA 1000

#define LB 1000

#define LC 1000

#define num 50

#define diva 3

#define divc 3

//we should use only 9 threads at most to run it

typedef struct{

int \*m;

int a,b;

}Matrix;

typedef struct{

int x,y;

int xlen, ylen;

}Point;

Matrix \*a,\*b,\*c;

pthread\_t hThread[diva][divc];

Point array[diva][divc];

int Myatoi(char \*c){

int sum = 0;

while(\*c != '\0'){

sum \*= 10;

sum += (\*c - '0');

c++;

}

return sum;

}

//(ax, ax+alen)\*(cy, cy+clen) with b

void \*MultMatrix(void \*arg){

int i, j, k;

int ax, alen, cy, clen;

Point \*p = (Point \*)arg;

ax = p->x;

alen = p->xlen;

cy = p->y;

clen = p->ylen;

for(i=ax; i<(ax+alen); i++)

for(j=cy; j<(cy+clen); j++)

for(k=0; k<a->b; k++)

\*(c->m+i\*c->b+j) += (\*(a->m+i\*a->b+k))\*(\*(b->m+k\*b->b+j));

}

void allocMm(Matrix \*\*m, int x,int y,int init){

\*m = (Matrix \*)malloc(sizeof(Matrix));

(\*m)->a = x;

(\*m)->b = y;

(\*m)->m = (int \*)malloc(sizeof(int)\*x\*y);

if(!init){

memset((\*m)->m,0,sizeof(int)\*x\*y);

return ;

}

int i,j;

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

for(j=0;j<y;j++)

\*((\*m)->m+i\*y+j) = rand()%num;

}

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

srand(time(0));

int la, lb, lc;

if(argc==2){

la = lb = lc = atoi(\*(argv+1));

}

else if(argc==4){

la = Myatoi(\*(argv+1));

lb = Myatoi(\*(argv+2));

lc = Myatoi(\*(argv+3));

}

else if(argc==1){

printf("use 1000, 1000, 1000\n");

la = LA; lb = LB; lc = LC;

}

else{

if(argc>1) printf("argc error,use 1000, 1000, 1000\n");

la = LA; lb = LB; lc = LC;

}

//alloc the matrix

allocMm(&a, la, lb, 1);

allocMm(&b, lb, lc, 1);

allocMm(&c, la, lc, 0);

int i,j;

int adis = la/diva;

int cdis = lc/divc;

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

for(j=0; j<divc; j++){//x, y. xlen, ylen

array[i][j].x = i\*adis;

array[i][j].y = i\*cdis;

if(i+1==diva) array[i][j].xlen = la - i\*adis;

else array[i][j].xlen = adis;

if(j+1==divc) array[i][j].ylen = lc - i\*cdis;

else array[i][j].ylen = cdis;

if(pthread\_create(&hThread[i][j], NULL, (void \*)MultMatrix, (void \*)&array[i][j])){

printf("create error!\n");

return 0;

}

}

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

for(j=0; j<divc; j++)

pthread\_join(hThread[i][j], NULL);

return 0;

}

//OpenMp.c

#include <stdio.h>

#include "omp.h"

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define LA 1000

#define LB 1000

#define LC 1000

#define num 50

#define diva 3

#define divc 3

//we should use only 9 threads at most to run it

typedef struct{

int \*m;

int a,b;

}Matrix;

int Myatoi(char \*c){

int sum = 0;

while(\*c != '\0'){

sum \*= 10;

sum += (\*c - '0');

c++;

}

return sum;

}

Matrix \*a,\*b,\*c;

void openmp\_multmatrix(){

int i,j,k;

int index;

int all = a->a \* b->b;

#pragma omp parallel private(i,j,k)

for(index=0; index<all; index++){

i = index/b->b;

j = index%b->b;

for(k=0; k<a->b; k++){

\*(c->m+i\*c->b+j) += (\*(a->m+i\*a->b+k))\*(\*(b->m+k\*b->b+j));

}

}

}

void allocMm(Matrix \*\*m, int x,int y,int init){

\*m = (Matrix \*)malloc(sizeof(Matrix));

(\*m)->a = x;

(\*m)->b = y;

(\*m)->m = (int \*)malloc(sizeof(int)\*x\*y);

if(!init){

memset((\*m)->m,0,sizeof(int)\*x\*y);

return ;

}

int i,j;

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

for(j=0;j<y;j++)

\*((\*m)->m+i\*y+j) = rand()%num;

}

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

srand(time(0));

int la, lb, lc;

if(argc==2){

la = lb = lc = atoi(\*(argv+1));

}

else if(argc==4){

la = Myatoi(\*(argv+1));

lb = Myatoi(\*(argv+2));

lc = Myatoi(\*(argv+3));

}

else if(argc==1){

printf("use 1000, 1000, 1000\n");

la = LA; lb = LB; lc = LC;

}

else{

if(argc>1) printf("argc error,use 1000, 1000, 1000\n");

la = LA; lb = LB; lc = LC;

}

//alloc the matrix

allocMm(&a, la, lb, 1);

allocMm(&b, lb, lc, 1);

allocMm(&c, la, lc, 0);

int i,j;

openmp\_multmatrix();

return 0;

}

//MPI.c

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#define LA 1600

#define LB 1600

#define LC 1600

#define num 50

//use 4\*4 = 16 pro

int Wp = 4;

int Hp = 4;

typedef struct{

int \*m;

int a,b;

}Matrix;

int Myatoi(char \*c){

int sum = 0;

while(\*c != '\0'){

sum \*= 10;

sum += (\*c - '0');

c++;

}

return sum;

}

Matrix \*a,\*b,\*c;

MPI\_Status status;

void allocMm(Matrix \*\*m, int x,int y){

\*m = (Matrix \*)malloc(sizeof(Matrix));

(\*m)->a = x;

(\*m)->b = y;

(\*m)->m = (int \*)malloc(sizeof(int)\*x\*y);

}

int posx, posy, xstep, ystep;

int xB, xE, yB, yE;

int \*buf, \*bufa, \*bufb;

void init\_Mm(int id, int x, int y, int same){

int i,j,k;

int all = Wp\*Hp;

xstep = x/all;

ystep = y/all;

xB = id \* xstep;

yB = id \* ystep;

xE = (id+1)\*xstep;

yE = (id+1)\*ystep;

bufa = (int \*)malloc(sizeof(int)\*ystep\*same);

bufb = (int \*)malloc(sizeof(int)\*xstep\*same);

int con = 0;

for(i=yB;i<yE;i++)

for(k=0;k<same;k++){

\*(a->m+i\*a->b+k) = rand()%num;

\*(bufa+con) = \*(a->m+i\*a->b+k);

con++;

}

con = 0;

for(i=xB;i<xE;i++)

for(k=0;k<same;k++){

\*(b->m+i\*b->b+k) = rand()%num;

\*(bufb+con) = \*(b->m+i\*b->b+k);

con++;

}

}

//Send - oid

//Recive - even

void init\_con(int id, int same){

int i,j,k;

posy = id/Wp;//get group

int bb = posy\*Hp, ee = (posy+1)\*Hp;

int con;//send once recive k-1

int \*buffa = (int \*)malloc(sizeof(int)\*ystep\*same);

int \*buffb = (int \*)malloc(sizeof(int)\*xstep\*same);

for(i=bb;i<ee;i++){

//i send

if(i==id){

for(j=bb;j<ee;j++){

if(j==id) continue;

MPI\_Send(bufa, ystep\*same, MPI\_INT, j, 1, MPI\_COMM\_WORLD);//send a

MPI\_Send(bufb, xstep\*same, MPI\_INT, j, 2, MPI\_COMM\_WORLD);//send b

}

}

else{

MPI\_Recv(buffa, ystep\*same, MPI\_INT, i, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(buffb, xstep\*same, MPI\_INT, i, 2, MPI\_COMM\_WORLD, &status);

con = 0;

xB = i \* xstep;

yB = i \* ystep;

xE = (i+1)\*xstep;

yE = (i+1)\*ystep;

for(j=yB;j<yE;j++)

for(k=0;k<same;k++){

\*(a->m+j\*a->b+k) = \*(buffa+con);

con++;

}

con = 0;

for(j=xB;j<xE;j++)

for(k=0;k<same;k++){

\*(b->m+j\*b->b+k) = \*(buffb+con);

con++;

}

}

}

}

void Fill(int id, int x, int y, MPI\_Status \*status){

int i,j;

posx = id%Wp;

posy = id/Hp;

xstep = x/Wp;

ystep = y/Hp;

xB = posx \* xstep;

yB = posy \* ystep;

xE = (posx+1)\*xstep;

yE = (posy+1)\*ystep;

//buf = (int \*)malloc(sizeof(int)\*xstep\*ystep);

if(id) MPI\_Recv(buf, xstep\*ystep, MPI\_INT, id, 0, MPI\_COMM\_WORLD, status);

int con = 0;

for(i=xB;i<xE;i++)

for(j=yB;j<yE;j++){

\*(c->m+i\*x+j) = \*(buf+con);

con++;

}

}

void count(int id, int x, int y, int over){

int i,j,k,con = 0;

posx = id%Wp;

posy = id/Hp;

xstep = x/Wp;

ystep = y/Hp;

xB = posx \* xstep;

yB = posy \* ystep;

xE = (posx+1)\*xstep;

yE = (posy+1)\*ystep;

buf = (int \*)malloc(xstep\*ystep\*sizeof(int));//

for(i=yB;i<yE;i++)

for(j=xB;j<xE;j++){

for(k=0;k<over;k++)

\*(buf+con) += (\*(a->m+i\*a->a+k)) \* (\*(b->m+k\*b->a+j));

con++;

}

}

int nPNum, Pid;

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

srand(time(0));

int la, lb, lc;

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_size(MPI\_COMM\_WORLD,&nPNum);

if(argc==2){

la = lb = lc = atoi(\*(argv+1));

}

else if(argc==4){

la = Myatoi(\*(argv+1));

lb = Myatoi(\*(argv+2));

lc = Myatoi(\*(argv+3));

}

else if(argc==1){

printf("use 1600, 1600, 1600\n");

la = LA; lb = LB; lc = LC;

}

else{

if(argc>1) printf("argc error,use 1600, 1600, 1600\n");

la = LA; lb = LB; lc = LC;

}

if(la%(Wp\*Hp)!=0||lc%(Wp\*Hp)!=0) {

printf("adjust the Wp, Hp or change the input");

}

//alloc the matrix

allocMm(&a, la, lb);

allocMm(&b, lb, lc);

if(Pid==0){

init\_Mm(Pid, lc, la, lb);

init\_con(Pid, lb);

count(Pid, lc, la, lb);

allocMm(&c, la, lc);

int i;

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

Fill(i, lc, la, &status);

}

}

else{

init\_Mm(Pid, lc, la, lb);

init\_con(Pid, lb);

count(Pid, lc, la, lb);

MPI\_Send(buf, xstep\*ystep, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

MPI\_Finalize();

return 0;

}

MPI\_Finalize();

return 0;

}

//CUDA.c

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <cuda\_runtime.h>

//#pragma comment(lib, "cudart.lib")

#define LA 1024

#define LB 1024

#define LC 1024

#define num 50

typedef struct{

int \*m;

int a,b;

}Matrix;

cudaError\_t error;

int block\_size = 16;

bool init(){

int count;

int devID = 0;

cudaDeviceProp deviceProp;

cudaGetDeviceCount(&count);

if(count == 0){

printf("no device, exit\n");

return false;

}

error = cudaGetDevice(&devID);

error = cudaGetDeviceProperties(&deviceProp, devID);

if(error != cudaSuccess){

printf("device not perpared,exit\n");

return false;

}

block\_size = (deviceProp.major < 2) ? 16 : 32;

return true;

}

int Myatoi(char \*c){

int sum = 0;

while(\*c != '\0'){

sum \*= 10;

sum += (\*c - '0');

c++;

}

return sum;

}

Matrix \*a,\*b,\*c;

void allocMm(Matrix \*\*m, int x,int y,int init){

\*m = (Matrix \*)malloc(sizeof(Matrix));

(\*m)->a = x;

(\*m)->b = y;

(\*m)->m = (int \*)malloc(sizeof(int)\*x\*y);

if(!init){

memset((\*m)->m,0,sizeof(int)\*x\*y);

return ;

}

int i,j;

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

for(j=0;j<y;j++)

\*((\*m)->m+i\*y+j) = rand()%num;

}

template <int BLOCK\_SIZE> \_\_global\_\_ void

matrixMulCuda(int \*c, int \*a, int \*b, int wA, int wB){

int bx = blockIdx.x;

int by = blockIdx.y;

int tx = threadIdx.x;

int ty = threadIdx.y;

int aBegin = wA \* BLOCK\_SIZE \* by;

int aEnd = aBegin + wA - 1;

int aStep = BLOCK\_SIZE;

int bBegin = BLOCK\_SIZE \* bx;

int bStep = BLOCK\_SIZE \* wB;

int Cadd = 0;

for (int i = aBegin, j = bBegin;i <= aEnd; i += aStep, j += bStep)

{

\_\_shared\_\_ float As[BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ float Bs[BLOCK\_SIZE][BLOCK\_SIZE];

As[ty][tx] = a[i + wA \* ty + tx];

Bs[ty][tx] = b[j + wB \* ty + tx];

\_\_syncthreads();

for (int k = 0; k < BLOCK\_SIZE; ++k)

{

Cadd += As[ty][k] \* Bs[k][tx];

}

\_\_syncthreads();

}

int w = wB \* BLOCK\_SIZE \* by + BLOCK\_SIZE \* bx;

c[w + wB \* ty + tx] = Cadd;

}

void matrixMul(Matrix \*a, Matrix \*b, Matrix \*c, int block\_size){

int \*d\_a, \*d\_b, \*d\_c;

cudaMalloc((void \*\*)&d\_a, sizeof(int)\*a->a\*a->b);

cudaMalloc((void \*\*)&d\_b, sizeof(int)\*b->a\*b->b);

cudaMalloc((void \*\*)&d\_c, sizeof(int)\*c->a\*c->b);

error = cudaMemcpy(d\_a, a->m, sizeof(int)\*a->a\*a->b, cudaMemcpyHostToDevice);

error = cudaMemcpy(d\_b, b->m, sizeof(int)\*b->a\*b->b, cudaMemcpyHostToDevice);

if(error != cudaSuccess) {

printf("copy failed from host to device, exit\n");

exit(0);

}

dim3 threads(block\_size, block\_size);

dim3 grid(b->b/threads.x, a->a/threads.y);

if(block\_size==16)

matrixMulCuda<16><<< grid, threads >>>(d\_c, d\_a, d\_b, a->b, b->b);

else

matrixMulCuda<32><<< grid, threads >>>(d\_c, d\_a, d\_b, a->b, b->b);

error = cudaMemcpy(c->m, d\_c, sizeof(int)\*c->a\*c->b, cudaMemcpyDeviceToHost);

if(error != cudaSuccess) {

printf("copy failed from device to host, exit\n");

exit(0);

}

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

}

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

srand(time(0));

if(!init()) exit(0);

int la, lb, lc;

if(argc==2){

la = lb = lc = atoi(\*(argv+1));

}

else if(argc==4){

la = Myatoi(\*(argv+1));

lb = Myatoi(\*(argv+2));

lc = Myatoi(\*(argv+3));

}

else if(argc==1){

printf("use 1024, 1024, 1024\n");

la = LA; lb = LB; lc = LC;

}

else{

if(argc>1) printf("argc error,use 1024, 1024, 1024\n");

la = LA; lb = LB; lc = LC;

}

if(!la%block\_size || !lb%block\_size || !lc%block\_size){

printf("set matrix's parameter%(%d) = 0\n",block\_size);

return;

}

//alloc the matrix

allocMm(&a, la, lb, 1);

allocMm(&b, lb, lc, 1);

allocMm(&c, la, lc, 0);

matrixMul(a, b, c, block\_size);

return 0;

}

//Monte-Carl.c

#include <stdio.h>

#include <stdlib.h>

#include "omp.h"

#include "mpi.h"

#define BORD 100

#define CON 10000

#define NumPro 16

int buf[NumPro];

int Myatoi(char \*c){

int sum = 0;

while(\*c != '\0'){

sum \*= 10;

sum += (\*c - '0');

c++;

}

return sum;

}

int nPNum, Pid;

int Ycon;

int yBord;

int fun(int x){

return x\*x;

}

void test(int t){

int i, x, y;

yBord = fun(BORD);

Ycon = 0;

#pragma omp parallel for

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

x = rand()%(BORD+1);

y = rand()%(yBord+1);

if(y <= fun(x)){

#pragma omp critical

Ycon = Ycon + 1;

}

}

}

MPI\_Status status;

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

int all;

srand(time(0));

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_size(MPI\_COMM\_WORLD,&nPNum);

if(argc == 2) all = Myatoi(\*(argv+1)) \* CON;

else all = CON;

if(Pid==0){

test(all/NumPro);

int i;

buf[0] = Ycon;

for(i=1; i<NumPro; i++){

MPI\_Recv(buf+i, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, &status);

}

int sum = 0;

#pragma omp parallel for reduction(+:sum)

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

sum += buf[i];

}

printf("the area of fun from [0,%d] is %.2f\n",BORD,(double)(sum)/(double)(all)\*(double)(BORD\*yBord));

}

else{

test(all/NumPro);

MPI\_Send(&Ycon, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

MPI\_Finalize();

return 0;

}

MPI\_Finalize();

return 0;

}